Blueprint Developer Manual

CoreMedia Content Cloud – v13

Copyright CoreMedia GmbH © 2026

CoreMedia GmbH

Altes Klöpperhaus, 5. OG

Rödingsmarkt 9

20459 Hamburg

International

All rights reserved. No part of this manual or the corresponding program may be reproduced or copied in any form (print, photocopy or other process) without the written permission of CoreMedia GmbH.

Germany

Alle Rechte vorbehalten. CoreMedia und weitere im Text erwähnte CoreMedia Produkte sowie die entsprechenden Logos sind Marken oder eingetragene Marken der CoreMedia GmbH in Deutschland. Alle anderen Namen von Produkten sind Marken der jeweiligen Firmen.

Das Handbuch bzw. Teile hiervon sowie die dazugehörigen Programme dürfen in keiner Weise (Druck, Fotokopie oder sonstige Verfahren) ohne schriftliche Genehmigung der CoreMedia GmbH reproduziert oder vervielfältigt werden. Unberührt hiervon bleiben die gesetzlich erlaubten Nutzungsarten nach dem UrhG.

Licenses and Trademarks

All trademarks acknowledged.
July 22, 2026 (Release 2512.1)

1. Preface 1
1.1. Audience 2
1.2. Typographic Conventions 3
1.3. CoreMedia Services 5
1.3.1. Registration 5
1.3.2. CoreMedia Releases 6
1.3.3. Documentation 7
1.3.4. CoreMedia Training 10
1.3.5. CoreMedia Support 10
1.4. Working with CoreMedia Content Cloud 12
1.4.1. Getting an Overview 12
1.4.2. Learning about Components 13
1.4.3. Working with the GUI 13
1.4.4. Operating the System 14
1.4.5. Extending the System 14
1.5. Change Chapter 17
2. Overview of CoreMedia Content Cloud 18
2.1. Components and Architecture 20
2.1.1. Content Management Environment 21
2.1.2. Content Delivery Environment 22
2.1.3. Shared Components 23
2.1.4. User Management 26
2.1.5. Communication Between the Components 26
2.2. CoreMedia Blueprint Sites 29
3. Getting Started 31
3.1. Prerequisites 32
3.1.1. Developer Setup 35
3.1.2. Test System Setup 36
3.1.3. Additional Software for eCommerce Blueprint only 37
3.2. Quick Start 38
3.2.1. Building the Workspace 38
3.2.2. Docker Compose Setup 44
4. Blueprint Workspace for Developers 57
4.1. Concepts and Architecture 58
4.1.1. Maven Concepts 58
4.1.2. Blueprint Base Modules 61
4.1.3. Application Architecture 62
4.1.4. Structure of the Workspace 67
4.1.5. Project Extensions 77
4.1.6. Application Plugins 82
4.2. Configuring the Workspace 105
4.2.1. Enabling Or Removing Optional Components 105
4.2.2. Configuring the Workspace 116
4.2.3. Configuring Local Setup 116
4.3. Build and Run the Applications 119
4.3.1. Starting Applications using IntelliJ IDEA 122
4.3.2. Starting Applications using the Command Line 123
4.3.3. Local Docker Test System 126

4.4. Development 127
4.4.1. Using Blueprint Base Modules 127
4.4.2. Extending Content Types 133
4.4.3. Developing with Studio 135
4.4.4. Developing with the CAE 138
4.4.5. Quality Assurance 140
4.4.6. Customizing the CAE Feeder 141
4.4.7. Handling Personal Data 141
5. CoreMedia Blueprint - Functionality for Websites 147
5.1. Overview of eCommerce Blueprint 148
5.2. Overview of Brand Blueprint 152
5.3. Basic Content Management 154
5.3.1. Common Content Types 154
5.3.2. Tagging and Taxonomies 161
5.4. Website Management 171
5.4.1. Folder and User Rights Concept 171
5.4.2. Navigation and Contexts 173
5.4.3. Settings 175
5.4.4. Page Assembly 177
5.4.5. Overwriting Product Teaser Images 189
5.4.6. Content Lists 190
5.4.7. View Types 191
5.4.8. CMS Catalog 193
5.4.9. Teaser Management 196
5.4.10. Dynamic Templating 197
5.4.11. View Repositories 200
5.4.12. Client Code Delivery 201
5.4.13. Managing End User Interactions 204
5.4.14. Images 208
5.4.15. URLs 211
5.4.16. Vanity URLs 212
5.4.17. External URLs 213
5.4.18. Content Visibility 214
5.4.19. Content Type Sitemap 215
5.4.20. Robots File 216
5.4.21. Sitemap 220
5.4.22. Website Search 224
5.4.23. Topic Pages 231
5.4.24. Search Landing Pages 236
5.4.25. Theme Importer 237
5.4.26. Tag Management 238
5.5. Localized Content Management 239
5.5.1. Concept 239
5.5.2. Administration 246
5.5.3. Development 252
5.6. Personalized Content Management 289
5.6.1. Concept 290
5.6.2. Development 293
5.7. Workflow Management 299

5.7.1. Publication 299
5.7.2. Translation Workflow 306
5.7.3. Deriving Sites 326
5.7.4. Synchronization Workflow 327
6. Editorial and Backend Functionality 328
6.1. Studio Enhancements 329
6.1.1. Content Query Form 329
6.1.2. Call-to-Action Button 331
6.1.3. Media Player Configuration 332
6.1.4. Displayed Date 333
6.1.5. Library 334
6.1.6. Bookmarks 336
6.1.7. External Preview 336
6.1.8. Settings for Studio 337
6.1.9. Content Creation 338
6.1.10. Create from Template 344
6.1.11. Open Street Map 346
6.1.12. Site Selection 347
6.1.13. Upload Files 348
6.1.14. Studio Preview Slider 352
6.1.15. Uploading Content to Salesforce Marketing Cloud 355
6.2. CAE Enhancements 357
6.2.1. Using Dynamic Fragments in HTML Responses 357
6.2.2. Image Cropping in CAE 361
6.2.3. RSS Feeds 362
6.3. Elastic Social 363
6.3.1. Configuring Elastic Social 364
6.3.2. Displaying Custom Information in Studio 370
6.3.3. Adding Custom Filters for Moderation View 373
6.3.4. Emails 374
6.3.5. Resend Registration Confirmation Mail from *Studio* 375
6.3.6. Curated transfer 376
6.3.7. 376
6.3.8. reCAPTCHA 376
6.3.9. Sign Cookie 377
6.4. Third-Party Integration 378
6.4.1. Open Street Map Integration 378
6.4.2. Personalization Hub 379
6.4.3. Analytics Connectors Integration 407
6.4.4. Salesforce Marketing Cloud Integration 411
6.4.5. Pendo Integration 412
6.5. Advanced Asset Management 413
6.5.1. Product Asset Widget 414
6.5.2. Replaced Product and Category Images 416
6.5.3. Extract Image Data During Upload 419
6.5.4. Configuring Asset Management 421
7. Reference 429

7.1. Content Type Model 430
7.2. Link Format 433
7.3. Predefined Users 439
7.4. Database Users 445
7.5. Cookies 447
Glossary 448
Index 456

List of Figures

2.1. System Overview 21
4.1. CoreMedia CMS's Four-Tier Architecture 68
4.2. CoreMedia CMS's Shared, Application-Specific, and Global Workspaces 69
4.3. Backend Tier Workspace Dependencies 70
4.4. Middle Tier Workspace Dependencies 70
4.5. CoreMedia Extensions Overview 79
5.1. Calista (Experience-led) start page for different devices: desktop, tablet, mobile 149
5.2. Hybris (commerce-led) start page for different devices: desktop, tablet, mobile 150
5.3. Chef Corp. start page for different devices: desktop, tablet, mobile 153
5.4. Dynamic list of articles tagged with "Black" 161
5.5. Taxonomy Administration Editor 163
5.6. Taxonomy Property Editor 164
5.7. Taxonomy Studio Settings 164
5.8. Taxonomy Localization Form 170
5.9. Navigation in the Site 173
5.10. The page grid editor 180
5.11. An inheriting placement 180
5.12. A locked placement 181
5.13. The layout chooser combo box 181
5.14. Layout Variant selector 192
5.15. CMS Catalog Settings 195
5.16. Default view and teaser view of an Article 196
5.17. Content Type Sitemap 216
5.18. `Robots.txt` settings 218
5.19. Channel settings with configuration for `Robots.txt` as a linked setting on a root page 219
5.20. Selection of a sitemap setup 222
5.21. Search Configuration Settings content item 226
5.22. Generated topic page for topic "Professionals" 231
5.23. The topic pages administration in Studio 233
5.24. Settings content items for topic pages 234
5.25. A Search Result for a Topic Page 235
5.26. Tag Management Configuration 238
5.27. Multi-Site Interdependence 245
5.28. Locales Administration in CoreMedia Studio 247
5.29. Derive Site: Setting site manager group 251
5.30. Site Indicator: Setting site manager group 251
6.1. Content Query Form 331
6.2. Call-to-Action-Button editor 332
6.3. Call-to-Action button in banner view 332
6.4. Video Options panel in the `DocumentForm` of a Video content 333

6.5. Displayed Date editor 334
6.6. Setting a Custom Date 334
6.7. Image Gallery Creation Button 335
6.8. Image Gallery Creation Dialog 335
6.9. Library List View 336
6.10. Bookmarks 336
6.11. External Preview 337
6.12. Create content menu on the Header toolbar 338
6.13. Create content dialog 338
6.14. Create content dialog for pages 338
6.15. New content dialog as button on a link list toolbar 339
6.16. New content dialog menu on a link list toolbar 340
6.17. Create from template dialog 344
6.18. OpenStreetMap Property Editor 347
6.19. The site selector on the Header bar 347
6.20. The upload files dialog 349
6.21. The slider of the Studio Preview 352
6.22. SFMC Uploadable Properties Setting 356
6.23. Example for an Open Street Map integration in a website 378
6.24. Evergage settings item 387
6.25. Dynamic Yield settings item 387
6.26. Create CMExperienceDefinitions in Studio 388
6.27. Configure experience definition in Studio 389
6.28. Evergage Experiences panel 390
6.29. Adding JavaScript code to the variants 390
6.30. Defining Segment experiences in Evergage 392
6.31. Evergage add JavaScript to experience 393
6.32. Create final experience for Evergage 394
6.33. Create campaign in Dynamic Yield 395
6.34. Edit Dynamic Yield experience 395
6.35. Creating campaign for Dynamic Yield segmentation 396
6.36. Dynamic Yield configure experience for segmentation 396
6.37. Dynamic Yield add JavaScript to segment 397
6.38. Properties for Monetate connection 399
6.39. Create Action in Monetate 401
6.40. Monetate action screen 401
6.41. Monetate enter data for new action 402
6.42. Create experience in Monetate 402
6.43. Copy JavaScript code from Studio 403
6.44. Insert JavaScript code into Monetate action 403
6.45. Define Monetate experience 404
6.46. Add code to variants 405
6.47. Add an experience for AI powered segmentation 405
6.48. Experience for AI powered content selection 406
6.49. Place Final experience 407
6.50. Product image gallery in HCL Commerce delivered by the CMS 414
6.51. Assign a product to a picture 415
6.52. Define Product Image URLs in Management Center 416

6.53. Screenshot from Adobe Photoshop for a Picture containing XMP Data 420
6.54. Picture linked to XMP Product Reference 420
6.55. Configuration of the download portal 427
6.56. Taxonomy for assets 428
7.1. CoreMedia Blueprint Content Type Model - CMLocalized 430
7.2. CoreMedia Blueprint Content Type Model - CMNavigation 431
7.3. CoreMedia Blueprint Content Type Model - CMHasContexts 431
7.4. CoreMedia Blueprint Content Type Model - CMMedia 431
7.5. CoreMedia Blueprint Content Type Model - CMCollection 432
7.6. A basic absoluteUrlPrefixes Struct 436
7.7. A complete absoluteUrlPrefixes Struct 437
7.8. An initial absoluteUrlPrefixes Struct 438

List of Tables

1.1. Typographic conventions 3
1.2. Pictographs 4
1.3. CoreMedia manuals 7
1.4. Changes 17
3.1. Overview of minimum / recommended Hardware requirements 36
4.1. Plugin Manager Properties 93
4.2. Optional modules and blueprints 105
4.3. Blueprint Extensions and Dependencies 107
4.4. Add-ons and the dependent extensions 112
4.5. Database Settings 117
4.6. Content type model dependencies 128
4.7. Parameters of the settings* methods 129
5.1. Overview of Content Types for common content 155
5.2. Commerce Content Types 156
5.3. Overview Commerce Content Properties 157
5.4. Overview Common Content Properties 157
5.5. `CMMedia` Properties 160
5.6. CMTaxonomy Properties 162
5.7. Additional CMLocTaxonomy Properties 162
5.8. CMLinkable Properties for Tagging 163
5.9. Properties of CMLinkable for Settings Management 176
5.10. Collection Types in CoreMedia Blueprint 190
5.11. CMS Catalog: Maven parent modules 193
5.12. Properties of `CMTeasable` 197
5.13. Properties of `CMTemplateSet` 199
5.14. Client Code - Properties of `CMAbstractCode` 201
5.15. Properties for Visibility Restriction 214
5.16. Brand website search settings 226
5.17. Page Grid Indexing Spring Properties 228
5.18. Options of the import-themes tool 237
5.19. Suggested Users and Groups for multi-site 249
5.20. Properties of the Site Model 253
5.21. Placeholders for Site Model Configuration 257
5.22. Example for server export and import for multi-site 261
5.23. Translation Workflow Properties 284
5.24. XLIFF Properties 284
5.25. Content variant property names 293
5.26. Variant support per content type 294
5.27. Stable property configuration 295
5.28. Publishing content item: actions and effects 301
5.29. Publishing folders: actions and effects 302
5.30. Predefined publication workflow definitions 304
5.31. Predefined publication workflow steps 304
5.32. User options. 306
5.33. Attributes of GetDerivedContentsAction 311
5.34. Attributes of CreateTranslationTreeData 312

5.35. Attributes of FilterDerivedContentsAction 314
5.36. Attributes of GetSiteManagerGroupAction 315
5.37. Attributes of GetMultiSiteManagerGroupAction 316
5.38. Attributes of ExtractPerformerAction 318
5.39. Attributes of AutoMergeTranslationAction 318
5.40. Attributes of AutoMergeSyncAction 320
5.41. Attributes of CompleteTranslationAction 322
5.42. Attributes of RollbackTranslationAction 323
5.43. Attributes of CleanInTranslationFinalAction 325
6.1. Upload Settings 350
6.2. Root Channel Context Settings 364
6.3. Context Settings for Every Channel 366
6.4. Mail Templates 374
6.5. Settings for Open Street Map Integration 379
6.6. Evergage naming 389
6.7. Monetate properties for site connection 399
6.8. Google Analytics Tracking Configuration Options 410
6.9. Google Analytics Studio Configuration Options 411
6.10. Path segments in the image URL 417
7.1. `CapBlobHandler` 433
7.2. `CodeHandler` 433
7.3. `ExternalLinkHandler` 433
7.4. `PageActionHandler` 434
7.5. `PageHandler` 434
7.6. `PageRssHandler` 434
7.7. `PreviewHandler` 434
7.8. `StaticUrlHandler` 435
7.9. `TransformedBlobHandler` 435
7.10. Global groups 439
7.11. Global users 440
7.12. Site specific groups for Salesforce Commerce 440
7.13. Site specific users for Salesforce Commerce 441
7.14. Site specific groups for SAP Commerce 441
7.15. Site specific users for SAP Commerce 442
7.16. Site specific groups for *HCL Commerce* 442
7.17. Site specific users for *HCL Commerce* 443
7.18. Site specific groups Brand web presence 443
7.19. Site specific users Brand web presence 444
7.20. Application-Specific Database Users 445
7.21. Feature-Specific Database Users 445

List of Examples

4.1. Dependencies for a CoreMedia application 59
4.2. Adding the Base Component 65
4.3. cmBannedDependencies example 71
4.4. oneRepoEnforcerRule example 71
4.5. modularOneRepoEnforcerRule example 72
4.6. Specify the extension point 78
4.7. com.acme.myplugin.MyPluginConfiguration 84
4.8. com.acme.myplugin.MyExtension 85
4.9. pom.xml 85
4.10. PluginA plugin.properties 95
4.11. PluginABeansForPluginsContainer 95
4.12. PluginABeansForPlugins 95
4.13. PluginAConfiguration 96
4.14. PluginB plugin.properties 96
4.15. PluginBConfiguration 96
4.16. PluginA plugin.properties 97
4.17. SomeExtensionPointForA 97
4.18. PluginAConfiguration 97
4.19. PluginB plugin.properties 98
4.20. PluginBConfiguration 98
4.21. SomeExtensionPointForAImpl 98
4.22. content-hub-adapter-rss-2.0.4.json 102
4.23. Enable CoreMedia Elastic Social Extension 114
4.24. Remove CoreMedia eCommerce Extension 114
4.25. Remove CoreMedia Corporate Extension 115
4.26. Remove CoreMedia Product Asset Management Extension 115
4.27. Remove CoreMedia Analytics Connectors Extension 115
4.28. The Spring Bean Definition for the Map of Settings Finder 130
4.29. Adding Custom Settings Finder 131
4.30. Business Logic API 131
4.31. Settings Address Adapter 132
4.32. Address Proxy 132
4.33. src/SampleStudioPlugin.ts 136
4.34. jangaroo.config.js 137
4.35. Adding custom stub classes 144
5.1. Pagegrid example definition 183
5.2. A `robots.txt` file 217
5.3. `robots.txt` file generated by the example settings 219
5.4. A sitemap file 220
5.5. A sitemap file with last modification dates 220
5.6. A sitemap index file 221
5.7. Usage of import-themes 237
5.8. Multi-Site Folder Structure Example 243
5.9. Site Folder Structure Example 244
5.10. XML of locale Struct 246
5.11. CMLocalized 258

5.12. CMTeasable 259
5.13. XLIFF fragment 263
5.14. Transforming to Translation Items 264
5.15. Function to Determine Locales 265
5.16. Exporting XLIFF 265
5.17. Importing XLIFF 266
5.18. Importing XLIFF 267
5.19. Example for CapTranslateItemException 269
5.20. TranslatePropertyTransformer for XHTML 269
5.21. Example for CapXliffExportException 270
5.22. PropertyExportHandler for XHTML 271
5.23. XhtmlToXliffConverter 271
5.24. XHTML Example Input 274
5.25. XHTML as XLIFF Example Output 275
5.26. XliffXhtmlPropertyImportHandler 275
5.27. XliffToXhtmlConverter 276
5.28. Attribute Export 278
5.29. XHTML Example Input (Attributes) 280
5.30. XHTML as XLIFF Example Output (Attributes) 280
5.31. XLIFF Validation Error 280
5.32. Custom XLIFF XSD 280
5.33. Custom XLIFF XSD (Bean) 281
5.34. Importing Translatable Attributes 281
5.35. Importing Non-Translatable Attributes 282
5.36. `Example for a customTranslationWorkflowDerivedContentsStrategy` 283
5.37. Configuration Example for Translatable Property Paths 285
5.38. Usage of GetDerivedContentsAction 312
5.39. Usage of CreateTranslationTreeDataAction 313
5.40. Usage of FilterDerivedContentsAction 314
5.41. Usage of GetSiteManagerGroupAction 315
5.42. Usage of GetMultiSiteManagerGroupAction 317
5.43. Usage of ExtractPerformerAction 318
5.44. Usage of AutoMergeTranslationAction 320
5.45. Usage of AutoMergeSyncAction 321
5.46. Usage of CompleteTranslationAction 323
5.47. Usage of CompleteTranslationAction (implicit clean-in-translation) 323
5.48. Usage of RollbackTranslationAction 324
5.49. Usage of RollbackTranslationAction (implicit clean-in-translation) 324
5.50. Usage of CleanInTranslationFinalAction 326
6.1. Using the content query form 330
6.2. Add content creation dialog to link list with `quickCreateLinkListMenu` 339
6.3. Predicate Example 358
6.4. Predicate Customizer Example 358
6.5. Dynamic Include Link Scheme Example 359

6.6. Dynamic Include Handler Example 359
6.7. Root Channel Context Settings 365
6.8. Root Channel Context Settings 366
6.9. Context Settings for Every Channel 370
6.10. Adding submodules 382
6.11. Checkout branch in submodule 382
6.12. Commit changes to submodules 383
6.13. Activate extensions 383
6.14. Updating an extension 383
6.15. Example Monetate configuration 398
6.16. Rendition Publication Configuration 425
7.1. Configuration of URL prefix type 437

1. Preface

This manual contains the basic knowledge you should have when you want to develop with *CoreMedia Content Cloud*. It describes the basic features and concepts of the development workspace, of the Commerce integration and of the Corporate Blueprint features.

- Chapter 2, *Overview of CoreMedia Content Cloud* [18] gives you an overview over the modules, functions and architecture of *CoreMedia Content Cloud*.
- Chapter 3, *Getting Started* [31] shows you step by step how to install and start the components using the *Blueprint* workspace.
- Chapter 4, *Blueprint Workspace for Developers* [57] explains in depth the concepts and patterns of the *Blueprint* workspace. You will learn how to release and deploy the system and how to develop in the workspace.
- Chapter 5, *CoreMedia Blueprint - Functionality for Websites* [147] explains the content types, the web functionality, details about localized content management and workflow management of *CoreMedia Content Cloud*.
- Chapter 6, *Editorial and Backend Functionality* [328] describes the extensions of *CoreMedia Blueprint* to the standard system.
- Chapter 7, *Reference* [429] contains reference information, such as the tag library, ports, the content type model or Maven profiles.

1.1 Audience

This manual is intended for architects and developers who want to work with *CoreMedia Content Cloud* or who want to learn about the concepts of the product. The reader should be familiar with *CoreMedia CMS*, the commerce system to connect with, *Spring*, *Maven* and *containerization*.

1.2 Typographic Conventions

CoreMedia uses different fonts and types in order to label different elements.
The following table lists typographic conventions for this documentation:

Element	Typographic format	Example
Source code Command line entries Parameter and values Class and method names Packages and modules	Courier new	`cm systeminfo start`
Menu names and entries	Bold, linked with |	Open the menu entry **Format|Normal**
Field names CoreMedia Components Applications	Italic	Enter in the field *Heading* The *CoreMedia Component* Use *Chef*
Entries	In quotation marks	Enter "On"
(Simultaneously) pressed keys	Bracketed in "<>", linked with "+"	Press the keys <Ctrl>+<A>
Emphasis	Italic	It is *not* saved
Buttons	Bold, with square brackets	Click on the **[OK]** button
Code lines in code examples which continue in the next line	\	`cm systeminfo \` `-u user`

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

Pictograph	Description
	Tip: This denotes a best practice or a recommendation.
	Warning: Please pay special attention to the text.
	Danger: The violation of these rules causes severe damage.

Table 1.2. Pictographs

1.3 CoreMedia Services

This section describes the CoreMedia services that support you in running a CoreMedia system successfully. You will find all the URLs that guide you to the right places. For most of the services you need a CoreMedia account. See Section 1.3.1, “Registration” [5] for details on how to register.

NOTE

CoreMedia User Orientation for CoreMedia Developers and Partners

Find the latest overview of all CoreMedia services and further references at:

http://documentation.coremedia.com/new-user-orientation

- Section 1.3.1, “Registration” [5] describes how to register for the usage of the services.
- Section 1.3.2, “CoreMedia Releases” [6] describes where to find the download of the software.
- Section 1.3.3, “Documentation” [7] describes the CoreMedia documentation. This includes an overview of the manuals and the URL where to find the documentation.
- Section 1.3.4, “CoreMedia Training” [10] describes CoreMedia training. This includes the training calendar,the curriculum and certification information.
- Section 1.3.5, “CoreMedia Support” [10] describes the CoreMedia support.

1.3.1 Registration

In order to use CoreMedia services you need to register. Please, start your initial registration via the CoreMedia website. Afterwards, contact the CoreMedia Support (see Section 1.3.5, “CoreMedia Support” [10]) by email to request further access depending on your customer, partner or freelancer status so that you can use the CoreMedia services.

1.3.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

CoreMedia provides its software as a Maven based workspace. You can download the current workspace or older releases via the following URL:

https://releases.coremedia.com/cmcc-13

Refer to our Blueprint Github mirror repository for recommendations to upgrade the workspace either via Git or patch files.

> **NOTE**
>
> If you encounter a 404 error then you are probably not logged in at GitHub or do not have sufficient permissions yet. See Section 1.3.1, "Registration" [5] for details about the registration process. If the problems persist, try clearing your browser cache and cookies.

Maven artifacts

CoreMedia provides parts of its release artifacts via Maven under the following URL:

https://repository.coremedia.com

You have to add your CoreMedia credentials to your Maven settings file as described in section Section 3.1, "Prerequisites" [32].

npm packages

CoreMedia provides parts of its release artifacts as npm packages under the following URL:

https://repository.coremedia.com/nexus/repository/coremedia-npm/

The `.npmrc` file must be configured to be able to use the registry (see Section 3.1, "Prerequisites" [32]).

License files

You need license files to run the CoreMedia system. Contact the support (see Section 1.3.5, "CoreMedia Support" [10]) to get your licences.

1.3.3 Documentation

CoreMedia provides extensive manuals, how-tos and Javadoc as PDF files and as online documentation at the following URL:

https://documentation.coremedia.com

The manuals have the following content and use cases:

Manual	Audience	Content
Blueprint Developer Manual	Developers, architects, administrators	This manual gives an overview over the structure and features of *CoreMedia Content Cloud*. It describes the content type model, the *Studio* extensions, folder and user rights concept and many more details. It also describes administrative tasks for the features. It also describes the concepts and usage of the project workspace in which you develop your CoreMedia extensions. You will find a description of the Maven structure, the virtualization concept, learn how to perform a release and many more.
Connector Manuals	Developers, administrators	This manuals gives an overview over the use cases of the eCommerce integration. It describes the deployment of the Commerce Connector and how to connect it with the CoreMedia and eCommerce system.
Content Application Developer Manual	Developers, architects	This manual describes concepts and development of the *Content Application Engine (CAE)*. You will learn how to write Freemarker templates that access the other CoreMedia modules and use the sophisticated caching mechanisms of the CAE.
Content Server Manual	Developers, architects, administrators	This manual describes the concepts and administration of the main CoreMedia component, the *Content Server*. You will learn about the content type model which lies at the heart of a CoreMedia system, about user and rights management, database configuration, and more.

Manual	Audience	Content
Deployment Manual	Developers, architects, administrators	This manual describes the concepts and usage of the CoreMedia deployment artifacts. That is the deployment archive and the Docker setup. You will also find an overview of the properties required to configure the deployed system.
Elastic Social Manual	Developers, architects, administrators	This manual describes the concepts and administration of the *Elastic Social* module and how you can integrate it into your websites.
Frontend Developer Manual	Frontend Developers	This manual describes the concepts and usage of the Frontend Workspace. You will learn about the structure of this workspace, the CoreMedia themes and bricks concept, the CoreMedia Freemarker facade API, how to develop your own themes and how to upload your themes to the CoreMedia system.
Headless Server Developer Manual	Frontend Developers, administrators	This manual describes the concepts and usage of the *Headless Server*. You will learn how to deploy the Headless Server and how to use its endpoints for your sites.
Multi-Site Manual	Developers, Multi-Site Administrators, Editors	This manual describes different otions to desgin your site hierarchy with several languages. It also gives guidance to avoid common pitfalls during your work with the multi-site feature.
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of *CoreMedia Studio*. You will learn about the underlying concepts, how to use the devel-

Manual	Audience	Content
		opment environment and how to customize *Studio* to your needs.
Studio User Manual	Editors	This manual describes the usage of *CoreMedia Studio* for editorial and administrative work. It also describes the usage of the *Native Personalization* and *Elastic Social* GUI that are integrated into *Studio*.
Studio Benutzerhandbuch	Editors	The Studio User Manual but in German.
Supported Environments	Developers, architects, administrators	This document lists the third-party environments with which you can use the CoreMedia system, Java versions or operation systems for example.
Unified API Developer Manual	Developers, architects	This manual describes the concepts and usage of the *CoreMedia Unified API*, which is the recommended API for most applications. This includes access to the content repository, the workflow repository and the user repository.
Utilized Open Source Software & 3rd Party Licenses	Developers, architects, administrators	This manual lists the third-party software used by CoreMedia and lists, when required, the licence texts.
Workflow Manual	Developers, architects, administrators	This manual describes the *Workflow Server*. This includes the administration of the server, the development of workflows using the XML language and the development of extensions.

Table 1.3. CoreMedia manuals

If you have comments or questions about CoreMedia's manuals, contact the Documentation department:

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

CoreMedia's training department provides you with the training for your CoreMedia projects either live online, in the CoreMedia training center or at your own location.

You will find information about the CoreMedia training program, the training schedule and the CoreMedia certification program at the following URL:

https://www.coremedia.com/training

Contact the training department at the following email address:

Email: training@coremedia.com

1.3.5 CoreMedia Support

CoreMedia's support is located in Hamburg and accepts your support requests between 9 am and 6 pm MET. If you have subscribed to 24/7 support, you can always reach the support using the phone number provided to you.

To submit a support ticket, track your submitted tickets or receive access to our forums visit the CoreMedia Online Support at:

https://support.coremedia.com/

Do not forget to request further access via email after your initial registration as described in Section 1.3.1, “Registration” [5]. The support email address is:

Email: support@coremedia.com

Create a support request

Support request

CoreMedia systems are distributed systems that have a rather complex structure. This includes, for example, databases, hardware, operating systems, drivers, virtual machines, class libraries and customized code in many different combinations. That's why CoreMedia needs detailed information about the environment for a support case. In order to track down your problem, provide the following information:

- Which CoreMedia component(s) did the problem occur with (include the release number)?
- Which database is in use (version, drivers)?
- Which operating system(s) is/are in use?
- Which Java environment is in use?

- Which customizations have been implemented?
- A full description of the problem (as detailed as possible)
- Can the error be reproduced? If yes, give a description please.
- How are the security settings (firewall)?

In addition, log files are the most valuable source of information.

Support checklist

To put it in a nutshell, CoreMedia needs:

1. a person in charge (ideally, the CoreMedia system administrator)
2. extensive and sufficient system specifications
3. detailed error description
4. log files for the affected component(s)
5. if required, system files

Log files

An essential feature for the CoreMedia system administration is the output log of Java processes and CoreMedia components. They're often the only source of information for error tracking and solving. All protocolling services should run at the highest log level that is possible in the system context. For a fast breakdown, you should be logging at debug level. See Section 4.7, "Logging" in *Operations Basics* for details.

Which Log File?

In most cases at least two CoreMedia components are involved in errors: the *Content Server* log files together with the log file from the client. If you know exactly what the problem is, solving the problem becomes much easier.

Where do I Find the Log Files?

By default, application containers only write logs to the console output but can be accessed from the container runtime using the corresponding command-line client.

For the *docker* command-line client, logs can be accessed using the **docker logs** command. For a detailed instruction of how to use the command, see docker logs. Make sure to enable the timestamps using the `--timestamps` flag.

```
docker logs --timestamps <container>
```

For the *kubectl* command-line client in a Kubernetes environment you can use the **kubectl logs** command to access the logs. For a detailed instruction of how to use the command, see kubectl logs. Make sure to enable the timestamps using the `--timestamps` flag.

```
kubectl logs --timestamps <pod>
```

1.4 Working with CoreMedia Content Cloud

This chapter guides you to the download area, other manuals and training courses depending on your skills and the tasks you want to accomplish. CoreMedia documentation is organized in such a way, that each component manual contains all required information for the configuration, operation and development of the component. Only the user manuals for editors and other users are in separate documents.

Chapters and sections that have only a noun in the title usually contain conceptual information while a title with an "-ing" indicates an instructional chapter.

1.4.1 Getting an Overview

To start with *CoreMedia Content Cloud* you should open the following address in your browser:

https://releases.coremedia.com/cmcc-13

Here, you will find a short quick start description and links to all resources for *CoreMedia Content Cloud*. You can download all software artefacts and demo content.

With *CoreMedia Content Cloud* you do not get a program to install and run, but a workspace to develop within, to build and to deploy artifacts from.

- Read the Supported Environments document available at https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf to learn which databases, browsers, operation systems, Java versions, Portal version and servlet container are supported by *CoreMedia Content Cloud*.
- Read the Deployment Manual to learn how to install CoreMedia components with *CoreMedia Blueprint*.
- Read the [Blueprint Developer Manual] to learn about the *Blueprint* features.
- Read the Operations Basics manual to learn basic operation tasks.
- Read the Utilized Open Source Software & 3rd Party Licenses if you want to know which open source software is used by *CoreMedia Content Cloud*.
- Attend the "CoreMedia Administrator training" at the CoreMedia Training Center, see https://www.coremedia.com/training for the current schedule.

1.4.2 Learning about Components

If you want to get familiar with the concepts and coverage of *CoreMedia Content Cloud*, then this manual is the starting point. Nevertheless, it only gives you a rough insight. If you want to learn more about all the components that comprise *CoreMedia Content Cloud* you should read the following chapters:

- Read the Chapter 2, *Overview* in *Content Server Manual* to learn something about the basic component of the CoreMedia system.
- Read the "Overview" chapter in the manual of every component you are interested in.
- Attend the "CoreMedia Fundamentals" training at the CoreMedia Training Center, see https://www.coremedia.com/training for the current schedule.

1.4.3 Working with the GUI

CoreMedia Content Cloud comes with different GUIs that support different tasks, such as managing content and user generated content or personalize the output. Their usage is described in separate manuals or chapters shown below. All these manuals are intended for editors and other non-technical staff.

CoreMedia Studio

CoreMedia *Studio* is the editor tool for all users. It is web based and requires no installation. Its easy-to-use interface with instant preview and form based editing makes content creation easier than ever. All other CoreMedia components integrate their GUI into *CoreMedia Studio*. Create new content, access your catalog data, manage your website or user generated content or publish new content to your customers.

- Read the Studio User Manual for details.

Elastic Social

The *Elastic Social* GUI is integrated with *CoreMedia Studio*.

- Read Chapter 8, *Working with User Generated Content* in *Studio User Manual* for details.

Native Personalization Management

CoreMedia Native Personalization comes with a management GUI that bases on the same technology as *CoreMedia Studio*. It lets you define variants, AB-rests and customer segments.

- Read Chapter 7, *Working with Personalized Content* in *Studio User Manual* for details.

1.4.4 Operating the System

The components of *CoreMedia Content Cloud* are configured using properties and you can use JMX to manage them. In addition, *CoreMedia Content Cloud* contains tools to monitor the status of its components. The following chapters are intended for operators and administrators but developers should read the chapters as well.

- Read the Operations Basics for some operational concepts and tasks.
- Each component manual contains a configuration chapter. Read this chapter if you want to learn details about a component's configuration.

1.4.5 Extending the System

CoreMedia Content Cloud is a very flexible software system, which you can adapt to all your needs. It integrates nicely with a Maven based development environment. CoreMedia is shipped with manuals that cover general development concepts such as the workspace and the *Unified API* and with manuals that cover the development with specific components.

General Concepts

- Read the [Blueprint Developer Manual] to learn how to develop extensions using *Blueprint* workspace.
- Read the Unified API Developer Manual in order to learn how to use the most fundamental CoreMedia API.
- Read Chapter 4, *Developing a Content Type Model* in *Content Server Manual* in order to learn how to define your own content types.

Developing editorial components

If you want to develop components for editorial purpose, you might refer to one of the following manuals:

- Read the Studio Developer Manual in order to learn how to extend *CoreMedia Studio*.
- Read the Unified API Developer Manual in order to learn how to develop client applications from the scratch accessing the *CoreMedia CMS* via the *Unified API*.
- Attend the *CoreMedia Studio Customization* training in order to learn how to extend *CoreMedia Studio*, see https://www.coremedia.com/training for details.

Developing workflows

CoreMedia CMS contains a customizable *Workflow Server* that you can adapt to your needs. *CoreMedia CMS* is delivered with workflows that support publishing tasks, but the *Workflow Server* can support much more complicated processes.

- Read the Workflow Manual in order to learn how to define your own workflows.

Developing websites

CoreMedia CMS is a web content management system and its main purpose is to deliver content to various devices. Not only to a PC but to all gadgets such as mobile phones or tablet PCs.

- Read the Content Application Developer Manual in order to learn how to develop fast, dynamic websites that support sophisticated caching. Learn how to use the *CAE*.
- Read the Frontend Developer Manual in order to learn to write FreeMarker applications using the Frontend Workspace.
- Read the Headless Server Manual in order to learn how to access CoreMedia content via the *Headless Server* for your websites written with the framework of your choice.
- Read the Elastic Social Manual in order to learn how to extend your websites with user generated content, such as comments or ratings.
- Read the ???? in order to learn how to deliver personalized content.
- Read the Search Manual in order to learn how to make your websites searchable.
- Attend the *Content Application Engineering* training, in order to get hands-on experience in the development of CAE applications. See https://www.coremedia.com/training for details.

- Attend the *Frontend Development* training, in order to learn how to implement a new theme with FreeMarker, JavaScript and CoreMedia components. See https://www.coremedia.com/training for details.
- Attend the *CoreMedia Headless* training, in order to learn how to implement sites with content from the Headless Server using GraphQL. See https://www.coremedia.com/training for details.

1.5 Change Chapter

In this chapter you will find a table with all major changes made in this manual.

Section	Version	Description

Table 1.4. Changes

2. Overview of CoreMedia Content Cloud

CoreMedia Content Cloud is the next-generation experience management platform from CoreMedia that lets you build highly engaging, multi-channel branded eCommerce experiences as well as corporate sites for your global customers.

Now, you can easily bridge the gap between a pure eCommerce system which is focused on the more transactional aspects of the buying process and content-driven brand sites that focus on engaging user experiences.

CoreMedia Studio allows your business users to efficiently create and manage engaging digital experiences across the customer journey by adding editorial content and media assets from the CoreMedia CMS and by enriching the basic product information with storytelling by adding editorial content and media assets from the CoreMedia CMS. You can seamlessly blend catalog content and CMS content to any degree and on any delivery channel - and ensure brand-consistency through multi-language and multi-site localization tools.

The *CoreMedia Content Cloud* platform bundles all components to help you manage every aspect of your blended digital experiences from content to commerce:

- CoreMedia CMS platform
- *CoreMedia Studio*
- CoreMedia Blueprints for eCommerce and corporate sites
- *CoreMedia Commerce Hub and eCommerce Connectors*
- *CoreMedia Headless Server*
- *CoreMedia Elastic Social*
- *CoreMedia Native Personalization*
- *CoreMedia Advanced Asset Management*

CoreMedia Content Cloud was designed to empower your team in creating and managing highly relevant and engaging experiences for your customers from a single, easy-to-use business user interface. Customers should always get the information they need, independent of the device they use or the time they connect - delivered in an optimized fashion for the current customer's context.

CoreMedia Studio allows business users to create and manage experiences based on context and to define and test rules and customer segments for per-

sonalization in real-time. Content can be easily mixed with eCommerce catalog items. Editors can intuitively select the products and categories from the catalog and place them on the site just as they are accustomed from other web content.

CoreMedia Content Cloud ships with CoreMedia Blueprints for eCommerce and corporate sites that provide a high-level of prefabrication of common features and use cases. The source code is provided for easy customization to your specific needs for competitive differentiation.

Built upon industry-leading best practices with a fully responsive and adaptive mobile first design plus a wealth of ready-to-use layout modules, your development team can jump start on a strong foundation proven in many customer projects whilst retaining full flexibility. A predefined Maven based development environment is provided.

Leveraging the CoreMedia CAE technology, you can dynamically and contextually combine relevant content from CoreMedia CMS, *CoreMedia Elastic Social* and your eCommerce system and deliver the combined experience in real-time on all channels with utmost performance using the sophisticated caching.

Headless Server allows you using *CoreMedia Content Cloud* in a headless way. *Headless Server* delivers content from the repository as JSON data over a GraphQL endpoint and is fully integrated in *CoreMedia Studio*. Therefore, you can preview and edit all changes in Studio.

Elastic Social allows your end users to contribute user-generated content such as product reviews, comments and ratings - whilst providing an intuitive moderation interface to your business users that also allows for editorial re-purposing of user-generated content.

2.1 Components and Architecture

CoreMedia Content Cloud has been developed to provide a universal solution for the creation and management of content.

The use of modern development tools and open interfaces enables the system to be flexibly adapted to enterprise requirements. For this purpose, worldwide standards for information processing, such as XML, HTML, HTTP, REST, Ajax, gRPC, CORBA and the Java Platform are used or supported.

CoreMedia Content Cloud is a distributed system, that consists of several components for different use cases.

- *CoreMedia Content Server*
  - Content Management Server
  - Master Live Server
  - Replication Live Server
- *CoreMedia Workflow Server*
- *CoreMedia Content Application Engine*
- *CoreMedia Headless Server*
- *CoreMedia Search Engine*
  - CoreMedia Content Feeder
  - CoreMedia CAE Feeder
- *CoreMedia Commerce Hub*
- eCommerce Connectors
- *CoreMedia Studio*
- *CoreMedia User Changes Application*
- *CoreMedia Elastic Social*
- *CoreMedia Native Personalization*
- *CoreMedia Advanced Asset Management*
- *CoreMedia Blueprints*

In addition, *CoreMedia Content Cloud* relies on some third-party systems:

- An *HCL Commerce* Server or SAP Commerce Server or Salesforce Commerce Cloud or commercetools for Commerce
- A relational database to store the content and user data
- A MongoDB NoSQL database to store the user generated content
- An LDAP server for user management

Conceptually, a CoreMedia system can be divided into the *Content Management Environment* where editors create and manage the content and the *Content Delivery Environment* where the content is delivered to the customers. Some components are used in both environments, mostly to give you a realistic preview

of your websites. Figure 2.1, “System Overview” [21] provides an overview of a *CoreMedia Content Cloud* system with all components installed:

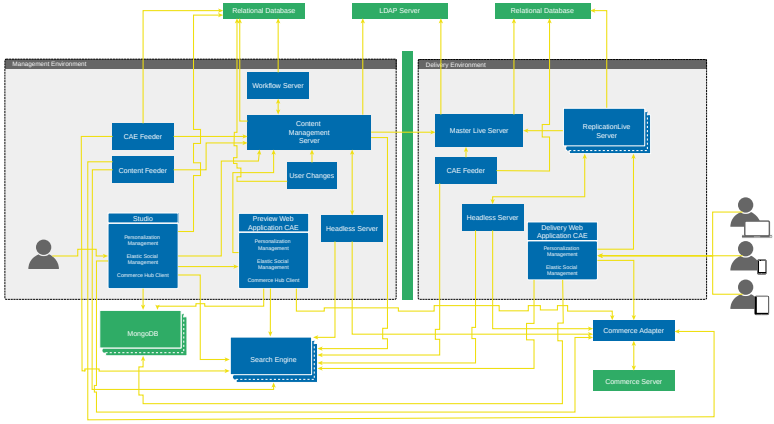

Figure 2.1. System Overview

The following sections describe in short the aim of all components, some main technologies used in *CoreMedia Content Cloud* and give a short overview over the communication between the components.

2.1.1 Content Management Environment

The Content Management Environment is the place where you create and manage your website with the *Content Management Server* and *Studio* at its heart. A freely adaptable content model allows you to manage and deliver every type of digital content including text, video, images, music and many more.

The following components are solely located in the Content Management Environment:

CoreMedia Content Management Server

The *Content Management Server* manages the content in *CoreMedia Content Cloud*.

CoreMedia Studio

Studio is a web application. It integrates the complete workflow used by online editors from the creation, over management to preview publication of digital

experiences with contextual content. *Studio* is a web application that bases on modern standards such as Ajax. Therefore, it can be used like a common desktop application; fast, reliable but without installation. *Studio* integrates the *CoreMedia Adaptive Personalization* and *Elastic Social* GUI and has an integrated preview window where you can see your content in its context. You can even see the effects of personalization or time-dependent publication.

With the use of eCommerce Connectors, *CoreMedia Studio* lets you access the content of the eCommerce system. Content can be mixed easily with commerce catalog items. Editors can intuitively select the items from the catalog and place them on the site just as they are accustomed from other web contents.

CoreMedia User Changes Application

The *CoreMedia User Changes Application* maintains lists of *My Edited Content*, which show the current work of the logged-in editor in *Studio*, and triggers *Notifications* for workflow activity in *Studio*. To that end, it listens to changes in the *Content Management Server* and *Workflow Server*.

CoreMedia Workflow Server

The *CoreMedia Workflow Server* is an application that executes and manages workflows. *CoreMedia Content Cloud* comes with predefined workflows for publication, translation and synchronization, but you can also define your own workflows.

CoreMedia Content Feeder

The *Content Feeder* is an application that collects the content from the *Content Management Server* and delivers it to the *Search Engine* for indexing. Thus, the *Content Feeder* is necessary to make content searchable in *Studio*. The *Content Feeder* listens for changes in the content and triggers the indexing of the changed or newly created content.

With the use of eCommerce Connectors, the *Content Feeder* lets you access items of the eCommerce system. This is needed if commerce issues should be integrated into the search in *Studio* as filter option. In this way, for example, invalid references to eCommerce items can be tracked down.

2.1.2 Content Delivery Environment

The Content Delivery Environment of *CoreMedia CMS* may consist of the *Master Live Server*, several *Replication Live Servers* (which are optional), the *CoreMedia*

CAE, the *Headless Server*, *CoreMedia Elastic Social*, the *Search Engine* and *Adaptive Personalization*. It manages the approved and published online data and adds user generated content.

CoreMedia Master Live Server

The *Master Live Server* manages the CoreMedia repository in the Content Delivery Environment. It receives this content from the *Content Management Server* during publication. The *Content Application Engine* or the *Headless Server* fetches the content from the *Master Live Server* or from the *Replication Live Servers*.

CoreMedia Replication Live Server

The optional *Replication Live Servers* replicate the content of the *Master Live Server* in order to enhance reliability and to add scalable performance.

2.1.3 Shared Components

Some components of *CoreMedia Content Cloud* are used in both environments. The *Commerce Hub*, for example, is used in the Management Environment to manage content from the eCommerce system in *Studio* and in the Delivery Environment to include content from the eCommerce system into the pages generated by *Content Application Engine*. Other components, like the *Content Application Engine*, are used to provide the editor with a preview of the live site.

Commerce Hub

Commerce Hub in combination with the eCommerce Connectors connects the CoreMedia CMS with the eCommerce server. It provides functionality to read catalog items, such as products or marketing spots, and to display them on web pages. You can also display price information and availability of products on the site. All commerce functions are provided by a commerce Java API that enables you to extend your shop application.

The eCommerce bridge also enables you to enrich pages rendered by the eCommerce system with content delivered by the CAE of *CoreMedia Content Cloud*. This way, you can enhance your shop pages with more engaging content.

Finally, the *CoreMedia eCommerce Bridge for IBM WebSphere Commerce* synchronizes user sessions between the *HCL Commerce* system and the CoreMedia system, so that users only have to sign in once.

CoreMedia Headless Server

CoreMedia Headless Server allows you to access CoreMedia content as JSON through a GraphQL endpoint. It provides clean APIs and easy access to content for all sorts of native apps, browser-based single-page applications or progressive web applications.

CoreMedia Studio integrates a preview of content delivered by the Headless Server.

With the use of eCommerce Connectors, the *CoreMedia Headless Server* lets you access items of the eCommerce system. This is mainly needed to compute page grid placements for commerce pages along the category hierarchy and to serve a mixed navigation.

CoreMedia Content Application Engine (CAE)

The *CoreMedia Content Application Engine* represents a stack for building client applications with *CoreMedia CMS*. It is a web application framework which allows fast development of highly dynamic, supportable and personalizable applications and websites. Sophisticated caching mechanisms allows for dynamic delivery even in high-load scenarios with automatic invalidation of changed content.

The *CoreMedia Content Application Engine* combines content from all CoreMedia components, from your eCommerce system and other third-party systems in so-called content beans and delivers the content to your customers in all formats. The preview in *Studio* and the website visited by your customers is delivered by the CAE

CoreMedia Search Engine

A *CoreMedia CMS* system comes with Apache Solr as the default search engine, which can be used from the editors on content management site and from the applications on content delivery site. The editor, for example, can perform a fast full text search in the complete repository. The pluggable search engine API allows you to use other search engines than Apache Solr for the website search.

CoreMedia CAE Feeder

The *CAE Feeder* makes content beans searchable by sending their data to the *Search Engine* for indexing.

CoreMedia Native Personalization

CoreMedia Native Personalization enables enterprises to deliver the most appropriate content to users depending on the 'context' – the interaction between the user, the device, the environment and the content itself. *CoreMedia Native Personalization* is a powerful personalization tool. Through a series of steps it can identify relevant content for individuals. It can draw on a user's profile, commerce segment, preferences and even social network behavior. Use *CoreMedia Native Personalization* to deliver highly relevant and personalized content to users, at any given moment in time.

The GUI is integrated into CoreMedia *Studio* for easy creation and testing of customer segments.

CoreMedia Elastic Social

CoreMedia Elastic Social enables enterprises to engage with users, entering a conversation with them and stimulating discussion between them. Use *Elastic Social* to enable Web 2.0 functionality for Web pages and start a vibrant community. It offers all the features it takes to build a community – personal profiles, preferences, relationships, ratings and comments. *CoreMedia Elastic Social* is fully customizable to reflect the environment you want to create, and offers unlimited horizontal scalability to grow with the community and your business vision. It also integrates with *CoreMedia Studio* so you can manage comments and external users right from your common workplace.

CoreMedia Advanced Asset Management

CoreMedia Advanced Asset Management is a module that adds asset management functionality to the system. Digital assets, such as images or documents, and their licenses can be managed in CoreMedia Studio. From an asset, you can create common content items that can be used in the eCommerce system.

CoreMedia Blueprint

For a quick start, *CoreMedia Content Cloud* is delivered with two fully customizable blueprint applications including best practices and example integration of available features. *CoreMedia Blueprint* contains a ready-made content model for navigation and multi-language support. It contains for instance solutions for eCommerce items, taxonomy, rating, integration with web analytics software and user created page layouts. *CoreMedia Blueprint* comes as a Maven based workspace for development.

The workspace is the result of CoreMedia's long year experience in customer projects. As *CoreMedia Content Cloud* is a highly customizable product adaptable to your specific needs, the first thing you used to do when you started to work with *CoreMedia Content Cloud* was to create a proper development environment on your own. *CoreMedia Content Cloud* addresses this challenge with a reference project in a predefined working environment that integrates all CoreMedia components and is ready for start.

Maven based environment

CoreMedia Blueprint workspace provides you with an environment which is strictly based on today's de facto standard for managing and building Java projects by using Maven.

For details on each component, please refer to the individual manuals. Online documentation for all these components is available online at https://documentation.coremedia.com/cmcc-13.

2.1.4 User Management

CoreMedia Content Cloud has an integrated user management, but also supports an LDAP server for user management.

Lightweight Directory Access Protocol (LDAP) is a set of protocols for accessing information directories. It is based on the standards within the X.500 standard, but is significantly simpler. Unlike X.500, LDAP supports TCP/IP, which is necessary for any type of Internet access. Because it's a simpler version of X.500, LDAP is sometimes called X.500-lite.

2.1.5 Communication Between the Components

Communication between the individual components on both the production side and the *Live Server* is performed via gRPC or CORBA and HTTP. MongoDB uses the Mongo Wire Protocol. The *Production* and *Live Systems* can be secured with a Firewall if the servers are located on different computers. The servers contact the databases over a JDBC interface,

CoreMedia Content Cloud and the commerce systems communicate over REST interfaces. The concrete communication differs slightly based on the selected deployment scenario which are the *content-led scenario* for *HCL Commerce* and the *commerce-led scenario*.

Internal Communication Protocol

The CoreMedia system supports gRPC and CORBA for internal communication. While servers always offer both protocols for clients to connect, clients can be configured to use *either* gRPC *or* CORBA. The protocol can be chosen freely on each start of a component. Currently, CORBA is the default protocol. See Section "Docker Compose Configuration" [47] and section "Internal Communication Protocol" [121] on how to set the protocol.

Note that Content Servers and the Workflow Server also contain client parts for internal communication, e.g., for publication, replication, etc. Choosing a protocol on a server therefore *does* make a difference although services *offered* by the servers to their clients always cover both protocols.

NOTE

With release 2512.0 of the CoreMedia system, the Workflow Server does not yet offer gRPC services to its clients. Activating gRPC on clients (e.g., Studio Server or command line tools) won't do any harm as long as these are of release 2512.0, too. *Future* clients, though, will fail to connect to that Workflow Server if gRPC is active on them. With releases 2512.1, 2606.0 and newer, the Workflow Server supports gRPC to the full extent.

Keep this in mind when operating a *mixed-release* CoreMedia system setup. In such scenarios (e.g., during upgrades), it is recommended to leave clients in their default protocol mode which is CORBA.

Processing

On the production side of the CoreMedia system, content is created and edited with *CoreMedia Studio* or custom clients. Once editing of contents is completed, they are approved and published via the *CoreMedia Workflow*. During the publication process, the content is put online onto the *Master Live Server*. If available, *Replication Live Servers* get noticed and reproduce the changes. Then the content is put online by the *Replication Live Server*. User generated content is produced via *Elastic Social* and is stored in MongoDB. Editors can use the *Studio* plugin to moderate this content.

Content from the commerce server is not copied into the CoreMedia system. Instead, references to the content are hold and are resolved when content is delivered.

The *CoreMedia CAE* in combination with *Native Personalization* and *Elastic Social* creates dynamic HTML pages or any other format (XML, PDF, etc.) from the internal and external content and CoreMedia templates.

Headless Server, on the other hand, delivers content from *CoreMedia Content Cloud* as JSON data via a GraphQL endpoint. This gives you full flexibility in choosing your frontend technology.

2.2 CoreMedia Blueprint Sites

CoreMedia Content Cloud Experience Platform contains Brand Blueprint and eCommerce Blueprint for a quick start. They come with four different sites that support different use cases.

Aurora Augmentation (en)

This site belongs to the eCommerce Blueprint. It is intended for a company that wants to extend their *HCL Commerce* B2C online shop with engaging assets and content from the CoreMedia system, the so called commerce-led scenario (see Chapter 6, *Commerce-led Integration Scenario* in *Connector for HCL Commerce Manual*). Editors can add inspiring content from the CMS such as images, videos, articles to the standard commerce pages. They do not need to enter the commerce system, but can use *CoreMedia Studio* for their work, taking advantage of the sophisticated preview of *Studio*.

Calista Augmentation (en)

This site belongs to the eCommerce Blueprint and implements the experience-led hybrid blended scenario, where pages are delivered by both systems, the corporate and the eCommerce system, transparent for the user. Therefore, it is intended for a company that wants to offer their corporate content as well as their eCommerce shop as one engaging experience for its users on all devices with a fully responsive design. You can manipulate the navigation through the catalog pages and add complete new navigation paths. You can augment product detail pages with content from the CMS. Categories are rendered from the CAE. However, content and settings are inherited along the catalog category structure.

Hybris Apparel (uk)

This site belongs to the eCommerce Blueprint. It is intended for a company that wants to extend their SAP Hybris Commerce B2C online shop with engaging assets and content from the CoreMedia system, the so called commerce-led scenario (see Chapter 5, *Com-*

	merce-led Integration Scenario in *Connector for SAP Commerce Cloud Manual*). Editors can add inspiring content from the CMS such as images, videos, articles to the standard shop pages. They do not need to enter the commerce system, but can use *CoreMedia Studio* for their work, taking advantage of the sophisticated preview of *Studio*.
Chef Corp. Site (en/de)	This site belongs to the Brand Blueprint. It is intended for a company that wants to offer their corporate site as an engaging experience for its users on all devices with a fully responsive design. The site contains no eCommerce shop, but the company can use the CoreMedia catalog to manage and present their products on the website.

Removing Sites

You can remove sites and features that you do not need from your workspace.

To remove the Aurora sites or Hybris Apparel site remove the eCommerce extension as described in Section 4.2.1.3, “Removing the eCommerce Blueprint” [114].

To remove the Brand Site, remove the corporate extension as described in Section 4.2.1.4, “Removing the Brand Blueprint” [114].

3. Getting Started

In this chapter you will learn the basics for the quickest way to get started using *CoreMedia Content Cloud*.

- Section 3.1, “Prerequisites” [32] describes the software and hardware requirements that you need to fulfill to work with *CoreMedia Content Cloud*.
- Section 3.2, “Quick Start” [38] describes the fastest way to get a CoreMedia system up and running.

3.1 Prerequisites

In order to work with the *Blueprint* workspace you need to meet some requirements.

NOTE

For an overview of exact versions of the supported software environments (Java, servlet container, databases, operating systems, directory services, web browsers) please refer to the Supported Environments document at https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf.

CoreMedia Account

In order to get access to the download page, to the CoreMedia contributions repository, the CoreMedia's Maven repository (https://repository.coremedia.com) and npm repository (https://repository.coremedia.com/nexus/repository/coremedia-npm/), you need to have a CoreMedia account. See Section 1.3.1, "Registration" [5] for details. If in doubt, contact CoreMedia support to validate your permissions (see Section 1.3.5, "CoreMedia Support" [10]).

NOTE

CoreMedia User Orientation for CoreMedia Developers and Partners

Find the latest overview of all CoreMedia services and further references at:

http://documentation.coremedia.com/new-user-orientation

Find the current online documentation at:

- https://documentation.coremedia.com/cmcc-13

Find the download links at the CoreMedia release page at:

- https://releases.coremedia.com/cmcc-13

Internet access

CoreMedia provides the *CoreMedia Content Cloud* components as Maven artifacts. These components in turn depend on many third-party components. If

your operator has not yet set up and populated a local repository manager, you need Internet access so that Maven can download the artifacts.

NOTE

Maven and npm Repositories and Internet Access

The *CoreMedia Blueprint* workspace relies heavily on Maven and pnpm to build the workspace. That is, Maven and pnpm will download CoreMedia artifacts, third-party components, npm packages and Maven plugins from the private CoreMedia repository and other, public repositories (Maven Central Repository, for example). This might interfere with your company's internet policy. Moreover, if a big project accesses public repositories too frequently, the repository operator might block your domain in order to prevent overload. The best way to circumvent both problems is to use a repository manager like Sonatype Nexus for Maven and npm (since Nexus 3), or Verdaccio (https://verdaccio.org/ for npm. Both decouple the development computers from direct Internet access.

Maven Repository Manager

CoreMedia strongly recommends to use a repository manager to mirror CoreMedia's Maven repository, for example Sonatype Nexus. Alternatively, if a repository manager is not available, configure your credentials for the CoreMedia Maven repositories in your `~/.m2/settings.xml` file as shown below. Simply replace `USERNAME` and `PASSWORD` with your CoreMedia user name and password. It is strongly recommended, that you do not enter the password in plaintext in the `settings.xml` file but encrypt the password. To do so, follow the instructions at https://maven.apache.org/guides/mini/guide-encryption.html or any other available Maven documentation.

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
          http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <interactiveMode>false</interactiveMode>
  <servers>
    <server>
      <id>coremedia.external.releases</id>
      <username>USERNAME</username>
      <password>PASSWORD</password>
    </server>
  </servers>
</settings>
```

MAVEN_OPTS

Maven requires the following minimal memory settings:

```
MAVEN_OPTS=-Xmx2048m
```

NPM registry

To be able to download the packages from https://repository.coremedia.com/nexus/repository/coremedia-npm/, you need to configure your .npmrc.

To tell pnpm to actually download CoreMedia and Jangaroo packages from the CoreMedia npm registry use the following commands:

First, create a base64 encoded string of your credentials in the format `username:password`. You can use the following command on Linux or macOS (the result is bmFtZUBjb3JlbWVkaWEuY29tOmFkbWluMTIz):

```
echo -n 'name@coremedia.com:admin123' | openssl base64
```

```
pnpm config set "@coremedia:registry"
"https://repository.coremedia.com/nexus/repository/coremedia-npm/"
pnpm config set "@jangaroo:registry"
"https://repository.coremedia.com/nexus/repository/coremedia-npm/"
pnpm config set
"//repository.coremedia.com/nexus/repository/coremedia-npm/:_auth"
bmFtZUBjb3JlbWVkaWEuY29tOmFkbWluMTIz
```

> **NOTE**
>
> Please note that https://repository.coremedia.com/nexus/repository/coremedia-npm/ does not mirror packages from https://www.npmjs.com and therefore cannot be used as the default registry for pnpm.

Configuring proxy for pnpm

In order to operate *pnpm* behind a proxy server, you need to configure it accordingly. See https://pnpm.io/npmrc#https-proxy

If your credentials include an `@` symbol, just put your username and password inside quotes. If you use any other special characters in your credentials, you have to convert them into equivalent hexadecimal unicode.

Active Directory users have to pass their credentials in the URL as follows:

```
pnpm config set proxy "http://domain\\username:password@proxy.domain.tld:port"
```

Configuring proxy for Git

Some setups require access to GitHub (e.g. when using Git submodules). You then need to configure `git` to use a proxy in a similar way:

```
git config --global http.proxy http://username:password@proxy.domain.tld:port
git config --global https.proxy http://username:password@proxy.domain.tld:port
```

NOTE

Many companies use a proxy auto-config (PAC) file which defines how browsers and other user agents choose the appropriate proxy server for fetching a given URL. Unfortunately neither `pnpm` nor `git` support these files. As a workaround, you can install a local proxy server which uses a PAC file to decide how to forward a request.

3.1.1 Developer Setup

These are the prerequisites for your local machine where you develop CAE templates or *CoreMedia Studio* extensions, for example.

Hardware

- At least a dual-core CPU with 2GHz, a quad-core CPU is recommended, because *CoreMedia CMS* code makes heavy use of multithreading.
- The minimum RAM you need is 8 GB which is enough if your locally tested components are connected to remote Test System Setup.

Required Software

- A supported Java SDK (see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf). The variable `JAVA_HOME` must be set.
- A supported browser (see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf)
- A supported Maven installation, (see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf).

- A supported Node installation, (see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf)
- A supported pnpm installation, (see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf)
- An IDE. CoreMedia suggests IntelliJ Idea because it has the best support for *CoreMedia Studio* development.
- A supported container environment, see Section 3.2.2, "Docker Compose Setup" [44] and https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf for details.
- If you want to build the workspace with tests, you need an up-to-date version of Google Chrome installed on your computer. It must be contained in your path.
- CoreMedia license files for starting the various *Content Servers*. If you do not already have the files, request your licenses from the CoreMedia support.

OEM Licenses

CoreMedia has an OEM license agreement in place for the following software components:

- CKEditor
- ExtJS by Sencha

These licenses are part of any CoreMedia license agreement, and allow use, and extension of, these components in the context of CoreMedia products. Therefore, you are free to extend CoreMedia Studio using ExtJS and the required tools by Sencha and to use and extend CKEditor in your CoreMedia projects.

3.1.2 Test System Setup

These are the prerequisites for the machine on which you want to install the Test System Setup

Hardware

CPU	Mem (GiB)	Storage (GiB)
4	16	32

Table 3.1. Overview of minimum / recommended Hardware requirements

Required Software

- A supported container environment (see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf).
- A supported Docker Compose release (see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf).

3.1.3 Additional Software for eCommerce Blueprint only

Depending on the eCommerce Connector you use, you need one of the following eCommerce systems:

- *HCL Commerce*
- *SAP Hybris Commerce*
- *Salesforce Commerce Cloud*
- *commercetools*

In Connector for HCL Commerce Manual you will learn how to install and configure the CoreMedia software in the *HCL Commerce* system.

In Connector for SAP Commerce Cloud Manual you will learn how to install and configure the CoreMedia software in the *SAP Hybris Commerce* system.

In Connector for Salesforce Commerce Cloud Manual you will learn how to install and configure the CoreMedia software in the *Salesforce Commerce Cloud* system.

In Commercetools Connector Manual you will learn how to install and configure the CoreMedia software in *commercetools*.

NOTE

For an overview of exact versions of the supported software environments (especially the eCommerce systems) please refer to the Supported Environments document at https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf.

3.2 Quick Start

With *CoreMedia Content Cloud* you do not get a program to install and run, but a workspace to develop within, to build with Maven and to deploy artifacts from. See Chapter 2, *Overview of CoreMedia Content Cloud* [18] for an overview.

Different deployment scenarios

By default, you have two ways to build and deploy the workspace. Both approaches base on the built of the Blueprint Workspace described in Section 3.2.1, "Building the Workspace" [38].

- The Docker Test System Setup is the recommended way. It uses the Docker images to start the systems components. See Section 3.2.2, "Docker Compose Setup" [44] for details.
- Starting the services application jars using SystemD or a different service initialization system.

The subsections guide you through all steps you have to perform in order to get the CoreMedia system running on a machine using the Docker Test System Setup approach. The quick start describes only one path, no options or advanced configurations are described. The "Further Reading" section of each step contains links to additional content, but you do not need to read these chapters for the purpose of the quick start.

NOTE

You need Internet access and a resolvable host name to get everything up and running.

3.2.1 Building the Workspace

What do you get?

When you are finished with all steps, you will have built the CoreMedia Blueprint Workspace and the required Docker images for all CoreMedia applications.

Step 1: Getting a Login for CoreMedia

Goal

You have a login to the CoreMedia software download page, the contributions GitHub repository, the documentation and the CoreMedia artifact repository.

Steps

1. Ask your project manager for your company's account details or contact the CoreMedia support. Keep in mind, that you have to ask explicitly for the access rights to the CoreMedia GitHub contributions repository. See CoreMedia's website for the contact information of the support at https://www.coremedia.com/support.

Check

Go to https://documentation.coremedia.com/cmcc-13 and https://github.com/coremedia-contributions/coremedia-blueprints-workspace and enter your credentials. You should be able to use the online documentation and see the contributions repository.

Step 2: Getting License Files for the CoreMedia System

Goal

You have licenses for the CoreMedia system.

Steps

Ask your project manager, your key account manager or your partner manager for the CoreMedia licenses.

Check

You have a Zip file that contains three zipped license files. In Section 3.2.2, “Docker Compose Setup” [44] you will learn where to put the license files.

Further Reading

- See Section 4.6, “CoreMedia Licenses” in *Operations Basics* for details about the license file format.

Step 3: Checking the Hardware Requirements

Goal

You are sure, that your computer meets the hardware requirements as described in Section 3.1, “Prerequisites” [32].

Step 4: Checking and Installing all Required Third-Party Software

Goal

All required third-party software (such as Java, Git, Maven, ...) is installed on your computer and has the right version.

Steps

1. Open the supported environments document at https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf and check that you have installed the right version of Java and that you have the right OS. The `JAVA_HOME` variable must be set.
2. Check that a supported Maven version is installed (see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf).
3. Check that a supported container environment is installed on your computer (see https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 - Supported Environments.pdf). See Section "Docker Installation" [44] for installation instructions.

Further reading

- Section 3.1, "Prerequisites" [32] describes the required software in more detail.

Step 5: Cloning the Workspace

Goal

You have the *CoreMedia Blueprint* workspace on your hard disk.

Steps

1. Make sure that you have access to https://github.com/coremedia-contributions/coremedia-blueprints-workspace. If you encounter a 404 error, then you are probably not logged in at GitHub or you do not have sufficient permissions yet.
2. When you use a Windows system, make sure that the Git configuration parameter *`core.autocrlf`* is set to "input". Otherwise, some init files will not run properly in your test machine. Because on checkout, Git would change the line endings to Windows style.
3. On your local machine, clone the repository into a directory `blueprint` using Git:

WARNING

Path length limitation in Windows

The *CoreMedia Blueprint* workspace contains long paths and deeply nested folders. If you install the *CoreMedia Blueprint* workspace in a Windows environment, keep the installation path shorter than 25 characters. Otherwise, unzipping the workspace might fail or might lead to missing files due to the 260 bytes path limit of Windows.

```
git clone
https://github.com/coremedia-contributions/coremedia-blueprints-workspace
 blueprint
```

4. In the cloned repository, get a list of all tags:

```
git tag
```

5. Create your working branch from the tag you want to use as your starting point:

```
git checkout -b <yourBranchName> <tagName>
```

Check

The Git clone command has succeeded.

Further reading

- Chapter 4, *Blueprint Workspace for Developers* [57] describes the structure of the workspace, the concepts behind the workspace and how you can work with the workspace.
- Section 4.2.2, "Configuring the Workspace" [116] describes further configuration of the workspace which is required for development and deployment.
- On https://releases.coremedia.com/cmcc-13 click the link to the latest download to find a description on how to download a specific release.

Step 6: Getting the blob Demo Content

The textual content and the themes are already part of the workspace you have cloned before. However, to keep the workspace small, the blob content is supplied in a separate file.

Goal

The workspace contains the blob files of the CoreMedia demo content (videos, images, ...).

Steps

1. Open the releases site https://releases.coremedia.com/cmcc-13 and click the link to the current release.
2. Click the "content-blobs archive" link on the site and download the file.
3. Extract the archive into the workspace you have cloned in step 5.

Step 7: Configuring the Repository Settings and Check Maven/NPM Configuration

Goal

Your Maven `settings.xml` file contains the settings required to connect with the CoreMedia Nexus repository.

The .npmrc is configured to access https://repository.coremedia.com/nexus/repository/coremedia-npm/.

Steps

1. Follow the steps described in Section 3.1, “Prerequisites” [32].

Check

When you build the workspace, all artifacts and packages are found.

Step 8: Building the Workspace with Maven

Goal

The workspace has been build, so that most of the artifacts and Docker images are built. The build takes some time. On an Intel i7 processor with 16GB RAM around 20 minutes.

If you want to build images for ARM processors, you need to pass the property `-Dapplication.image-arch=arm64` to the Maven build.

Steps

In the main directory of the workspace call:

```
mvn clean install -DskipTests -Pdefault-image
```

Check

The Maven build ends with message "Build successful".

Further reading

- Section 4.2.1, "Enabling Or Removing Optional Components" [105] describes how you can remove parts of the workspace that you do not need.

Step 9: Building the Studio Client with pnpm

Goal

The Studio Client has been build, so that you can start the Docker container.

Steps

1. Switch into the Studio Client directory:

```
cd workspace/apps/studio-client
```

2. Build the Studio Client:

```
pnpm install
pnpm -r run build
pnpm -r run package
```

3. Build the Docker image:

```
docker buildx build --secret id=npmrc,src=.npmrc --tag
coremedia/studio-client:latest --load .
```

For more detailed instructions and possible build options consult `apps/studio-client/README.adoc`.

Step 10: Building the Frontend

Goal

The frontend has been build, so that you can use the themes and bricks.

Steps

1. Switch in the frontend directory with:

```
cd workspace/frontend
```

2. Build the frontend parts with:

```
pnpm install
pnpm run build
pnpm run build-frontend-zip
```

For more detailed instructions and possible build options consult `frontend/README.adoc`.

Now, you have build the Blueprint workspace and the Docker images. Continue with Section 3.2.2, "Docker Compose Setup" [44] in order to configure and start the Docker deployment.

3.2.2 Docker Compose Setup

This tutorial will guide you through the first steps to start the CoreMedia Content Cloud Services using `docker compose`, which is a tool to simplify the deployment of development environments using docker.

3.2.2.1 Prerequisites

Docker knowledge is not required, but for first starters with this technology, it is highly recommended to start with simpler projects until the infrastructure is running and basic knowledge about the tooling has been acquired. As a good start, you can play around online in one of the free tutorials on learndocker or katacoda.

Docker Installation

For the Docker Compose setup to work, you need a running container runtime and Docker client and Docker Compose to be installed. The default is to use Docker Desktop, a commercial development tooling suite. Please check their pricing options first. There are free alternatives available for all major operating systems.

Rancher Desktop

Rancher Desktop is a full open source replacement of Docker Desktop supporting all platforms and chipsets. Rancher Desktop is our preferred choice for running containers on developer workstations. At the start of the installation, you can disable the inbuilt Kubernetes in favor of our Kubernetes in Docker setup.

- Installer https://rancherdesktop.io/

After the installation, you might want to adapt the provisioning configuration of the VM to:

- increase file limits doc
- add registry mirrors doc
- increase inotify limits see snippet below

override.yaml:

```
provision:
- mode: system
  script: |
    #!/bin/sh
    sysctl -w fs.file-max=200000
    sysctl -w fs.inotify.max_user_watches=524288
    sysctl -w fs.inotify.max_user_instances=512
```

Don't forget to restart Rancher Desktop to apply the changes.

Docker Desktop

Docker Desktop is a commercial development tooling suite

- Installer Mac | Windows
- Getting Started Mac | Windows

Colima - Containers in Linux Machines

Colima is a free virtualization tooling for macOS to provide the same seamless developer experience as Docker Desktop. It is based on Lima (Linux Machines), which is using the same QEMU stack as Docker Desktop. Lima ist also the foundation of Rancher Desktop, the Kubernetes developer tooling setup by Rancher.

To install Colima, Docker and Docker Compose run the brew installation formulae for each app:

- colima
- docker
- docker compose

```
brew install colima docker docker-compose
```

To start Colima run:

```
colima start --cpu 4 --memory 14
```

After the VM has started, you should be able to use the Docker client. Be aware, that instead of `~/.docker/daemon.json`, Colima uses `~/.colima/docker/daemon.json` to configure the runtime.

If you are using the Spotify `dockerfile-maven-plugin`, you also need to set the `DOCKER_HOST` environment variable. Colima exposes the Docker socket at `~/.colima/docker.sock` and Spotifys Docker client only works, when this is set.

```
DOCKER_HOST=unix:///Users/<YOUR USER NAME>/.colima/docker.sock
```

Windows Subsystem

Instead of using Docker Desktop, it is also possible to install Docker directly within the Windows Subsystem (WSL2) Linux.

1. Install WSL2 with an Ubuntu system, by following the instructions here.
2. Install a Linux subsystem, by running `wsl --install -d Ubuntu`
3. Install Docker Engine on Ubuntu, by following the instructions at https://docs.docker.com/engine/install/ubuntu/.
4. Increase security and user experience by following the post-installation steps, described at https://docs.docker.com/engine/install/linux-postinstall/.

Docker Configuration

After the installation was successful and Docker has been started, proceed with the following configurations. If you read the getting started documentation of Docker, you should easily find the corresponding settings.

- Increase RAM to at least 16 GB
- Increase CPUs to at least 4
- On Windows, you also need to:
  - share the Drive C in the "Shared Drives" settings page if you use Docker desktop.
  - enable the "Expose the daemon without TLS" toggle in the general settings page.

Docker Compose Configuration

Configure your docker compose environment by creating or editing your `.env` file. All environment variable references in the Docker Compose files, can be configured using this file. Be aware, that environment variables in the current process environment have precedence over variables defined in the `.env` file. Below, you will find an example `.env` file.

In the `.env` file you can configure the following properties. All relative paths shown here are relative to the `global/deployment/docker` directory.

- Make sure `compose/development.yml` is included in the `COMPOSE_FILE` variable, it is required to expose the container internal ports to the docker host.
- Make sure `compose/development-local.yml` is included in the `COMPOSE_FILE` variable, it is required for content import from Blueprint and optionally for loading licenses from local `coremedia-licenses` directory.
- If you want to configure gRPC as the internal communication protocol, include `compose/grpc.yml` in the `COMPOSE_FILE` variable. Otherwise, CORBA will be used.
- For the `docker compose` development setup, make sure that you have the licenses placed at the following locations:

  ```
  coremedia-licenses/cms-license.zip
  coremedia-licenses/mls-license.zip
  coremedia-licenses/rls-license.zip
  ```

  Zip files added below this directory are by default excluded from Git version control. If you place the license files in this directory, you must not set an environment variable for the license location!

  Alternatively, you may define environment variables with license URLs, and the server containers will download them at runtime. You will find the corresponding environment variables in the `.env` example below.
- For the development setup, make sure that you have created or provide themes. Set either `THEMES_ARCHIVE_URL` or `THEMES_ARCHIVE_FILE` in the `.env` file. To re-import the themes set `FORCE_REIMPORT_THEMES` to `true`.

  If you use the provided CoreMedia themes, you must not set these environment variables because the default setting is sufficient.
- For the development setup, make sure that you have created or provide content. Set either `CONTENT_ARCHIVE_URL` or `CONTENT_IMPORT_DIR`

in the `.env` file. To re-import the content set `FORCE_REIMPORT_CONTENT` to `true`.

If you use the provided CoreMedia test data, you must not set these environment variables because the default setting is sufficient.

- To start MongoDB and Elastic Worker (for example if you want to use Elastic Social), include the following compose files:

```
COMPOSE_FILE=...:compose/elastic.yml:compose/development-elastic.yml
```

If gRPC shall be used, add this:

```
COMPOSE_FILE=...:compose/elastic.yml:compose/development-elastic.yml:grpc-elastic.yml
```

- Depending on the eCommerce system(s) you want to connect to, you will need to set these additional variables:

HCL WebSphere Commerce

```
SPRING_PROFILE=dev-wcs
COMPOSE_FILE=compose/default.yml:compose/development-wcs.yml
WCS_HOST=your.wcs.host
```

SAP Hybris

```
COMPOSE_FILE=compose/default.yml:compose/development-hybris.yml
```

Salesforce Commerce Cloud

```
COMPOSE_FILE=compose/default.yml:compose/development-sfcc.yml
```

- Depending on if you want to use the optional KIO Copilot, you will need to install the KIO Studio Plugin and set these additional variables:

```
COMPOSE_FILE=compose/default.yml:compose/development-kio.yml

// mandatory key to access the OpenAI API
KIO_OPENAI_API_KEY=CONFIGURE_ME

// version matching to your KIO Studio Plugin Version
KIO_VERSION=latest

// Optional access to CM default playbooks
KIO_GITHUB_TOKEN=CONFIGURE_ME
```

Consult the documentation for further details on how to configure the KIO Copilot.

By default, you can start with this file:

```
# This sets the compose path separator to ":" for all OS.
COMPOSE_PATH_SEPARATOR=:

# Configure a list of Docker Compose files you want to apply and
```

```
# separate them using the value of the COMPOSE_PATH_SEPARATOR.
# Be advised that ordering is crucial and last definitions
# override preceeding ones.
#  compose/default.yml - unconfigured services
#  compose/development.yml - development configuration
#  compose/development-local.yml - local licenses / content / studio-client
 build
#  compose/grpc.yml - enable gRPC for internal communication
#
# for most cases this should be your default.
COMPOSE_FILE=compose/default.yml:compose/development.yml:compose/development-local.yml

# Optional properties

# With this variable, you can set the prefix of the image repository.
# Set this to use images from a remote registry, that is,
# REPOSITORY_PREFIX=my.registry/cmcc would result in a studio-server
# image my.registry/cmcc/studio-server
# REPOSITORY_PREFIX=my.registry/cmcc

# With this variable, you can set the prefix of the image repository for
# the Commerce Adapter Docker images. Set this to use images from a remote
# registry.
# COMMERCE_REPOSITORY_PREFIX= my.registry/cmcc

# The version tags of the commerce adapter service images to be used.
# COMMERCE_ADAPTER_MOCK_VERSION=1.2.3
# COMMERCE_ADAPTER_SFCC_VERSION=1.2.3
# COMMERCE_ADAPTER_HYBRIS_VERSION=1.2.3
# COMMERCE_ADAPTER_WCS_VERSION=1.2.3

# The environment fully qualified domain name to use for the system.
# If not set, docker.localhost will be used.
# ENVIRONMENT_FQDN=docker.localhost

# enable debug agent for all spring boot apps. If you want to enable
# this only for a single service, you need to set the environment
# variable explicitly at that service.
# JAVA_DEBUG=true

# Service Specific variables

# The license url/path for the content-management-server
# CMS_LICENSE_URL=/coremedia/licenses/cms-license.zip

# The license url/path for the master-live-server
# MLS_LICENSE_URL=/coremedia/licenses/mls-license.zip

# The license url/path for the replication-live-server
# RLS_LICENSE_URL=/coremedia/licenses/rls-license.zip

# The mail server for elastic social registration mails
# ELASTIC_SOCIAL_MAIL_SMTP_SERVER=localhost

# Theme Import

# Themes can be imported from a file location or from an URL
# pointing to an zip archive containing the themes.
# By default, the variable points to the path
# /coremedia/import/frontend.zip within the management-tools
# container. To pass in an archive from your hosts file system
# include the developmemt-local.yaml file in your
# COMPOSE_FILE environment variable and configure only the path
# on your host system using the THEMES_ARCHIVE_FILE env var.
# If you don't configure that variable, the default will point
# to the frontend.zip in your workspace.
# THEMES_ARCHIVE_URL=
# THEMES_ARCHIVE_FILE=

# Force reimport of themes when set to true
# FORCE_REIMPORT_THEMES=false
```

```
# Content Import

# The directory from which the content should be imported. By default,
# this points to the content/test-data/target/ directory of the test-data
module
# in the CoreMedia Bluprints workspace.
# CONTENT_IMPORT_DIR=

# Forces the reimport of the content when set to true
# FORCE_REIMPORT_CONTENT=false

# The url of a webserver, serving all content blobs during the
# server-import. If you added the content blobs to the workspace,
# you can leave this field empty. This is a CI development
# optimization to keep content image blobs out of the VCS history.
# BLOB_STORAGE_URL=

# The url to a zip archive containing content, users and optionally
# themes for import. The layout in the archive should be the same
# as the test-data module creates. This is a CI development feature
# to import content from a separated build process.
# CONTENT_ARCHIVE_URL=

# Skips the whole content and theme import when set to true
# SKIP_CONTENT=false
```

Note that you cannot set arbitrary environment variables in the `.env` file and expect, that they will be picked up by the CoreMedia Spring Boot applications. Only the variables, being referenced in Docker Compose files, can be used here.

For more information about this tooling option, visit the official Docker Compose documentation.

DNS Configuration

To access the applications, you need to configure your hosts DNS resolution. Changing this requires admin rights.

Without Administrator rights: Without administrator rights, you need to set the following environment variables in the `.env` file to the DNS resolvable host name of your computer:

- `ENVIRONMENT_FQDN`
- `CMS_ORB_HOST`
- `MLS_ORB_HOST`
- `WFS_ORB_HOST`
- `WFS_GRPC_HOST`

With Administrator rights: With administrator rights edit the configuration file for the host mappings at the following locations:

- On Linux / Mac OS `/etc/hosts`

- On Windows `%SystemRoot%\System32\drivers\etc\hosts`

Make sure that it contains the following mappings:

```
# Development names to connect from Maven / IDEA
127.0.0.1 workflow-server
127.0.0.1 content-management-server

# Administrative Hosts
127.0.0.1 docker.localhost
127.0.0.1 overview.docker.localhost
127.0.0.1 monitor.docker.localhost

# Corporate Hosts
127.0.0.1 corporate-de.docker.localhost
127.0.0.1 corporate.docker.localhost

# Management Hosts
127.0.0.1 studio.docker.localhost
127.0.0.1 preview.docker.localhost

# Commerce Hosts
127.0.0.1 helios.docker.localhost
127.0.0.1 calista.docker.localhost
127.0.0.1 apparel.docker.localhost
127.0.0.1 sitegenesis.docker.localhost
127.0.0.1 shop-preview-ibm.docker.localhost
127.0.0.1 shop-ibm.docker.localhost
127.0.0.1 shop-preview-production-ibm.docker.localhost
127.0.0.1 shop-preview-hybris.docker.localhost
127.0.0.1 shop-hybris.docker.localhost
127.0.0.1 shop-preview-sfcc.docker.localhost
127.0.0.1 shop-sfcc.docker.localhost
127.0.0.1 shop-tools-sfcc.docker.localhost
```

Reducing the Setup

If you do not want to start the whole stack, you can start only the required components. All services define their dependencies using the `depends_on` directive. Running a simple `docker compose up -d content-management-server workflow-server` therefore will also start `mysql`, `mongodb` and `solr`.

You can get all available services by running `docker compose config --services`

Alternatively, you can render the current setup to a config file and opt-out the services by deleting them from the rendered file.

```
docker compose config > docker-compose.yml
```

You can then remove everything you don't want. `docker-compose.yml` is ignored by Git with the default `.gitignore` file. You only have to make sure, that in your `.env` file

```
COMPOSE_FILE=docker-compose.yml
```

is set, otherwise the file won't be loaded.

Of course, there are a lot of toggles for your convenience:

- `JAVA_DEBUG` - default ports `XXX06` for JDWP
- `FORCE_REIMPORT_CONTENT` - once imported, the content won't reimport unless forced
- `SKIP_CONTENT` - same as not running the `management-tools` container.

There is also an option to define profiles to match a set of services. Visit the Docker documentation if you are interested in this feature.

Having multiple backends in parallel or keep multiple backend data volumes

In order to work on multiple tasks in an interleaved mode, you may want to keep the example content of each setup and switch back and forth. In order to do so, you can use the `COMPOSE_PROJECT_NAME`. If set docker compose will prefix all resources with the set value, that is, a volume will be named `JIRA-55_db-data` if `COMPOSE_PROJECT_NAME=JIRA-55`. The only thing to keep in mind with this approach is to never use the `-v` flag when running `docker compose down`.

3.2.2.2 Starting the Docker Setup

Make sure that you have build the workspace, and the Docker images. To build the Docker images the Maven profile `default-image` must be activated. To check whether you have built the images you can list the available images using the following command:

```
docker images
```

The result should look like this but should contain image names like `cae-live` or `content-server`:

```
REPOSITORY                  TAG     IMAGE ID      CREATED        SIZE
coremedia/cae-preview       latest  a8f9d245fbbe  10 hours ago  296MB
coremedia/content-server    latest  8f6045472222  10 hours ago  272MB
```

If there are no images listed, you probably did not activate the Maven profile. To build with the active Maven profile, run the following command:

```
mvn clean install -Pdefault-image
```

After the maven build images have been built, you can change the directory to the docker setup:

```
cd global/deployment/docker
```

To build the studio-client image, you can either follow the steps in the `apps/studio-client/README.adoc` or add a working `.npmrc` to the `apps/studio-client` folder as described in that README.adoc and include the `development-local-build.yml` to the `COMPOSE_FILE` environment variable to build the image directly from docker compose.

Prestart Check

In order to make sure that there are no conflicts with preexisting setups, you can run the following steps to delete all preexisting setups:

- `docker compose down -v` this command should stop and remove all running containers and volumes that are associated with the project defined by the Docker Compose files. The execution of this command needs to be successful to start up a new clean system. The warnings can be ignored, because if the volume does not exist, it cannot be removed.
- `docker ps --format "table {{.Names}}\t{{.Ports}}" -a` this command lists all running containers and their mapped ports. Make sure that these ports do not conflict with the standard port mappings of the CoreMedia applications, when run on a single node.

Start the services

To list the services that will be started execute the following command:

```
docker compose config --services
```

The result should look similar to this, depending on the value of `COMPOSE_FILE`:

```
mysql
mongodb
solr
content-management-server
master-live-server
replication-live-server
workflow-server
content-feeder
cae-feeder-preview
cae-feeder-live
user-changes
```

```
elastic-worker
studio-server
studio-client
cae-preview
cae-live
headless-server-preview
headless-server-live
traefik
```

Now you can start the services by running the following command:

```
docker compose up -d --build
```

The flag `--build` forces `docker compose` to build the images, which are not included into the Maven build and which are only required in the development setup, such as `overview`, `traefik` and the commerce proxies. You can omit this flag on subsequent builds, if there aren't any changes to these images.

The result should look like the following output.

```
Creating elastic-worker            ... done
Creating traefik                   ... done
Creating mongodb                   ... done
Creating management-tools          ... done
Creating master-live-server        ... done
Creating headless-server-preview   ... done
Creating studio-client             ... done
Creating cae-live                  ... done
Creating cae-preview               ... done
Creating replication-live-server   ... done
Creating cae-feeder-live           ... done
Creating solr                      ... done
Creating overview                  ... done
Creating content-management-server ... done
Creating mysql                     ... done
Creating studio-server             ... done
Creating workflow-server           ... done
Creating headless-server-live      ... done
Creating content-feeder            ... done
Creating user-changes              ... done
Creating cae-feeder-preview        ... done
```

Wait until the services are healthy

To make sure that the system is up and running and all services are healthy, you can use the `docker ps` command. By default, it will print out the healthy state for each service.

```
docker ps
```

The result should look like this excerpt:

```
cae-preview    coremedia/cae-preview    Up About a minute (healthy)
cae-live       coremedia/cae-preview    Up About a minute (unhealthy)
studio-server  coremedia/studio-server  Up About a minute (health:starting)
```

Now you have to wait until all services are healthy. You can track this by either running a command loop in your shell, or by visiting the overview page in your browser.

Watch health state using the overview page

Open the https://overview.docker.localhost and scroll down to the table where the status is shown on the right side. In case the service is healthy or unhealthy, a green or red icon is shown.

Watch health state using the shell

- On Linux / Mac OS

```
watch -n 1 "docker ps"
```

- On Windows (PowerShell)

```
while($true) { Clear-Host; docker ps; sleep 5 }
```

For better visibility of this command, you can configure the formatting by editing/creating the `~/.docker/config.json` with the following content:

```
{
  "psFormat" : "table {{.Names}}\\t{{.Image}}\\t{{.Status}}"
}
```

Login to CoreMedia Studio

Click on the **Open Studio** link at the top of the overview page or simply open the link to Studio directly by using the previously configured domain name:

- https://overview.docker.localhost
- https://studio.docker.localhost

NOTE

The import of the test data may take some time. So, you may not be able to log in as one of the predefined users like Rick C, or some content is missing.

Cleanup Services

To shut down the services, simply run the following command:

```
docker compose down
```

Cleanup Services and Content

To shut down all services and clear all created volumes including the databases, simply run the following command:

```
docker compose down -v
```

4. Blueprint Workspace for Developers

CoreMedia Blueprint workspace is the result of CoreMedia's long year experience in customer projects. As *CoreMedia CMS* is a highly customizable product that you can adapt to your specific needs, the first thing you used to do when you started to work with *CoreMedia CMS* was to create a proper development environment on your own. *CoreMedia Blueprint* workspace addresses this challenge with a reference project in a predefined working environment that integrates all CoreMedia components and is ready for start.

The CoreMedia workspace contains two blueprints, the eCommerce Blueprint and the Brand Blueprint. Both blueprints can be used together as demonstrated in the example sites (see Section 2.2, "CoreMedia Blueprint Sites" [29]). However, you can also use both blueprints separately. Deactivate the not used blueprint as described in Section 4.2.1, "Enabling Or Removing Optional Components" [105].

Maven based environment

CoreMedia Blueprint workspace provides you with an environment which is strictly based on today's de facto standard for managing and building Java projects by using Maven. You do not get a program to install and run, but a workspace to develop within, to build and to deploy artifacts from.

NOTE

Unless specified otherwise command line examples are given in Unix style. The path to the root of the *Blueprint* workspace directory will be referenced by the variable `$CM_BLUEPRINT_HOME`.

4.1 Concepts and Architecture

This chapter describes concepts and architecture of *CoreMedia Content Cloud*.

- Section 4.1.1, "Maven Concepts" [58] describes how the Maven concepts are implemented within the *CoreMedia Blueprint* workspace.
- Section 4.1.3, "Application Architecture" [62] describes how CoreMedia applications are build from library and component artifacts and how deployable artifacts are build with package artifacts.
- Section 4.1.4, "Structure of the Workspace" [67] describes the folder structure of the *CoreMedia Blueprint* workspace.
- Section 4.1.5, "Project Extensions" [77] describes the extensions mechanism which lets you enable and disable extensions in one single location.
- Section 4.1.6, "Application Plugins" [82] describes how you can create plugins for CoreMedia applications.

4.1.1 Maven Concepts

The *Maven* build and dependency system is the foundation of the *CoreMedia Blueprint* workspace. This section will introduce you into the concepts CoreMedia used with *Maven* to provide you with the best development experience as possible.

4.1.1.1 Packaging Types

By default, *Maven* provides you with several packaging types. The most important ones are the `pom`, `jar` and the `war` type. They should be sufficient for the most common kinds of development modules but whenever you try to either support proprietary formats or try to break whole new ground, those three packaging types aren't sufficient. Using only the `pom` packaging type together with custom executions of arbitrary plugins, gives you flexibility but adding and maintaining your `pom.xml` files is going to be a complex and costly process.

To reduce complexity, but even more important to enforce standards, CoreMedia came up with a custom tailored packaging type for the *CoreMedia Blueprint* workspace. The `coremedia-application` packaging type provides a build lifecycle and dependency profile for a proprietary application format.

coremedia-application

The `coremedia-application` packaging type is provided by the `coremedia-application-maven-plugin`. When you take a look at the root `pom.xml` file and search for this plugin, you will find two occurrences, one in the `pluginManagement` section and one in the `build` section. The latter definition contains the line `<extensions>true</extensions>` within its plugin body, telling Maven that it extends Maven functionality. In this case, Maven will register the custom lifecycle bound to the custom packaging type.

```
<plugin>
  <groupId>com.coremedia.maven</groupId>
  <artifactId>coremedia-application-maven-plugin</artifactId>
  <extensions>true</extensions>
</plugin>
```

Besides lifecycle, a custom packaging type can also influence if Maven dependencies of this type have transitive dependencies or not. Because CoreMedia wanted to keep the `coremedia-application` packaging type to be the pendant of the `war` packaging type, it does not have transitive dependencies either. For your modules to depend on other `coremedia-application` modules and their dependencies as well, this means, that you need to define an additional dependency to the same GAV (groupId, artifactId, version) coordinates but with packaging type `pom`.

```
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>application</artifactId>
  <type>coremedia-application</type>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>application</artifactId>
  <type>pom</type>
  <scope>runtime</scope>
</dependency>
```

Example 4.1. Dependencies for a CoreMedia application

You may know this pattern from working with `war` overlays if they are skinny too, which means that they contain no further versioned artifacts.

For further information about the `coremedia-application-maven-plugin`, you should visit the plugins documentation site at CoreMedia Application Plugin.

4.1.1.2 BOM files

BOM stands for "bill of material" and defines an easy way to manage your dependency versions. The BOM concept depends on the `import` scope introduced with *Maven 2.0.9*, that allows you to merge or include the `dependencyManagement` of a foreign POM artifact in your POMs `dependencyManagement` section without inheriting from it.

```
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>core-bom</artifactId>
  <version>CURRENT_RELEASE_VERSION</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
```

The inclusion or merge is done before the actual dependency resolution of your project is done. By the time the actual resolution starts *Maven* does not see any BOM imports but only the merged or included dependencies.

For projects using a framework that provides many artifacts like *CoreMedia* does, this means, that you can fix the versions for all dependencies that are part of that BOM, by simply declaring one dependency.

Of course there are pitfalls when using BOMs and the `import` scope, but the benefits of using BOMs overcome any disadvantages. To prevent you from falling into one of the pitfalls, the following paragraphs will show you how to use the BOM approach correctly.

Chaining BOMs and artifact procurement

Artifact procurement is a feature that some repository management tools like *Nexus* or *Artifactory* offer you to allow your project to use only explicitly configured versions of their dependencies. In addition to the local dependency management in your POM files, artifact procurement is done remotely in your artifact repository. Because of this fact, artifact procurement is much stricter and most commonly only applied in organizations, where securing build infrastructure has the highest priority.

When you chain BOM files, which means that the BOM you import, imports another BOM and so forth, you cannot achieve complete artifact procurement if any POM enforces a different version of a BOM than the version that is used within that chain of its predecessors. This problem stems from the fact that all import scoped dependencies must be resolved in any case, even if your topmost project enforces a different version. Luckily this only affects POM artifacts, you cannot compile against or which have no effect when deployed to the classpath.

BOM import order

Because the import scope is more likely an xinclude on XML basis, ordering of these imports is crucial if the BOMs content is not disjoint, which is most likely the case in presence of chained BOMs.

As a result, it is important to list the BOM imports in reverse order of the BOM import chain. To make sure your update is correct you should therefore always create the effective POM and check the resulting `dependencyManagement` section. To do so execute:

```
$mvn help:effective-pom -Doutput=effective-pom.xml
```

4.1.2 Blueprint Base Modules

CoreMedia Content Cloud introduces a new way of providing default features for *CoreMedia Blueprint*, *Blueprint Base Modules*. Step by step CoreMedia will move features from the *Blueprint* workspace to the *Blueprint Base Modules*. All features of the *Blueprint Base Modules* will be described by a public API. The reasons why CoreMedia decided to do so are:

- Less source code means faster Maven builds.
- Less source code in *Blueprint* workspace leads to easier migration paths when updating to new versions.

As its name implies, this new module contains Blueprint logic and thus depends on the Blueprint's content model. The content model is still part of the Blueprint workspace, hence you may customize it. Be aware, that some changes will break the Blueprint Base modules. Read Section 4.4.1.1, “Content Type Model Dependencies” [128] for the list of Blueprint Base dependencies to the Blueprint content type model.

NOTE

Read Section 4.4.1, “Using Blueprint Base Modules” [127] for a detailed description of how to develop with the various *Blueprint Base Modules*.

4.1.3 Application Architecture

CoreMedia applications are based on Spring Boot and as such support Spring Java configuration, Spring Boot auto-configuration, and Spring Boot configuration properties.

The recommended way to develop, extend, and configure CoreMedia applications is to follow the Spring Boot conventions. However, for backward-compatibility reasons, the CoreMedia component loader may still be used to activate component XML bean definitions and to load component properties files.

Spring Bean Customizer

A customizer is a mechanism which enables you to change an existing bean definition without touching the actual configuration file of the bean. Technically speaking, a customizer is a `BeanPostProcessor` bean which adjusts the bean during startup of the `ApplicationContext`. The recommended way is to use Spring Java configuration and to annotate beans acting as customization sources using the annotation class `com.coremedia.springframework.customizer.Customize`. However, a customizer can also be declared in a Spring XML file.

Examples

```
@Configuration(proxyBeanMethods = false)
  static class AddEntriesToSomeMap {
    @Bean(autowireCandidate = false)
    @Customize("someMap")
    Map<String, String> append() {
      return Map.of("key1", "value1", "key2", "value2");
    }
  }
```

Here, two more entries `key1` and `key2` are added to a bean of type `Map`.

Although the recommended way to replace a predefined bean is to use Spring `Primary` annotation (or XML attribute), there is also a customizer to replace the target bean:

```
@Configuration(proxyBeanMethods = false)
  static class ReplaceLoginInterceptor {
    @Bean(autowireCandidate = false)
    @Customize(value = "loginInterceptor", mode = Customize.Mode.REPLACE)
    MyLoginInterceptor myLoginInterceptor() {
      return new MyLoginInterceptor();
    }
  }
```

Here, a predefined bean `loginInterceptor` is replaced with the bean `myLoginInterceptor`.

XML Syntax

NOTE

Important: The recommended way to customize beans provided by CoreMedia applications is to use the first mechanism of the following list that can be applied to the target bean.

- Use external configuration (for example, system properties) to configure the bean.
- Override injection of the target bean by using Spring's `Primary` annotation (or XML attribute).
- Define a source bean in Spring Java configuration style (for example, using `Component`/`ComponentScan` or `Configuration`/`Bean`) and add the `Customize` annotation.
- Only use the XML customizers if none of the above can be used in your project.

The syntax to define a customizer is as follows (id attribute omitted):

```
<customize:operation
              bean="beanname" [property="propertyname"]
              custom-value="value"/>
```

or

```
<customize:operation
              bean="beanname" [property="propertyname"]
              custom-ref="custom-beanname"/>
```

or

```
<customize:operation bean="beanname" [property="propertyname"]>
    <bean, map, set, list or properties>
</customize:operation>
```

Basically, an operation (`<customize:operation>`) is performed on a bean (`bean="..."`) or on a property of a bean (`bean="..." property="..."`). As a parameter of an operation, you can use a value (`custom-value="..."`) or a reference to a bean (`custom-ref="..."`). Instead of a bean reference, you can also use an element `<map>`, `<list>`, `<set>`, `<properties>`or `<bean>` as a parameter. The customizer can be disabled by an attribute `enabled="false"`.

The following operations are supported:

- **Replace** - Depending on the context, a bean will be replaced by another bean or a bean property will be set to another value. If a bean is replaced, it is removed from the context and its name will be added as alias to the replacing

bean. This way, wiring logic using the original name will work as expected, except for autowiring Maps where the name of the replacing bean will be used as key.

- **Append/Prepend** - This operation works on beans or properties which are comprised of multiple elements, thus are of type List, Set, Map, String array and the like. The elements you add must be wrapped with the type of the property or bean that you modify (such as list, map or set). As you can see in the listing below you cannot add an element directly but instead, even if it is only one element that you wish to add, you have to wrap it. You can add elements to the start ("prepend") or end ("append").

```
<customize:append id="registerMyService" bean="myServices"
                  property="serviceList">
  <list>
    <ref bean="myServiceBeanId">
  </list>
</customize:append>
```

- **Wrap** - This operation wraps a bean by another bean: it replaces a bean and injects the original bean into the new bean. The following example replaces the bean "service" by an instance of `WrapperService` and injects the original "service" bean as a property "delegate" into `WrapperService`:

```
<customize:wrap id="wrapService" bean="service"
                wrapper-property="delegate">
  <bean class="com.mycompany.WrapperService"/>
</customize:wrap>
```

If different customizers work on the same bean or property, conflicts may arise. Therefore, you can use the attribute `order` to define the order of execution of the customizers.

```
<customize:replace id="registerMyService-1" bean="myService"
                   property="name"
                   custom-value="myService-1"
                   order="10"/>
<customize:replace id="registerMyService-2" bean="myService"
                   property="name"
                   custom-value="myService-2"
                   order="20"/>
```

The example shows two customizers, both working on the property `name` of the bean `myService`. Due to the lower order value (10), the first customizer has a higher priority and is executed first. Afterward, the second customizer overwrites this setting again.

NOTE

The rest of this chapter describes the component loader and related topics. This is mostly relevant for projects migrating from an old CoreMedia version to a recent one.

Component Loader

To activate the component loader, add the following dependency to your component or web application module `pom.xml` file:

```
<dependency>
 <groupId>com.coremedia.cms</groupId>
 <artifactId>base-component</artifactId>
 <scope>runtime</scope>
</dependency>
```

Example 4.2. Adding the Base Component

CoreMedia applications are hierarchically assembled from artifacts:

- Library artifacts are used by
- Component artifacts which are used by
- Application artifacts.

Library Artifacts

Library artifacts contain JAR artifacts with Java classes, resources and Spring bean declarations.

An example is the artifact `cae-base-lib.jar` that contains *CAE* code as well as the XML files which provide Spring beans.

Component Artifacts

Component artifacts provide a piece of business (or other high level) functionality by bundling a set of services that are defined in library artifacts. Components follow the naming scheme `<componentKey>-component.jar`. The component artifact `cae-component.jar` for example, bundles all services that are typically required by a *CAE* web application based project.

Component artifacts are automatically activated on application startup, in contrast to library artifacts. That is, Spring beans and properties are loaded into the application context and servlets and so on will be instantiated. Therefore, you can add a component by simply adding a Maven dependency. No additional steps (such as adding an import to a Spring file) are necessary.

The following files allow you to declare services for a component which are automatically activated:

- `/META-INF/coremedia/component-<componentname>.xml`:

  An entry point for all component Spring beans. Either declared directly or imported from library artifacts.

- `/META-INF/coremedia/component-<componentname>.properties`:

  All configuration options of the component as key/value pairs. These properties might be overridden by the concrete application.

Redundant Spring Imports

> **NOTE**
>
> This section is about Spring XML configuration files. The recommended way to configure Spring bean definitions is to use Spring Java configuration.

Due to the design of the Spring Framework and the CoreMedia component loader, it may often be necessary to declare many `<import/>` elements in Spring XML configuration files, often pointing to the same resource. This slows down the startup of the ApplicationContext.

Unfortunately, `org.springframework.beans.factory.xml.XmlBeanDefinitionReader` does not track imported XML files, so redundant `<import/>` elements will lead to Spring parsing the same XML files over and over again (in most cases, those XML files will contain more `<import/>` elements leading to even more parsing, ...) After moving to Servlet 3.0 resources, for each `<import/>`, the JAR file containing the XML file has to be unpacked. Also, every time that an XML file is completely parsed, Spring reads all Bean declarations, creates new `org.springframework.beans.factory.config.BeanDefinition` instances, overwriting any existing bean definitions for the same bean ID.

The optional Spring Environment property `skip.redundant.spring.imports` controls handling of redundant imports. If set to `true`, the first `<import/>` element will be used and all following, duplicate `<import/>` elements pointing to the same resource will be ignored. The property is `true` by default. The time saved depends on the number of duplicated `<import/>` elements.

CAUTION

Even though this setting is recommended, it may change which bean definitions are loaded. (As explained above, normally, bean definitions may be overwritten by subsequent imports, depending on how `<import/>` elements are used in a web application).

4.1.4 Structure of the Workspace

Starting with *CoreMedia Content Cloud* major version 10 (CMCC 10), this repository has been restructured to better reflect that the overall software system consists of several applications.

Overview

Since CoreMedia applications have been developed monolithically for years, there are lots of dependencies and shared code between the applications. Also, the build process of different applications was not independent, because they shared build configuration (through parent POMs).

The new *CoreMedia Blueprint* workspace structure is modular in the sense that it consists of many (sub-)workspaces that can be built independently, only interacting through Maven artifacts. Shared code still exists, and shared workspaces must be built before application workspaces, but workspaces of different applications are independent.

Besides shared workspaces (`shared/*`) and application-specific workspaces (`apps/*`), there are global Workspaces (`global/*`) that depend on several to all applications.

Workspace Concepts and Terminology

Workspaces

To reduce build-time dependencies and allow modular builds, the concept of workspaces has been introduced. A workspace is a Maven multi-module project that can be built independently, only relying on artifacts from the Maven repository, but not on anything else being present in the same Git repository. This means one Git repository hosts several workspaces. Since a workspace is a group of Maven modules and each module only belongs to one workspace, dependencies between modules of different workspaces lead to dependencies between their workspaces. In other words, workspaces are a coarsening of Maven

modules and their dependencies, just like modules (and their dependencies) are a coarsening of classes (and their dependencies).

Applications (Apps) and Shared Code

CoreMedia Content Cloud is a software system that consists of several applications. Here, an application is a piece of software running in the same execution environment (usually a JVM), serving a certain (business) objective, and communicating with other applications via remote calls. Examples of CoreMedia applications are CAE, Studio Server, Studio Client (execution environment: browser!), Content Server, and all Commerce Adapters.

An application consists of one or more application-specific workspaces and reuses shared code from arbitrary many other workspaces, but not from other application workspaces. This means that all code and resources used by more than one application must not be located in an application-specific workspace, but in a shared code workspace.

Putting all shared code into one workspace would have been too coarse-grained. One has to consider that shared code changes are much more expensive, since they potentially affect any application, thus after changes, all applications have to be rebuilt, retested, redeployed, and re-released.

CoreMedia CMS has a four-tier architecture: Between frontend and persistent data storage, unlike most architectures that use one "backend" tier, CoreMedia CMS features two tiers. The backend tier consists of Content Server, Workflow Server and Search (a specifically configured Solr). The middle tier acts as a frontend façade to the backend for delivery (CAE, Headless Server), editorial interface (Studio Server, *User Changes Application*, Studio Package Proxy), search (CAE Feeder, Content Feeder), and other tasks (Elastic Worker).

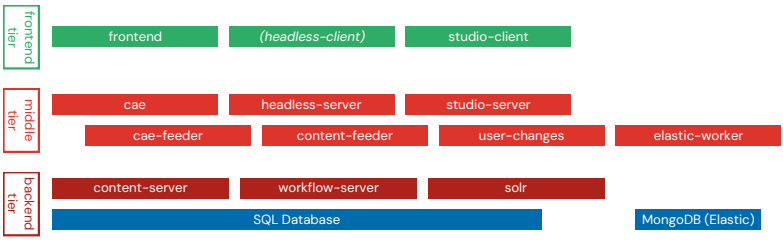

Figure 4.1. CoreMedia CMS's Four-Tier Architecture

Analyzing code reuse between applications in our code base validated the assumption that this four-tier architecture has a major influence on code sharing. Middle-tier servers share code not reused by backend servers, and vice versa. Computing the set of shared modules, it turned out that there was only one module shared by the backend servers (cap-serverbase), so CoreMedia decided

not to create a workspace with just one module and ended up with two shared code workspaces:

- shared/middle - contains all modules shared by two or more middle-tier servers, but not by backend server
- shared/common - all other shared code, shared by two or more servers of any tier

Global Modules

Despite the clear separation of application development, there is the need to unite all applications to a complete CoreMedia CMS software system. There are two use cases for doing so:

- Run system tests. A system (integration) test is a test that verifies the interaction of two or more applications and as such cannot be located in any application-specific workspace (and of course is not shared code, either).
- Deploy a complete CMS software system.

  CoreMedia offers prefabrication for setting up the complete system of all applications in form of a Docker compose file.

The system deployment workspace is called global/deployment.

All Workspaces

The following diagram shows all workspaces, grouped into shared, apps, and global.

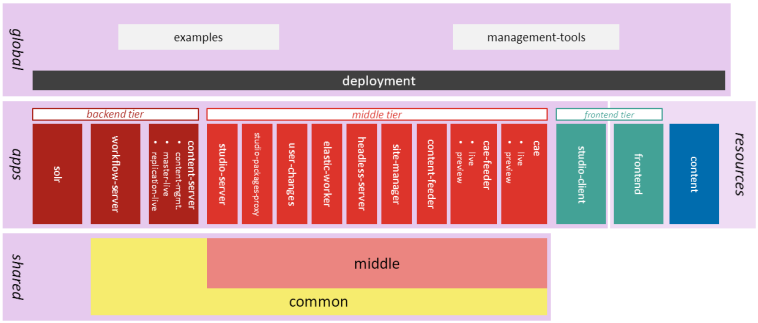

Figure 4.2. CoreMedia CMS's Shared, Application-Specific, and Global Workspaces

Dependency Management

Putting applications into focus leads to the idea that dependency management can also be done modularly, namely for each application, because each runs in its own execution environment. Since application-specific code only runs in one

execution environment, there are application-specific external dependencies that are managed centrally for each application. This means that not every (sub-)workspace needs its own external dependency management.

However, shared code needs to run in all applications that use it, so by reusing shared code, an application also inherits the shared code's dependency management.

Maven implements reused dependency management through "bill of material" (BOM) POMs. This means that there are third-party dependency management BOM POMs for each shared workspace and for each application.

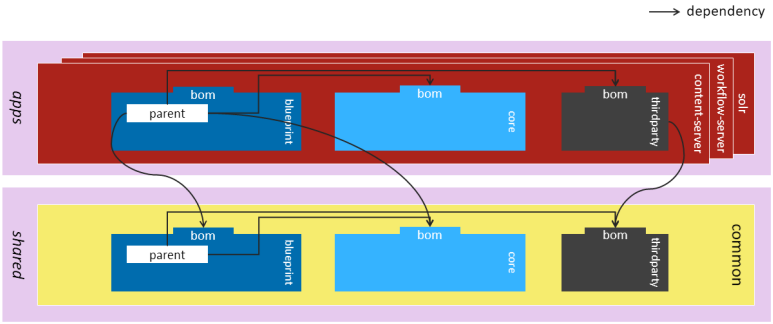

Figure 4.3. Backend Tier Workspace Dependencies

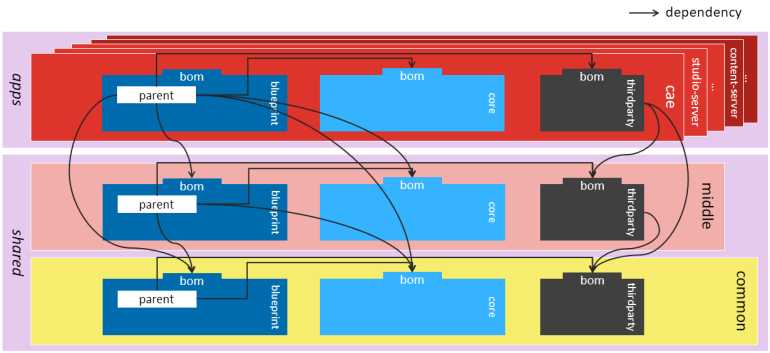

Figure 4.4. Middle Tier Workspace Dependencies

Enforcer

The `cmBannedDependencies` rule is used for global management of banned dependencies. It reads the banned dependencies from a configuration file on

cmBannedDependencies

the classpath, which by default comes with a dependency on `com.coremedia.cms:common-banned-dependencies`. It is an XML file which contains the `bannedDependencies` configuration element that you would normally include in the configuration of the enforcer-plugin. It is also possible to add additional includes and excludes directly in the custom rule element.

```
<rules>
  <cmBannedDependencies>
    <!-- configuration file from classpath with bannedDependencies -->

<configurationFile>/com/coremedia/cms/maven/enforcer/bannedDependencies.xml</configurationFile>

    <!-- additional banned dependencies -->
    <excludes>com.acme:some-banned-artifact</excludes>
    <!-- allow dependencies that are configured as banned in configuration
file -->

<includes>com.acme:some-allowed-artifact,com.acme.some-other-allowed-artifact</includes>

  </cmBannedDependencies>
</rules>
```

Example 4.3. cmBannedDependencies example

oneRepoEnforcerRule

The `oneRepoEnforcerRule` enforces some basic consistency of `groupId` and `version` in dependency elements: When the groupId of a dependency is `${project.groupId}` then the version should be set to `${project.version}` and vice versa. Also, when the dependency is already managed, the version should not be set directly. If you have for some reason a different groupId in your workspace and still want to use `${project.version}` for a dependency with that groupId, you can configure that groupId as a *sibling* with the parameter `siblingGroupIds` which takes a regular expression.

```
<rules>
  <oneRepoEnforcerRule>
    <!-- allows using project.version for dependencies with groupId matching
 this regex -->
    <siblingGroupIds>com\.acme\.groupA|com\.acme\.groupB</siblingGroupIds>
  </oneRepoEnforcerRule>
</rules>
```

Example 4.4. oneRepoEnforcerRule example

modularOneRepoEnforcerRule

The `modularOneRepoEnforcerRule` mainly enforces that one workspace always manages its dependencies on other workspaces. You should always manage this kind of dependencies by importing the BOM of the other workspace instead of using a versioned dependency directly. On the other hand, for dependencies inside one workspace you should use `project.version`.

When you have changed your blueprint groupId, you have to configure your groupId with the parameter `blueprintGroupId`.

If you have to violate these rules (for a hotfix, for instance), you can ignore certain dependencies, by adding an `ignoredDependencies` element to the rule, which works the same way as in the maven-dependency-plugin. The filter syntax is: `[groupId]:[artifactId]:[type]:[version]` where each pattern segment is optional and supports full and partial * wildcards. An empty pattern segment is treated as an implicit wildcard.

```
<rules>
  <modularOneRepoEnforcerRule>
    <!-- your blueprint groupId -->
    <blueprintGroupId>com.acme.blueprint</blueprintGroupId>
    <!-- do not analyze these dependencies -->
    <ignoredDependencies>
      <ignoredDependency>com.acme:some-artifact</ignoredDependency>
    </ignoredDependencies>
  </modularOneRepoEnforcerRule>
</rules>
```

Example 4.5. modularOneRepoEnforcerRule example

Following these patterns enables you to build the workspaces independently and to even use different versions for the separate workspaces.

Remark on Group and Artifact IDs

With the introduction of separate workspaces some aggregator and parent modules have to be copied to more than one workspace (`blueprint-parent`, for instance). To make the Maven coordinates unique the artifact IDs of these modules were prefixed with the name of the workspace (for example, `cae.blueprint-parent`), while the directory of the modules stayed as they were (for example, `blueprint-parent/`). The `groupId` could have been used for this, which would have been the more natural solution, but in order to get consistent group IDs for a workspace this would have meant new group IDs for every single artifact, which was refrained from changing for now.

There are some exceptions where the modules are copied to many workspaces, but got a real distinct artifact ID and directory (for example, `cae-core-bom`). These modules distinguish themselves as they are also relevant outside of a workspace, in contrast to the parents and aggregators, where the focus is put more on the similarity to the old structure and the other workspaces.

Development Use Cases

The new repository structure encourages working on a single workspace at a time, or at least on few workspaces.

Currently, you have to build workspace common and in most cases, that is when working on a middle tier app, workspace middle. Later, there should be a CI that produces SNAPSHOT artifacts for all modules from branch master, so that you

can let Maven fetch artifacts from there and only do local builds of workspaces you actually work on.

Working with Application-Specific Code Only

When your task only involves one application, build common, (if it is a middle tier app) middle, and the application's workspace on the command line:

```
  for ws in shared/common shared/middle apps/<some-app>; do mvn clean
source:jar install -f $ws -DskipTests <more-options>; done
```

Then, open only the application's workspace in IDEA. The goal source:jar allows browsing sources of shared code, even though they are not part of the IDEA project.

All Java applications are Spring Boot applications and can be started locally like so:

```
    mvn spring-boot:run -pl :<someapp>[-<variant>]-app
-Dspring-boot.run.jvmArguments=<jvmArgs>
-Dspring-boot.run.profiles=dev,local,private
```

The Spring Boot Maven plugin forks a JVM which means that system properties passed to the maven call are not passed to the forked JVM. Arguments can only be passed via the `spring-boot.run.jvmArguments` flag as described in the official documentation of the Spring Boot Maven plugin. In the example given above `<jvmArgs>` may have the following value to enable remote debugging on port `5005` as also described in the official documentation of the Spring Boot Maven plugin:

```
    -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=*:5005
```

The value of `<jvmArgs>` may also contain system properties given via `-Dkey=value`. The recommended approach for external configuration is to place a profile specific application properties file, such as `application-private.properties`, under `src/main/resources/config` of the app's spring-boot module. Note that this path is ignored by git so that you don't check in local or private configuration. This profile is activated in addition to the `dev` and `local` spring profiles with the `-Dspring-boot.run.profiles=dev,local,private` argument. As an alternative, you may also use system environment variables such as `INSTALLATION_HOST=<FQDN>` to connect your local application to a CI reference system.

NOTE

The `private` profile is just an example which may fit for setups with recurring configuration or a single remote test system to connect to. It's also valid to have various differently named profiles defined below `src/main/resources/config` such as `application-env1.property` and `application-env2.property` according to the available backend services or the individual configurations applied to the apps.

An alternative to using the Spring Boot Maven plugin is to run the application directly from your IDE. To run the application from IDEA, the IDEA run configuration provided in the `ideaRunConfiguration` subfolder of the Spring Boot folder can be copied to `.idea/runConfigurations`.

Working with Shared Code Only

This use case is quite similar to the first one.

When working with shared/middle, you have to build shared/common first.

When working with shared/common, nothing needs to be built before.

Keep in mind that changes in shared code have impact on many, sometimes even all CoreMedia applications. Treat shared code like public API!

- Refrain from unnecessary breaking changes.
- Write unit tests for new functionality.
- If a change in shared code passes unit tests, but CI alerts you that it breaks an application, write a regression test before fixing shared code.
- Document what you change.
- If possible, put shared code changes and application code changes in separate commits.

Working with Application-Specific and Shared Code

There is still a lot of shared code, so it might happen more often than not that part of the code you must touch to implement an application feature is located in a shared workspace. The advantage of the new multi-workspace structure is that you can immediately tell that code is shared by the fact that it is located under a path starting with `shared/`.

The idea of modularization is to not fall into monolithic development mode (see below) just because you change shared code. In an ideal world, all shared code's contracts would be checked by unit tests. So if you change shared code in a non-breaking fashion and no tests fail, you can use new API in the application you actively work on and need not worry about other applications also using the changed code.

Even if you do not have sufficient unit tests coverage of shared code, you might have integration tests that should detect shared code changes that break other applications. Thus, if you push your shared code changes and your application-specific changes to a feature branch, your local CI should take care of validating that no other applications are (negatively) affected by your changes. Treat shared code as having an API, and you should be fine.

Working in IDEA, the most convenient way is to add the needed shared code workspace(s) to the application's IDEA project.

After that, you have to run "Reimport Maven Projects" to update the dependencies on shared code from references into your local Maven repository to references to the corresponding IDEA modules. This enables a fast development turn-around after changes in shared code, including source-level debugging and hot deploy.

Working with (Almost) All Code

If your task requires global changes, for example, a shared third-party library is updated to a new (major) version, you can still use the multi-workspace repository like the old monolithic workspace.

You can simply build the whole workspace through the root POM and then open it in IDEA.

Now, having one big IDEA project, you can do global refactorings or search and replace.

It is not recommended to work like this for normal feature implementation, because importing the large overall project into IDEA takes quite some time, and after switching to a different branch or merging in master, this process has to be repeated over and over again.

Even if you have to perform application-spanning changes, try to find a subset to work on:

- Do the changes affect Java code "only"? Even though most of your code is Java, when restricting the IDEA project to Java workspaces, you can leave out Studio Client, Frontend, and Content. Although these are only three workspaces, they use quite different tooling and in case of Studio Client a custom IDEA Maven import process, which you may be glad to avoid.
- Are the changes located in backend-tier servers only? If so, you can leave out `shared/middle`, which contains a large fraction of the workspace modules and code.

The *CoreMedia Blueprint* workspace contains the `modules` and `test-data` top level aggregator modules.

modules

Almost every workspace, be it an application, shared or global workspace, has a `modules` top-level aggregator module which is the most important space for project developers. All code, resources, templates and the like is maintained here. You can start all components locally in the modules area.

The `modules` hierarchy consists of modules that build libraries and modules that assemble these libraries to applications. Library modules are being built with the standard Maven *`jar`* packaging type.

Most applications created by the modules below the modules folder are Spring Boot applications using the standard Maven *`jar`* packaging type. The *CoreMedia Studio* client is a browser application and uses *pnpm* instead of Maven. All other applications are command line tools built with the custom *`coremedia-application`* packaging type. *`coremedia-application`* modules are built with the *coremedia-application-maven-plugin*, a custom plugin tailored to the CoreMedia `.jpif` based application runtime.

The modules folder is structured in sub-hierarchies by grouping modules due to their functionality. There is a dedicated group `cmd-tools` for command line tools and functional groups like `ecommerce`. Since the introduction of application-oriented workspaces, the groups for these applications (`cae`, `studio`, ...) are mostly redundant, but kept for structural similarity to previous releases of *CoreMedia Content Cloud*. The same holds true for the group named `shared`, whose modules now are in most cases part of one of the two `shared` workspaces. The remaining two groups `extension-config` and `extensions` are required for the extensions functionality of *CoreMedia Blueprint* workspace.

By default, *CoreMedia Blueprint* workspace ships preconfigured with no extensions. Typically, extensions do not extend one, but many applications. *CoreMedia Project Extension*s decouple the application from the dependencies it is extended by and lets you automatically manage these dependencies. Not all extensions will be used in a project right from the start. In this case, the *CoreMedia Extension Tool* allows you to easily deactivate or enable features like *Elastic Social*. See Section 4.1.5, "Project Extensions" [77] for details.

test-data

The `content/test-data` folder contains test content to run *CoreMedia Blueprint* with. It can be imported into the content repository by using the CoreMedia *serverimport* tool. Extensions may contain additional test-data folders.

4.1.5 Project Extensions

One of the main goals of *CoreMedia Content Cloud* is to offer a developer friendly system with a lot of prefabricated features, that can simply be extended modularly. To this end, CoreMedia provides the Maven based *CoreMedia Blueprint* workspace and the extensions mechanism.

An extension adds new features to one or more CoreMedia applications. Assume, for example, that a feature requires a new content type. In this case the extension affects at least three applications:

- The Content Server needs the new content type
- The CAE needs according content beans
- Studio needs according content forms

Manually enabling or disabling a feature in all the affected applications would be cumbersome, tedious and error-prone. Therefore, CoreMedia provides the extensions mechanism, which allows you to enable or disable a feature for all affected applications by a single configuration switch. The extensions mechanism is based on Maven modules and runtime dependencies. Adhering to some structural conventions enables you to use the *CoreMedia Extension Tool* to manage those dependencies.

Extensions Mechanism Structure

The extension mechanism structure consists of two parts:

- extensions which extend different applications to provide a new feature
- extension points of CoreMedia applications to which the features are added.

Extensions and extension points are separated for each application workspace (`apps/*`). In each workspace, extension points (usually one) can be found at the local path `./modules/extension-config/*-extension-dependencies` and extensions are located below `./modules/extensions`.

extension points

Any Maven module with an artifact ID using the pattern `{prefix}-extension-dependencies` constitutes an extension point, for example the artifact ID `studio-server-extension-dependencies` defines an extension point named studio-server. This name serves as an ID by which an extension point can be referenced from an extension.

extension point names backwards compatibility

In the new version of *CoreMedia Content Cloud*, the extension point names have been aligned with the application (workspace) names. For backwards compatibility, the old extension point names may still be used, but are deprecated.

Extensions points collect runtime dependencies on app extensions. Applications have explicit dependencies on extension points, so they get transitive dependencies on the actual app extensions.

extensions and application extensions

Since an extension extends several applications, it consists of so-called application extensions or for short app extensions, where one app extension uses exactly one extension point and thus extends exactly one CoreMedia application. The desired extension point is marked in the `pom.xml` by the property named `coremedia.project.extension.for` and has the name of the extension point as value.

```
<properties>

<coremedia.project.extension.for>studio-client</coremedia.project.extension.for>
</properties>
```

Example 4.6. Specify the extension point

In previous versions of *CoreMedia Content Cloud*, one module was allowed to use several extension points. Since now, a clear separation of applications is enforced and only one extension point may be used. To migrate a multi-extension-point extension, you must move the extension module to a shared code location (`shared/middle/modules/extensions/...`) and create two app extensions that both have a dependency on the shared-code module.

Short overview of the extensions structure of each application workspace `apps/{application}`:

- ./modules/extensions - contains application extensions as Maven submodules
  - one extension - aggregator of all application extensions (usually only one per workspace)
    - one application extension - Maven module with property `coremedia.project.extension.for`
    - another application extension - Maven module with property `coremedia.project.extension.for`
    - extension library - Maven module with application-specific code that is used by one or more of the application extensions
- ./modules/extension-config - Maven aggregator module
  - {extension-point-name}-extension-dependencies - extension point for application with the identifier {extension-point-name}
- ./spring-boot/{application}-app - has a runtime dependency on {extension-point-name}-extension-dependencies.

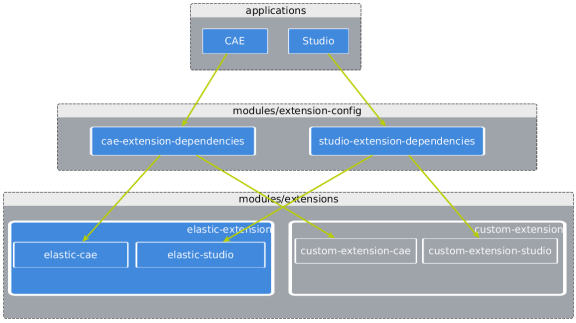

Figure 4.5. CoreMedia Extensions Overview

Usually, each workspace below `apps/` contains exactly one application with exactly one extension point. However, there are two exceptions to this one-to-one rule.

The CAE comes in two application "flavors", cae-preview-app and cae-live-app, and consequently offers two extension points. Because only preview allows dedicated extensions, the extension points are called cae-preview (preview only) and cae (both preview and live).

The second exception is *CoreMedia Studio* client: The studio-app (an Ext JS app, not a Spring Boot app) allows statically linked as well as dynamically linked extension modules. Depending on your choice, use the extension point studio-client or studio-client-dynamic.

Extension points do not contain any code or configuration directly and should never be edited manually, because they are modified by the *CoreMedia Extension Tool* to contain all collected dependencies on active application extensions.

Usage of the *CoreMedia Extension Tool*

Usage of the CoreMedia Extension Tool

Within the extension mechanism structure, the extensions and the dependencies from the applications onto the extension points are maintained manually. The extension point modules must already be present and are updated by the *CoreMedia Extension Tool*. The tool lets you synchronize the dependencies from the extension points to application extensions by their status (enabled/disabled). You can disable, enable, remove and add prefabricated and custom extensions.

For convenience, the *CoreMedia Extension Tool* is implemented as a Maven plugin, which is preconfigured for usage in the *CoreMedia Blueprint* workspace in the Maven POM file `workspace-configuration/extensions/pom.xml`. The tool is used by invoking it through this POM, by running Maven either from that directory or from the project root directory, adding `-f`

`workspace-configuration/extensions`. All relevant use cases and command line examples can be found in `workspace-configuration/extensions/README.md`. The most important advice is to call `mvn -f workspace-configuration/extensions extensions:help` to see full usage instructions of all available goals (commands) and their options (parameters). Especially the extensive help text of goal `sync` is important to understand how the tool works.

When disabling or removing extensions, note that extensions may depend on each other. Therefore, when you enable an extension all the extensions it depends on must also be enabled. For example, lc-asset makes only sense if lcis enabled too. Otherwise, you would encounter runtime errors like missing Spring beans.

To prevent such situations, the *CoreMedia Extension Tool* does not allow enabling or disabling extensions that would result in such an inconsistent state. The tool will stop with an error message that tells you exactly what went wrong and how to fix the problem. For example, if you try to disable the extension es, the tool outputs the following error (the concrete list of dependent modules may vary in future releases):

```
[ERROR] Inconsistent set of extensions to disable/remove. These extensions
would need to be
  disabled or removed, too:
  [lc-es]
```

Implementing a Custom Extension

The following steps summarize how to add a custom extension to the *CoreMedia Blueprint* workspace. Let's call the extension my-feature.

1. Plan your feature: Which applications (workspaces) do you need to extend? Do you have shared code? Let's assume you need to extend cae and studio-server and want to have one shared code module.
2. For shared code, add a new Maven module at `shared/middle/modules/extensions/my-feature/`. Its parent must be set to `com.coremedia.blueprint:middle.extensions:1-SNAPSHOT`. If you have multiple shared modules, add an aggregator module at that location and place the other shared modules below that aggregator.
3. For each application you want to extend, here cae and studio-server, create a new Maven module in its application workspace at `apps/{application}/modules/extensions/my-feature`. Its parent must be set to `com.coremedia.blueprint:{application}.extensions:1-SNAPSHOT`. Again, if you have multiple application-specific modules, instead create an aggregator at that location and place all modules below that aggregator.

4. Set the `coremedia.project.extension.for` property in the `pom.xml` file of all application extension modules, that is all modules that are supposed to be added as `{extension-point}-extension-dependencies`. Usually, there is exactly one such module per extension point/application.
5. This would be a good time to do a VCS commit. This helps you to see what modifications the *CoreMedia Extension Tool* applies in the next step, and if anything goes wrong, allows you to revert to this state.
6. Run the *CoreMedia Extension Tool* with goal `sync` (`mvn -f workspace-configuration/extensions extensions:sync` and enable your new extension by adding `-Denable=my-feature`.
7. Use `mvn -f workspace-configuration/extensions extensions:list` to check that your extension has been added (`my-feature` appears in the list) and activated (it does *not* start with a hash ("`#`")).
8. Check the changes the tool has applied to each affected workspace:
   - Your extension's workspace-specific root module is added as a `<module>` to the corresponding workspace extensions aggregator, `{workspace}.extensions/pom.xml`.
   - All affected extension points `{extension-point}-extension-dependencies/pom.xml` files should now contain dependencies on your application extensions.
   - All modules of your extension that now belong to this workspace are added to the workspace's extensions BOM, `{workspace}-extensions-bom/pom.xml`, so that their version is managed for others who import this BOM.
9. Rebuild your project, at least all affected workspaces.

If your extension becomes obsolete, you can disable it:

1. Run the *CoreMedia Extension Tool* with goal `sync` and option `-Ddisable=my-feature`.
2. Checkpoint: In all affected workspaces, all three types of POMs (extensions aggregator, extension point, extensions BOM) no longer refer to any of your extension modules.
3. Rebuild your project, at least all affected workspaces.

If you want to reactivate your extension, just call the `sync` goal with `-Denable=my-feature` again and rebuild your project. Otherwise, if you are sure that you will never need your extension again, rerun the *CoreMedia Extension Tool* with goal `sync` and option `-Dremove=my-feature`, which removes all files of your extension from all workspaces.

NOTE

Removing is not recommended for extensions that come with the Blueprint, because when updating to a new Blueprint release, deleted files lead to merge conflicts for all files updated by CoreMedia.

Best Practice

In a particular project the set of active extensions is usually not changed frequently. Therefore, CoreMedia recommends applying the *CoreMedia Extension Tool* only manually and to check in all changed files into the VCS. You should then call the tool's `sync` goal without any additional parameters on a regular basis to check that all generated extension dependencies are in a consistent state.

Alternatively, you could integrate the tool into the CI and synchronize the extension points in every build. However, since the result is almost always the same, this would unnecessarily increase the roundtrip time.

4.1.6 Application Plugins

Application Plugins offer a way to extend *CoreMedia Content Cloud* applications. The focus of plugins is clear APIs and strong isolation to increase reusability and decrease maintenance effort.

Difference between extensions and plugins

In contrast to classic Blueprint Extensions (see Section 4.1.5, "Project Extensions" [77]), which are usually part of a project's Blueprint and are built together with the application, plugins are meant to be developed and released separately. This way plugins can be packaged with the application at a later time, for example, when creating a Docker image or even later, when deploying the application.

Similar to Blueprint Extensions, to implement some feature through plugins, it must be decomposed into parts that plug into exactly one CoreMedia application and optional parts that are reused in different application plugins (called shared code). The plugin artifacts resulting from these parts can then be bundled via a *Plugin Descriptor* (see Section 4.1.6.3, "Plugin Descriptors and Bundled Plugins" [101]).

The technology to implement plugins is different for *Studio Client* than for the Java based applications, so there are dedicated sections for these two types of plugins, followed by a section explaining how to bundle plugins that together implement some feature.

- Section 4.1.6.1, "Plugins for Java Applications" [83]

- Section 4.1.6.2, "Plugins for Studio Client" [98]
- Section 4.1.6.3, "Plugin Descriptors and Bundled Plugins" [101]

4.1.6.1 Plugins for Java Applications

Some CoreMedia Java applications provide *extension points* that are interfaces for which you can provide implementations (*extensions*, see Section "Plugin Extensions" [85]) via a *plugin*.

Plugins and their application context

For the integration of plugins with an application, the Spring framework is used. Every plugin has its own application context with its own class loader, which has the app's class loader as parent and resolves classes in the plugin first. A plugin cannot access the application context of the application directly but only use a well-defined subset of beans that the application has copied into the plugin context (see Section "Application Beans in Plugins" [87]). The application also takes care of starting the plugin's context and then collects all beans of extension point types. A plugin extension is very similar to a Java Service Provider (SPI), but is based on beans (instances) instead of classes.

Creating Plugins

A plugin is a (zipped) folder with the following structure:

classes/	The classes of your plugin
lib/	Third-party dependencies (JAR files) used by your plugin
plugin.properties	File for plugin configuration and metadata

The `plugin.properties` file provides metadata of your plugin. Most importantly, the properties give your plugin an identifier and configure a Spring configuration class that will be registered with the application context. These are the supported properties:

plugin.id	The ID of the plugin, must be unique, for example, **`MyPlugin`** (required)
plugin.version	The version of the plugin following the Semantic Versioning Specification, for example, **`1.2.3`** or **`1.0.0-SNAPSHOT`** (required)
plugin.configuration-class	The Spring Configuration class, for example, **`com.acme.myplugin.MyPluginConfiguration`** (optional; required for Java exten-

	sions but not needed for plugins that only provide resources)
plugin.provider	The provider/author of the plugin, for example, **ACME** (optional)
plugin.dependencies	Comma-separated list of IDs of plugins this plugin depends on, for example, **some-plugin,some-other-plugin**. See Section "Plugin Dependencies" [94] for details. (optional)
plugin.add-on-for	The plugin ID of the plugin that this plugin is an add-on for, for example, **some-plugin**. See Section "Add-Ons" [97] for details. (optional) (optional)
plugin.independent	This Boolean property determines whether the plugin operates independently of other plugins and application beans. If set to **true**, the plugin's configuration class is registered early in the application's lifecycle, which is essential for some plugins that provide common or foundational application beans required by other plugins or the application itself. By default, this property is **false**, meaning the plugin will be initialized in the standard order and may rely on other plugins or application beans. (optional)

Maven Plugin

To create a plugin with Maven, you can use the `coremedia-plugin-maven-plugin`. To this end, the packaging type of the POM must be set to `coremedia-plugin`, and the `coremedia-plugin-maven-plugin` has to be added with `extensions` set to `true`. The plugin will then create a plugin-zip during the package phase. It is possible to provide a custom `plugin.properties` file or to generate one using the configuration from the POM.

Example

```
@Import(SomePluginBeansConfiguration.class)
@Configuration
public class MyPluginConfiguration {
    @Bean
    public MyExtension myExtension(SomeBean someBean) {
        return new MyExtension(someBean);
```

```
    }
}
```

Example 4.7. com.acme.myplugin.MyPluginConfiguration

```
public class MyExtension implements SomeCoremediaExtensionPoint {
    public MyExtension(SomeBean someBean) {
        ...
    }
    ...
}
```

Example 4.8. com.acme.myplugin.MyExtension

```
<project>
  <groupId>com.acme.myplugin</groupId>
  <artifactId>MyPlugin</artifactId>
  <version>1.2.4-SNAPSHOT</version>
  <packaging>coremedia-plugin</packaging>

  <build>
    <plugins>
      <plugin>
        <groupId>com.coremedia.maven</groupId>
        <artifactId>coremedia-plugins-maven-plugin</artifactId>
        <version>1.4.0</version>
        <extensions>true</extensions>
        <configuration>
          <pluginId>${project.artifactId}</pluginId>
          <pluginVersion>${project.version}</pluginVersion>
         <pluginConfigurationClass>com.acme.myplugin.MyPluginConfiguration</pluginConfigurationClass>

          <pluginProvider>ACME</pluginProvider>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Example 4.9. pom.xml

Plugin Extensions

Plugin extensions must not be confused with *CoreMedia Project Extensions* (see Section 4.1.5, “Project Extensions” [77]). Project extensions are just normal Maven modules in your CoreMedia Blueprint project and have no isolation. They use the same dependencies, Spring application context and class loader as the rest of the application and all other extensions. Plugin extensions only use an explicit set of beans from the application, can bring their own dependencies and do not interfere with other parts or plugins of the application.

Plugin Extension Points

Extension points are CoreMedia interfaces or classes that are annotated with `@ExtensionPoint`.

Typically, extension points are strategies like content hub adapters or validators (for an overview, see section "Plugin Extension Points Reference" [86]). A plugin provides instances of extension points as Spring beans in its configuration class. The plugin framework detects them at runtime by their type and passes them to the components that apply them.

Plugin Extension Points Reference

CoreMedia Content Cloud features the following extension points (grouped by applications):

Every application using Spring Web MVC

- `PluginRestController`

Content Feeder / Studio Server

- `CapTypeValidator`
- `CapTypeValidatorFactory`
- `NotificationListener`
- `PropertyValidatorFactory`
- `Validator`
- `ValidatorFactory`

Studio Server

- `CSPSettings`
- `ContentHubAdapterFactory`
- `EntityController`
- `FeedbackHubAdapterFactory`
- `FeedbackProviderFactory`
- `JobFactory`

Headless Server

- `CaasWiringFactory`
- `CopyToContextParameter`
- `CustomFilterQuery`

- `CustomScalarType`
- `FacetedSearchServiceProvider`
- `FilterPredicate`
- `GrapQLLinkComposer`
- `PluginConverter`
- `PluginSchemaAdapterFactory`
- `PluginSchemaGenerator`
- `SearchServiceProvider`
- `SuggestionSearchServiceProvider`
- `UriLinkComposer`

For details, consult the API documentation of the particular extension point.

Plugin Resource Extension Points

Besides providing Beans for a given extension point interface, plugins can also provide resource files for a given pattern from their classpath.

CoreMedia Content Cloud features the following resource patterns (grouped by applications):

Headless Server
: Section 4.17.3, "Resource file loading" in *Headless Server Manual*

Content Server
: Section 4.2.4, "Content Types from Plugins" in *Content Server Manual*

Application Beans in Plugins

Each plugin has its own Spring application context. However, in order to implement your feature, you might need some services from the application, for example the `CapConnection`.

The applications of *CoreMedia Content Cloud* expose dedicated sets of beans for usage in plugins, which are provided as Spring configuration classes that you can import in your plugin's configuration class. These configuration classes are annotated with `@BeansForPlugins`.

Other plugins can also provide such configuration classes. To access them, you have to add these plugins as dependencies.

Beans for Plugins Reference

The following `@BeansForPlugins` configuration classes are available:

- `CommonBeansForPluginsConfiguration` Available in all apps with extension points. Dependency:

```
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>common.beans-for-plugins-container</artifactId>
  <scope>provided</scope>
</dependency>
```

- `CommerceBeansForPluginsConfiguration` Provides access to commerce beans. Available in plugins for commerce-enabled apps. Dependency:

```
<dependency>
  <groupId>com.coremedia.blueprint.base</groupId>
  <artifactId>middle.bpbase-lc-beans-for-plugins-container</artifactId>
  <scope>provided</scope>
</dependency>
```

- `HeadlessBlueprintBaseBeansForPluginsConfiguration` Provides access to Headless Server beans. Available in `headless-server` plugins. Dependency:

```
<dependency>
  <groupId>com.coremedia.blueprint.base</groupId>
  <artifactId>bpbase-headless-server-core</artifactId>
  <scope>provided</scope>
</dependency>
```

- `ContentHubBeansForPluginsConfiguration` Provides access to Content Hub beans. Available in `studio-server` plugins. Dependency:

```
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>content-hub-api</artifactId>
  <scope>provided</scope>
</dependency>
```

- `FeedbackHubBeansForPluginsConfiguration` Provides access to Feedback Hub beans. Available in `studio-server` plugins. Dependency:

```
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>feedback-hub-api</artifactId>
```

```
  <scope>provided</scope>
</dependency>
```

- `SettingsBeansForPluginsConfiguration` Provides access to the settings service bean. Available in `studio-server`, `headless-server`, and `cae` plugins. Dependency:

```
<dependency>
  <groupId>com.coremedia.blueprint.base</groupId>
  <artifactId>middle.bpbase-settings-beans-for-plugins</artifactId>
  <scope>provided</scope>
</dependency>
```

The API documentation of the `@BeansForPlugins` configuration classes shows the actual beans they provide.

Application Properties

All Spring properties are passed from the application to its plugins, including those from `application.properties`, system properties, environment variables, etc. You can access these properties in your Plugin's `Application Context` for example via `@Value` or `ConfigurationProperties` as usual.

Although it is technically possible, you should not rely on existing properties of the application but instead create new dedicated properties for your plugin.

CoreMedia / Third-party Dependencies

Plugins have their own class loader

Each plugin has its own class loader and can bring its own third-party libraries. This gives you control over the versions of the third-party libraries you use and makes you independent of third-party version changes in the particular *CoreMedia Content Cloud* application, so that your plugin will reliably continue to work with *CoreMedia Content Cloud* updates. The plugin's libraries are included in the plugin ZIP file, where the plugin class loader discovers them and loads the classes from.

Shared classes

However, at runtime your extensions are integrated into the particular *CoreMedia Content Cloud* component that features the extension point. Technically, this requires some classes (especially the *Spring Framework*) to be shared with the application.

Moreover, some frameworks like *SLF4J* have application wide aspects, which do not work with objects of classes from different class loaders. And last but not

least, any *CoreMedia Content Cloud* classes must be shared with the application. Even if it would work to use a particular *CoreMedia Content Cloud* feature just like an independent third-party library, CoreMedia does not guarantee this for updates, so just don't do it.

The following subsections describe how you can handle these issues.

Dependencies in Practice

Plugin developers must handle the difference between plugin libraries and shared classes appropriately. You have to develop your plugin in a Maven project. Each extension has its own module. Add a `dependencyManagement` section with at least the following entries to your POM file:

- The BOM that manages the extension point
- The BOM that manages the shared classes for the particular application (Each application with extension points provides such a BOM.)

Now, if you add a new dependency to your POM file, check whether it is managed by this dependency management. If it is not, it does not need to be shared, and you can simply declare the version of your choice in the dependency. Otherwise, omit the version and set the dependency scope to `provided`. Scoping all shared dependencies as `provided`, you can easily use the *maven-dependency-plugin* and the *maven-assembly-plugin* to build the plugin ZIP file including the non-shared libraries.

For example, a POM file for a *studio-server* plugin looks like this:

```
<project>

  <!-- [...] -->

  <dependencyManagement>
    <dependencies>

      <!-- For the extension point, e.g., ContentHubAdapterFactory -->
      <dependency>
        <groupId>com.coremedia.cms</groupId>
        <artifactId>studio-server-core-bom</artifactId>
        <version>${studio-server.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>

      <!--
        For the third-party classes that must be
        shared with the studio-server application
      -->
      <dependency>
        <groupId>com.coremedia.cms</groupId>
        <artifactId>studio-server-thirdparty-for-plugins-bom</artifactId>
        <version>${studio-server.version</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
```

```
  </dependencyManagement>

  <dependencies>
    <!-- An independent third-party library, managed right here. -->
    <dependency>
      <groupId>jakarta.activation</groupId>
      <artifactId>jakarta.activation-api</artifactId>
      <version>2.1.3</version>
    </dependency>

    <!-- coremedia dependencies must always be shared. -->
    <dependency>
      <groupId>com.coremedia.cms</groupId>
      <artifactId>content-hub-api</artifactId>
      <scope>provided</scope>
    </dependency>

    <!--
      The Spring framework is managed by studio-server-thirdparty-for-plugins-bom,
      thus, it must be shared by the plugin.
    -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <!-- [...] -->

</project>
```

More Dependency Subtleties

Further dependency problems might occur when you try to link instances of plugin classes with objects of application classes. The following example illustrates this. Assume, that you want to use the FasterXML Jackson libraries in your plugin. Those are not managed in the `studio-server-thirdparty-for-plugins-bom`, so you add a normal dependency to your POM:

```
<dependencies>
  <dependency>
    <groupId>tools.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>3.1.0</version>
  </dependency>

  <dependency>
    <groupId>com.coremedia.cms</groupId>
    <artifactId>content-hub-api</artifactId>
    <scope>provided</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <scope>provided</scope>
  </dependency>

  <!-- [...] -->
```

```
<dependency>
```

You code your plugin, you build your plugin, you run your plugin. Everything works fine.

The next change in your plugin involves Spring's `JacksonJsonHttpMessageConverter`:

```
import org.springframework.http.converter.json.JacksonJsonHttpMessageConverter;
import tools.jackson.databind.json.JsonMapper;

private HttpMessageConverter<Object> createMessageConverter() {
  JsonMapper jsonMapper = JsonMapper.shared();
  return new JacksonJsonHttpMessageConverter(jsonMapper);
}
```

You code your plugin, you build your plugin, you run your plugin. Everything wor… but wait, what's this?

```
the class loader org.pf4j.PluginClassLoader @227b4be7 of the current class,
com/acme/my/famous/StudioPlugin,
and the class loader 'app' for the method's defining class,
org/springframework/http/converter/json/AbstractJacksonHttpMessageConverter,
have different Class objects for the type
tools/jackson/databind/json/JsonMapper
```

The `JacksonJsonHttpMessageConverter` class is shared with the application (because the `spring-web` dependency has the scope `provided`) and is loaded by the application class loader. `JacksonJsonHttpMessageConverter` references `JsonMapper` as a constructor argument, so the application class loader also loads the `JsonMapper` class.

Your plugin declares an ordinary (scope `compile`) `jackson-databind` dependency. Thus, the plugin class loader also loads the `JsonMapper` class, independently of the application. Now, you invoke `new JacksonJsonHttpMessageConverter(jsonMapper)`. The argument is an instance of the plugin's `JsonMapper`, but the method expects an instance of the application's `JsonMapper`. These two classes are not assignment compatible, not even if the plugin and the application use the same version of the `jackson-databind` library! This causes the observed error.

The solution is to set the scope `provided` for the `jackson-databind` dependency, in order to share the `JsonMapper` class with the application, even though `jackson-databind` is not managed by `studio-server-thirdparty-for-plugins-bom`. However, this has some drawbacks that you know already from the CoreMedia Extensions:

- You are bound to the particular version of `jackson-databind` that is used by Spring, so you cannot use the new features of later versions.

- Other *CoreMedia Content Cloud* versions may use other Spring/Jackson versions, which may differ in functionality and make your plugin less portable.

This combination of Spring and Jackson is just an example. You may need to use other libraries with scope `provided` too. CoreMedia refrained from adding all such transitive dependencies to the `thirdparty-for-plugins` BOMs, because of the following reasons:

- The appropriate scope depends on the particular usage of the library. It is not generally `provided`. As long as you use a library only with other plugin classes, you can use a `compile` scope dependency.
- Normally, BOMs do not list transitive dependencies explicitly, so CoreMedia adheres to this convention.

Due to the drawbacks mentioned above, CoreMedia recommends that you use scope `compile` for dependencies that are not managed by the `thirdparty-for-plugins` BOMs whenever possible, and only switch to `provided` when necessary.

Using Plugins

To use your plugins, you have to provide the paths to the directories containing plugins for the application with the Spring property `plugins.directories`.

If you put a zipped plugin in there, it will be automatically extracted on the first start of the application and the extracted directory is used afterwards. See Section 4.3, "Build and Run the Applications" [119] for how to configure and run CoreMedia Java applications.

Normally, if a plugin could not be started, an ERROR is logged and the application continues without the respective plugin. With the property `plugins.required-plugins` it is possible to abort the application start if one of the listed plugins is missing or could not be started.

Plugin Manager Properties

`plugins.directories`

Type	`List<String>`
Description	The directories from where plugins are loaded.

`plugins.directory`

Type	`String`
Description	The directory from where plugins are loaded.
Deprecation	This property has been deprecated and will be removed in a future version. Use `directories` instead. **Reason:** Introduction of new list-valued property 'directories'

`plugins.required-plugins`

Type	`List<String>`
Description	A list of ids for plugins that are required. If one of these plugins is missing or could not be started, the application startup will be aborted. This property is only effective when 'plugins.directories' is set.

Table 4.1. Plugin Manager Properties

Plugins Actuator

Applications that use plugins provide information about their registered plugins via the actuator endpoint `actuator/plugins`.

Plugin Dependencies

Plugins can also have dependencies on other plugins to make use of their classes and beans. Dependencies are declared in the property `plugin.dependencies`. You can declare just the dependent plugin's ID or a specific version or version range separated by `@`. Multiple dependencies can be declared separated by commas.

Examples

- `some-plugin`
- `some-plugin@1.0.0`
- `some-plugin@>=1.0.0 & <2.0.0`

Providing Beans to Dependents

The configuration classes described in Section "Application Beans in Plugins" [87] are a kind of public beans API for your plugin. For each of those configuration classes, you need to create a class implementing the `BeansForPluginsContainer` marker Interface. This implementation serves as a container for your beans and is injected into the dependents application context.

For every bean definition in the Configuration class there should be a field, a constructor argument and a getter in the `BeansForPluginsContainer` class. The configuration class should have a field and a constructor with one argument of the concrete `BeansForPluginsContainer` type, so it can delegate to this field in its bean defining methods. This allows Spring to resolve the bean dependencies for the `BeansForPluginsContainer` classes and dependents of your plugin get a simple IDE supported way to find and inject your beans.

Example

PluginB depends on PluginA; PluginA provides a bean of type `SomeBeanFromA` to its dependencies and PluginB uses this bean.

PluginA

```
plugin.id=pluginA
plugin.version=1.2.3
plugin.configuration-class=com.acme.plugin_a.PluginAConfiguration
```

Example 4.10. PluginA plugin.properties

```
// Used to collect beans to be injected into dependent plugins
public class PluginABeansForPluginsContainer implements BeansForPluginsContainer {
  private final SomeBeanFromA someBeanFromA;

  public PluginABeansForPluginsContainer(SomeBeanFromA someBeanFromA) {
    this.someBeanFromA = someBeanFromA;
  }

  public SomeBeanFromA getSomeBeanFromA() {
    return someBeanFromA;
  }
}
```

Example 4.11. PluginABeansForPluginsContainer

```
// Defines a public API of Beans to be used by dependents
// and provides convenient access to beans from PluginABeansForPluginsContainer.
@BeansForPlugins
@Configuration(proxyBeanMethods = false)
public class PluginABeansForPlugins {
  private final PluginABeansForPluginsContainer pluginABeansForPluginsContainer;
```

```
  @SuppressWarnings("SpringJavaInjectionPointsAutowiringInspection")
  public PluginBeansAConfiguration(PluginABeansForPluginsContainer pluginABeansForPluginsContainer)
  {
    this.pluginABeansForPluginsContainer = pluginABeansForPluginsContainer;
  }

  @Bean
  public SomeBeanFromA someBeanFromA() {
    return pluginABeansForPluginsContainer.getSomeBeanFromA();
  }
}
```

Example 4.12. PluginABeansForPlugins

```
...
  @Bean
  SomeBeanFromA someBeanFromA() {
    return new SomeBeanFromA();
  }

  // Beans of type PluginBeans will be collected and injected into
  // dependents by the framework.
  @Bean
  PluginABeansForPluginsContainer pluginBeansA(SomeBeanFromA someBeanFromA) {
    return new PluginABeansForPluginsContainer(someBeanFromA);
  }
...
```

Example 4.13. PluginAConfiguration

PluginB

```
plugin.id=pluginB
plugin.version=0.1.0
plugin.configuration-class=com.acme.plugin_b.PluginBConfiguration
plugin.dependencies=pluginA
```

Example 4.14. PluginB plugin.properties

```
@Import(PluginABeansForPlugins.class)
@Configuration(proxyBeanMethods = false)
class PluginBConfiguration {
  ...
  @Bean
  SomeBeanForB someBeanForB(SomeBeanFromA someBeanFromA) {
    return new SomeBeanForB(someBeanFromA);
  }
  ...
}
```

Example 4.15. PluginBConfiguration

Add-Ons

Plugins can also extend other plugins. To this end plugins can define extension points for which they collect implementing beans from *add-on* plugins using the `AddOnManager`.

A plugin can have multiple dependencies but can only be an add-on for one other plugin. Add-ons can also inject beans from the extended plugin in the same way as if the add-on had a dependency on the extended plugin. The `AddOnManager` can be injected by importing the configuration class `com.coremedia.cms.common.plugins.plugin_framework.addons.AddOnConfiguration`.

Example

PluginB is an add-on for PluginA and provides an extension for PluginA's extension point.

PluginA

```
plugin.id=pluginA
plugin.version=1.2.3
plugin.configuration-class=com.acme.plugin_a.PluginAConfiguration
```

Example 4.16. PluginA plugin.properties

```
import com.coremedia.cms.common.plugins.plugin_base.ExtensionPoint;

@ExtensionPoint
public class SomeExtensionPointForA {}
```

Example 4.17. SomeExtensionPointForA

```
@Import(AddOnConfiguration.class)
@Configuration(proxyBeanMethods = false)
class PluginAConfiguration {
  @Bean
  UsageOfExtensionPoints usageOfExtensionPoints(AddOnManager addOnManager)
{
    return new
UsageOfExtensionPoints(addOnManager.getExtensions(SomeExtensionPointForA.class));

  }
}
```

Example 4.18. PluginAConfiguration

PluginB

```
plugin.id=pluginB
plugin.version=0.1.0
plugin.configuration-class=com.acme.plugin_b.PluginBConfiguration
plugin.add-on-for=pluginA
```

Example 4.19. PluginB plugin.properties

```
...
  @Bean
  SomeExtensionPointForAImpl someExtensionPointForAImpl() {
    return new SomeExtensionPointForAImpl();
  }
...
```

Example 4.20. PluginBConfiguration

```
public class SomeExtensionPointForAImpl extends SomeExtensionPointForA {}
```

Example 4.21. SomeExtensionPointForAImpl

4.1.6.2 Plugins for Studio Client

The word plugin is used in the Studio client context with various meanings. While `StudioPlugin` and `EditorPlugin` are a possible way to customize the Studio client (see Section 9.3, "Studio Plugins" in *Studio Developer Manual*) the plugins that are described here are merely a way to package your code, so that you are able to add customizations at deployment time similar to the Java applications.

A Studio client plugin is just a normal npm package utilizing the `jangaroo.npm` build tooling containing code which should be executed when the plugin is loaded at runtime. The only specialty of the package in contrast to normal code packages is, that you need to execute the script `package` using `pnpm run package` to package your code together into a ZIP file instead of adding a dependency to the corresponding app in the Studio client.

The created ZIP file contains a directory `packages` that contains the corresponding plugin package as a subfolder which can be added to a *Plugin Descriptor* (see Section 4.1.6.3, "Plugin Descriptors and Bundled Plugins" [101]).

Setting-up a Plugin

As a starting point for the development of a new plugin it is recommended to use the Jangaroo package `@jangaroo/create-project`. The package is

a so called starter kit, which can be utilized via `pnpm create`. By executing `pnpm create @jangaroo/project my-plugin` an interactive command line tool is started that leads through the steps necessary to create a new npm package containing the basic structure for a Studio client plugin. It also adds some convenience to run the local development state of your Studio client plugin. Make sure you confirm the steps that ask if `start` and `package` scripts should be added.

The resulting package does not yet contain a surrounding workspace. While a workspace is not necessary if you only have a single package it might become useful if you want to maintain multiple plugins. Please copy and adapt the `package.json` file and the `pnpm-workspace.yaml` file of the Studio client workspace for a basic setup. If you create the workspace before triggering the tool it will automatically detect it and ask if the newly created package should be added.

NOTE

The tool can also be triggered without any interactive elements by providing the required parameters via command line. Please check `pnpm create @jangaroo/project --help` for possible options.

Limitations of Plugins

Due to being added after the static part of the Studio Client has already been built, it has the same limitations as so called *dynamic packages*, which are also added after the app builds have already been executed.

- You can not adjust anything theming related inside SCSS files in the `sencha/sass` folder as the CSS for the theming has already been build.
- All dependencies added to the `dependencies` entry inside the plugin's `package.json` file are considered to be provided by the `app` the plugin is added to. This means that (for now) a plugin cannot bring its own dependencies and can only utilize dependencies that are already part of the corresponding `app`.
- While it is technically possible, it is not intended that a plugin utilize a Blueprint dependency. This would lead to a cyclic dependency between the Blueprint workspace and the corresponding plugin (see Section "Working with the Plugin Workspace" [100]).

Working with the Plugin Workspace

Like the Maven workspaces for the (Java) CoreMedia applications, plugins can be built independently of the Blueprint workspace. For example, it does not utilize any dependencies of the Blueprint workspace. This is important, so that your plugin can be updated and released independently of your Blueprint customizations and a concrete CoreMedia release.

Building the Studio plugin workspace

A Studio client package containing a plugin is built just like a normal Studio client package in the Blueprint workspace: You can compile the code with `pnpm run build`, resulting in JavaScript files and resources in the `dist` directory. The code can also be watched and linted as usual. What is different is, that by executing `pnpm run package`, a `zip` archive is created inside the `build` directory, containing all resources needed by the plugin. This file can be seen as a "binary" release of your Studio client plugin.

The same plugin can be added to multiple apps as long as the dependencies are satisfied by the corresponding app.

Local development

The package created by the starter kit also contains configuration to conveniently run the local development state of your plugin. You need a ready-to-run Studio app as described in Section 4.3, “Build and Run the Applications” [119].

NOTE

If the feature you develop as a plugin also plugs into Studio server (which is a common case), take care that this Studio client is connected to a Studio server including the corresponding plugin.

To activate your Studio client plugin, you have the following options:

- If you want to use the Studio client Docker image, you only need to mount your plugin's `dist` directory as a volume under `/coremedia/plugins/APP_NAME` where `APP_NAME` is the app the plugin should be added to (for example, `studio-client.main`). After running the Studio client Docker image, the corresponding app will pick up your plugin. If you used an invalid identifier for `APP_NAME` it will be logged as a warning during container startup.

  In order to add multiple plugins just put them into separate subfolders. The mechanism will just scan any nested directory structure below `/coremedia/plugins/APP_NAME` until a package is encountered.
- If you want to run a Studio client app from your Blueprint workspace via pnpm with your plugin, you must configure the path to your plugin's location for

`jangaroo run` (which is started via the `start` script). There is a Blueprint file `apps/studio-client/apps/main/app/jangaroo.config.js`, which contains a `jangarooConfig` structure with a property `additionalPackagesDirs`. The value of the property is a list. Add the absolute path of the studio-client plugin's directory (usually `studio-client/apps/main/<the-plugin-name>` in the plugin workspace) to this list.

- If you do want to start a Studio client app locally, you can also connect via HTTP(S) against a Studio client app that is already deployed on a webserver. In order to do so the package containing the plugin also has a `start` script utilizing `jangaroo run` that can be called with the parameter `proxyTargetUri` containing the URL to the deployed Studio Client app, for example:

```
pnpm run start --proxyTargetUri http://some-host/studio
```

NOTE

All parameters provided to Jangaroo commands like `jangaroo run` can alternatively also be put into environment variables or the `jangaroo.config.js` file (see Section 4.1, "Setting Up the Workspace and IDE" in *Studio Developer Manual* for details).

4.1.6.3 Plugin Descriptors and Bundled Plugins

This section describes how multiple application plugins are combined with a so called *plugin-descriptor*, and how plugins can be integrated or *bundled* with the CoreMedia Blueprint.

Plugin Descriptors

As a feature often requires the extension of multiple applications, a *plugin-descriptor* JSON file is used to bundle up plugins for different applications. If you write plugins only for a specific project, you might not need to create such a file. It simply lists all individual plugin urls by application, and must follow the schema at https://releases.coremedia.com/plugins/plugin-schema-3.2.0.json. If you want to provide plugins for others, bundle plugins with the Blueprint or want to use plugins provided by CoreMedia or partners, this file is intended to be the entry point to access those plugins.

Example

```
{
  "$schema": "https://releases.coremedia.com/plugins/plugin-schema-3.2.0.json",

  "plugins": {
    "studio-server": {
      "url":
"https://github.com/CoreMedia/content-hub-adapter-rss/releases/download/
v2.0.4/studio-server.content-hub-adapter-rss-2.0.4.zip"
    },
    "studio-client.main": {
      "url":
"https://github.com/CoreMedia/content-hub-adapter-rss/releases/download/
v2.0.4/studio-client.main.content-hub-adapter-rss-2.0.4.zip"
    }
  },
  "minimum-cms-version": "2110.1"
}
```

Example 4.22. content-hub-adapter-rss-2.0.4.json

Plugin Releases

Plugins are released as simple file attachments on GitHub Releases. A release of e.g. the RSS-Adapter (https://github.com/CoreMedia/content-hub-adapter-rss/releases/tag/v2.0.4) contains a plugin-descriptor json file and two plugin zip files, one for studio-server and one for studio-client.

Using Plugin Descriptors and Releases

There are two approaches to use plugin-descriptors with the Blueprint:

- During deployment
- During Blueprint build (so called *Bundled Plugins*)

These approaches are described in detail in the following sections.

NOTE

The default Blueprint already contains some bundled plugins. If you add or exchange plugins by either approach, make sure that finally each plugin is included at most once.

During Deployment: Download and Mount Plugins

Plugins are zip artifacts and the applications can load plugins from the file system on startup. This gives great flexibility as you can combine different sets of plugins with the already-built Docker images of your applications without re-compiling or re-building anything, but only restarting the affected applications.

To this end there is a Compose file `deployment/docker/compose/plugins.yml` configuring the applications to use plugins and a shell script `deployment/docker/download-plugins.sh` to download plugins from provided descriptor urls. For details how to use the script, please see the comment at the beginning of the script.

During Blueprint Build: Download and Include Plugins with Docker Images

It is also possible to include plugins directly with your Docker images. The process to do that is similar to the one of the Blueprint extensions, but more lightweight.

A plugin consists of one or more *application-plugins* for different applications of the CoreMedia system. The plugin-descriptor maps the application-plugin links to the applications.

Next to `workspace-configuration/extensions` is a `workspace-configuration/plugins` directory where you can configure the plugins you want to bundle. You can either provide plugin-descriptor files in the `plugin-descriptors` subdirectory directly, or reference them as URLs in the `plugin-descriptors.json` file.

To add the referenced application-plugins to the respective application workspaces, execute **mvn generate-resources** in the `workspace-configuration/plugins` directory. This will parse the plugin-descriptors and put the links of the application-plugins into `plugins.json` files in the corresponding application workspaces. The `plugins.json` files are then used in the build process of the distinct applications to download the application-plugins and add them to the Docker images.

In a particular project, you usually do not add or remove plugins very frequently, so this step is not part of the standard build turnaround. Therefore, it is recommended that you commit the changed `plugins.json` files to your version control system.

Workflow to add a Bundled Plugin

The recommended workflow to add a bundled plugin to your Blueprint is as follows:

1. Go to `workspace-configuration/plugins`.
2. Add the URL of the plugin-descriptor to the `plugin-descriptors.json` file.
3. Execute **mvn generate-resources**.
4. Commit the changes.
5. Build the applications.

Removing a Bundled Plugin

To remove a bundled plugin, simply remove the descriptor from the `plugin-descriptors.json` file or the `plugin-descriptors` directory and re-run **mvn generate-resources** in `workspace-configuration/plugins`.

4.2 Configuring the Workspace

Before you can start with development, you have to do some configurations, in part, depending on the Blueprint you want to work with.

- Enable or remove optional Blueprints and modules as described in Section 4.2.1, "Enabling Or Removing Optional Components" [105].
- Adapt your Maven settings to the required repositories as described in Section 3.1, "Prerequisites" [32].
- Adapt the workspace to your own project as described in Section 4.2.2, "Configuring the Workspace" [116]. Configure, for example, groupId, version, deployment repositories and CoreMedia licenses.
- If you want to use a local setup, then you have to do some database configuration and host mapping as described in Section 4.2.3, "Configuring Local Setup" [116].

4.2.1 Enabling Or Removing Optional Components

The *CoreMedia Content Cloud* workspace contains a complete CoreMedia system with all the core components and optional modules which have to be licensed separately. Some of these modules are disabled by default. See Section 2.1, "Components and Architecture" [20] for an overview of all components. Before you start with development, enable all modules that you need and remove all modules that you do not need.

Name	Description	Removal
CoreMedia Elastic Social	Module for the work with external users and user generated content, such as ratings or comments (disabled by default).	See Section 4.2.1.2, "Enabling the Elastic Social Extension" [113]
Advanced Asset Management	Module for the work with assets, such as images or documents.	See Section 4.2.1.5, "Removing the Advanced Asset Management Extensions" [115]
eCommerce Blueprint	Blueprint for the integration with an eCommerce system.	See Section 4.2.1.3, "Removing the eCommerce Blueprint" [114]

Name	Description	Removal
Brand Blueprint	Blueprint for a brand website with responsive templates.	See Section 4.2.1.4, "Removing the Brand Blueprint" [114]
Analytics Connectors	Module for the integration of connectors to thirdparty analytics providers.	See Section 4.2.1.6, "Removing the Analytics Connectors Extension" [115]

Table 4.2. Optional modules and blueprints

Section 4.2.1.1, "Extensions and Their Dependencies" [106] lists all extensions and their mutual dependencies.

As described in Section 4.1.5, "Project Extensions" [77] the *CoreMedia Blueprint* workspace provides an easy way to enable or disable existing extensions in one place. This chapter shows you how to disable and remove extensions from the *Blueprint*.

The command is always the same, only the list of extension names differs depending on the feature to remove. There are two different commands to deactivate an extension:

disable — Removes the dependencies from applications to any extension modules, but keeps the modules in the Maven aggregator(s) in a profile named `inactive-extensions`.

remove — Removes the dependencies from applications to any extension modules, but keeps the modules in the Maven aggregator(s) in a profile named `inactive-extensions`.

All commands in the following sections use `disable`, which can be replaced by `remove` according to your needs. All `mvn` commands have to be executed in directory `workspace-configuration/extensions`.

4.2.1.1 Extensions and Their Dependencies

This section sums up the existing extensions in *CoreMedia Blueprint* and shows their mutual dependencies, and their dependencies with licensed product add-ons. This information is required when removing extensions completely or when you want to know the licences required for some extensions.

To obtain a list of all extensions in the *CoreMedia Blueprint* workspace, invoke the extensions tool with the following command:

```
mvn -f workspace-configuration/extensions extensions:list -q -DshowDependencies
                                    -Dverbose
```

More information on how to use the extensions tool can be found in the `README.md` in `workspace-configuration/extensions`.

Dependencies between extensions and licences

alx	
Description	General Analytics Integration
Required by Extension	`alx-google`
Required licences	No add-on licence required

alx-google	
Description	Specific Analytics Integration for Google Analytics. This extension can be enabled or disabled independently.
Required by extension	None
Required licences	None

am	
Description	*CoreMedia Asset Management* allows you to store digital assets (for example high-resolution pictures of products) in the content repository.
Required by extension	None
Required licences	Advanced Asset Management

catalog	
Description	Internal catalog.
Required by extension	`corporate`
Required licences	None

corporate	
Description	Extension with the features for the Brand Blueprint.
Required by extension	None
Required licences	None

create-from-template	
Description	Create a Page in Studio with predefined content.
Required by extension	None
Required licences	None

custom-topic-pages	
Description	Create custom topic pages in Studio.
Required by extension	None
Required licences	None

ecommerce-adapter	
Description	eCommerce Adapter

Required by extension	None
Required licences	Commerce Hub
ecommerce-ibm	
Description	*HCL Commerce* specific demo content
Required by extension	None
Required licences	Commerce Hub, Connector for HCL Commerce
ec-augmentation	
Description	eCommerce augmentation extension for headless-server
Required by extension	none
Required licences	Commerce Hub
hybris	
Description	SAP Hybris specific demo content
Required by extension	None
Required licences	Commerce Hub, Connector for SAP Commerce Cloud
ecommerce-sfcc	
Description	Salesforce Commerce Cloud specific demo content
Required by extension	None

Required licences	Commerce Hub, Connector for Salesforce Commerce Cloud
ecommerce-commercetools	
Description	commercetools specific demo content
Required by extension	None
Required licences	Commerce Hub, Connector for commercetools
es	
Description	CoreMedia Elastic Social Integration
Required by extension	`lc-es`
Required licences	Elastic Social
feedback-hub	
Description	Feedback Hub
Required by extension	None
Required licences	Experience Feedback Hub
lc	
Description	Generic eCommerce Extension
Required by extension	`ecommerce-ibm`, `hybris`, `lc-es`, `lc-asset`, `ec-augmentation`
Required licences	Commerce Hub

lc-asset	
Description	Feature allows you to manage images and image variants (or crops) for categories, products and products variants (products for short) in the CoreMedia system. These extensions depend on *CoreMedia Commerce*.
Required by extension	`ec-augmentation`
Required licences	Advanced Asset Management, Commerce Hub

lc-es	
Description	Elastic Social features for eCommerce
Required by extension	None
Required licences	Elastic Social, Commerce Hub

notification-elastic	
Description	*Notifications* Note, that despite suggested by the name, this extension is not necessarily bound to Elastic Core as a persistence layer. You may also use an SQL database.
Required by extension	None
Required licences	None

osm	
Description	OpenStreetMap Integration
Required by extension	None
Required licences	None

sfmc	
Description	Salesforce Marketing Cloud basics
Required by extension	None
Required licences	Marketing Automation Hub, Connector for Salesforce Marketing Cloud
taxonomy	
Description	Taxonomy
Required by extension	taxonomies required by am, and custom-topic-pages
Required licences	No add-on licence required

Table 4.3. Blueprint Extensions and Dependencies

Add-on licences and their corresponding extensions

The following Table 4.4, " Add-ons and the dependent extensions " [112] shows the CoreMedia add-on licences, and the extensions that need this license.

> **NOTE**
>
> This section only describes the technical dependencies between add-ons and extensions. If you can use a CoreMedia system in a specific configuration, depends on your contract.

Add-On	Extensions
Analytics Connectors	`alx`, `alx-google`,
Elastic Social	`es`, `lc-es`
Content Hub	Up to three Content Hub adapters, including the default ones, are already included in the product. Content Hub adapters use extension `content-hub-default`. Default

Add-On	Extensions
	adapters are packages in extension `content-hub-default-adapters`.
Experience Feedback Hub	`feedback-hub`
Commerce Hub	A general commerce hub license is required for any of CoreMedia's vendor-specific commerce adapters as well as to implement a custom commerce adapter, based on extension ecommerce-adapter. The license for a vendor-specific commerce adapter allows using the corresponding extension:
Connector for HCL Commerce	`ecommerce-ibm` (starting with CMCC 2004: demo content only, adapter ships separately)
Connector for SAP Commerce Cloud	`hybris` (demo content only, adapter ships separately)
Connector for Salesforce Commerce Cloud	`ecommerce-sfcc` (demo content only, adapter ships separately)
Connector for commercetools	`ecommerce-commercetools` (demo content only, adapter ships separately)
Marketing Automation Hub	Not yet sold separately / no corresponding extension yet.
Connector for Salesforce Marketing Cloud	`sfmc`

Table 4.4. Add-ons and the dependent extensions

4.2.1.2 Enabling the Elastic Social Extension

This section describes the required steps to enable the *CoreMedia Elastic Social* extension in the *Blueprint*.

NOTE

Enabling the Elastic Social Integration is optional and its usage requires a separate license.

1. Enable the listed extensions. The `lc-es` of course only makes sense if you are using the optional eCommerce Blueprint.

```
mvn extensions:sync -Denable=es,lc-es
```

Example 4.23. Enable CoreMedia Elastic Social Extension

2. Although the Elastic Social extensions are disabled by default, the frontend workspace which isn't separated into extensions still provides the frontend brick `elastic-social` and uses it in themes, so they are already prepared if you enable Elastic Social.

4.2.1.3 Removing the eCommerce Blueprint

This section describes the required steps to remove the *CoreMedia eCommerce* Blueprint from the *CoreMedia Blueprint* workspace depending on your eCommerce Connector.

1. Remove the listed extensions from the managed extensions.

```
mvn extensions:sync
-Ddisable=lc,lc-asset,lc-es,ecommerce-ibm,hybris,ecommerce-sfcc,
ecommerce-adapter,ec-augmentation,campaigns
```

Example 4.24. Remove CoreMedia eCommerce Extension

2. Delete the following themes: `aurora-theme`, `calista-theme`, `hybris-theme`, `sfra-theme`, `sitegenesis-theme` in the frontend module (`themes`).

4.2.1.4 Removing the Brand Blueprint

This section describes the required steps to remove the *CoreMedia Brand* Blueprint from the *CoreMedia Blueprint* workspace.

1. Remove the listed extensions from the managed extensions.

```
mvn extensions:sync -Ddisable=corporate
```

Example 4.25. Remove CoreMedia Corporate Extension

2. Delete the `corporate-theme` in the Frontend Workspace (`themes`).

4.2.1.5 Removing the Advanced Asset Management Extensions

This section describes the required steps to remove *Advanced Product Asset Management* from *CoreMedia Blueprint*. *Advanced Asset Management* consists of two extensions.

1. Remove the listed extensions from the managed extensions.

```
mvn extensions:sync -Ddisable=lc-asset,am
```

Example 4.26. Remove CoreMedia Product Asset Management Extension

2. If you use the CoreMedia example content, you also have to remove the links to `Asset` content in the `CMPicture` files. You can use a tool like sed.
3. Disable or remove the frontend brick `download-portal` and make sure, it is not used in any existing theme.

4.2.1.6 Removing the Analytics Connectors Extension

This section describes the required steps to remove the *CoreMedia Analytics Connectors* extension from *Blueprint*.

- Remove the listed extensions from the managed extensions.

```
mvn extensions:sync -Ddisable=alx,alx-google
```

Example 4.27. Remove CoreMedia Analytics Connectors Extension

4.2.2 Configuring the Workspace

The *Blueprint* workspace comes ready to use. However, there are some environment specific configurations to be adjusted at the very beginning of a project. You may skip these steps only if you are just going to explore the workspace, you will neither share your work with others nor release it, and you will start over from scratch again with your actual project.

Changing the group IDs and versions

The groupId of the *CoreMedia Blueprint* workspace is `com.coremedia.blueprint`. While this works from the technical point of view, you have to change it to a project specific groupId, because CoreMedia reserves the possibility to provide versioned artifacts of this groupId.

Since the groupId is needed to denote the parent POM file, it cannot be inherited but occurs in every `pom.xml` file. CoreMedia provides a script for this task. It does not only change the group ID in the POM files but also in the deployment environment when necessary. Execute the following command in the workspace:

```
./workspace-configuration/scripts/set-blueprint-groupId.sh <YourGroupId>
```

To keep the ModularOneRepoEnforcerRule functional after changing the groupId, you should adjust its configuration like this:

```
<modularOneRepoEnforcerRule>
  <blueprintGroupId><YourGroupId></blueprintGroupId>
</modularOneRepoEnforcerRule>
```

See Section 4.1.4, “Structure of the Workspace” [67] for more information on the ModularOneRepoEnforcerRule.

You can use the script `set-blueprint-version.sh` to also change the versions.

4.2.3 Configuring Local Setup

Relational Database Setup

You need to create different databases and users used by the various *CoreMedia CMS* components (see Section 3.1, “Prerequisites” [32]). In the `workspace-configuration/database` folder of the *Blueprint* workspace you will find

SQL scripts for creating and dropping all database entities needed for the relational database.

The scripts are suitable for a local MySQL instance in a developer environment. You can easily adapt them for other databases or remote users. There are also Bash scripts and Windows batch files to apply the SQL scripts. If the MySQL server is running and the *mysql* command line client is executable via the `PATH` variable, you only need to execute the following in order to prepare the databases for *CoreMedia Content Cloud*.

Windows:

```
> cd $CM_BLUEPRINT_HOME\workspace-configuration\database\mysql\
createDB.bat
```

Linux:

```
$ cd $CM_BLUEPRINT_HOME/workspace-configuration/database/mysql
./createDB.sh
```

The command was successful if the following databases have been created:

Database	User	Password	Description
cm_management	cm_management	cm_management	Database for the *Content Management Server*
cm_master	cm_master	cm_master	Database for the *Master Live Server*
cm_replication	cm_replication	cm_replication	Database for the *Replication Live Server*
cm_mcaefeeder	cm_mcaefeeder	cm_mcaefeeder	Database for the *CAE Feeder* connected to the *Content Management Server*
cm_caefeeder	cm_caefeeder	cm_caefeeder	Database for the *CAE Feeder* connected to the *Master Live Server*
cm_editorial_comments	cm_editorial_comments	cm_editorial_comments	Database for the *Studio Server* that runs the feature *Editorial Comments*
cm_notifications	cm_notifications	cm_notifications	Database for the *Notifications* based on SQL persistence layer. For details see Notifications SQL Persistence Configuration for CoreMedia Studio in *Deployment Manual*.

Database	User	Password	Description
cm_projects	cm_projects	cm_projects	Database for the *Projects/To-Dos* based on SQL persistence layer. For details see Section 3.4.12, "Projects/To-Dos SQL Persistence Configuration" in *Deployment Manual*.
cm_caplist	cm_caplist	cm_caplist	Database for the CapLists based on SQL persistence layer. Relevant for the features *My Edited Content* and *Workflow Lists*. For details see Section 3.8, "My Edited Content and Workflow Lists Properties" in *Deployment Manual*.

Table 4.5. Database Settings

4.3 Build and Run the Applications

The Blueprint workspace provides Maven modules to build Spring Boot applications for Blueprint applications.

Prerequisites

Before you can run the applications, you have to build the CoreMedia Blueprint Workspace in advance:

```
mvn clean install -Pdefault-image
cd apps/studio-client
pnpm install
pnpm -r run build
cd ../..
```

See Section 3.2.1, “Building the Workspace” [38] for more details.

Workspace Structure

- `apps/<app-name>/spring-boot` - below this folder there is a Maven module for each service application. Each of these modules will build a single Spring Boot application packaged as a JAR file.
- `shared/common/spring-boot/blueprint-spring-boot-autoconfigure`, this module encapsulates common configuration aspects for all Spring Boot service modules.

See Section 4.1.4, “Structure of the Workspace” [67] for more details.

Application Structure

Each Spring Boot application module is structured the same way:

- A source folder containing at least the application starter class. It could also contain other classes implementing Spring configuration classes.
- A resources folder containing the properties files and the logging configuration file.

Spring Configuration

Each application can define properties in multiple Spring Boot profiles:

- The **default** profiles with properties defined in `application.properties`.

- The **development** profiles defined in `application-dev*.properties`.
- The **local** profiles with properties defined in `application-.*-local.properties`. These properties can contain paths only available on a local workstation. The default local profile is named `local` and should be used when starting the application using either IDEA or Maven.
- **Commerce specific** settings:
  - `application-dev-wcs.properties`
- **Development profiles** activating development features for local and CI environments.
  - The default development profile is named `dev` and should be used when starting the application using either IDEA or Maven.

Logging

Logging is configured using the standard Spring-Boot logging properties. If you want to modify the logback configuration, you need to place a `logback-spring.xml` inside your classpath root directory i.e. `src/main/resources` in the application modules.

By default, the logging is configured with different logging patterns depending on the output and the active Spring Boot profile:

- No active profile: Only console logging without timestamps or coloring. Timestamps are added by all container runtimes. If file logging is activated by setting the `logging.file.name` property, the log file will contain timestamps but no coloring.
- `dev` profiles: File and console logging are active without coloring. Only file logging will contain a timestamp. The log file is created at `/coremedia/log/application.log` in the container file system. For excessive logging, the directory should be mounted to a container volume.
- `local` profiles: File and console logging are active with timestamps. The console logging will contain coloring. The log file is created in the `project.build.directory` of the module. The name of the application or an abbreviation will be used as the filename to differentiate two disctinct applications, started with different local profiles.

To gather logs from the command-line using `docker` or `kubectl`, please use the `--timestamp` or `-t` flag. This is especially important, if you collect logs to be included in a support request:

Docker:

```
docker logs --timestamp <container>
```

Kubernetes:

```
kubectl logs --timestamp <pod>
```

For Kubernetes there is also a timestamp flag for the `kubectl` command-line utility:

Internal Communication Protocol

CORBA is the default protocol for communication between components of the CoreMedia system. To choose gRPC as the communication protocol for a component, Java system property `repository.use-grpc` or, respectively, environment variable `REPOSITORY_USEGRPC` must be set to `true`. See Section 4.3.1, "Starting Applications using IntelliJ IDEA" [122] and Section 4.3.2, "Starting Applications using the Command Line" [123] on how to set the protocol.

Developing with CoreMedia Blueprint applications

Currently there are two different approaches to start the Spring Boot apps you want to develop locally:

- Using the spring-boot-maven-plugin
- Using IntelliJ IDEA run configurations together with the Run Dashboard

The apps you don't want to alter, can be provided using the local Docker development deployment.

Application Configuration Facade

To configure the locally started applications, Spring Boot profiles are being used:

- `dev` - this profile activates development features like actuators, monitoring etc. and should not add filesystem or localhost features. This profile will be active by default if you include the `development.yml` in the docker compose setup included with this workspace.
- `local` - this profile configures local paths within the workspace like paths to licenses or source folders of other modules. This profile should only be activated for locally started apps using Maven or IDEA but not the docker compose setup.

To configure which application should be used from a remote system, there is a list of convenience host properties, forming a simple configuration facade. The

intent of these properties is to use them only in private Spring profiles, on the command-line or in IDEA run configurations when you are developing locally. Do not use these properties outside of the `application-local.properties` files except for the cases mentioned above.

The main property is `installation.host` which when set implicates all other services / endpoints are running remotely. By default, all other convenience host properties derive their default from `installation.host`. If you start more than one service locally, lets say studio-server and preview, then you need to tell studio-server to use the locally started preview instead of a remote one and you have to set `cae-preview.host` to `localhost`.

```
- installation.host
    |- db.host
    |- mongodb.host
    |- solr.host
    |- content-management-server.host
    |- master-live-server.host
    |- workflow-server.host
    |- cae-preview.host
    `- cae-live.host
```

4.3.1 Starting Applications using IntelliJ IDEA

Since IDEA 2018.2 Spring Boot applications are natively supported in the *Run* dashboard.

CoreMedia delivers predefined run configurations for the apps with all required settings. In order to use these configurations, do as follows:

1. Copy the predefined files from `apps/<app-name>/spring-boot/ideaRunConfigurations` into the `.idea/runConfigurations` folder in your Blueprint workspace. You can use the script `spring-boot/copy-run-configurations.sh` to copy all configurations at once.

   If you copy the configuration files manually, you might have to create the `runConfigurations` folder.
2. Close and open IDEA, so that it finds the new run configurations.

Now, you can edit the run configurations in IDEA by setting the `installation.host` or any of the other convenience properties. To activate gRPC communication, set system property `repository.use-grpc=true` or set environment variable `REPOSITORY_USEGRPC=true`.

Services/Run Dashboard not Visible

In order to see the run configurations in IDEA's *Services* dashboard (in version 18 called *Run Dashboard*) you might have to do some configuration. Open the *Run* menu and select *Edit Configurations*. Select the *Templates* folder, and add *Spring Boot* to the *Configurations available in Services* field in the main field of the window.

4.3.2 Starting Applications using the Command Line

As an alternative to the IDEA integration, you can start most of the applications using the spring-boot-maven-plugin.

Exception: The Studio Client must be started via pnpm.

Using CoreMedia's configuration facade makes it very simple to use remote services when developing a single app. Simply run the application via Spring Boot Maven plugin, like:

```
mvn spring-boot:run \
-Dspring-boot.run.jvmArguments="-Dinstallation.host=<FQDN> ..." \
-Dspring-boot.run.profiles=<profiles>
```

The Spring profiles are most likely to be `dev,local,private` (in that particular order) but can vary depending on the spring profiles defined for the application. If more than one app is started locally, simply add the required convenience host properties to the apps' private Spring profile or the environment variables.

To activate gRPC on the started application, export the corresponding environment variable `REPOSITORY_USEGRPC` set to `true` or precede the command with a local variable definition:

```
REPOSITORY_USEGRPC=true mvn spring-boot:run \
-Dspring-boot.run.jvmArguments="-Dinstallation.host=<FQDN> ..." \
-Dspring-boot.run.profiles=<profiles>
```

NOTE

Maven's `mvnDebug` command does not work with current Spring Boot versions. Instead, you can enable debugging by adding the debug configuration like `-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=*:8000` to the JVM arguments.

4.3.2.1 Starting the Studio Client

To start the Studio Client, use `pnpm` to run the `start` script.

You have two possibilities to connect your Studio Client with a Studio server:

1. **Connect Remote Studio Server**
2. **Connect Local Studio Server**

Working Directory:

```
apps/studio-client/global/studio
```

Connect Remote Studio Server

Start the Studio Client and connect against a remote Studio running at `<URL>` via

```
pnpm run start --proxyTargetUri <URL>
```

With this command line call, only Rest requests are proxied to/from the remote Studio Server. No remote static Studio Client resources are proxied, that is, all Studio Client resources are served locally.

Connecting Local Studio Server

First: Start Studio Server locally.

Then just start the Studio Client via

```
pnpm run start
```

With this command line call, the Rest requests are proxied to/from the locally started Studio Server. Again, no remote static Studio Client resources are proxied, that is, all Studio Client resources are served locally.

4.3.2.2 Starting the Studio Server

Working Directory:

```
apps/studio-server/spring-boot/studio-server-app
```

Start the Studio Server locally via Spring Boot Maven plugin, as explained in Section 4.3.2, “Starting Applications using the Command Line” [123].

Connecting with Local CAE

When you want to connect Studio Server with a local CAE instance, start Studio Server as above, but add `-Dcae-preview.host=localhost` and/or `-Dcae-live.host=localhost` to the JVM arguments.

Links

- Actuators

4.3.2.3 Starting the CAE Preview App

Working Directory:

```
apps/cae/spring-boot/cae-preview-app
```

Start the CAE Preview App locally using the Spring Boot Maven plugin, as explained in Section 4.3.2, “Starting Applications using the Command Line” [123].

Links

- Actuators
- CAE Preview
- Log File

4.3.2.4 Starting the CAE Live App

Working Directory:

```
apps/cae/spring-boot/cae-live-app
```

Start the CAE Live App locally using the Spring Boot Maven plugin, as explained in Section 4.3.2, “Starting Applications using the Command Line” [123].

Links

- Actuators
- CAE Live
- Log File

4.3.3 Local Docker Test System

For the setup of a local Docker test system see Section 3.2.2, “Docker Compose Setup” [44].

4.4 Development

This chapter describes how you can customize your CoreMedia system in the *CoreMedia Blueprint* workspace. However, it does not describe how you, for example, write a *Studio* plugin or a *CAE* template; this is explained in the component's specific manual. Instead, it describes how you can use the workspace mechanisms to include your extensions and where you can add your own code or configuration.

- Section 4.2.1, "Enabling Or Removing Optional Components" [105] describes how you can add and remove extensions using the *CoreMedia Project Maven Build Extension*. The extensions mechanism is explained in detail in Section 4.1.5, "Project Extensions" [77].
- Section 4.4.2, "Extending Content Types" [133] describes how you can add your own content types. You will find more details on content types in the Content Server Manual.
- Section 4.4.3, "Developing with Studio" [135] describes how you can add Studio modules to the list of studio plugins.
- Section 4.4.4, "Developing with the CAE" [138] describes how you can add extensions to the CAEs.
- Section 4.4.7, "Handling Personal Data" [141] describes how you can document and check personal data usage in Java code.

4.4.1 Using Blueprint Base Modules

This section describes how the *Blueprint Base Modules* are integrated into *CoreMedia Blueprint* and how a developer might customize and configure all the various modules or even replace certain modules completely.

NOTE

CoreMedia Blueprint uses *Blueprint Base Modules* as binary Maven dependencies but CoreMedia provides access to the source code via Maven source code artifacts. IDE's like Jetbrains IntelliJ Idea are able to download those sources automatically for a certain class by evaluating its correspondent Maven POM file.

4.4.1.1 Content Type Model Dependencies

As its name implies, the *Blueprint Base Modules* contain Blueprint logic and thus depend on the Blueprint's content type model. The content type model is still part of the Blueprint workspace, hence you may customize it. Be aware, that changes might affect or even break the *Blueprint Base Modules*. The following table shows an overview of the content types which are relevant for the *Blueprint Base Modules*. Details are explained in the sections about the particular modules.

Content Type (Properties)	Module
`CMLinkable` (`localSettings, linkedSettings`)	Settings
`CMTeaser` (target)	Settings
`CMNavigation`	Settings
`CMSettings`	Settings

Table 4.6. Content type model dependencies

4.4.1.2 The Settings Service

Settings are a flexible way to enable editors to configure application behavior via content changes within *CoreMedia Studio* without the need to redeploy a web application. *CoreMedia Blueprint* uses the `com.coremedia.blueprint.base.settings.SettingsService` to read certain settings from various different sources. This section describes how you can use the settings service in your own projects.

NOTE

Read Section 5.4.3, “Settings” [175] for a description of why you want to use settings and how to do it from an editors perspective.

The setting* Methods

```
public interface SettingsService {
  <T> T setting(
          String name,
          Class<T> expectedType,
          Object... beans);

  <T> T settingWithDefault([...]);

  <T> List<T> settingAsList([...]);

  <K, V> Map<K, V> settingAsMap([...]);

  [...]
}
```

All `setting*` methods are actually just variants of the basic `setting` method. Some provide additional convenience like `settingWithDefault`, others have complex return types which cannot be expressed as a simple type parameter, for example `settingAsList`. All `setting*` methods have some common parameters which are described in Table 4.7, "Parameters of the settings* methods" [129]. For detailed descriptions of the `setting*` methods please consult the API documentation of the SettingsService.

Parameter	Description
name	The name (or key) of the setting to fetch.
expectedType	The type of the returned object. This parameter allows for type safety and prevents you from unchecked casts of the result. For the `settingAsList` method, the *expectedType* parameter determines the type of the list entries, not the list itself. `settingAsMap` has separate type arguments for keys and values of the result map.
beans	Settings are always fetched for one or multiple targets, which are passed by the *beans* `vararg` parameter. In the *Blueprint*'s default configuration the `SettingsService` supports content objects, content beans, pages, sites and some other kinds of beans.

Table 4.7. Parameters of the settings methods*

Configuring the Default Settings Service via SettingsFinders

The *Blueprint Base Modules* not only defines the interface of how to evaluate settings but also provides an implementation and a Spring bean.

```
<beans>
  <bean id="settingsService"
        class="c.c.b.base.settings.impl.SettingsServiceImpl">
    <property name="settingsFinders" ref="settingsFinders"/>
  </bean>

  <util:map id="settingsFinders">
  </util:map>
</beans>
```

The plain `SettingsService` has no lookup logic for settings at all, but it must be configured with `SettingsFinders`. A `SettingsFinder` implements a strategy how to determine settings of a particular type of bean. *CoreMedia Content Cloud* provides some preconfigured `SettingsFinder`s for popular beans like content objects. It can be modified and enhanced with custom `SettingsFinders` for arbitrary bean types. As you can see, the default settings service only needs one property, which is a map named `settingsFinders`. The keys of that map must be fully qualified Java class names and its values are references to concrete `SettingsFinder` beans.

```
<util:map id="settingsFinders">
  <entry key="com.coremedia.cap.content.Content"
         value-ref="cmlinkableSettingsFinder"/>
  <entry key="com.coremedia.cap.multisite.Site"
         value-ref="siteSettingsFinder"/>
</util:map>

<bean id="cmlinkableSettingsFinder"
      class="c.c.b.base.settings.CMLinkableSettingsFinder">
  <property name="cache" ref="cache"/>
  <property name="hierarchy" ref="navigationTreeRelation"/>
</bean>

<bean id="siteSettingsFinder"
      class="c.c.b.base.settings.SiteSettingsFinder"/>
```

Example 4.28. The Spring Bean Definition for the Map of Settings Finder

The example above shows a map with two settings finders. One is supposed to be used for target beans of type `com.coremedia.cap.content.Content` and the other for targets of type `com.coremedia.cap.multisite.Site`.

In order to determine the appropriate settings finder for a given target bean the settings service calculates the most specific classes among the keys of the settings finders map which match the target bean. For the above example this is trivial, since `Content` and `Site` are disjointed. The lookup gets more interesting with content beans which usually constitute a deeply nested class structure. Assume, you configured settings finders for `CMLinkable` and `CMTaxonomy`. If you invoke the settings service with a `CMTaxonomyImpl` bean, only the settings finder for `CMTaxonomy` is effective. There is no automatic fallback to the `CMLinkable` finder. If you need such a fallback, let your special settings finder extend the intended fallback finder and call its `setting` method explicitly.

The easiest way to provide a custom way of fetching settings for certain content items or even for objects that do not represent a CoreMedia content item, is, to add a corresponding settings finder, that does the trick. Therefore, you should use CoreMedia's Spring bean customizer, that you can use anywhere within your Spring application context as follows:

```
          @Customize(value = "settingsFinders")
@Bean
public Map<Class<?>, SettingsFinder> mySettingsFinders(example.org.MyClass
mySettingsFinder) {
  return Map.of(example.org.MyClass.class, mySettingsFinder);
}
```

Example 4.29. Adding Custom Settings Finder

Via Spring you can configure one settings finder per class. This is a tradeoff between flexibility and simplicity which is sufficient for most use cases. However, on the Java level the `SettingsServiceImpl` provides the method `addSettingsFinder(Class<?>, SettingsFinder)` which allows you to add multiple settings finders for a class.

Typed Settings Interfaces

The `SettingsService` is a powerful multi-purpose tool. However, genericity always comes at the price of abstraction. Assume, there is some business logic which is based on a domain specific interface `Address`:

```
public interface Address {
  String getName();
  String getCity();
}
```

```
public class Messages {
  public static String getHelloMessage(Address address) {
    return "Hello " +
           address.getName() +
           ", are you living in " +
           address.getCity() + "?";
  }
}
```

Example 4.30. Business Logic API

If your actual address data is provided by the `SettingsService`, it must be adapted to the address interface.

```
class SettingsBackedAddress implements Address {
  // [...] constructor and fields for service and provider bean
  public String getName() {
    return settingsService.setting("name", String.class, bean);
  }
  public String getCity() {
    return settingsService.setting("city", String.class, bean);
  }
}
```

Example 4.31. Settings Address Adapter

Cumbersome, isn't it? Especially, if the interfaces are larger or not yet final. Fortunately, you don't need to implement such interfaces manually, but `SettingsService.createProxy` does the job for you:

```
class MyCode {
  private SettingsService settingsService;

  void doSomething(BusinessBean beanWithSettings) {
    Address address = settingsService.createProxy(Address.class,
beanWithSettings);
    String message = Messages.getHelloMessage(address);
  }
}
```

Example 4.32. Address Proxy

Internally the default settings service intercepts the call to `getName()` and `getCity()`. The operation `getCity()` is translated to `settingsService.setting("city", String.class, bean)`. Note: The property name "city" will be derived from the operation `getCity()` in the interface. Be aware of this dependency when choosing names for your settings properties and for the operations of your business objects if you want to use the proxy mechanism.

Content types Requirements

The `SettingsService` itself does not depend on particular content types, since the actual lookup strategies are implemented in `SettingsFinders`. CoreMedia Blueprint provides (among others) the `LocalAndLinkedSettingsFinder`, which fetches settings from the `localSettings` and `linkedSettings` properties of `Content` objects. It is a naming convention that originates from the `CMLinkable` content type, but applies also to other content types, for example `CMSite`.

CoreMedia recommends to yield the `localSettings` and `linkedSettings` properties exclusively to the `SettingsService`. If you need struct data which is not to be handled by the `SettingsService`, do not put it into `localSettings` and `linkedSettings`, but add new struct properties to the content type model.

CMNavigation content items inherit their settings along the hierarchy up to the root navigation. `CMTeaser` content items inherit the settings of their targets. If you rename these content types, this functionality gets lost.

4.4.2 Extending Content Types

Developing a new software almost always starts by analyzing the domain model. This is not different for *CoreMedia CMS*. Here the domain model is the source for modeling the content type model. The content type model is the backbone of *CoreMedia CMS* as it describes what content means to you. Read Chapter 4, *Developing a Content Type Model* in *Content Server Manual* for details on the content types.

Basically, there are two places within the *Blueprint* workspace you may use if you define your own content type model or extend the *Blueprint's* one. You will learn both of them by defining a new content type `CMHelloWorld` as a child of `CMTeaser` within a new file `mydoctypes.xml` as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<DocumentTypeModel
  xmlns="http://www.coremedia.com/2008/documenttypes"
  Name="my-doctypes">

 <ImportGrammar Name="coremedia-richtext-1.0"/>
 <ImportDocType Name="CMTeaser"/>

 <DocType Name="CMHelloWorld" Parent="CMTeaser">
   <StringProperty Name="message" Length="32"/>
```

```
 </DocType>
</DocumentTypeModel>
```

Defining content types in contentserver-blueprint-component

The first and a little easier way of defining `CMHelloWorld` is to put the new file `mydoctypes.xml` shown above into the directory `apps/content-server/modules/server/contentserver-blueprint-component/src/main/resources/framework/doctypes/my/`. It is good style to create a subfolder under `doctypes` for your customization, here named "my".

After doing so, you can test your new content type. To do so, you have to build the `contentserver-blueprint-component` module and the `content-server-app` module as follows. Remember to stop the server if you have not already.

```
$ cd apps/content-server/modules/server/contentserver-blueprint-component
$ mvn clean install
$ cd apps/content-server/spring-boot/content-server-app
$ mvn clean install
```

Now, start the *Content Management Server* application and take a look into its log file. You should see the following message, telling you that the *Content Server* created a new database table for the new content type.

```
[INFO]  SQLStore - DocumentTypeRegistry: creating table:
CREATE TABLE CMHelloWorld( id_ INT NOT NULL, version_ INT NOT NULL,
isApproved_ TINYINT, isPublished_ TINYINT, editorId_ INT,
approverId_ INT, publisherId_ INT, editionDate_ DATETIME,
approvalDate_ DATETIME, publicationDate_ DATETIME,
"locale" VARCHAR(32), "masterVersion" INT, "keywords" VARCHAR(1024),
"validFrom" DATETIME, "validFrom_tz" VARCHAR(30), "validTo" DATETIME,
"validTo_tz" VARCHAR(30), "segment" VARCHAR(64), "title" VARCHAR(512),
"teaserTitle" VARCHAR(512), "notSearchable" INT, "message" VARCHAR(32),
PRIMARY KEY (id_, version_),  FOREIGN KEY (id_) REFERENCES Resources(id_))
```

Using a Separate Module in the Context of an Extension

The second possibility is the more flexible way. You build your own module in the context of an extension. The following steps assume that an extension module `my-extension` already exists and requires a new content type. Proceed as follows:

1. Create a new subfolder `my-extension-server` in the `apps/content-server/modules/extensions/my-extension` directory.
2. Create a `pom.xml` file and add the following contents.:

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

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.coremedia.blueprint</groupId>
    <artifactId>my-extension</artifactId>
    <version>${project.version}</version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <artifactId>my-extension-server</artifactId>

  <properties>
    <coremedia.project.extension.for>
      server
    </coremedia.project.extension.for>
  </properties>

</project>
```

3. Adjust the `groupId` and `artifactId` of the parent declaration according to your project settings.
4. Add this module's Maven coordinates to the Extension Descriptor of the extension.
5. Create the subfolder `src/main/resources/framework/doctypes/myextension`.
6. Copy the content type definition file from above into the folder created in the last step.
7. Refer to Section 4.2.1, "Enabling Or Removing Optional Components" [105] to enable the extension.

Defining content types in Plugins

It is also possible to add content type definitions with Plugins. To this end you can add Plugins to the Content Server that contain doctype definitions xml files matching this pattern: `framework/doctypes/**/*.xml`. Note that you cannot provide a custom grammar in a Plugin. For details please refer to Section 4.2.4, "Content Types from Plugins" in *Content Server Manual*. For more information about Plugins, see Section 4.1.6, "Application Plugins" [82].

4.4.3 Developing with Studio

New *CoreMedia Studio Client* packages can be added to the project using the Blueprint extensions mechanism or by adding them as direct dependencies of the `@coremedia-blueprint/studio-client.main.base-app` or `@coremedia-blueprint/studio-client.main.app` package. Using

the extension mechanism is the preferred way. But as it is based on the same steps of adding a package directly, this is firstly covered and the additional required steps for adding *Studio Client packages* as extensions are described at the end of the section.

pnpm Configuration

Create a jangaroo project in `apps/studio-client/apps/main/extensions` in a directory named after your extension, e.g. for the extension "sample", by triggering the starter kit and following the steps:

```
pnpm create @jangaroo/project apps/studio-client/apps/main/extensions/sample
```

The starter kit will also offer to add the newly created package to the studio-client workspace. Confirm this option.

Make sure to also run `pnpm install` from the workspace root after the package has been added.

Source Files and Folders

A Studio plugin package contains at least two files: the plugin descriptor file located in the package's root folder (`jangaroo.config.js`) and the initializing plugin class (`src/SampleStudioPlugin.ts`).

The plugin class only implements the `init` method of the `EditorPlugin` interface:

```
import EditorPlugin from
"@coremedia/studio-client.main.editor-components/sdk/EditorPlugin";
import IEditorContext from
"@coremedia/studio-client.main.editor-components/sdk/IEditorContext";

class SampleStudioPlugin implements EditorPlugin {

  init(editorContext: IEditorContext): void {
    // ...
  }
}

export default SampleStudioPlugin;
}
```

Example 4.33. src/SampleStudioPlugin.ts

Now it's time to add the plugin descriptor to the `jangaroo.config.js` file. The plugin descriptor specified therein is read after a user logs in to the Studio web app. It contains a reference to your plugin class and a user-friendly name of the Studio plugin.

```
module.exports = jangarooConfig({
  // ...
  sencha: {
    // ...
    namespace: "com.coremedia.blueprint.studio.sample",
    studioPlugins: [
      {
        name: "Sample Plug-in",
       mainClass: "com.coremedia.blueprint.studio.sample.SampleStudioPlugin"

      }
    ]
  }
}
```

Example 4.34. jangaroo.config.js

Each JSON object in the `studioPlugins` array may use the attributes defined by the class `EditorPluginDescriptor`, especially `name` and `mainClass` as shown above. In addition, the name of a group may be specified using the attribute `requiredGroup`, resulting in the plugin only being loaded if a user logs in who is a member of that group.

Additional CSS files or other resources required for the plugin can be declared in the `sencha` configuration of the `jangaroo.config.js`.

When set up correctly, your project structure should build successfully using

```
pnpm -r run build
```

NOTE

Additional steps would be adding resource bundles and plugin rules to your plugin. For more details about this and developing Studio plugins and property editors have a look at the Studio Developer Manual.

Adding Studio Client Packages as Blueprint extensions

How to work with *Blueprint* extensions is described in detail in Section 4.1.5, “Project Extensions” [77]. For *Studio* client packages there are three extension points:

- `studio-client.main`
- `studio-client.main-static`
- `studio-client.workflow`

The *Studio* client packages are packaged into *apps*. CoreMedia distinguishes between so-called *(base) apps* and *app overlays*. An app is a Sencha Ext JS app and includes the Ext JS framework, *Studio* core packages and generally all

packages that participate in theming. Packages of an app are included in the build of the Sencha Ext JS app and are thus *statically* linked into the app. An app overlay in contrast references an app and adds further packages to this app. These packages are not included in the initial build of the app and instead can be loaded at runtime. Consequently, they are *dynamically* linked into the app.

The *CoreMedia Blueprint* features one *Studio* app, namely the `@coremedia-blueprint/studio-client.main.base-app` package with Jangaroo type `app`. In addition, there is one app overlay, namely the `@coremedia-blueprint/studio-client.main.app` package with Jangaroo type `app-overlay`. It references the `studio-client.main.base-app`. If something is wrong with the overall *Studio* app, it is typically sufficient to just re-compile `studio-client.main.base-app`.

Both apps come with their own extension points. Use the extension point `studio-client.main-static` (for the `studio-client.main.base-app`) for new packages that do theming and the extension point `studio-client.main` (for the `studio-client.main.app`) for packages that come without theming. Also, note that there must never be a dependency of a `studio-client.main-static` extension package to a `studio-client.main` extension package.

Adding Studio Server Packages as Blueprint extensions

Additional packages for the *Studio* REST Service (Java / Maven) use the `studio-server` extension point.

4.4.4 Developing with the CAE

The *CAE* can be extended with new capabilities by using the *Blueprint* extension mechanism or by just creating a new module with the required resources. In both cases the extension will be activated by adding a Maven dependency on the new module. This section describes how to add a new Blueprint module which contains an additional view template and a new view repository using this template.

Maven Configuration

First you have to create a new module which contains the required resources. The location of the new module inside the workspace is not important to enable the new features provided by the module. But to keep cohesion in the aggregation modules of the *CoreMedia Blueprint* workspace the new module should be cre-

ated next to other CAE functions. In this example the new module `sample-cae-extension` will be created in the `apps/cae/modules/cae` module.

1. First, add a new module entry named `sample-cae-extension` to the modules section of the `cae pom.xml` file:

```
<modules>
          <module>cae-base-lib</module>
          <module>cae-base-component</module>
          <module>cae-live-webapp</module>
          <module>cae-preview-webapp</module>
          <module>contentbeans</module>

          <!-- add module -->
          <module>sample-cae-extension</module>
          </modules>
```

2. After that create a new subdirectory `sample-cae-extension` and add the `pom.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
          <project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
          <modelVersion>4.0.0</modelVersion>

          <parent>
          <groupId>com.coremedia.blueprint</groupId>
          <artifactId>cae</artifactId>
          <version>BLUEPRINT_VERSION</version>
          <relativePath>../pom.xml</relativePath>
          </parent>

          <artifactId>sample-cae-extension</artifactId>
          <packaging>jar</packaging>

          <dependencies>
          <dependency>
          <groupId>com.coremedia.cms</groupId>
          <artifactId>coremedia-spring</artifactId>
          <scope>runtime</scope>
          </dependency>
          </dependencies>

          </project>
```

Now the basic structure for the extension exists.

Enabling the Extension

To enable the extension the target component has to depend on the created extension module.

To enable the new capabilities in all CAEs add the following dependency to the `pom.xml` of the `cae-base-component` module:

```
<dependency>
  <groupId>${project.groupId}</groupId>
```

```
  <artifactId>sample-cae-extension</artifactId>
  <version>${project.version}</version>
</dependency>
```

Creating Source Files and Folders

The sample extension for the CAE provides a new view template for the content type `CMArticle` to display external content and a new view repository configuration which includes this view template.

1. Create the new view template `CMArticle.ftl` in the module `sample-cae-extension` in the directory `src/main/resources/META-INF/resources/WEB-INF/templates/external-content-view-repository/com.coremedia.blueprint.common.contentbeans`.
2. To include the new view repository add a Spring configuration defining the following beans:

```
@Configuration(proxyBeanMethods = false)
  static class AddSampleViewRepositories {
    @Bean(autowireCandidate = false)
    @Customize(value = "viewRepositories", mode = Customize.Mode.PREPEND)

    String sampleViewRepositories() {
      // Add repository name, relative to /WEB-INF/templates/
      return "sample-cae-extension";
    }
  }
```

4.4.5 Quality Assurance

CoreMedia Blueprint ships with a bunch of unit tests. While these may serve as examples for your own tests to write, *CoreMedia Blueprint* also provides some example tests, that are just meant to be used as reference.

Tests closely related to Unified API development can be found at `global/examples/uapi-tests`. They will demonstrate how to use the `XmlRepoConfiguration` as well as examples related to unit testing the CAE (see Section 4.3.10, "Unit Testing a CAE Application" in *Content Application Developer Manual* for details)

For further details consult the corresponding `README.md` files as well as the documentation within the tests and their resources.

Do Not Adapt Examples

The examples are meant serve as read-only resource for diving into unit testing *CoreMedia Content Cloud*. In subsequent releases, these examples may change in structure, may get updated to new best practices or removed if outdated.

See Also

- Section 4.3.10, "Unit Testing a CAE Application" in *Content Application Developer Manual*

4.4.6 Customizing the CAE Feeder

Before customizing the *CAE Feeder*, you should be familiar with the content of Section 4.4.4, "Developing with the CAE" [138] about the *CAE* modules. Details about how the *CAE Feeder* works and how it may be customized are presented in the Search Manual.

4.4.7 Handling Personal Data

NOTE

All features how to handle personal data, from annotations to Maven profiles are in experimental stage. For details what this means to you read the API documentation of the `@Experimental` annotation.

Note, that feedback on this feature set is very welcome.

Personal data needs to be handled carefully, as it can be subject to regulations such as the European Union's General Data Protection Regulation (GDPR). Therefore, it is important to know how and where personal data is used in your code.

CoreMedia provides annotations to mark personal data and document the flow of personal data in Java code. You can also enable compile-time checks to validate that personal data is not passed accidentally to methods or libraries that are not prepared for personal data. Compile-time checking can be enabled with a Maven profile as described in Section 4.4.7.1, "Running Personal Data Checker" [142].

CoreMedia public Java API and *CoreMedia Blueprint* code already use personal data annotations, especially in the context of *Elastic Social*. Note that applied annotations are not necessarily complete; there can be more places where personal data is used. The personal data annotations and the corresponding checker are tools to help to document and restrict access to personal data, but they cannot solve this for each and every case. The existing annotations also do not state whether some data is personal data in the sense of some legal act or regulation.

The annotations for personal data are:

- `@PersonalData`,
- `@PolyPersonalData` and
- `@NonPersonalData`

of package `com.coremedia.common.personaldata` in module `com.coremedia.cms:coremedia-personal-data`. Please read the `API documentation` of that annotations first. It describes the usage of these annotations with examples in detail. Below you will find more details like Section 4.4.7.1, "Running Personal Data Checker" [142], Section 4.4.7.3, "Annotating Third-Party Libraries" [144] and more.

4.4.7.1 Running Personal Data Checker

To run the *Personal Data Checker* in *Blueprint* you need to enable the Maven profile `checkPersonalData` while compiling.

The checker for personal data will analyze all modules with a direct or transitive dependency to Maven module `coremedia-personal-data`. In your Maven output you may recognize the message `Checking @PersonalData` once such a module is found in your Maven build. That message may be missing as long as the `maven-compiler-plugin` bug MCOMPILER-536 has not been fixed.

By default, the *Checker Framework* will trigger a failure once it detects a violation in using personal data objects. To change the behavior to print only warnings instead, add the compiler argument `-Awarns`. Configure the `maven-compiler-plugin` for the `checkPersonalData` profile in Maven `pom.xml` files accordingly. For more configuration options have a look at the documentation at checkerframework.org.

4.4.7.2 Using Personal Data Annotations

You can find documentation and examples how to use the annotations for personal data in the `API documentation`. This section covers some further best practices.

Logging Personal Data

You might want to take extra care of logging personal data, which is either to prevent it from being logged or to ensure that such log entries go to a secured environment.

To do so, you may use markers of the *Simple Logging Facade for Java* (SLF4J). Markers enable filtering your logs by cross-cutting concerns, such as authentication and authorization or for log entries which contain personal data. Find more about filtering in the corresponding *Logback* documentation.

The CoreMedia API provides a set of predefined markers also for personal data. You will find them as part of `com.coremedia.common.logging.BaseMarker` which are:

- `PERSONAL_DATA`,
- `UNCLASSIFIED_PERSONAL_DATA` and

Read the corresponding documentation for more details how and when to use them.

In order to log personal data explicitly it is recommended to use the logger `com.coremedia.common.logging.PersonalDataLogger`. It provides the very same logging methods as a standard SLF4J `Logger` having its parameters already annotated with `@PersonalData`. For details and usage examples have a look at the `API documentation`.

Logging Exceptions

To log exceptions which might (or will) contain personal data, you should consider using the helper class `com.coremedia.common.logging.PersonalDataExceptions`. It will log the original exception securely, using the markers mentioned above and rethrow a new exception which is directly under your control. The tool will ensure that there is a reference between the new exception and the logged one which eases tracing the exceptions although the cause hierarchy is not available by intention.

4.4.7.3 Annotating Third-Party Libraries

When handing over personal data to third-party dependencies, you will most likely get a compile time error raised by the *Checker Framework*. You have two options then: Either suppress the check as stated in the Javadoc or add so-called stub classes as described in The *Checker Framework* Manual.

CoreMedia Personal Data Checker already comes with some predefined stub classes. But they may not be sufficient for your needs. Adding your own stub classes can extend or even override the predefined stub classes as your explicitly mentioned stubs have a higher priority. And more: You may add stub classes for CoreMedia API as well.

To add custom stubs, just extend the annotation processor arguments in the `checkPersonalData` profile. In the example Example 4.35, “Adding custom stub classes” [144] you see how you may add two directories which will then be scanned for files named `*.astub`. Using `${path.separator}` ensures that your path will work across multiple platforms.

```
<profile>
  <id>checkPersonalData</id>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <executions>
          <execution>
            <id>default-compile</id>
            <!-- ... -->
            <configuration>
              <compilerArgs>
                <!-- ... -->
                <arg>-Astubs=/stubs/a${path.separator}/stubs/b</arg>
              </compilerArgs>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</profile>
```

Example 4.35. Adding custom stub classes

4.4.7.4 Stubbing: Best Practices

You can use stub files to add annotations to classes and methods which are not part of your source code, that is especially third-party API and of course CoreMedia API.

Below you will find some practices which have proven to ease managing such stub files.

Naming Convention

While you may put all your stubs into one file, it is recommended to split it into smaller chunks to ease maintenance. For example, create a stub file for each third-party library.

A possible naming pattern is to use the Maven group and artifact ID within the filename. So for the artifact `mongodb-driver-legacy` within group `org.mongodb` a stub filename could be: `org.mongodb.mongodb-driver-legacy.astub`.

Stub Structure

Start with imports and especially start with importing the personal data annotations:

```
import com.coremedia.common.personaldata.*;
```

Then add packages and their classes, each in alphabetical order. Add methods in alphabetical or logical order to group getters and setters for example.

Stubbing Rules

- In general, do not annotate parameters of methods that are intended for override. This may cause errors for existing code.

  ```
  class Collection<E> {
    boolean contains(@PersonalData Object o); // BAD!
  }
  ```

  While this sounds useful, it actually breaks existing code. Custom Collection implementations would cause errors as long as their parameter is not annotated as well. Because of that, it is better to use `@SuppressWarnings("PersonalData")` at usages of `Collection#contains`.
- The stub parser of the ignores method bodies and modifiers and it is recommended to omit them for readability. However, adding `static` and `final` modifiers to methods and classes will make it easier to think about possible overrides (see previous rule). Parameters of static or final methods or final classes can easily be annotated because there cannot be any overrides.
- There is no need to annotate all methods of a class or interface. However, it often makes sense to annotate similar methods equally. Overloaded conveni-

ence methods which just differ in the number of parameters should be annotated together to avoid confusion.

- Sometimes it makes sense to annotate classes itself when instances of this class contain personal data and are passed to third-party methods.

```
package java.security;

@PersonalData interface Principal {
  @PersonalData String getName();
}
```

Reasoning: A `Principal` may be passed to some third-party method, which then possibly calls `getName()` internally. Because third-party internals are not subject to checking, you should already check the transfer of the `Principal` object to third-party libraries and either avoid it or allow it with `@SuppressWarnings("PersonalData")`.

Note that this rule just applies to stubbing: It does not make sense to use `@PersonalData` at interface source files when all code that uses that interface can be checked for personal data use. It is easier to just annotate method return values then. However, if a custom class or interface extends a third-party class/interface that is already annotated with `@PersonalData`, then the extending class/interface needs to be annotated in the same way:

```
@PersonalData class MyPrincipal implements Principal {
  @Override
  public @PersonalData String getName() {
    return name;
  }
  // ...
}
```

5. CoreMedia Blueprint - Functionality for Websites

This chapter describes all aspects of *CoreMedia Blueprint* that you can use to manage your web sites.

- Section 5.1, "Overview of eCommerce Blueprint" [148] gives a short overview of the *eCommerce Blueprint* frontend.
- Section 5.2, "Overview of Brand Blueprint" [152] gives an overview of the *Brand Blueprint* frontend.
- Section 5.3, "Basic Content Management" [154] describes aspects of the content type model of *CoreMedia Blueprint*.
- Section 5.4, "Website Management" [171] describes all features relevant for website management, such as layout, search and navigation.
- Section 5.5, "Localized Content Management" [239] describes all aspects of localized content management.
- Section 5.7, "Workflow Management" [299] describes all aspects of multi-site management.

5.1 Overview of eCommerce Blueprint

The *eCommerce Blueprint* provides a modern, appealing, highly visual website template that can be used to start a customization project. It demonstrates the capability to build localizable, multi-national, experience-driven eCommerce web sites. Integration with HCL Commerce, SAP Hybris Commerce and Salesforce Commerce Cloud ships out of the box. Other eCommerce systems can be integrated via the CoreMedia eCommerce API as a project solution.

The following integration patterns are available with the product:

- Commerce-led fragment-based approach like the Hybris example
- Experience-led hybrid blended approach shown in the Calista store example

Based on a fully responsive, mobile-first design paradigm, the *eCommerce Blueprint* leverages most of our bricks and the FreeMarker templating framework. It scales from mobile via tablet to desktop viewport sizes and uses the CoreMedia Adaptive and Responsive Image Framework to dynamically deliver the right image sizes in the right aspect ratios and crops for each viewport.

NOTE

For more information about the themes please see the Section 6.1, “Example Themes” in *Frontend Developer Manual*

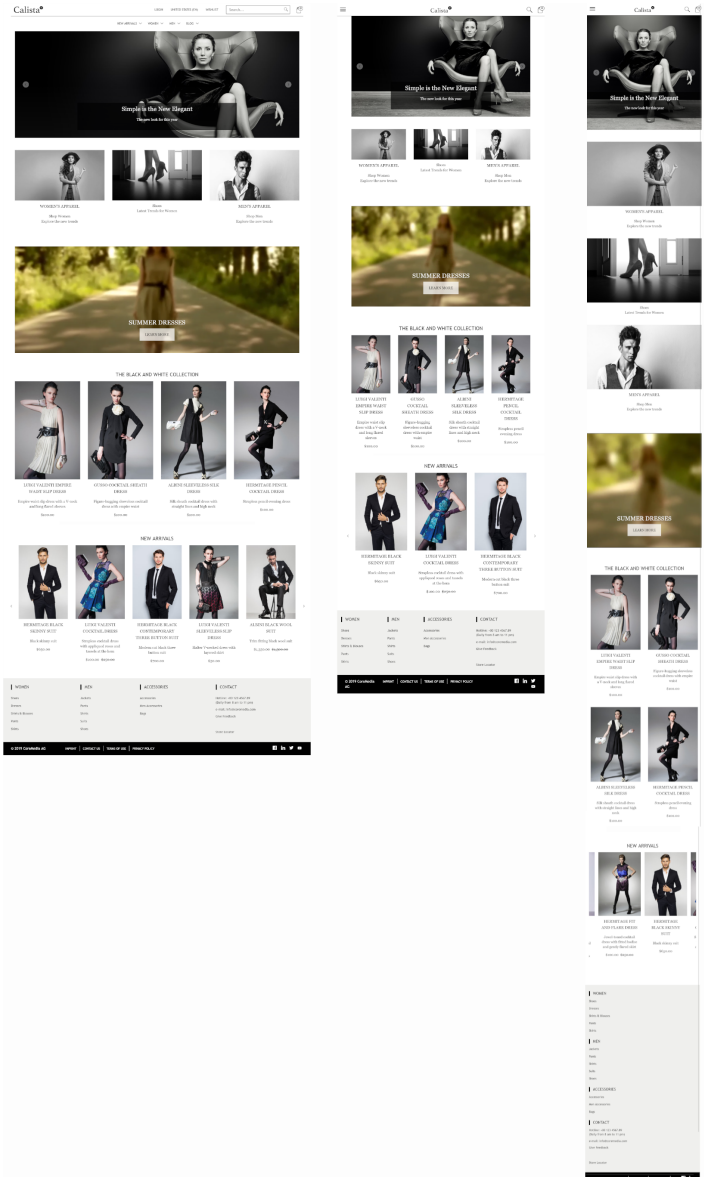

Figure 5.1. Calista (Experience-led) start page for different devices: desktop, tablet, mobile

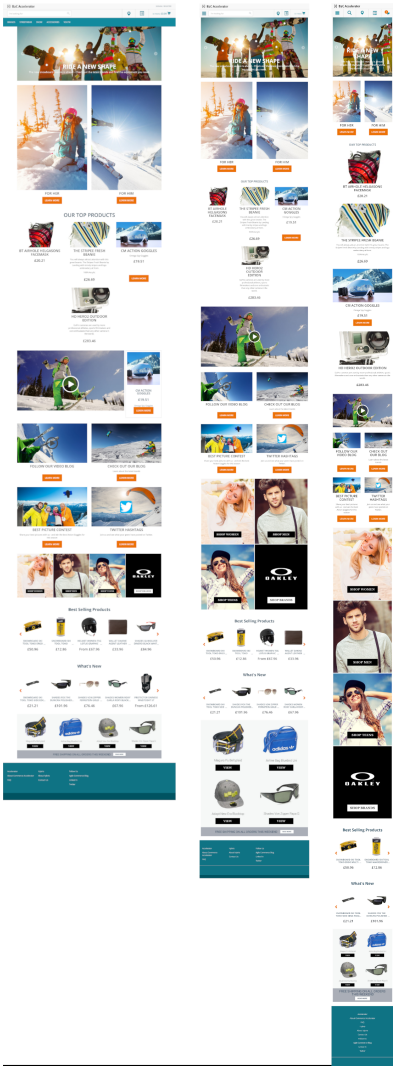

Figure 5.2. Hybris (commerce-led) start page for different devices: desktop, tablet, mobile

The responsive navigation can blend commerce as well as content categories and content pages seamlessly and in any user-defined order that does not have

to follow the catalog structure. Navigation nodes with URLs to external sites can be added in the content.

5.2 Overview of Brand Blueprint

The *Brand Blueprint* provides a modern, appealing, highly visual website template that can be used to start a customization project. It demonstrates the capability to build localizable, multi-national, non-commerce web sites.

Based on a fully responsive, mobile-first design paradigm, the *Brand Blueprint* leverages most of our bricks and Design framework for easy customization and adaptation by frontend developers. It is a child theme, inherited from the Shared-Example Theme.

NOTE

For more information about the themes please see the Section 6.1, "Example Themes" in *Frontend Developer Manual*

It scales from mobile via tablet to desktop viewport sizes and uses the CoreMedia Adaptive and Responsive Image Framework to dynamically deliver the right image sizes in the right aspect ratios and crops.

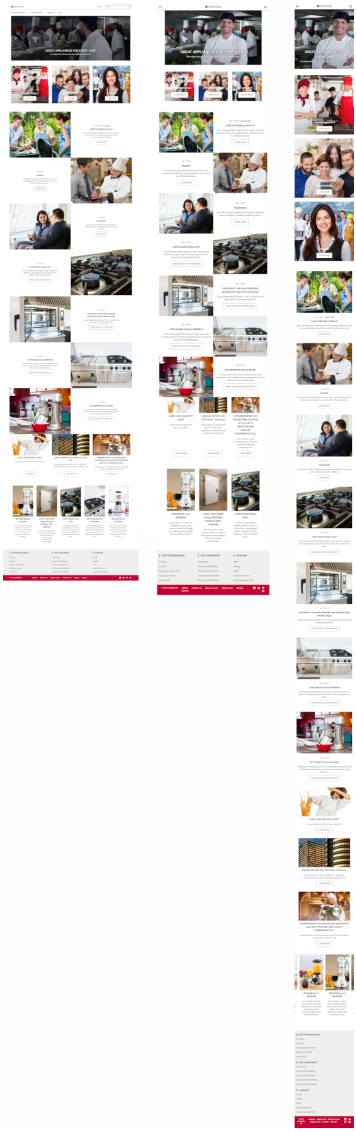

Figure 5.3. Chef Corp. start page for different devices: desktop, tablet, mobile

5.3 Basic Content Management

The basis of the information structure of a CoreMedia system are content types. Content types organize your content and form a hierarchy with inheritance.

See the Content Server Manual, Developing a Content Type Model and Section 7.1, "Content Type Model" [430] for more details.

CoreMedia Blueprint comes with a comprehensive content type model that covers the following topics:

- Common content such as Articles or Pictures.
- Placeholder types that you can use to link to commerce content
- Taxonomies are used to tag content.

5.3.1 Common Content Types

Requirements

An appealing website does not only contain text content but has also images, videos, audio files or allows you to download other assets such as brochures or software.

In addition, current websites aim to reuse content in different contexts. An article about the Hamburg Cyclassics might appear in Sports, Hamburg and News section, for example. An image of the St. Michaelis church (the "Hamburger Michel") on the other hand might appear in Articles about sights in Hamburg or religion. Nevertheless, it's not a good idea to copy the article to each section or the image to each article because this is error prone, inefficient and wastes storage.

Therefore, content should be reusable across different contexts (different sites, customer touchpoints for instance) by just applying the context specific layout and without having to duplicate any content. This increases the productivity by reducing redundancy and keeps management effort at a minimum.

Solution

CoreMedia Blueprint is shipped with content types that model common digital assets such as articles, images, videos or downloads. All these types inherit from a common parent type and can be used interchangeably. In addition, none of these types has fixed information about its context so that it can be used re-

peatedly and everywhere in your site. The context is first determined through the page which links to the content item or through the position in the folder hierarchy of the website (see Section 5.4.2, “Navigation and Contexts” [173] for more details).

Common Content Types

CoreMedia Blueprint defines the following types for common content. Using CoreMedia's object oriented content model projects can define their own content types or add to the existing ones.

`CMArticle`	
UI Name	Article
Description	Contains mostly the textual content of a website combined with images.
`CMPicture`	
UI Name	Picture
Description	Stores images of the website. The editor can define different crops of the image which can be used in different locations of the website.
`CMVideo`	
UI Name	Video
Description	Stores videos which can be viewed on the website.
`CMAudio`	
UI Name	Audio
Description	Stores audio/podcast information which can be heard on the website.
`CMDownload`	
UI Name	Download
Description	Stores binary data for download. You can add a description, image and the like.
`CMGallery`	

UI Name	Gallery
Description	Aggregates images via a linklist. You can add a description, teaser text and the like.

Table 5.1. Overview of Content Types for common content

eCommerce Placeholder Types

Blueprint comes with some additional content types required to build representations of entities of a commerce system.

`CMProductTeaser`	
UI Name	Product Teaser
Description	A teaser for products of the commerce system. It inherits from `CMTeasable`
`CMMarketingSpot`	
UI Name	e-Marketing Spot
Description	A placeholder for an e-Marketing spot. It inherits from `CMTeasable`.
`CMExternalChannel`	
UI Name	Category Placeholder
Description	Content items of this type are used to build a CMS representation of commerce categories. It inherits from `CMAbstractCategory` which in turn inherits from `CMChannel`.
`CMExternalPage`	
UI Name	Placeholder for other shop pages such as Help pages or the main page.
Description	Content items of this type are used to build a CMS representation of other commerce pages. It inherits from `CMChannel`.

Table 5.2. Commerce Content Types

Commerce Content Properties

A short description of the properties provided for eCommerce scenarios is provided below.

externalId	
UI Name	External ID
Description	The ID of the corresponding entity in the commerce system. For a `CMProduct-Teaser` this id is the technical id of the product in the catalog.
localSettings.shopNow	
UI Name	'Shop Now' flag
Description	This Boolean flag is stored in the local settings of the content types `CMProduct-Teaser` and `CMExternalChannel` and is used in the content-led scenario. If enabled the 'Shop Now' overlay is visible for product teasers. This configuration is extendable via `CMExternalChannels` and may be overwritten for every `CMProductTeaser`.

Table 5.3. Overview Commerce Content Properties

Common Content Properties

All common content types extend the abstract type `CMTeasable` to share common properties and functionality. Teasable means that you can show for each content that inherits from `CMTeasable` a short version that "teases" the reader to watch the complete article, site or whatever else.

A short description of the core properties of content is provided below. Properties specific for certain Blueprint features such as teaser management etc. are described in their respective sections (follow the link in the *Description* column).

title	
UI Name	(Asset) Title
Description	The name or headline of an asset, for example the name of a download object or the headline of an article.
detailText	
UI Name	Detail Text

Description	A detailed description, for example the article's text, a description for a video or download.
`teaserTitle, teaserText`	
UI Name	Teaser Title and Text
Description	The title and text used in the teaser view of an asset. See Section 5.4.9, "Teaser Management" [196].
`pictures`	
UI Name	Pictures
Description	A reference to `CMPicture` items that illustrate content. Examples include a photo belonging to the article, a set of images from a video etc. Usage of the pictures depends on the rendering. In *Blueprint* the pictures are used for teasers and detail views of content.
`related`	
UI Name	Related Content
Description	The related content list refers to all items that an editor deems related to the content. For an article for a current event this list could include a video describing of the event, a download with event brochure, an audio/podcast file with an interview with the organizers, an image gallery with photos of the previous event and many more.
`keywords`	
UI Name	Keywords
Description	Keywords for this content. *CoreMedia Blueprint* currently uses keywords as meta information for the HTML <head>.
`subjectTaxonomy`	
UI Name	locationTaxonomy
Description	Tags for this content. See Section 5.3.2, "Tagging and Taxonomies" [161] for details.

`viewType`	
UI Name	Layout Variant
Description	The layout variant influences the visual appearance of the content on the site. It contains a symbolic reference to a view that should be used when the content is rendered. For more information see Section 5.4.7, "View Types" [191]
`segment`	
UI Name	URL Segment
Description	A descriptive segment of a URL for this content. Used for SEO on pages displaying the content. See Section 5.4.15, "URLs" [211]
`locale, master, masterVersion`	
UI Name	Locale, Master, Master Version
Description	See Section 5.5, "Localized Content Management" [239] for details. Properties for the Localization of this asset.
`validFrom, validTo`	
UI Name	Valid From, Valid To
Description	Meta information about the validity time range of this content. Content which validity range is not between validFrom and validTo will not be displayed on the website. See Section 5.4.18, "Content Visibility" [214] for details.
`notSearchable`	
UI Name	Not Searchable Flag
Description	Content with this flag will not be found in end user website search. See Section 5.4.22, "Website Search" [224] for details.

Table 5.4. Overview Common Content Properties

Media Content

The abstract content type `CMMedia` defines common properties for all media types. Media types for content such as pictures (`CMPicture`), video (`CMVideo`), audio (`CMAudio`), and HTML snippets (`CMHTML`) inherit from `CMMedia`.

`data`	
UI Name	Data
Description	The core data of the content. Either a `com.coremedia.cap.common.Blob` or in the case of `CMHTML` a `com.coremedia.xml.Markup`.
`copyright`	
UI Name	Copyright
Description	Allows you to store arbitrary copyright information in a string property.
`alt`	
UI Name	Alternative Representation
Description	Allows managing alternative representations of an image, for example a description of an image that can be used to enable a website accessible for the visually impaired.
`caption`	
UI Name	Caption
Description	The caption of a content. Unused property in *Blueprint*.

Table 5.5. `CMMedia` Properties

A common feature of all `CMMedia` objects is the ability to generate and cache transformed variants of the underlying object (see `CMMedia#getTransformedData`). This ability is extensively used for rendering images without the need to store image variants and renditions as distinct blobs in the system.

5.3.2 Tagging and Taxonomies

Requirements

Most websites define business rules that require content to be classified into certain categories. Typical examples include use cases such as "Display the latest articles that have been labeled as press releases" or "Promote content tagged with 'Travel' and 'London' to visitors of pages tagged with 'Olympic Games 2012'" etc.

Keywords or tags are common means to categorize content. Employing a controlled vocabulary of tags can be more efficient than allowing free-form keyword input as it helps to prevent ambiguity when tagging content. Furthermore, a system that supports the convenient management of tags in groups or hierarchies is required for full editorial control of the tags used within a site.

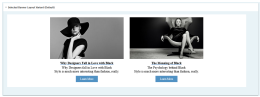

Figure 5.4. Dynamic list of articles tagged with "Black"

Solution

Blueprint currently uses tag information in various ways:

- It is possible to use the taxonomies of a content item as conditions for dynamic lists of content (such as "5 latest articles tagged with 'London').
- It is possible to display related content for a content item based on content that shares a similar set of tags (see `CMTeasableImpl#getRelatedBySimilarTaxonomies`).

In *CoreMedia Blueprint* tags are represented as `CMTaxonomy` content items which represent a controlled vocabulary that is organized in a tree structure. *CoreMedia Blueprint* defines two controlled vocabularies: Subject and location taxonomies that can be associated with all types inheriting `CMLinkable`.

5.3.2.1 Taxonomy Management

Subject taxonomies can be used to tag content with "flat" information about the content's topic (such as Olympic Games 2012). They can also enrich assets with

hierarchical categorization for fine-grained drill down navigation (such as Hardware / Printers / Laser Printer).

Subject Taxonomies are represented by the content type `CMTaxonomy` which defines the following properties:

`value`	
Type	`String`
Description	Name of this taxonomy node
`children`	
Type	`Link list`
Description	References to subnodes of this taxonomy node
`externalReference`	
Type	`String`
Description	Reference of an equivalent entity in an external system in the form of an ID / URI etc.

Table 5.6. CMTaxonomy Properties

Location taxonomies allow content to be associated with one or more locations. Location taxonomy hierarchies can be used to retrieve content for a larger area even if it is only tagged with a specific element within this area ("All articles for 'USA'" would include articles that are tagged with the taxonomy node North America / USA / Louisiana / New Orleans).

Location taxonomies are represented by the content type `CMLocTaxonomy` which inherits from `CMTaxonomy` and adds geographic information for more convenient editing and visualization of a location.

`latitudeLongitude`	
Type	`String`
Description	Latitude and longitude of this location separated by comma
`postcode`	

Type	`String`
Description	The post code of this location

Table 5.7. Additional CMLocTaxonomy Properties

The taxonomy administration editor can be used to create a taxonomy and build a tree of keywords.

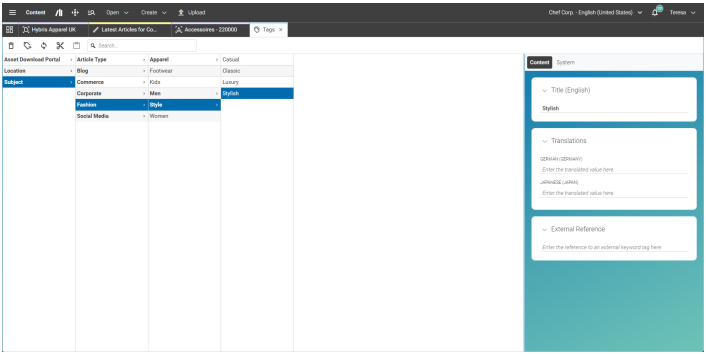

Figure 5.5. Taxonomy Administration Editor

The taxonomy administration editor displays taxonomy trees and provides drag and drop support and the creation and deletion of keywords.

5.3.2.2 Taxonomy Assignment

To enable tagging of content two properties are available the `CMLinkable` content type.

`subjectTaxonomy`	
Type	`Link list`
Description	Subject(s) / topic(s) of that content item
`locationTaxonomy`	
Type	`Link list`

Description	Geographic location(s) of that content item

Table 5.8. CMLinkable Properties for Tagging

Editors can assign taxonomies to content items using *CoreMedia Studio* and the Blueprint taxonomy property editor. It allows for the following:

- adding/removing references to taxonomy
- autocompletion
- suggestions

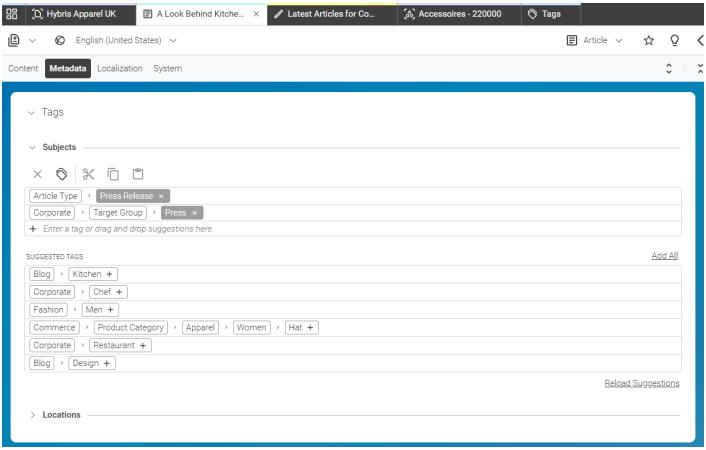

Figure 5.6. Taxonomy Property Editor

The user can add taxonomy keywords to the corresponding property link list using the taxonomy property editor. The editor also provides suggestions that are provided by a name matching algorithm. The strategy type can be configured in the preferences dialog of *CoreMedia Studio*.

Figure 5.7. Taxonomy Studio Settings

How taxonomies are loaded

A Blueprint taxonomy tree is built through content items located in a specific folder of the content repository. The default taxonomy resolver will then look for a "_root" settings content item inside these folders and uses the taxonomy content items linked inside the LinkListProperty "roots" as top level nodes. If the

"_root" content item is not found, the taxonomy resolver checks for empty referrers of all taxonomy content items inside a taxonomy folder, to determine which node is a top level node of the given taxonomy tree. In that case, top level node content items must be placed directly within the root folder. Taxonomies of subsequent levels can also be placed in subfolders. The name of the folder in which the taxonomy tree is placed defines the name of the taxonomy tree and is visible as a root node in the taxonomy administration UI.

The lookup folders for taxonomies and the strategy used to build the tree are configured in the Spring configuration class `TaxonomyConfiguration` of the `shared/taxonomies` module. The bean properties

```
siteConfigPath
```

and

```
globalConfigPath
```

of the `strategyResolver` bean configure the folders that are used to find taxonomies. `TaxonomyResolverImpl` implements the `TaxonomyResolver` interface so that it is possible to implement other taxonomy detection strategies.

WARNING

The default taxonomy implementation (`DefaultTaxonomy.java`) checks the taxonomy folder for write permissions. If these permissions are not granted, the taxonomy won't appear in *Studio*. Therefore, ensure that taxonomy administrators have `Folder` rights for taxonomy folders.

How to implement a new taxonomy resolver strategy

The `TaxonomyResolverImpl` implements the interface `TaxonomyResolver` and is injected to the `TaxonomyResource` in the `component-taxonomies.xml`. For every taxonomy request, the `TaxonomyResource` instance looks up the corresponding `Taxonomy` object using the resolver instance. To change the resolver strategy, inject another instance of `TaxonomyResolver` to the `TaxonomyResource`.

How to configure the content properties used for semantic strategies

The content properties that are used for a semantic evaluation are configured in the method `SemanticTaxonomyConfiguration#semanticDocumentProperties` of the `shared/taxonomies` module. The Spring configuration declares the abstract class `AbstractSemanticService` that new

semantic service can extend from. The default properties used for a semantic suggestion search are:

- `title`
- `teaserTitle`
- `detailText`
- `teaserText`

How to implement a new suggestion/semantic strategy

To add a new semantic strategy to *Studio*, it is necessary to implement the corresponding strategy for it and add it to *CoreMedia Studio*.

A new semantic strategy can easily be created by implementing the interface `SemanticStrategy`. The result of a strategy is a `Suggestions` instance with several `Suggestion` instances in it. Each `Suggestion` instance must have a corresponding content instance in the repository whose content type matches that one used for the taxonomy. *Blueprint* uses `CMTaxonomy` content items for keywords of a taxonomy, so suggestions must be fed with these content items. Additionally, a float value `weight` can be set for each suggestion, describing how exactly the keyword matches from 0 to 1. After implementing the semantic strategy, the implementing class must be added to the Spring configuration, for example:

```
<customize:append id="semanticStrategyExamplesCustomizer"
bean="semanticServiceStrategies" order="1000">
      <list>
      <ref bean="myMatching"/>
      </list>
      </customize:append>
```

Next, the new suggestion strategy must to be added to *Studio*, so that is selectable in *CoreMedia Studio*.

1. Open the file `TaxonomyStudioPlugin.ts`
2. Add an entry to the configuration section that configures the `AddTaggingStrategyPlugin`:

   ```
   <taxonomy:AddTaggingStrategyPlugin serviceId="{TAXONOMY_NAME_MATCHING_KEY}"

   label="{resourceManager.getString('com.coremedia.blueprint.studio.taxonomy.TaxonomyStudioPlugin',
    'TaxonomyPreferences_value_nameMatching_text')}" />
   ```

   Make sure that the `serviceId` matches the one you configured for the implementation of the `SemanticStrategy`.

How to add a site specific taxonomy

The logic how a site-depending taxonomy tree is resolved, is implemented in the `TaxonomyResolver#getTaxonomy(String siteId, String taxonomyId)` method.

To create a new site-depending taxonomy proceed as follows:

1. Open *Studio*, create and select the site specific folder `Options/Taxonomies/` from the library.
2. Create a new sub folder with the name of the new taxonomy.

   The location for the new taxonomy has been created now.
3. To identify the type of taxonomy (such as `CMTaxonomy` or `CMLocTaxonomy`) you have to create at least one taxonomy content item in the new folder. Alternatively, create a `_root` settings content item and link a newly created `CMTaxonomy` content item to the StructList `roots` to it.

Once the taxonomy has been set up, additional nodes can be created using the taxonomy manager. If the new taxonomy does not appear as new element in the column on the left, press the reload button. It ensures that the `TaxonomyResolver` rebuilds the list of available taxonomy trees. The new taxonomy is shown in the root column afterwards, include the site name it is created in.

Creating site specific taxonomies allows you to overwrite existing ones. For example, you create a new taxonomy tree called `Subject` for site `X` and open an article that is located in a sub folder of site `X`, the regular `Subject` taxonomy property editor on the `Taxonomies` tab in *CoreMedia Studio* will access the `Subject` taxonomy of your new site, not the one that is located in the global `Settings` folders. The suggestions and the chooser dialog will also work in the new taxonomy tree.

Taxonomy Localization Workflow Integration

When translating content items that belong to a taxonomy tree, parent taxonomy nodes and the taxonomy root settings document are also considered for the translation set. This ensures that the taxonomy hierarchy remains intact in the derived site. If your content type model uses a custom taxonomy type instead of `CMTaxonomy`, the configuration can be adjusted accordingly. For details see Section 4.5, "Referrer Relations in Localization" in *Multi-Site Manual* and Section 3.6.2, "Shared Multi-Site Configuration" in *Deployment Manual* as well as in Section 3.4.3, "Shared Multi-Site Configuration" in *Deployment Manual*.

How to configure the taxonomy property editor for a taxonomy

CoreMedia Blueprint comes with two types of taxonomies: `Subject` and `Location`. The name of the taxonomy matches the folder name they are located in, which is `/Settings/Taxonomies`. When the taxonomy property editor for a *Studio* form is configured, these IDs are passed to the property editor, for example

```
<taxonomy:taxonomyPropertyField propertyName="subjectTaxonomy"
                                                taxonomyId="Subject"/>
      <taxonomy:taxonomyPropertyField itemId="locTaxonomyItemId"
                                      propertyName="locationTaxonomy"
                                      taxonomyId="Location"/>
```

As mentioned in the previous section, it is possible to overwrite the existing location or subject taxonomy with a site-depending variant. In this case, it is *not* necessary to change the configuration for the property field. The taxonomy property editor will always try to identify the site-depending taxonomy with the same name first. If this one is not found, the global taxonomy with the given id will be looked up and used instead. For custom site-specific taxonomy trees, the attribute value `taxonomyId` must match the name of the newly created taxonomy folder.

How to configure access to the taxonomy content / taxonomy administration

To ensure that the taxonomies are working properly, ensure that the user has the corresponding read and write rights to the settings and taxonomy folders. For taxonomy folders, ensure that also the *Folder* rights are set.

How to configure contexts for taxonomy trees

Since global taxonomies are not part of the "regular" content hierarchy, they do not have a context. That's why the `IsPartOfNavigationValidator` of *CoreMedia Studio* excludes global settings paths, as well as the underlying taxonomy folders from the context validation. If you, however, created a taxonomy tree where this validator complains about a missing context, make sure that the folder of your taxonomy root folder is excluded from the context validation (please refer to Section 3.4.5, “Navigation Validator Configuration” in *Deployment Manual*).

Taxonomy Manager Access

You can configure the list of user groups that can access the taxonomy manager in the `jangaroo.config.js` file of the `taxonomy` extension. By default, the following groups are allowed to open the manager:

- global-manager
- taxonomy-manager
- developer

This list cannot be customized during runtime. To add or remove access for users, ensure that they are a member of the corresponding groups.

Taxonomy Visibility

To allow a more fine-granular access to taxonomy nodes in the Taxonomy Manager, it is possible to configure roles for different taxonomy trees in the `TaxonomySettings` settings document. The example below shows how to configure the visibility of a global "Subject" taxonomy tree for the user group `taxonomy-admin` and for the site specific "Subject" taxonomy (with site id "MY_SIDE_ID") for the user group `taxonomy-admin`.

```
<Struct xmlns="http://www.coremedia.com/2008/struct"
xmlns:xlink="http://www.w3.org/1999/xlink">
    <StructProperty Name="rules">
      <Struct>
        <StructProperty Name="Subject">
          <Struct>
            <StructProperty Name="global">
              <Struct>
                <StringListProperty Name="groups">
                  <String">taxonomy-admin</String>
                </StringListProperty Name="groups">
              </String>
              </Struct>
            </StructProperty>
            <StructProperty Name="MY_SIDE_ID">
              <Struct>
                <StringListProperty Name="groups">
                  <String">taxonomy-admin</String>
                </StringListProperty Name="groups">
              </String>
              </Struct>
            </StructProperty>
          </Struct>
        </StructProperty>
      </Struct>
    </StructProperty>
```

Note that this configuration only configures the visibility of the taxonomies inside the Taxonomy Manager. User are still able to see matching contents of the taxonomy in the library if they have the corresponding rights.

How to configure taxonomy localization

By default, the localization for taxonomies is enabled and only supports global taxonomy trees. The logic which language should be used when a tag is translated is implemented in the class `TaxonomyLocalizationStrategyImpl` which implements the interface `TaxonomyLocalizationStrategy`. The class is located in the `content-services` modules of the `shared/middle`

workspace. Note that this class is responsible for the localization of taxonomies in all apps: *Studio*, *Headless Server* and *CAE*.

The `TaxonomyLocalizationStrategy` reads configuration values from global content item `/Settings/Options/Settings/TaxonomySettings`. The configuration `TaxonomySettings` contains the `StringListProperty translations` that contains the list of target locales. The additional `defaultLanguage StringProperty` defines the locale of the `value` property of taxonomy content items. This property contains the name of the tag. In a localized context, this name is used when the request language is not available.

```
<Struct xmlns="http://www.coremedia.com/2008/struct"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <StringListProperty Name="translations">
    <String>en</String>
    <String>de</String>
    <String>ja</String>
  </StringListProperty>
  <StringProperty Name="defaultLocale">en<StringProperty>
</Struct>
```

If the localization shouldn't be used, the field `defaultLanguage` can be left empty. This will hide the list of localized input fields from the document form. Otherwise, additional `StringProperty` input fields will be shown for every locale of the `translations` list. The values are stored in the `localSettings` of each taxonomy content item.

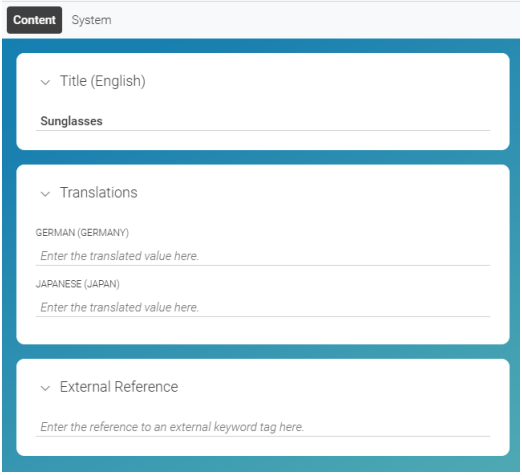

Figure 5.8. Taxonomy Localization Form

5.4 Website Management

Website management comprises different features. For example:

- Layout
- Navigation
- Search

5.4.1 Folder and User Rights Concept

It is good practice to organize the content of a content management system in a way that separates different types of content in different locations and to have user groups that attach role depending rights to these locations. This fits with CoreMedia access rights, which are assigned to groups and grant rights to folders and their content, including all sub folders, to all members of that group.

See Section 3.15, "User Administration" in *Content Server Manual* for details about the CoreMedia rights system.

CoreMedia Blueprint comes with demo sites that provide a proposal on how to structure content in a folder hierarchy and how to organize user groups for different roles. A more fine grained folder and group configuration can easily be built upon this base.

For details on site specific groups and roles have a look at Groups and Rights Administration for Localized Content Management [247] and for a set of predefined users for that groups and roles see Reference - Predefined Users [439].

CoreMedia Blueprint distinguishes between the following types of content in the repository:

Different content types for different uses

- **Content:** These are the "real" editorial contents like Articles, Images, Videos, and Products. They are created and edited by editorial users. In a multi-site environment editors are usually working on one of the available sites and they can only access that site's content.
- **Navigation and page structure:** These types represent the site's navigation structure - both the main navigation and the on-page navigation elements like collections or teasers linking to other pages. They are readable by every editorial user, but only the site manager group may maintain them.
- **Technical content types** like options, settings and configuration: These types provide values for drop down boxes in the editorial interface, like view types. They also bundle reusable sets of context settings, for example API keys for

external Services. These types are readable by every editorial user but can only be created and edited by Administrators or other technical staff.

- **Client code:** Consists of JavaScript and CSS and is maintained by technical editors.

CoreMedia Blueprint comes with a folder structure that simplifies groups and rights management in that way that users taking specific roles only get rights to those contents they are required to view or change. Most notably you will find a `/Sites` folder which contains several sites and several other folders which contain globally used content like global or default settings.

For details on the structure of the `/Sites` folder have a look at Section 5.5.1.2, "Sites Structure" [243].

Commonly used content is stored below dedicated folders directly at root level. Web resources like CSS or JavaScript is stored under `/Themes`. Global settings, options for editorial interfaces, and the like are stored under `/Settings`.

Site-Independent Groups

Along with the site specific groups which are described in Groups and Rights Administration for Localized Content Management [247] there are also groups representing roles for global permissions required by some of the predefined workflows. These workflows are especially dedicated to the publication process and are bound to the following roles:

- composer-role

  This site-independent group allows members to participate in a workflow as a composer, that is each member of this group may compose a change set for a publication workflow.

- approver-role

  This site-independent group allows members to participate in a workflow as an approver, that is each member of this group may perform approval operations within a publication workflow.

- publisher-role

  This site-independent group allows members to participate in a workflow as a publisher, that is each member of this group may publish the content items involved in a workflow.

For details on these groups and how to connect them to a LDAP server have a look at Workflow Manual.

5.4.2 Navigation and Contexts

Requirements

Websites are structured into different sections. These sections frequently form a tree hierarchy. For example, a news site might have a Sports section with a Basketball subsection. The website of a bank might have different sections for private and institutional investors with the latter having subsections for public and private institutions.

Sections are also often called "navigation" or "context". Usually the sections of a site are displayed as a navigable hierarchy (a "navigation" or "site map"). The current location within the tree is often displayed as a "breadcrumb navigation".

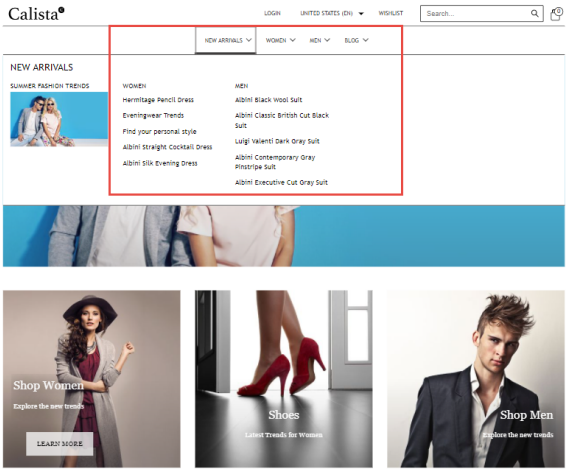

Figure 5.9. Navigation in the Site

Additionally, efficient content management requires reuse of content in different contexts.

For example, reusage of an article for a different section, a mobile site or a micro site should not require inefficient and error-prone copying of that article.

Solution

A site section (or "navigation" or "context") is represented by a content item of type `CMChannel` or `CMExternalChannel` which is a child of `CMChannel`.

Sections span a tree hierarchy through the child relationships of `CMChannel#children`. If a `CMChannel` is referenced by a `CMSite` item it is considered a root channel, that is an entry into a channel hierarchy representing a website. The `CMChannel` content items fulfill the following purposes:

- Hierarchy: They form a hierarchy of site sections which can be displayed as a navigation, sitemap, or bread crumb. Each site consists of exactly one section tree.
- Context: They function as contexts for content. Content can be reused within different contexts in different layouts and visual appearance. For example, an article's layout may differ in a company's blog section from its layout in the knowledge base.
- Page: Each `CMChannel` can be rendered as an overview page of the section it represents. Therefore, the `CMChannel` contains information about the page structure (the "grid") for this overview page and the pages generated when content items are displayed in the content of the `CMChannel`.

  For more information on how web pages are assembled in *Blueprint* also refer to the Section 5.4.4, “Page Assembly” [177] section.
- Configuration: `CMChannel` content items contain settings which configure various aspects of the site section they represent. Each `CMChannel` can override parent configuration by defining its own layout settings, content visibility, and other context settings. If for example, the "News" section of a site is configured for post-moderation of comments this configuration can be overwritten to premoderation in the subsection "News/Politics".

  For more information on settings see the section Section 5.4.3, “Settings” [175].

The context in which a content should be displayed is determined whenever a URL to the content is created. In a simple website with no content reuse all contents only have a single context and link building is very simple. For more complex scenarios *Blueprint* includes a `ContextStrategy` for the following purposes:

- Generate a list of the available contexts for a content (the `ContextFinder`).
- Determine the most appropriate context for the specific link to be built (the `ContextSelector`).

The `DefaultContextStrategy` in *Blueprint* uses a list of `ContextFinders` to retrieve all possible contexts for a content item and a single `ContextSelector` to determine the most appropriate one from the list.

Most notably, there is a `ContextFinder` that utilizes special configuration contents, so-called "folder properties". Its logic to retrieve contexts is as follows:

1. Determine the folder of the content item.

2. Traverse the folder hierarchy starting from the folder in step 1 to the root folder looking for a content item of type `CMFolderProperties` named `_folderProperties`.
3. Return the contents of the linklist property `contexts` of the found `CMFolderProperties` content item.

The `ContextSelector` in *CoreMedia Blueprint* is the `NearestContextSelector`. From the list of possible contexts for a content it selects the context closest to the current context.

Navigation and Localization Workflows

When translating a navigation channel to a derived site, its parent channels are also considered for the translation set so that the navigation hierarchy remains intact. For configuration options and custom referrer relations see Section 4.5, "Referrer Relations in Localization" in *Multi-Site Manual* and Section 3.6.2, "Shared Multi-Site Configuration" in *Deployment Manual* as well as in Section 3.4.3, "Shared Multi-Site Configuration" in *Deployment Manual*.

5.4.3 Settings

Requirements

Editorial users must be able to adjust site behavior by editing content without the need to change the code base and redeploy the application. For example:

- Enable/disable comments for a certain section or the whole site.
- Set the number of dynamically determined related content items that are shown in an article detail view.
- Configure the refresh interval for content included from an external live source.

Administrative users must be able to adjust more technical settings through content, for example:

- Manage API keys for external services
- Image rendering settings
- Localization of message bundles

Solution

CoreMedia Blueprint uses Markup properties following the CoreMedia Struct XML grammar to store settings. Struct XML offers flexible ways to conveniently store typed key-value pairs where the keys are Strings and the values can be any of the following: String, Integer, Boolean, Link, Date, Secret, Struct (allows for nested sub Structs).

For more information on the Structs and CoreMedia Struct XML see Section 4.4.4, "Structs" in *Unified API Developer Manual*.

Settings can be defined on all content types inheriting from `CMLinkable`.

`localSettings`	
UI Name	Local Settings
Description	The settings defined specifically on this `CMLinkable`.
`linkedSettings`	
UI Name	Linked Settings
Description	A list of reusable `CMSettings` content items that contain a bundle of settings.

Table 5.9. Properties of CMLinkable for Settings Management

The local settings are easiest to edit. However, if you want to share common settings across multiple contents, you should spend the few extra steps to put them into a separate `Settings` content item and add it to the linked settings in order to facilitate maintenance and ensure consistency. Some projects make use of settings quite extensively.

Multiple `Settings` content items are a good instrument to structure settings of different aspects. You can still override single settings in the local settings, which have higher precedence.

The application also considers settings of the content's page context. If you declare a setting in a page, it is effective for all contents rendered in the context of this page.

Settings are inherited down the page hierarchy, so especially settings of the root page are effective for the whole site, unless they are overridden in a subpage or a content.

For more detailed information and customization of the settings lookup strategy see Section 4.4.1.2, "The Settings Service" [128] and the `SettingsService` related API documentation.

Settings as Java Resource Bundles

In a typical web application there is the need to separate text messages (such as form errors or link texts) from the rendering templates as well as rendering them according to a certain locale. The Spring framework provides a solution for these needs by the concepts of `org.springframework.context.MessageSource` for retrieving localized messages and by `org.springframework.web.servlet.LocaleResolver` for retrieving the current locale. Certain FTL macros such as `<@spring.showErrors%gt;` or `<@spring.message>` are built on top of these concepts.

In *CoreMedia Blueprint*, localized messages are stored as settings in Structs as described above and can be accessed as `java.util.ResourceBundle` instances.

A handler interceptor (`com.coremedia.blueprint.cae.web.i18n.ResourceBundleInterceptor`) is used to make these content backed messages (as well as the current locale) available to the rendering engine: They are extracted from the content and passed to a special Spring `MessageSource`, the `RequestMessageSource` by storing it in the current request. As a consequence, using FTL macros like `<@spring.message>` or `<@spring.showErrors>` will transparently make use of these messages.

5.4.4 Page Assembly

Requirements

Requirements

For a good user experience a website should not layout each and every page in a different fancy manner but limit itself to a few carefully designed styles. For example, most pages consist of two columns of ratio 75/25, where the left column shows the main content, and the right column provides some personalized recommendations.

In the best case an editor needs to care only for the content of a page, while the layout and collateral contents are added automatically, determined by the context of the content. However, there will always be some special pages, so the editors must be able to change the layout or the collateral contents. For example for a campaign page which features a new product they may omit the recommendations section and choose a simple one-column layout without any

distracting features. In order to preserve an overall design consistency of the site, editors are not supposed to create completely new layouts. They can only choose from a predefined set.

Solution

CoreMedia Blueprint addresses these requirements with the concept of a page grid and placements.

The page grid does not handle overall common page features such as navigation elements, headers, footers and the like. Those are implemented by Page templates with special views. Neither does the page grid control the layout of collections on overview pages. This is implemented by `CMCollection` templates with special views and view types.

Page grid defines layout of a page

You can think of a page grid as a table which defines the layout of a page with different sections. Each section has a link to a symbol content item which will later be used to associate content with the section. Technically, the layout of a page is defined in form of rows, columns and the ratio between them. A page grid contains no content and can be reused by different pages. So you might define three global page grids from which an editor can select one, for instance.

CMChannel contains content for page

The content for the page grid on the other hand, is defined in a `CMChannel` content item in so called placements, realized as link lists in structs. Each placement is associated with a specific position of the page grid through a link to a symbol content item. The editor can add content to the placement, collections for example, which will be shown at the associated position of the page grid.

Inheriting placements

Placements can also be shared between channels because a child inherits the placements of its parent. A prerequisite for inheritance is that the page grids of the parent and child page must have sections with the same name. For example, the parent channel has a two-column layout with the sections "main" and "sidebar". The child channel has a three-column layout with the sections "main", "sidebar" and "leftcolumn".

For the placements this means:

- The child must fill a placement with content for the "leftcolumn" section, because the parent has no such section.
- The child will override the placement for the "main" section with its content. Inheritance makes no sense for the "main" section.
- The child does not need to declare a "sidebar" placement but can inherit the "sidebar" placement of the parent, even though it uses a different layout.

Before going into the implementation details of the page grid, you will see how to work with page grids in *CoreMedia Studio*.

Page grids in CoreMedia Studio

Inheriting placements and locking

Editors can manage pages directly by editing the "placements" in the page grid in `CMChannel` content items (localized as `Page` in *CoreMedia Studio*). A placement is a specific area on a page such as the navigation bar, the main column or the right column.

A `CMChannel` can inherit page grid placements of its parent channel. For example, the Sports/Football section of a site can inherit the right column from the Sports section. Editors can also choose to "lock" certain placements and thus prevent subchannels from overwriting them. Each page grid editor provides a combo box to choose between different layouts for a page. Depending on the selected layout, placement may inherit their content if the same placement is defined in the layout of the parent page.

Layout of placement via view type

Each placement link list can configure a view type. The view type determines how the placement is rendered.

To define which placement view types are available for a page in some site, view type (`CMViewtype`) content items are placed in view type folders under a site-relative path or at global locations. The default paths are the site-relative path `Options/Viewtypes/` and the absolute path `/Settings/Options/Viewtypes/`. This can be configured via the application property `pagegrid.viewtype.paths`, which contains a comma-separated list of repository paths. Each path may start with a slash (`/`) to denote an absolute path or with a folder name to denote a path relative to a site root folder. When changing these values, please make sure that the existing view type content items are moved or copied to the new target location.

Web pages are represented in the CAE using the `com.coremedia.blueprint.common.contentbeans.Page` object which consists of two elements: the content to be rendered and the context in which to render the content.

Pages where the content to be rendered is the same as the context (for example, section overview) display the page grid of the context. Pages where content items (such as Articles) are displayed within a context use display the context's page grid but replace the "main" placement with the content item.

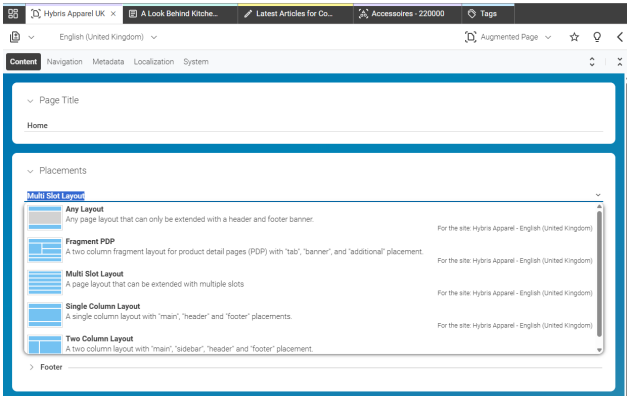

Figure 5.10. The page grid editor

Placement structure

Each placement contains a link list and several additional buttons on top of it. The order of the linked elements can be modified using drag and drop.

Inheriting content from parent page

Instead of adding own content, a placement can inherit the linked content from a parent's page placement. If you inherit the content, you cannot edit the placement in the child page. You have to deactivate the "override" button to change the content of the placement.

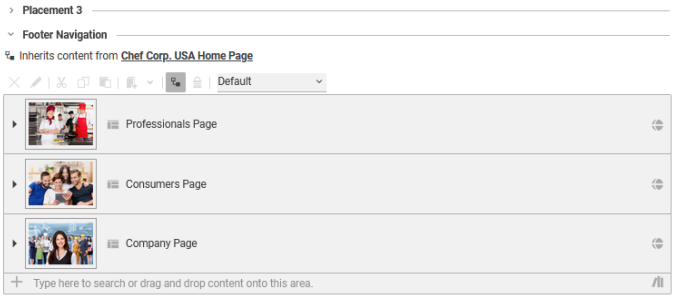

Figure 5.11. An inheriting placement

Locking placement

A placement can be locked using the "lock" button. In this case all child placements are not able to overwrite this placement with own content.

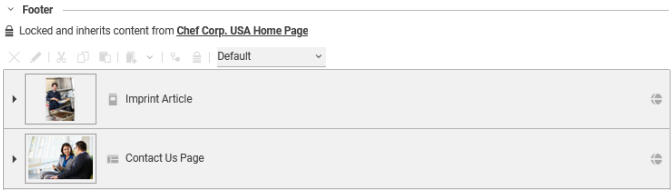

Figure 5.12. A locked placement

The page grid editor provides a combo box with predefined layouts to apply to the current page. After changing the layout, the *Studio* preview will immediately reflect the new page layout.

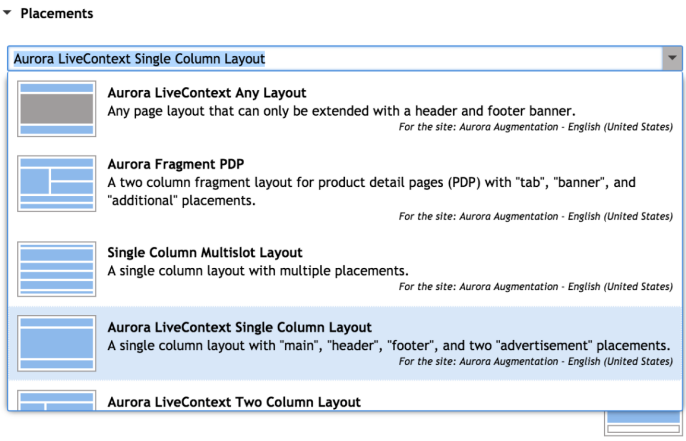

Figure 5.13. The layout chooser combo box

Inconsistency between parent and child page grid

The layout of a parent page grid may be changed so that it does not fit anymore with the layout of a child page which inherits some settings. A child may use a three-column layout and inherit most of its content from its parent page that also uses a three-column layout. Then, the layout of the parent may be changed to another layout with a single column that doesn't contain any of the needed layout sections. The child configuration is invalid in this case and the user has to reconfigure all child pages.

No check for inconsistency

Currently there is no kind of detection for these cases in *Studio*, so the user has to check manually if the child configurations are still valid.

How to configure a page grid editor

The Blueprint base module `bpbase-pagegrid-studio-plugin` provides an implementation of the page grid editor shown above through the config class

`pageGridPropertyField` in the package `com.coremedia.blueprint.base.pagegrid.config`. In many cases, you can simply use this component in a content item form by setting only the standard configuration attributes `bindTo`, `forceReadOnlyValueExpression`, and `propertyName`

If you want to adapt the columns shown in the link list editors for the individual section, you can also provide fields and columns using the attributes `fields` and `columns`, respectively. The semantics of these attributes match those of the `linkListPropertyField` component.

How to configure the layout location

Pages look up layouts from global and site specific folders. By default, the site specific page grid layout path will point to `Options/Settings/Pagegrid/Layouts` and the global one to `/Settings/Options/Settings/Pagegrid/Layouts`. This can be changed via the application property `pagegrid.layout.paths`, which contains a comma-separated list of repository paths. Each path may start with a slash ('`/`') to denote an absolute path or with a folder name to denote a path relative to a site root folder. When changing these values, please make sure that the existing page layout content items are moved or copied to the new target location. Also, mind that when looking for the default page layout (see below), paths mentioned first take precedence, so it usually makes sense to start with site-relative paths and continue with absolute paths.

CAUTION

The default layout settings content item `PagegridNavigation` must be present in at least one of the available layout folders. The page grid editor will show an error message if the content item is not found.

CAUTION

If several layout folders are used, make sure that the layout settings content items have unique names.

How to configure a new layout

Every `CMSettings` content item in a layout folder is recognized as a layout definition. The `settings` struct property defines a table layout with different sections. The struct defines two integer properties with the overall row and

column count. The struct data may also contain two string properties `name` and `description`, which are used for the localization of page grid layout content items (see section "How to localize page grid objects" [188]).

The `items` property contains a list of substructs, each defining a section of the page grid. The order in which the sections appear in the struct list matches the order in which the link lists of the individual sections are shown by the page grid editor.

The sections are represented by `CMSymbol` content items. The layout definition is inspired by the HTML table model, even though *CoreMedia Blueprint*'s default templates do not render page grids as HTML tables but with CSS means. The sections support the following attributes:

- *`col`*: The column number where the section is placed or, if the *`colspan`* attribute is set, the column number of the leftmost part of the section.
- *`row`*: The row number where the section is placed or, if the *`rowspan`* attribute is set, the row number of the topmost part of the section.
- *`colspan`*: The number of columns spanned by the section.
- *`rowspan`*: The number of rows spanned by the section.
- *`width`*: The width of this section in percent of the total width.
- *`height`*: The height of this section in percent of the total height.

The *`col`*, *`row`* and *`rowspan`* attributes of the section must match the grid layout defined by the *`colCount`* and *`colRow`* attributes (see Example 5.1, "Pagegrid example definition" [183]). That is, when *`colCount`* and *`colRow`* are "3" and "4", for example, then you have 12 cells in the page grid table layout which must all be filled by the sections. No cell can be left empty, and no section can overlap with other sections.

The height attribute is only used for the preview of the layout in the page form. It has no impact on the delivered website.

The default `PagegridNavigation` layout settings content item with a 75%/25% two column layout looks as follows:

```
<Struct xmlns="http://www.coremedia.com/2008/struct"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <IntProperty Name="colCount">2</IntProperty>
  <IntProperty Name="rowCount">1</IntProperty>
  <StructListProperty Name="items">
    <Struct>
      <LinkProperty Name="section" xlink:href="coremedia:///cap/content/550"
 LinkType="coremedia:///cap/contenttype/CMSymbol"/>
      <IntProperty Name="row">1</IntProperty>
      <IntProperty Name="col">1</IntProperty>
      <IntProperty Name="height">100</IntProperty>
      <IntProperty Name="width">75</IntProperty>
      <IntProperty Name="colspan">1</IntProperty>
    </Struct>
    <Struct>
      <LinkProperty Name="section" xlink:href="coremedia:///cap/content/544"
```

```
 LinkType="coremedia:///cap/contenttype/CMSymbol"/>
      <IntProperty Name="row">1</IntProperty>
      <IntProperty Name="col">2</IntProperty>
      <IntProperty Name="height">100</IntProperty>
      <IntProperty Name="width">25</IntProperty>
      <IntProperty Name="colspan">1</IntProperty>
    </Struct>
  </StructListProperty>
  <StringProperty Name="name">2-Column Layout (75%, 25%)</StringProperty>
  <StringProperty Name="description">Two column layout with main and sidebar
 sections</StringProperty>
</Struct>
```

Example 5.1. Pagegrid example definition

> **CAUTION**
>
> The main content of a content item will always be rendered into the `main` section of a layout. Therefore, every layout must define a `main` section.

How to configure a read-only placement

The page grid layout definition provides the possibility to declare a read-only section. Such sections are typically filled with content from third-party integrations. If unspecified, a section is editable. In order to disable editing, you have to declare the Boolean property `editable` for the `struct` element of the corresponding section and set it to "false", for example:

```
<Struct>
  <LinkProperty Name="section" xlink:href="coremedia:///cap/content/120"
LinkType="coremedia:///cap/contenttype/CMSymbol"/>
  <IntProperty Name="row">2</IntProperty>
  <IntProperty Name="col">1</IntProperty>
  <IntProperty Name="colspan">1</IntProperty>
  <IntProperty Name="height">75</IntProperty>
  <IntProperty Name="width">25</IntProperty>
  <BooleanProperty Name="editable">false</BooleanProperty>
</Struct>
```

The section that matches the given symbol will be shown as disabled in *Studio*. The matching placements will not appear in the editor.

How to disable the inheritance of the placement

The page grid layout definition provides the possibility to declare a section for which the inheritance of placements is disabled. For such section the placement will be never inherited from parent but must be provided for each children. In order to disable the inheritance, you have to declare the Boolean property `disableInheritance` for the `struct` element of the corresponding section and set it to "true", for example:

```
<Struct>
  <LinkProperty Name="section" xlink:href="coremedia:///cap/content/120"
LinkType="coremedia:///cap/contenttype/CMSymbol"/>
```

```
  <IntProperty Name="row">2</IntProperty>
  <IntProperty Name="col">1</IntProperty>
  <IntProperty Name="colspan">1</IntProperty>
  <IntProperty Name="height">75</IntProperty>
  <IntProperty Name="width">25</IntProperty>
  <BooleanProperty Name="disableInheritance">true</BooleanProperty>
</Struct>
```

For the section that matches the given symbol the inheritance and locking in the *Studio* are disabled.

You can also disable the inheritance for all sections for a given layout by declaring the Boolean property `disableInheritance` to 'true' in the first level as shown in the following example:

```
<Struct xmlns="http://www.coremedia.com/2008/struct"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <IntProperty Name="colCount">2</IntProperty>
  <IntProperty Name="rowCount">1</IntProperty>
  <BooleanProperty Name="disableInheritance">true</BooleanProperty>
  <StructListProperty Name="items">
  <Struct>
  <LinkProperty Name="section" xlink:href="coremedia:///cap/content/550"
LinkType="coremedia:///cap/contenttype/CMSymbol"/>
  <IntProperty Name="row">1</IntProperty>
  <IntProperty Name="col">1</IntProperty>
  <IntProperty Name="height">100</IntProperty>
  <IntProperty Name="width">75</IntProperty>
  <IntProperty Name="colspan">1</IntProperty>
  </Struct>
  <Struct>
  <LinkProperty Name="section" xlink:href="coremedia:///cap/content/544"
LinkType="coremedia:///cap/contenttype/CMSymbol"/>
  <IntProperty Name="row">1</IntProperty>
  <IntProperty Name="col">2</IntProperty>
  <IntProperty Name="height">100</IntProperty>
  <IntProperty Name="width">25</IntProperty>
  <IntProperty Name="colspan">1</IntProperty>
  </Struct>
  </StructListProperty>
  <StringProperty Name="name">2-Column Layout (75%, 25%)</StringProperty>
  <StringProperty Name="description">Two column layout with main and sidebar
 sections</StringProperty>
  </Struct>
```

How to disable the default inheritance of the placement

For a new page grid the placement is per default inherited from a parent. You can declare a section for which the inheritance of placements is not default. For such section the placement will be empty first before you activate the inheritance or fill the placement on your own. In order to make the non-inheritance default, you have to declare the Boolean property `defaultInheritance` for the `struct` element of the corresponding section and set it to "false", for example:

```
<Struct>
    <LinkProperty Name="section" xlink:href="coremedia:///cap/content/120"
LinkType="coremedia:///cap/contenttype/CMSymbol"/>
    <IntProperty Name="row">2</IntProperty>
    <IntProperty Name="col">1</IntProperty>
    <IntProperty Name="colspan">1</IntProperty>
    <IntProperty Name="height">75</IntProperty>
```

```
    <IntProperty Name="width">25</IntProperty>
    <BooleanProperty Name="defaultInheritance">false</BooleanProperty>
    </Struct>
```

You can also make the non-inheritance default for all sections for a given layout by declaring the Boolean property `defaultInheritance` to 'false' in the first level - similar as the `disableInheritance`.

CAUTION

For an existing layout which is used for many page grids changing the flags `disableInheritance` and `defaultInheritance` will change the contents of the page grids. Especially if the page grids are indexed by the *CAE Feeder* they all be re-indexed. Use the flags with caution.

How to populate a page grid with content

Page grids are defined in the struct property `CMNavigation.placement` of a channel. Such structs are typically created using the page grid editor shown above. Example:

```
<Struct xmlns="http://www.coremedia.com/2008/struct"
xmlns:xlink="http://www.w3.org/1999/xlink">
  <StructListProperty Name="placements">
    <Struct>
      <LinkProperty Name="section"
LinkType="coremedia:///cap/contenttype/CMSymbol"
xlink:href="coremedia:///cap/content/550"/>
      <LinkProperty Name="viewtype"
LinkType="coremedia:///cap/contenttype/CMViewtype"
xlink:href="coremedia:///cap/content/1784"/>
      <LinkListProperty Name="items"
LinkType="coremedia:///cap/contenttype/CMArticle">
        <Link xlink:href="coremedia:///cap/content/134"/>
        <Link xlink:href="coremedia:///cap/content/498"/>
      </LinkListProperty>
    </Struct>
    <Struct>
      <LinkProperty Name="section"
LinkType="coremedia:///cap/contenttype/CMSymbol"
xlink:href="coremedia:///cap/content/544"/>
      <LinkListProperty Name="items"
LinkType="coremedia:///cap/contenttype/CMArticle">
        <Link xlink:href="coremedia:///cap/content/776"/>
      </LinkListProperty>
    </Struct>
  </StructListProperty>
  <StructProperty Name="placements_2">
    <Struct>
      <LinkProperty Name="layout"
LinkType="coremedia:///cap/contenttype/CMLayout"
xlink:href="coremedia:///cap/content/3488"/>
    </Struct>
```

```
  </StructProperty>
</Struct>
```

A placement struct contains a list of section structs `placements`. The `placements_2` struct contains another struct, `placements` and a link property `layout`, which determines the layout for this channel.

The `placements` struct property consists of substructs for the single placements, each of which refers to a section and lists its contents in the items property. Additionally, each placement can declare a view type.

Layouts and placements are connected by the section content items. Let's assume you have two sections, "main" and "sidebar". Your channel declares some latest news for the main section and some personalized recommendations for the sidebar. The layout definition consists of one row with two columns, the left column refers to the "main" section, the right column refers to the "sidebar". This will make your channel be rendered with the main content left and the recommendations on the right. If you don't like it, you can simply choose another layout, for example with a different width ratio of the columns or with the sidebar left to the main section.

The rendering of a page grid is layout-driven, because the sections of the table-like layout model must be passed to the template in an order which is suitable for the output format (usually HTML). CoreMedia Blueprint's web application processes a page grid as follows:

1. The `PageGridServiceImpl` determines the layout content item of the channel. If there is no layout link in the `placements_2` struct, a fallback content item `PagegridNavigation` is used. This name can be configured by setting the application property `pagegrid.layout.defaultName`. The fallback layout content item can be located in any of the configured layout folders (see "layout locations"), usually it will be located under the site relative path `Options/Settings/Pagegrid/Layouts`. The layout definition is evaluated and modeled by a `ContentBackedStyleGrid`.
2. The `PageGridServiceImpl` collects the placements of the channel itself and the parent channel hierarchy. The precedence is obvious, for example a channel's own placement for a section ("sidebar" for instance) overrides an ancestor's placement for that section.
3. Both layout and placements are composed in a `ContentBackedPageGrid` which is the backing data for a `PageGridImpl`. `PageGridImpl` implements the `PageGrid` interface and prepares the data of the `ContentBackedPageGrid` for access by the templates. Basically
   - it wraps the content of the placements into content beans,
   - it arranges the placements in rows and columns, according to the layout

- it replaces the channel's main placement with the requested content.

Blueprint's default templates (namely `PageGrid.ftl`) do not render page grids as HTML tables but as nested elements and suitable CSS styles. The beginning of a rendered page grid looks like this:

```
<div id="row1" class="row">
    <div id="main" class="col1 column col1of2 width67">
```

The outer elements represent the rows of the page grid, the inner elements represent the columns. The ids of the rows are generated by the template as an enumeration. The ids of the columns are the section names of the placements. The column elements are rendered with several class attributes:

- `column`: A general attribute for column elements
- `col1`: The absolute index of the column in its row
- `col1of2`: The *colspan* of this column (1) and the absolute number of columns of the page grid (2)
- `width67`: The relative width of this column

You can use these attributes to define appropriate styles for the columns. *CoreMedia Blueprint's* default CSS provides styles which reflect the width ratios of some typical multi-column layouts. You find them in the content item `/Themes/basic/css/basic.css` in the content repository where you can enhance or adapt them to your needs.

In the inner elements the placements are included, and their section names determine the views. For example a "sidebar" placement is included by the `PageGridPlacement.sidebar.ftl` template.

How to localize page grid objects

To localize a layout name, create a resource bundle entry with the key `<layout name>_text` in the resource bundle `PageGridLayouts_properties`, where `<layoutname>` is the name of the layout content item or, preferably, the `name` property of the settings struct of the layout. Similarly, a layout description can be localized with entries of the form `<layoutname>_description`. If no corresponding resource bundle entries are found, the `description` property of the settings struct of the layout is used. If that property is empty, too, the name is used as the description. The resource bundle is available in the package `com.coremedia.blueprint.base.pagegrid` of module `bp base-pagegrid-studio-plugin`.

For the purposes of localization, placements are treated as pseudo-properties and localized according to the standard rules for content properties as described

in the Studio Developer Manual. The name of the pseudo-property is `<structname>-<placementname>`, where `<structname>` is the name of the struct property storing the page grid and `<placementname>` is the name of the section content item. For example, a placement with the name `main` that is referred from the standard page grid struct `placement` of a `CMChannel` content item would obtain its localization using the key `CMChannel_placement-main_text`. You can add localization entries to the resource bundle `BlueprintDocumentTypes_properties` of module `blueprint-forms`, which is applied to the built-in resource bundle `ContentTypes_properties` at runtime.

To localize a view type name or a view description, you can add a property `<viewtypename>_text` or `<viewtypename>_description` to the bundle `Viewtypes_properties`. Here `<viewtypename>` is the name of the view type content item or, preferably, the string stored in its `layout` property. Because view types are also used in other contexts, this bundle has been placed in the package `com.coremedia.blueprint.base.components.viewtypes` of module `bpbase-studio-components`.

CoreMedia Blueprint defines three resource bundles `BlueprintPageGridLayouts_properties`, `BlueprintPlacements_properties`, and `BlueprintViewtypes_properties`. Entries of these bundles are copied to the bundles described above, providing a convenient way to add custom entries.

5.4.5 Overwriting Product Teaser Images

> NOTE
>
> Feature is only supported in *eCommerce Blueprint*

Requirements

You have put a product teaser on your home page, which is displayed with the default product image coming from the eCommerce system but you want to highlight that teaser by changing its default image to a more engaging one.

Solution

CoreMedia Content Cloud allows you to either use the content from the eCommerce database or overwrite this image with your own image in the `Teaser` content type.

5.4.6 Content Lists

Requirements

Websites frequently display content items that share certain characteristics as lists, for example, the top stories of the day, the latest press releases, the best rated articles or the recommended products. Some of these lists are managed editorially while others should be compiled dynamically by business rules defined by editors. It is a common requirement to reuse these content lists across different web pages and use common functionality to place lists on pages and assign different layouts to lists.

Solution

CoreMedia Blueprint defines different content types for lists of content which differ in how they determine the content items. Leveraging CoreMedia's object oriented content modeling these lists can reuse view templates and can be placed interchangeably on web pages.

Type	Purpose
`CMCollection`	A common base type for lists, which all other list types extend. It provides functionality for editorially managed lists.
`CMGallery`	A distinct content type for lists of `CMMedia` content items which should be displayed as a gallery.
`CMQueryList`	Dynamic lists that are based on content metadata, such as "latest 5 articles in sport".
`ESDynamicList (part of Elastic Social)`	Dynamic lists that are based on Elastic Social metadata, such as "5 best rated articles in news."

Table 5.10. Collection Types in CoreMedia Blueprint

5.4.7 View Types

Requirements

A common pattern for CoreMedia projects is to reuse content and display the same content item on various pages in different layouts and view variants. A content list, for example, could be rendered as simple bulletin list or as a list of teasers with thumbnails. Similarly, an article can be displayed in a default ("full") view or as a teaser.

Usually the rendering layer decides what view should be applied to a content item in different use cases. For example, the view rendering results of a search on the website could use the `asListItem` view to render the found items.

Editors still need a varying degree of control to influence the visual appearance of content in specific cases. They might want to decide whether a list of content items should be displayed as a teaser list or a collapsible accordion on a page, for example.

Solution

A dedicated content type called `CMViewtype` is available that can be associated with all `CMLinkable` content types.

During view lookup a special `com.coremedia.objectserver.view.RenderNodeDecorator`, the `ViewTypeRenderNodeDecorator`, augments the view name by the `layout` property of the view type referenced by the content item.

The `BlueprintViewLookupTraversal` then evaluates this special view name and falls back to the default view name without the view type if the view could not be resolved.

In the example above the template responsible for rendering search results would include all found content with the `asListItem` view. If the content is of type `CMArticle` there would be a lookup for a `CMArticle.asListItem.ftl` (among others in the content object's type hierarchy, see Section 4.3.3, "Views" in *Content Application Developer Manual* for more CoreMedia's object-oriented view dispatching). If the article has a view type assigned (such as *breakingnews*) there would be a lookup for `CMArticle.asListItem[breakingnews].ftl` before falling back to `CMArticle.asListItem.ftl`. This allows for very fine-grained editorially driven layout selection for any created content.

Selecting a view type in CoreMedia Studio

You can use the view type selector which is associated with the view type property to select a specific view type for a content item, a collection for instance. The view type selector is implemented as a combo box providing an icon preview and a description text about the view type. View types can be defined globally or site specific. If the view type item is configured for a site, the name of the site is also displayed in the combo box item.

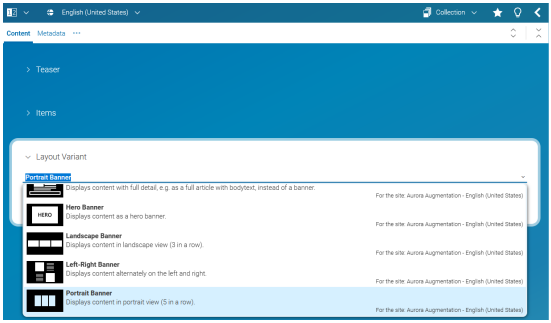

Figure 5.14. Layout Variant selector

How to configure a view type selector

There are several content forms that include the view type selector form. The `ViewTypeSelectorForm` bundles the view type selector combo box and its configuration parameters. The parameter `paths` defines which items are shown in the combo box. The combo box assumes that each of the items (`CMViewtype` here) has a property `icon` that contains the thumbnail view of the view type.

If no paths are defined (default), folders based on the content type and hierarchy are used for lookup.

```
<bpforms:ViewTypeSelectorForm propertyName="viewtype"
  paths="{['/Settings/Options/Viewtypes/CMTeasable',
  'Options/Viewtypes/CMTeasable']}"/>
```

In this example all `CMViewtype` content items of the folders `/Settings/Options/Viewtypes/CMTeasable` and `Options/Viewtypes/CMTeasable` (depending on site) are shown in the view type selector combo (without content type hierarchy lookup).

An additional view type selector form is the class `ContainerViewTypeSelectorForm`. It inherits from `ViewTypeSelectorForm` and sets

the `paths` parameter to `['/Settings/Options/Viewtypes/Container/', 'Options/Viewtypes/Container/']`. The same folders are used for the PageGrid placement view type selector.

The following content types extend `ContainerViewTypeSelectorForm` with a lookup for paths based on the content type name:

- `CMCollection`
- `CMProductList`
- `CMQueryList`
- `ESDynamicList`

How to localize view types for the view type selector

The view type selector displays two fields of a view type: The name (which is the name of the content item in the repository) and the `description` property. These string can be localized as described earlier in section "How to localize page grid objects" [188].

5.4.8 CMS Catalog

Requirements

Some companies do not run an online store. They do not need a fully featured shopping system. Nonetheless, they want to promote some products on their corporate site.

Solution

CoreMedia Content Cloud provides the *CMS Catalog*, an implementation of the *eCommerce API*, which is backed only by the CMS and does not need a third-party eCommerce system. It allows maintaining a smaller number of products and categories for presentation on the website. It does not support shopping features like availability or payment. The *CMS Catalog* is based on *Blueprint* features. It is already integrated in the `Corporate` extension, so you can use it out of the box.

Maven Module	Description
`com.coremedia.blueprint.base:bpbase-ecommerce`	Contains the *eCommerce API* implementation for the CMS. The implementation is content type independent.

Maven Module	Description
`com.coremedia.blueprint:ecommerce`	Contains the content types, content beans and the studio catalog component.
`com.coremedia.blueprint:corporate`	Example usage of the catalog in the corporate page.

Table 5.11. CMS Catalog: Maven parent modules

Content Types

In the *CMS Catalog* products and categories are modeled as content. There are two new content types, `CMProduct` and `CMCategory`, which extend the well known *Blueprint* content types `CMTeasable` and `CMChannel`, respectively. So you can seamlessly integrate categories into your navigation hierarchy and place products on your pages, just like any other content. In order to activate the new content types you have to add a Maven runtime dependency on the `catalog-doctypes` module to your Content Server components.

Content Beans

The modules `catalog-contentbeans-api` and `catalog-contentbeans-lib` provide content beans for `CMProduct` and `CMCategory`. The content beans integrate into the class hierarchy according to their content types, that is they extend `CMTeasable` and `CMChannel`, respectively. The content beans do not implement the *eCommerce API* interfaces `Product` and `Category`, though. Instead, they provide delegates via `getProduct` and `getCategory` methods. While this may look inconvenient at first glance, it has some advantages concerning flexibility:

- The content bean interfaces remain independent of future changes in the *eCommerce API*.
- You have better control over the view lookup by explicitly including the content bean or the delegate.

Configuration

First, you need three settings in the root channel to activate a CMS Catalog for your site. *Blueprint Base* provides a commerce connection named cms1 which is backed by the content repository. You can activate this connection by the `livecontext.connectionId` setting. Moreover, your catalog needs a

name, which is specified by the `livecontext.store.name` setting. Finally, your catalog needs a root category, which is specified by the `livecontext.rootCategory` setting. In case you didn't choose a root category, you need to reload the site to complete the linking of the settings to the site.

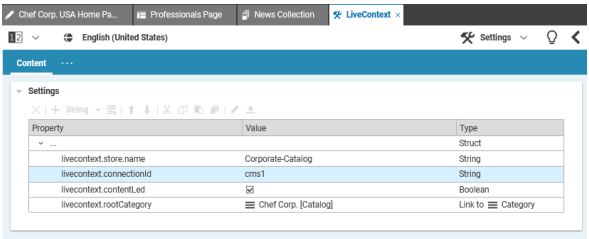

Figure 5.15. CMS Catalog Settings

Although the catalog indicator is a `CMCategory` content item, it does not represent a category but serves only as a technical container for the actual top categories (see *eCommerce API*, `CatalogService#findTopCategories`). The concept resembles the site indicator, which is the point of entry to the navigation without being part of it.

While the product → category relation is modeled explicitly with the `contexts` link list, the reverse relation uses the search engine. Therefore, you need to extend the *contentfeeder* component with some Spring configuration from the `bpbase-ec-cms-contentfeeder-lib` module:

```
<import resource="classpath:/framework/spring/bpbase-ec-cms-contentfeeder.xml"/>
```

Templating

You can use both, `Product` or `CMProduct` templates. You can also use a mixture of both for different views or fallback to `CMTeasable` templates for views that do not involve `CMProduct` specific features.

Using `Product` templates you can easily switch to a third-party eCommerce system later, since the interface remains the same. Otherwise, you are more flexible with `CMProduct` templates:

- You can easily enhance the `CMProduct` content type and interface and access the new features immediately.
- You benefit from all the inherited features (like multi-language) and fallback capabilities along the content type driven interface hierarchy.

- You can easily switch from `CMProduct` to `Product` just by calling `CMProduct#getProduct` anywhere you need a `Product` object. The reverse direction is more cumbersome.

5.4.9 Teaser Management

Requirements

Most websites present short content snippets as "teasers" on various pages. Content and layout for teasers should be flexible but manageable with minimum effort.

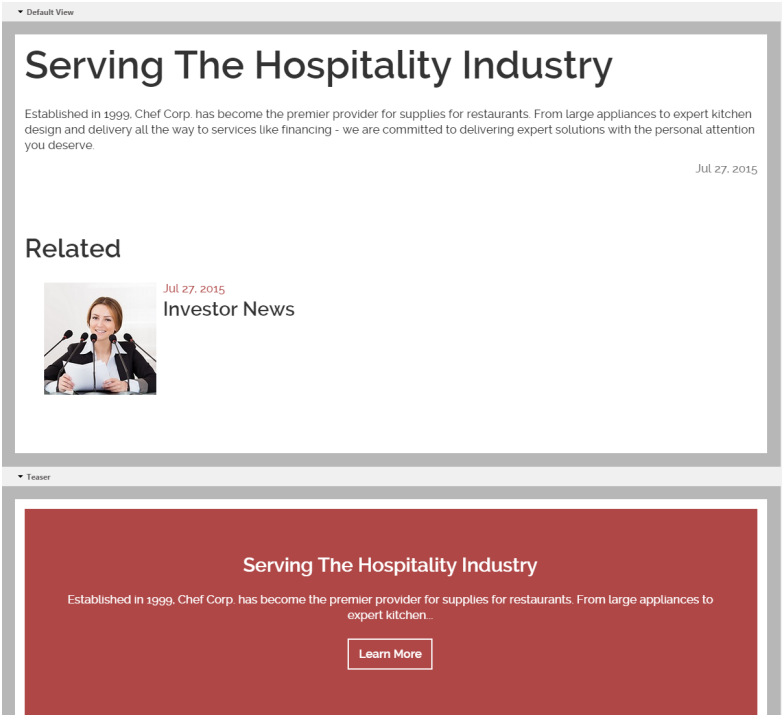

Figure 5.16. Default view and teaser view of an Article

It is a common requirement to automatically derive abbreviated content teasers without the need to duplicate any content items. In some cases, editors wish to

create distinct teasers for a content item that don't reuse any information from that item.

Example: An editor wants to point to an article using a specific image that is not part of that article. Or: An editor wants to promote an article on a page with a teaser that is not the default teaser (using different text, image, or layout).

Solution

In *CoreMedia Blueprint* all content types for content and pages extend from the abstract content type `CMTeasable`. It defines common properties and business rules which provide all types inheriting from `CMTeasable` with a default behavior when displayed as a teaser.

Type	Purpose
`teaserTitle`	The title of the content item when displayed as a teaser.
`teaserText`	The text of the content item when displayed as a teaser.

Table 5.12. Properties of `CMTeasable`

Fallbacks to automatically display the shorter teaser variant of properties are implemented in the content bean implementation for `CMTeasable`. For example, the `teaserText` of a content reverts to the `detailText` if no `teaserText` has been entered by an author.

For distinct teasers *CoreMedia Blueprint* includes a `CMTeaser` content type that can be used for this purpose. It provides all properties required to display a teaser and can be linked to the content that it promotes. Teasers without a link are also supported to create non-interactive brand promotions etc.

5.4.10 Dynamic Templating

Requirements

In order to quickly implement microsites, campaigns, or specialized channels with unique template requirements, templates can be updated without interrupting the service or requiring a redeployment of the application.

Solution

Views can be implemented as FreeMarker templates and uploaded to the Content Repository in a container file, preferably a JAR. For details, consult the Section 4.3.3.4, "Loading Templates from the Content Repository" in *Content Application Developer Manual*.

Prerequisites

In order for the CAE to find the FreeMarker templates, the property `delivery.local-resources` must be set to "false".

Create the archive containing the templates

A template set archive, preferably a JAR file, can contain FreeMarker templates which must be located under the path: `/META-INF/resources/WEB-INF/templates/siteName/packageName/`

The easiest way to create the JAR is to create a new Maven module with a POM like this one:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.example.groupId</groupId>
  <artifactId>templates</artifactId>
  <version>--insert version here--</version>
  <packaging>jar</packaging>
  <description>
    CAE templates to be uploaded to a CMTemplateSet document in
    /Themes/*my.package*/templates/ with name *my.package*-templates.jar.

    Use the *my.package* as a reference in a Page's
    "viewRepositoryNames" settings (list of strings).
  </description>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>
        <configuration>
          <archive>
            <addMavenDescriptor>true</addMavenDescriptor>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Put your templates below the path `src/main/resources/META-INF/resources/WEB-INF/templates/--themeName--/--package`

`Name--/`, for example `src/main/resources/META-INF/resources/WEB-INF/templates/corporate/com.coremedia.blueprint.common.contentbeans/Page.ftl`

Upload the template set

CoreMedia Blueprint provides the content type `Template Set` (`CMTemplateSet`) which is used for this purpose. Create a content item of type `Template Set` in folder `/Themes/--themeName--/templates` and upload the JAR to its `archive` property. Its name is significant and is used to reference template sets from channel settings, as explained see below.

Name	Description
`description`	A description of the purpose / contents of the code.
`archive`	blob property that contains the archive (preferably a JAR) that contains the templates.

Table 5.13. Properties of `CMTemplateSet`

Add the template set to a page

A Page context can be configured to add additional template sets to all pages rendered in its context. The names of additional template sets are configured in a string list setting `viewRepositoryNames` of a Page. Like all settings, a Page will inherit this list of names form its parent context, if it is not set. See Section 5.4.11, “View Repositories” [200] for more details.

NOTE

The CAE will resolve view repository names automatically according to the predefined name pattern. For instance, if a Page sets its `viewRepositoryNames` to the list **`["christmas", "campaigns"]`**, each page rendered in this context will use templates implemented in the Template Sets `/Themes/christmas/templates/christmas-templates.jar` and `/Themes/campaigns/templates/campaigns-templates.jar` before falling back to the default templates defined for the web application.

5.4.11 View Repositories

Requirements

A CoreMedia deployment can host multiple sites which frequently differ in layout and functionality. It is a common requirement to use different view templates for those sites but still be able to define reused templates across sites flexibly.

Solution

The *CoreMedia CAE* offers a very flexible view selection mechanism by providing the `ViewRepositoryNameProvider` and `ViewRepositoryProvider` abstraction (see Section 4.3.3, "Views" in *Content Application Developer Manual*).

CoreMedia Blueprint offers the `BlueprintViewRepositoryNameProvider` implementation which for each lookup of model and view generates a list of view repository names to query. The list is created based on

- the specific view repository names defined in the String list setting `viewRepositoryNames` of the navigation context of the provided model,
- the view repository names defined via Spring in the property `commonViewRepositoryNames` on the `BlueprintViewRepositoryNameProvider` Java bean.

This allows for more fine-grained control of the used view repositories as view repositories can be configured not only specific for a site but also for each site section.

CoreMedia Blueprint uses the standard CAE `TemplateViewRepositoryProvider` to create from the list of view repository names the list of actual view repositories to query. *CoreMedia Blueprint* configures the following `templateLocationPatterns` for the `TemplateViewRepositoryProvider`:

- `jar:id:contentproperty:/Themes/%1$s/templates/%1$s-templates.jar/archive!/META-INF/resources/WEB-INF/templates/%1$s`
- `jar:id:contentproperty:/Themes/%1$s/templates/%1$s-templates.jar/archive!/META-INF/resources/WEB-INF/templates/sites/%1$s`
- `/WEB-INF/templates/sites/%s`
- `/WEB-INF/templates/%s`

Example: For a content of the corporate site the `BlueprintViewRepositoryNameProvider` yields the view repository names "corporate". The `TemplateViewRepositoryProvider` would then return the following view repositories which are queried for the responsible view:

- A FreeMarker template view repository in the CMS located in the `/Themes/corporate/templates/corporate-templates.jar` (a `CMTemplateSet`) content item's blob property `archive`
- A FreeMarker Template file system view repository below `/WEB-INF/templates/sites/corporate`
- A FreeMarker Template file system view repository below `/WEB-INF/templates/corporate`

5.4.12 Client Code Delivery

Requirements

Client code such as JavaScript and CSS is changing more rapidly than frontend templates and backend business rules. To deliver JS and CSS changes conveniently it is a common pattern to consider those as content and use the common editorial workflow (create, approve, publish) to deploy these to the live environment.

Solution

CoreMedia Blueprint provides the content types `CMCSS` and `CMJavaScript` which both inherit from the common super type `CMAbstractCode`.

Name	Description
`description`	A description of the purpose / contents of the code.
`code`	The code stored in a CoreMedia XML property following the CoreMedia RichText schema. This allows for embedding images directly in a code fragment and enables quick fixes of client code in the standard CoreMedia editing tools.
`include`	Other code elements that should be deployed together with this one.

Name	Description
`dataUrl`	An (optional) URL of the code on an external system. Allows to also manage all code included from third-party servers as if it was part of the CoreMedia repository.

Table 5.14. Client Code - Properties of `CMAbstractCode`

Client code is associated with themes or site sections. `CMTheme` and `CMNavigation` content items contain references to the CSS and JavaScript items to be used within the section. Child sections inherit code from their parent if this code is defined in a theme. They can extend it to refine their section layout. This enables editorial users to quickly associate new design to sections that stand out from the rest of the page, or even roll out a site wide face lift without having to redeploy the application itself.

NOTE

CSS and JavaScript added to a page will only apply to this page and will not be inherited. To apply layout changes to all subpages of a page, it is recommended to create a new theme.

Additional web resources for preview and fragment preview

Additional resources for preview

Settings to add resources for preview

NOTE

For preview and fragment preview settings and resources it is recommended to manage them in the theme, since it is now possible to define settings there. For more information see Frontend Developer Manual.

Additional CSS and JavaScript can be added to sites for use in *CoreMedia Studio* and the embedded preview. CSS will be included in `Page._additionalHead.ftl` and JavaScript in `Page._bodyEnd.ftl` after the regular web resources.

The settings are organized as linklist properties. The name of the `linklist` for CSS itself must be `previewCss` and `previewJs` for JavaScript. The settings must be attached to the root channel of a site.

> **WARNING**
>
> In earlier versions the `css/preview.css` and `js/preview.js` of the theme were attached via this setting as well. This is no longer needed as the theme build mechanism will handle adding preview related resources itself.

Additional resources for fragment preview

Settings to add resources for fragment preview

Additional CSS and JavaScript can be added to sites for use in *CoreMedia Studio* and the embedded preview for fragments, for example, Articles. CSS will be included in `Page._additionalHead.ftl` and JavaScript in `Page._bodyEnd.ftl` before the regular web resources.

The settings are organized as linklist properties. The name of the `linklist` for CSS itself must be `fragmentPreviewCss` and `fragmentPreviewJs` for JavaScript. The settings must be attached to the root channel of a site.

> **NOTE**
>
> Keep in mind: The CSS and JavaScript for preview are loaded **after** the regular web resources and the ones for the fragment preview are loaded **before** them! Both additional web resources can be combined.

Web Performance Optimization

Besides the concepts for managing and deploying client code from within the content repository, *CoreMedia Blueprint* also features mechanisms to both speed up site loading and reduce request overhead during the delivery of web resources.

Reducing the overhead of both client request count and data transfer sizes for client codes and web resources such as JavaScript and/or CSS.

Merging

CoreMedia Blueprint offers a merging process which merges all JavaScript and CSS files into a single one each.

The process of merging only applies to source files, that don't have a Data URL property. If a Data URL is set, the file will be skipped and result in each file rendered separately into the source code of the page.

Configure merging

To enable merging of JavaScript and CSS files you can use the property `cae.merge-code-resources` switch to control the behavior. The default is false.

For debugging purposes during the development, it might come in handy to disable the merging feature. You do that by turning on the `delivery.developer-mode` property switch, either provided with a standard property file, or via a Maven switch. Inside the `cae-preview-webapp` module, all you have to do is to start the preview CAE web application locally using the Maven Tomcat plugin.

> **NOTE**
>
> Instead of merging resources in the CAE, it is generally recommended to do it during the build process in the frontend workspace.

5.4.13 Managing End User Interactions

Requirements

For a truly engaging experience website visitors need to be able to interact with your website. Interactions can reach from basic ways to search content, register and give feedback to enabling user-to-user communication and facilitating business processes such as product registration and customer self care.

End user interactions should be configurable in the editorial interface by non-technical users in the editorial interface of the system. It should, for example, be possible to place interaction components such as Login and Search buttons on pages just like any other content, configure layout and business rules etc.

Solution

For the Blueprint website, the term "action" denotes a functionality that enables users to interact with the website.

Examples:

- Search: The "search" action lets user to enter a query into a form field. After processing the search, a search result is displayed to the user.
- Login: This action can be used by users to login to the website by adding user name and password credentials. A successful login changes the state web

application's state for the user and offers him additional actions such as editing his user profile.

From an editor's perspective, all actions are represented by content objects of type `CMAction`. This enables an editor to add an action content to a page, for example by inserting it to the navigation `linklist` property. When rendering the page, this action object is rendered by a certain template that (for example) renders a search form. The submitted form data (the query, for instance) is received by a handler that does some processing (passing the query to the search engine, for instance) and that provides a model containing the search action result.

This section demonstrates the steps necessary to add new actions to *CoreMedia Blueprint*. It also helps to understand the currently available actions.

Standard Actions

As stated above, all actions are represented as `CMAction` contents in the repository. These contents can be used as placeholders in terms of the "substitution" mechanism described in the Content Application Developer Manual. An example for adding a new action: Consider an action where users can submit their email addresses in order to receive a newsletter.

1. Create a bean that represents the subscription form and add an adequate template.

```
  public class SubscriptionForm {
  public String email;

  public void setEmail(String email) {
    this.email = email;
  }

  public String getEmail() {
    return email;
  }
}
```

`SubscriptionForm.asTeaser.ftl`

```
        <%--@ftlvariable id="self" type="com.mycompany.SubscriptionForm
"--%>
<%--@ftlvariable id="subscriptionForm" type="com.mycompany.SubscriptionForm
 "--%>
<%--@ftlvariable id="cmpage"
type="com.coremedia.blueprint.common.contentbeans.Page"--%>
<#assign redirectUri=cm.getLink(cmpage.linkable)/>
<#assign subscriptionUri=cm.getLink(cmpage, "", {"return" : redirectUri})/>
<form method="post" action="${subscriptionUri}">
    <@spring.formInput path="subscriptionForm.email"/>
  <input type="submit" value="Subscribe"/>
```

```
</form>
```

2. Add a handler that is able to process the subscription as well as a link scheme that builds links pointing to the handler.

```
        @Link
@RequestMapping
public class SubscriptionHandler {

  @RequestMapping(value="/subscribe", method=RequestMethod.POST)
  public ModelAndView handleSubscription(@RequestParam(value="return",
required=true) String redirectUri,
                                         @ModelAttribute("subscriptionForm")
 SubscriptionForm form,
                                            HttpServletRequest request,
HttpServletResponse response)
          throws IOException {
    doSubscribe(request.getSession(), form.getEmail());
    response.sendRedirect(redirectUri);
    return null;
  }

  @Link(type=SubscriptionForm.class, parameter="return", uri="/subscribe")

  public UriComponents createSubscriptionLink(UriComponentsBuilder uri,
Map<String,Object> parameters) {
    return uri.queryParam("return", (String)
parameters.get("return")).build();
  }

  ...
}
```

Don't forget to register this class as a bean in the Spring application context.

3. Define an action substitution.

```
        public class SubscriptionHandler {
  ...
  @Substitution("com.coremedia.subscription",
modelAttribute="subscriptionForm")
  public SubscriptionForm createSubscriptionSubstitution(CMAction original,
 HttpServletRequest request) {
    return new SubscriptionForm();
  }
  ...
}
```

Notes

- The parameters `original` as well as `request` are optional and might be omitted here. But in a more proper implementation it might be useful to have access to the original bean and the current request.
- The optional `modelAttribute` causes the substitution to be become available as a request attribute `subscriptionForm`. This is useful when using dealing with the Spring form tag library (see above).

4. Create a newsletter action content
   - Create a content of type `CMAction`
   - Set the `id` property to value `com.coremedia.subscription`
   - Insert this content to a page's teaser link list.

Here is what happens when opening the page by sending an HTTP request:

1. The request will be accepted by the `PageHandler` that builds a `ModelAndView` containing the `Page` model. This model's tree of content beans contains the new `CMAction` instance.
2. The model will be rendered by initially invoking `Page.ftl` for the `Page` bean.
3. When the `CMAction` is going to be rendered in the teaser list, the template `CMAction.asTeaser.ftl` is invoked. This template substitutes the `CMAction` bean by invoking the `cm.substitute()` function while using the ID `com.coremedia.subscription`.
4. The substitution framework invokes the method `#createSubscriptionSubstitution` after checking whether `SubstitutionRegistry#register` has been invoked by any handler for his ID (which hasn't happened here). As the result, the substitutions result is a bean of type `SubscriptionForm`.
5. The above mentioned template `CMAction.asTeaser.ftl` therefore delegates to `SubscriptionForm.asTeaser.ftl` then.
6. While rendering `SubscriptionForm.asTeaser.ftl`, a link pointing to this form bean is going to be built. The method `#createSubscriptionLink` is chosen as a link scheme so that the link points to the handler method `#handleSubscription`.
7. After the user has received the rendered page, he might enter his email address and press the submit button.
8. This new (POST) request is accepted by the mentioned handler method `#handleSubscription` that performs the subscription and redirects the original page then so that the first step of this flow is repeated.

Of course, a more proper implementation could mark the subscription state (subscribed or not) in a session/cookie and would return an `UnsubscribeForm` from `#createSubscriptionSubstitution` depending on this state.

Webflow Actions

Spring Webflow (https://spring.io/projects/spring-webflow) is a framework for building complex form based applications consisting of multiple steps. Webflow

based actions can be integrated into *Blueprint* as well. This section describes the steps of how to integrate this kind of actions.

In *CoreMedia Blueprint* the `PageActionHandler` takes care of generally handling Webflow actions. The flow's out coming model is automatically wrapped into a bean `WebflowActionState`. A special aspect of this bean is that it implements `HasCustomType` and therefore is able to control the lookup of the of the matching template.

1. Place your flow definition file somewhere below a package named `webflow` somewhere in the classpath. The name of the flow definition file should be `<action_id>.xml`. Example: For an action `com.mycompany.MyFlowAction` you might create a file `com.mycompany.MyFlowAction.xml` that can be placed below a package `com.coremedia.blueprint.mycompany.webflow`.
2. For every flow view (such as "success" or "failure") create a ftl template. The template name needs to match the action id. Example: The action `com.mycompany.MyFlowAction` requires templates to be named `.../templates/com.mycompany/MyFlowAction.<flowView>.ftl`. These templates will be invoked for the mentioned beans of type `WebflowActionState`.
3. Create (and integrate) a new content item of type `CMAction` and set the property `id` to the action id (such as `com.mycompany.MyFlowAction`) and the property `type` to **`webflow`**.

5.4.14 Images

Requirements

For a website, images are required in different sizes and formats. For example, teaser need a small image with an aspect ratio of 1:1 in the sidebar and an aspect ratio of 4:3 in the main section. Images in articles and galleries are shown in 5:2 or 4:3 with a large size. And even these sizes are different on mobile devices and desktop displays.

Solution

CoreMedia Blueprint supports different formats combined with different sizes. It comes with four predefined cropping definitions.

- portrait_ratio3x4 (aspect ratio of 3:4)
- portrait_ratio1x1 (aspect ratio of 1:1)

- landscape_ratio4x3 (aspect ratio of 4:3)
- landscape_ratio16x9 (aspect ratio of 16:9)

A list of sizes can be defined for each format in the `Responsive Image Settings` which are usually defined in the settings of a site's theme as described in Frontend Developer Manual. They can also be located in the `Options/Settings/CMChannel` folder of the site, in the global settings below `/All Content/Settings/Options/Settings` or in the Spring configuration in the backend server. For more details see Section 9.5.3, "Image Cropping and Image Transformation" in *Studio Developer Manual*. The website will automatically choose the best matching image depending on the viewport of the client's browser.

How to configure image sizes

The struct `responsiveImageSettings` contains a list of string properties. This string must contain the name of a cropping format. For example `portrait_ratio1x1`. Each format contains a list of string properties, representing one size of this format. The name and the order of this list is not important and will be ignored. Every size must contain two integer properties `width` and `height`.

If site specific image variants are enabled, the `Responsive Image Settings` will be used for the image editor as well. In this case the additional integer property fields `widthRatio`, `heightRatio`, `minWidth` and `minHeight` must be defined. Additionally, the field `previewWidth` and/or `previewHeight` should be defined to define the preview size in Studio.

For example a `Responsive Image Settings` with two formats. `portrait_ratio1x1` with just one size and `landscape_ratio4x3` with three sizes.

```
<Struct xmlns="http://www.coremedia.com/2008/struct">
  <StructProperty Name="responsiveImageSettings">
    <Struct>
      <StructProperty Name="portrait_ratio1x1">
        <IntProperty Name="widthRatio">1</IntProperty>
        <IntProperty Name="heightRatio">1</IntProperty>
        <IntProperty Name="minWidth">200</IntProperty>
        <IntProperty Name="minHeight">200</IntProperty>
        <IntProperty Name="previewWidth">400</IntProperty>
        <StringProperty Name="jpegQuality">0.8</StringProperty>
        <Struct>
          <StructProperty Name="0">
            <Struct>
              <IntProperty Name="width">60</IntProperty>
              <IntProperty Name="height">60</IntProperty>
            </Struct>
          </StructProperty>
        </Struct>
      </StructProperty>
      <StructProperty Name="landscape_ratio4x3">
        <IntProperty Name="widthRatio">4</IntProperty>
```

```
        <IntProperty Name="heightRatio">3</IntProperty>
        <IntProperty Name="minWidth">1180</IntProperty>
        <IntProperty Name="minHeight">885</IntProperty>
        <IntProperty Name="previewWidth">400</IntProperty>
        <StringProperty Name="jpegQuality">0.7</StringProperty>
        <Struct>
          <StructProperty Name="0">
            <Struct>
              <IntProperty Name="width">200</IntProperty>
              <IntProperty Name="height">150</IntProperty>
            </Struct>
          </StructProperty>
          <StructProperty Name="1">
            <Struct>
              <IntProperty Name="width">320</IntProperty>
              <IntProperty Name="height">240</IntProperty>
            </Struct>
          </StructProperty>
          <StructProperty Name="2">
            <Struct>
              <IntProperty Name="width">640</IntProperty>
              <IntProperty Name="height">480</IntProperty>
            </Struct>
          </StructProperty>
        </Struct>
      </StructProperty>
    </Struct>
  </StructProperty>
</Struct>
```

CAUTION

Every image cropping format must contain one image size, otherwise the default size and format, defined in `ImageFunctions`, will be used.

Image dimensions supported in Cloud Installations

The editing capabilities of Self-Managed and new Cloud Installations since CMCC 11 (2307) differ in terms of supported image formats, image sizes, and image editing capabilities. The image dimensions supported in Cloud installations are listed here. Make sure that the sizes defined in the `Responsive Image Settings` match the supported dimensions.

High Resolution/Retina Images

CoreMedia Blueprint supports high resolution images. Set the BooleanProperty `enableRetinaImages` to true. If enabled, the JavaScript `jquery.coremedia.responsiveimages.js` is choosing a larger image according to the `devicePixelRatio` of the browser.

For Example the website wants to render an image with an aspect ratio of 4:3 and the best responsive image size is `400px : 300px`. With a `devi`

`cePixelRatio` of 2, the JavaScript `jquery.coremedia.responsiveimages.js` is now choosing the size of `800px : 600px`.

Default JPEG Compression Quality

The default JPEG compression quality is 80% in *CoreMedia Blueprint*. This parameter is configured in `blueprint-handlers.xml` for the `transformedBlobHandler`. For further information consult the "CAE Application Developer Manual", chapter "Image Transformation API".

MIME Type Mapping

When building links to image variants in the CAE, the MIME type of the original image is used by default to determine the file extension of the links. To adjust these MIME types you can provide a mapping of original MIME types to desired MIME types in the setting `linkMimeTypeMapping`. The struct `linkMimeTypeMapping` contains String properties where the key is the MIME type of the original image and the value is the desired MIME type for the links to variants of this image.

You could for instance add this setting to the `Responsive Image Settings` content item next to the `responsiveImageSettings` struct like so:

```
<Struct xmlns="http://www.coremedia.com/2008/struct">
  <StructProperty Name="linkMimeTypeMapping">
    <Struct>
      <StringProperty Name="image/jpeg">image/png</StringProperty>
      <StringProperty Name="image/gif">image/png</StringProperty>
    </Struct>
  </StructProperty>
  <StructProperty Name="responsiveImageSettings">
    <Struct>
      <StructProperty Name="portrait_ratio1x1">
        ...
      </StructProperty>
      ...
    </Struct>
  </StructProperty>
</Struct>
```

With these settings all links to variants of images with MIME type `image/jpeg` or `image/gif` would be created with MIME type `image/png` and the file extension `.png` instead.

5.4.15 URLs

Link generation and request handling is based on the concepts of the CAE web application. For further information consult the "CAE Application Developer Manual". *CoreMedia Blueprint* offers a simple mechanism for link building and

parsing that is based on regular expressions. The out of the box configuration has been made with "SEO Search Engine Optimization" in mind:

- URLs show to which site section the currently displayed page belongs
- URLs for asset detailed pages – opposed to section overview pages – contain the title of the asset

See Section 7.2, "Link Format" [433] for link schemes and controllers of *CoreMedia Blueprint* as well as existing post processors.

5.4.16 Vanity URLs

Requirements

Editors should be able to define special URLs to special content objects which are easy to remember.

Solution

Vanity URLs are special human readable URLs which do not contain any technical identifiers like content item IDs. *CoreMedia Blueprint* provides a means to assign vanity URLs to content objects.

Vanity URLs are configured in channel settings. Typically, there is one Vanity URL settings content item for the root channel of a given site. This is the setup chosen for *CoreMedia Blueprint* demo content. To find the Vanity URL settings content item, open the root channel of a site and switch to the `Settings` tab. You will find the Vanity URL settings content item link inside the `Linked Settings` section.

Vanity URLs are defined as a relative URI path. The path might consist of several segments, but if you would like to keep your Vanity URLs simple, just use only one path segment. The URI path is then prepended with a path segment consisting of the site name. For example, for the site `corporate`, a URI path of `my/special/article` would yield the Vanity URL `/corporate/my/special/article`.

To add a Vanity URL for a content item, follow these steps:

1. Select the `StructListProperty` *`vanityUrlDefinition`* and create a new child Element Struct by clicking the **[Add item to List Property]** symbol in the toolbar.
2. Create a new `LinkProperty` and name it "target".

3. Set the content type field to the type of your target content item.
4. Click on the value field, this will open the library window. Drag your target content item from the library window into the value field.
5. Create a `StringProperty`, name it "id" and type your vanity URI path inside the value field.

Once the settings content item is published, the new Vanity URL is reachable on the live site, and it is used for all generated links referring to the target content item.

NOTE

While it is possible to define multiple Vanity URLs for a single target content item it might cause confusion when looking on the links of the website. By default, the first "id" in the settings content item that refers to the target content item will be used for link creation.

NOTE

Defining multiple target content items for a single Vanity URL is not prevented by the Studio but it is usually not desired. A link will be generated for every target but the resulting URL will only point to the content item referenced by the first occurrence of the "id" in the settings.

CAUTION

A Vanity URL can overlap with other dynamic URLs on your website. A Vanity URL will always take precedence. Consequently, the content item behind the other dynamic URL will not be reachable on the website anymore.

5.4.17 External URLs

External URLs can be managed manually in documents of type *External Link* (technical type `CMExternalLink`).

A value in field *URL* of these documents is checked for valid URL syntax. If property `url-validator.strict` is set to `true` for the *Studio Server*, the URL is also validated for proper encoding. The URL validator will then report URLs with special characters (non-ASCII) as invalid. Examples:

- `https://www.coremedia.com/über-uns` must be encoded as `https://www.coremedia.com/%C3%BCber-uns` (percent-encoding).
- `https://www.münchen.de` must be encoded as `https://www.xn--mnchen-3ya.de` (punycode-encoding for International Domain Names).

By default, the encoding is not checked.

Your sites can be configured to sanitize special characters (non-ASCII) in URLs from External Link documents for delivery by the *Content Application Engine*. For this, a boolean content setting `sanitize-external-links` is available. If set to `true`, non-ASCII characters in External Link URLs will be properly encoded for delivery (see examples above). Default is `false` so that URLs are delivered as-is. To activate, add boolean content setting `sanitize-external-links` to the settings of your sites' root pages in Studio and tick the checkbox on each.

Note that while *http(s)* URLs will be properly encoded according to general rules (punycode for domain, percent-encoding for all other parts), other schemes (like *mailto*) may not be handled properly and may need to be encoded manually before entering them in Studio.

The Headless Server does not support this sanitization and will deliver External Link URLs as-is.

5.4.18 Content Visibility

Requirements

Content should become available online only within a specific time frame. For example, editors need to ensure that a press release only becomes public at a certain day and time or an article should expire after a specific day. In addition, editors want to preview their preproduced content in the context of the website as if it was already available.

Solution

CoreMedia Blueprint supports restricting the visibility of content items by setting the optional `validFrom` and `validTo` date properties of content of type `CMLinkable`.

validFrom	
UI Name	Valid From

Description	Content where the "valid from" date has not been reached yet is not displayed on the site yet.
validTo	
UI Name	Valid To
Description	Content where the "valid to" date has passed is not displayed on the site anymore. By not specifying either of validTo or validFrom, an open interval can be specified to define just a start or end date.

Table 5.15. Properties for Visibility Restriction

Content is filtered in the CAE during the following two stages of request processing:

- The controller is resolving content from a requested URL. See `ContentValidityInterceptor`.
- In the content bean layer whenever references to other content beans that implement `ValidityPeriod` are returned.

In the CAE visibility checking is implemented as part of an extensible content validator concept. The generic `ValidationService` is configured with a `ValidityPeriodValidator` to filter content when it is requested.

To allow editors to preview content for a certain preview date and time a `PreviewDateSelector` component has been added to *Studio*, which sets the request parameter `previewDate`. This parameter is respected by the `ValidityPeriodValidator`.

5.4.19 Content Type Sitemap

Configuration

The content type Sitemap has three fields you can configure:

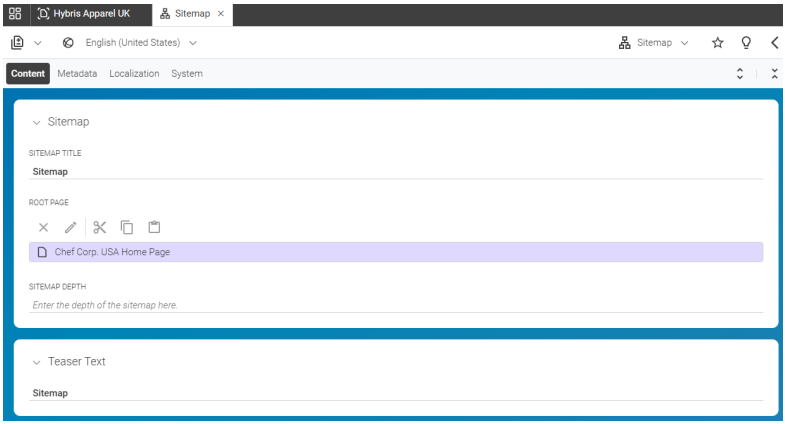

Figure 5.17. Content Type Sitemap

Enter a Sitemap Title which will be rendered as the headline of the Sitemap section in the site. The Root Page field defines the root node from where the content for the Sitemap will be rendered. Additional the Sitemap can be rendered to a specific depth which can be set here. This depth is three by default.

5.4.20 Robots File

Requirements

Technical editors should be able to adjust site behavior regarding robots (also known as crawlers or spiders) from search engines like Google. For example:

- Enable/disable crawling of certain pages including their sub pages.
- Enable/disable crawling of certain single content items.
- Specify certain bots to crawl different sections of the site.

To support this functionality most robots follow the rules of `robots.txt` files like explained here: http://www.robotstxt.org/.

For example, the site "Corporate" is accessible as `http://corporate.blueprint.coremedia.com`. For all content of this site the robots will look for a file called `robots.txt` by performing an HTTP GET request to `http://corporate.blueprint.coremedia.com/robots.txt`.

A sample `robots.txt` file may look like this:

```
User-agent: Googlebot,Bingbot
Disallow: /folder1/
Allow: /folder1/myfile.html
```

Example 5.2. A `robots.txt` file

Solution

Blueprint's `cae-base-lib` module provides a `RobotsHandler` which is responsible for generating a `robots.txt` file. A `RobotsHandler` instance is configured in `blueprint-handlers.xml`. It handles URLs like `http://corporate.blueprint7.coremedia.com:49080/blueprint/servlet/service/robots/corporate`

This is a typical preview URL. In order to have the correct external URL for the robots one needs to use Apache rewrite URLs that forwards incoming GET requests for `http://corporate.blueprint7.coremedia.com/robots.txt` to `http://corporate.blueprint7.coremedia.com:49080/blueprint/servlet/service/robots/corporate`

The `RobotsHandler` will be responsible for requests like this due to the path element `/robots` The last path element of this URL (in this example `/corporate` will be evaluated by `RobotsHandler` to determine the root page that has been requested. In this example "corporate" is the URL segment of the Corporate Root Page. Thus, `RobotsHandler` will use Corporate root page's settings to check for `Robots.txt` configuration.

To add configuration for a `Robots.txt` file the corresponding root page (here: "Corporate") needs a setting called `Robots.txt`

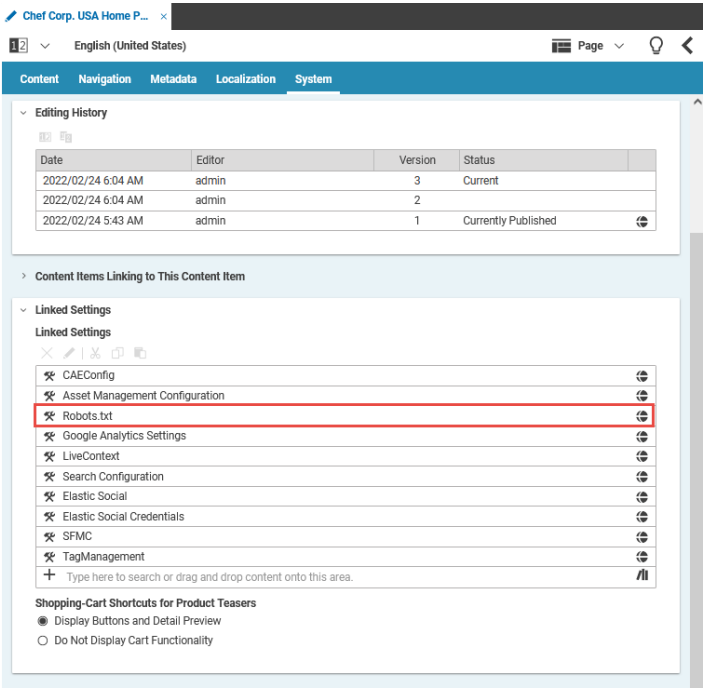

Figure 5.18. `Robots.txt` settings

Example configuration for a Robots.txt file

The settings content item itself is organized as a `StructList` property like in this example:

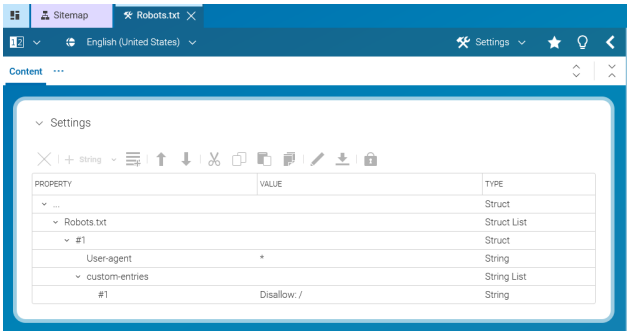

Figure 5.19. Channel settings with configuration for `Robots.txt` as a linked setting on a root page

For any specified user agent the following properties are supported:

- `User-agent`: Specifies the user agent(s) that are valid for this node.
- `Disallow`: A link list of items to be disallowed for robots. This list specifies a black list for navigation elements or content: Elements that should not be crawled. Navigation elements will be interpreted by "do not crawl elements below this navigation path". This leads to two entries in the resulting `robots.txt` file: one for the link to the navigation element and one for the same link with a trailing '/'. The latter informs the crawler to treat this link as path (thus the crawler will not work on any elements below this path). Single content elements will be interpreted as "do not crawl this content"
- `Allow`: A link list of items to be explicitly allowed for robots. This list specifies navigation elements or content that should be crawled. It is interpreted as a white list. Usually one would only use a black list. However, if you intend to hide a certain navigation path for robots but you want one single content item below this navigation to be crawled you would add the navigation path to the disallow list and the single content item to the allow list.
- `custom-entries`: This is a String List to specify custom entries in the `Robots.txt`. All elements here will be added as a new line in the `Robots.txt` for this node.

The example settings content item will result in the following `robots.txt` file:

```
User-agent: *
Disallow: /corporate/corporate-information/
Allow: /corporate/corporate-information/contact-us

User-agent: Googlebot
Disallow: /corporate/embedding-test
```

Example 5.3. `robots.txt` file generated by the example settings

5.4.21 Sitemap

Requirements

If you run a public website, you want to get listed by search engines and therefore give web crawlers hints about the pages they should crawl. https://www.sitemaps.org/ declares an XML format for such sitemaps which is supported by many search engines, especially from Google and Microsoft.

"Sitemap" in terms of https://www.sitemaps.org/ is not to be mistaken with a human-readable sitemap which visualizes the structure of a website (see Section 5.4.19, “Content Type Sitemap” [215]). It is rather a complete index of all pages of a site.

A simple sitemap file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.sitemaps.org/schemas/sitemap/0.9
http://www.sitemaps.org/schemas/sitemap/0.9/sitemap.xsd">
  <url>
    <loc>http://helios.coremedia.com/corporate/spicy-duck-694</loc>
  </url>
  <url>
    <loc>http://helios.coremedia.com/corporate/share-your-recipes-696</loc>
  </url>
  ...
</urlset>
```

Example 5.4. A sitemap file

If generation of last modification dates for sitemaps is active (see below), a sitemap file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.sitemaps.org/schemas/sitemap/0.9
http://www.sitemaps.org/schemas/sitemap/0.9/sitemap.xsd">
  <url>
    <loc>http://helios.coremedia.com/corporate/spicy-duck-694</loc>
    <lastmod>2026-03-02</lastmod>
  </url>
  <url>
    <loc>http://helios.coremedia.com/corporate/share-your-recipes-696</loc>
    <lastmod>2026-03-04</lastmod>
  </url>
  ...
</urlset>
```

Example 5.5. A sitemap file with last modification dates

Maximum number of URLs

The size of a sitemap is limited to 50,000 URLs. Larger sites must be split into several sitemap files and a sitemap index file which aggregates the sitemap files.

A sitemap index file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <sitemap>
    <loc>http://helios.coremedia.com/sitemap1.xml.gz</loc>
    <lastmod>2014-03-31T15:33:26+02:00</lastmod>
  </sitemap>
  ...
</sitemapindex>
```

Example 5.6. A sitemap index file

Solution

A sitemap consists of multiple entities (the index and the sitemap files) and has dependencies on almost the whole repository. If a new content is created, which "coincidentally" occurs in the first sitemap file, the entries of all subsequent sitemap files are shifted.

In border cases even the number of sitemap files may change, which affects the sitemap index file. So you cannot generate single sitemap entities on crawler demand, asynchronously and independent of each other, but you must generate a complete sitemap which represents a snapshot of the repository. Moreover, the exhaustive dependencies make sitemaps practically uncacheable, and the generation is expensive. For these reasons *Blueprint* does not render sitemaps on demand but pregenerates them periodically. So you must distinguish between sitemap generation and sitemap service. Both are handled by the live web application, though.

Sitemap Generation

CoreMedia Blueprint features separated sitemaps for each site. Sitemap generation depends on some site specific configuration, like the content types to include or paths to exclude, amongst others. This configuration is specified by `SitemapSetup` Spring beans.

The `corporate` extension provides a `SitemapSetup` bean suitable for their particular sites. Projects can declare their own sitemap setups. The setups are collected in the `sitemapConfigurations` Spring map in the Spring management context. To append the `sitemapConfigurations`, declare a customizer bean in the management context configuration. Consider the Spring Boot API documentation for requirements to register a configuration in the

management context (Spring Boot API Documentation: ManagementContextConfiguration).

```
@ManagementContextConfiguration(proxyBeanMethods = false)
@Import({
  CustomizerConfiguration.class,
  CaeManagementConfiguration.class,
})
public class MyCustomManagementConfiguration {
...

  @Bean(autowireCandidate = false)
  @Customize("sitemapConfigurations")
  Map<String, SitemapSetup> appendCorporateSitemapConfiguration(
          SitemapSetup corporateSitemapConfiguration) {
    return Map.of("corporate", corporateSitemapConfiguration);
  }

  @Bean
  public SitemapSetup corporateSitemapConfiguration(
          CaeSitemapConfigurationProperties properties,
          SitemapRendererFactory sitemapIndexRendererFactory,
          SitemapUrlGenerator corporateSitemapContentUrlGenerator) {
    SitemapSetup sitemapSetup = new SitemapSetup(properties);
    sitemapSetup.setSitemapRendererFactory(sitemapIndexRendererFactory);
    sitemapSetup.setUrlGenerators(List.of(corporateSitemapContentUrlGenerator));
    return sitemapSetup;
  }

...
}
```

If you want to generate a sitemap for a site, you have to specify the setting `sitemapOrgConfiguration` at the root channel. It is a `String` setting, and the value must be a key of the `sitemapConfigurations` map.

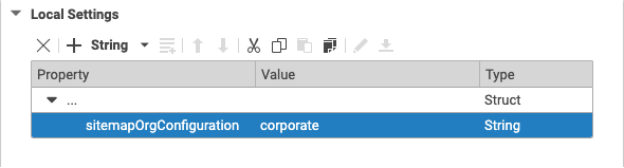

Figure 5.20. Selection of a sitemap setup

By default, the Corporate sites are sitemap-enabled. The eCommerce sites are not sitemap-enabled, since they serve only as backend for *HCL Commerce* applications, there is no need for sitemaps.

Sitemaps are generated periodically in the Delivery CAE by a `SitemapGenerationJob`. You can specify the initial start time and the period as application properties `cae.sitemap.starttime` and `cae.sitemap.period-minutes`, respectively. For details about the values see the Javadoc of the setters in `SitemapGenerationJob`. The *Blueprint* is preconfigured to run the sitemap generation nightly at 01:30. You can also trigger sitemap generation for a particular site manually by the management URL

```
http://live-cae:42181/internal/corporate-de-de/sitemap-org
```

where `corporate-de-de` stands for the segment of the site's root channel. Note that it is an internal URL which can only be invoked directly on the CAE's servlet container. Sitemap generation is an expensive administrative task, which is not to be exposed to end users.

The sitemaps are written into the file system under a directory which is specified by the `cae.sitemap.target-root` application property. That means, the CAE needs write permissions for this directory.

Last Modification Dates

If last modification dates for URLs are to be included in sitemaps, system property `cae.sitemap.include-last-modification-date` must be set to `true`. This is a system-wide setting, affecting all sitemaps alike. Activating the feature may double the time for sitemap generation, depending on your content structure.

The *Blueprint* provides a mechanism to compute last modification dates based on simple rules to minimize computational overhead. You may need to adjust the mechanism to tailor last modification date computation to your sites' content structures and your SEO requirements.

The rules for last modification date computation are designed as follows:

- Each editorial content addressed by a URL in the sitemap receives
  - the *most recent* modification date of itself and all its linked contents if it is a `CMTeasable` or descendant thereof, or
  - just its own modification date if it is a `CMLinkable`.

  Linked contents are all contents linked either through *link list properties* or through *struct properties*, *markup properties*, or *(inherited) page placements*.
- The modification date of a content is the date of its most recent publication when generating a sitemap on the live side.
- The following contents are not taken into account when computing the last modification date:
  - *technical* contents (symbols, CSS, JavaScript, themes, etc; all that are not of type `CMLinkable` or descendants thereof),
  - *linked* channels (type `CMChannel` or descendants thereof),
  - currently not valid contents (with respect to `validFrom` and `validTo` properties), and
  - *master* content from which a content is derived via localization workflows.

If there is need to adjust the computation of the last modification date, take a look at implementations of Java interface `ModificationDateAggregator`. Adjust existing implementations to your requirements or introduce new implementations.

Sitemap Service

The generated sitemaps are available by the URL pattern

`/service-sitemap-siteID-sitemap_index.xml`

This pattern consists only of a single segment without a path, so there are no path restrictions for the URLs included in the sitemap.

In order to inform search crawlers, the sitemap URLs are included in the `robots.txt` files. Since there is only one robots file per web presence, you will see multiple sitemap entries for the localized sites:

```
User-agent: *
Disallow: /

Sitemap: http://corporate.acme.com/service-sitemap-ab...ee-sitemap_index.xml
Sitemap: http://corporate.acme.com/service-sitemap-1c...7a-sitemap_index.xml
```

5.4.22 Website Search

There are two search types in *HCL Commerce* integration scenarios: *HCL Commerce Search* and *CoreMedia CMS Search*. You can view the results of both searches by switching between tabs "shop" or "content".

> **NOTE**
>
> For *HCL Commerce* Search the CMS content must be crawled by the HCL Solr Search engine. Please refer to the HCL documentation. A configuration file for each example site is part of the *HCL Commerce Workspace* archive (for example, `WCDE-ZIP/components/foundation/subcomponents/search/solr/home/droidConfig-cm-aurora-en-US.xml`).

The CoreMedia CMS Search is introduced further in this section.

Requirements

In order to make content more accessible for their audience virtually all websites have full-text search capabilities. To improve the search experience some websites also offer features such as search term autocompletion, suggestions in case of misspelled search terms, more advanced filtering options or even metadata based drilldown navigation in search results.

Solution

CoreMedia CMS has built-in integration with the Apache Solr search engine. *Blueprint* comes with a small abstraction layer that offers unified search access to Solr for all CAE based code. It provides the following features, all based on standard Solr functionality:

- Full text search: Search for content across all fields
- Field based filters: Filter results by metadata such as the content type, the site section it belongs to, etc.
- Facets: Display facets, that is the number of results in a field for certain values
- Spellcheck suggestion: "Did you mean" suggestions for misspelled terms
- Search term highlighting: All words are highlighted in your text
- Validity range filtering: Automatically filter for only visible results (see section Section 5.4.18, “Content Visibility” [214]
- Filter non-searchable: Automatically filter content that should not be part of search results.
- Caching: Search results can be optionally cached for a certain amount of time.

The search integration can be found in the modules `com.coremedia.blueprint.cae.search` and `com.coremedia.blueprint.cae.search.solr`.

`com.coremedia.livecontext.fragment.CMSearchFragmentHandler` is used in *HCL Commerce* integration scenarios to process *fragment search requests*. The handler requires no specific configuration in content settings and uses the general CAE Search configuration as explained in Section 5.4.22, “Website Search” [224].

Configuring search in content settings

Some aspects of website search are configurable in a site-specific Settings content item. The site's root channel links to the Settings content item `Search Configuration` with the settings used for that site.

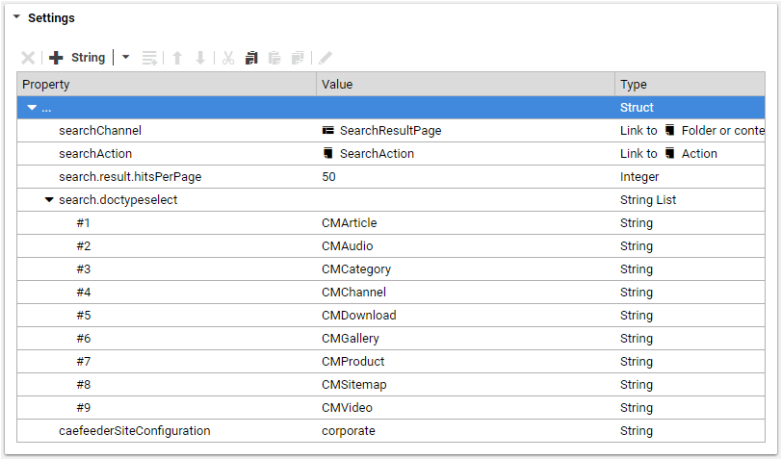

Figure 5.21. Search Configuration Settings content item

It contains the following settings:

Settings Property	Description
searchChannel	The channel used to render the search result page.
searchAction	Content of type CMAction with ID "search".
searchResultHitsPerPage	The number of hits shown on the search result page. If not set, default is 10.
searchResultPagination	Boolean parameter to enable pagination instead of "load more" functionality. Default is false.
searchDisableSpellingSuggestions	Boolean parameter to disable the spelling suggestion offered by Solr. Default is false.
searchDoctypeSelect	The content types that appear in the search result. Sub-types must be listed explicitly.
searchChannelSelect	The Categories that appear in the filter panel based on configured channels.
searchFacets	A substruct that maps symbolic search facet names to Solr index field names. This enables search faceting on the website with the possibility to filter search results based

Settings Property	Description
	on values indexed in the configured fields. You can choose arbitrary names for facets, but note that these names will appear as request parameters in search URLs.
searchFacetLimit	An integer parameter that controls how many values will be displayed for each search facet, if `searchFacets` is configured. Default is 100.
caefeederSiteConfiguration	Contains the value `corporate` to select the *CAE Feeder Brand Blueprint* configuration for indexing content of the site. This enables page grid indexing as described in the next section.

Table 5.16. Brand website search settings

Configuring page grid indexing

The *Brand Blueprint CAE Feeder* feeds CMChannel content items to the search engine so that pages can be found on the website. To this end, the CAE Feeder configuration specifies which parts of a page grid need to be indexed. This includes the configuration of relevant page grid sections, content types of linked contents and their properties.

NOTE

Read section Section 5.4.4, “Page Assembly” [177] for an introduction to page grids.

The *Brand Blueprint CAE Feeder* is configured in the Spring configuration class `CorporateCaeFeederAutoConfiguration` and its accompanied properties class `CorporatePageGridConfigurationProperties` in package `com.coremedia.blueprint.caefeeder.corporate` of the Blueprint module `apps/cae-feeder/modules/extensions/corporate/corporate-caefeeder-component`. The Spring configuration class imports the content bean definitions and defines the following `FeedablePopulators` to index the page grid:

The `PageGridFeedablePopulator` takes properties from content linked in the page grid and adds them to the `textbody` index field when feeding a `CMChannel`. It is configured to feed the teaser properties of linked content items except for articles linked with view type "Detail" in which case the full article

text is indexed with the channel. The `PageGridInlineContentFeedablePopulator` ensures that articles that are linked with view type "Detail" are not returned by the website search in addition to their page. To this end, it sets the index field `notsearchable` to `true` for such articles.

If a page grid placement contains a `CMCollection` content item, then the contents linked in its `items` property are included as well - just as if they were linked directly in the page grid.

The mentioned FeedablePopulators are only used for content items if their site has a settings content item that defines the setting `caefeederSiteConfiguration` with value `corporate`. This is the case for *Brand Blueprint* sites. The Spring configuration class `CorporateCaeFeederAutoConfiguration` configures the site-specific activation of page grid feeding by adding the FeedablePopulators to the bean `siteSpecificFeedablePopulatorMap` for the value `corporate`.

The *Brand Blueprint* comes with a default configuration for indexing page grids of `CMChannel` content items. If needed, you can adapt the configuration in class `CorporateCaeFeederAutoConfiguration` and `CorporatePageGridConfigurationProperties`. The following table describes the used Spring properties. All properties start with the prefix `corporate.search.page-grid` which is abbreviated with `[c.s.p]` below.

Property	Description
[c.s.p].content-type	The type of the contents with indexed page grid. Default: `CMChannel`
[c.s.p].name	The name of the struct property that contains the page grid. Default: `placement`
[c.s.p].excluded-sections	Comma-separated list of ignored page grid sections. Default: `header, footer, sidebar`
[c.s.p].item-content-types	Comma-separated list of content types of considered page grid items. Contents of other types that are linked in the page grid are ignored and not indexed with the page grid. Default: `CMChannel, CMArticle, CMTeaser, CMCollection, CMVideo, CMDownload, CMExternalLink, CMProduct`
[c.s.p].item-text-properties.[*content type*]	The content properties of page grid items with a view type other than "Detail" that are indexed in the index field `textbody`

Property	Description
	of the page. This property takes a Map from content type name to list of content property names. You can configure a Map entry with the content type at the end of the Spring property key, and a comma-separated list of content property names as value. The configuration for the most specific content type decides which item properties are used. Property lists are not merged with configurations for super types. This makes it possible to ignore properties in subtypes. Default: `corporate.search.page-grid.item-text-properties.CMTeasable=teaserTitle,teaserText`, and `corporate.search.page-grid.item-text-properties.CMProduct=productName,shortDescription`
[c.s.p].item-valid-from-property [c.s.p].item-valid-to-property	The name of the date properties for visibility as described in Section 5.4.18, "Content Visibility" [214]. Content that is not currently visible is not indexed with the page. The *CAE Feeder* automatically reindexes after visibility has changed. Default: `validFrom` / `validTo`
[c.s.p].inline-content-types	Comma-separated list of content types used in the page grid with view type "Detail" for which the text properties are indexed with the page grid instead of the teaser properties. Default: `CMArticle`
[c.s.p].inline-content-view-type	The technical name of the "Detail" view type. Default: `full-details`
[c.s.p].inline-content-text-properties.[*content type*]	The content properties of page grid items with view type "Detail" that are indexed in the index field `textbody` of the page. This property takes a Map from content type name to list of content property names. You can configure a Map entry with the content type at the end of the Spring property key, and a comma-separated list of content property names as value. The configuration for the most specific content type decides which item properties are used. The property lists are not merged with configurations for super types. This makes it possible to ignore properties in subtypes.

Property	Description
	Default: `corporate.search.page-grid.inline-content-text-properties.CMArticle=title,detailText`
[c.s.p].collection-content-type	The content type of collection content items used in the page grid. Default: `CMCollection`
[c.s.p].collection-items-property	The link property of collection content items to get the items of a collection. Default: `items`
[c.s.p].collection-view-type-property	The link property of collection content items to get the view type for the items of a collection. Default: `viewtype`
[c.s.p].config-id	An identifier that represents the configuration options. Default: `corporate`

Table 5.17. Page Grid Indexing Spring Properties

NOTE

You must reindex from scratch with an empty *CAE Feeder* database to apply the changes of the above configuration properties to all indexed content items. If it is okay to just apply the changes to newly indexed content items and if you don't reindex with an empty *CAE Feeder* database, then you need to change the value of the `[c.s.p].config-id` property to some other string constant, if you've changed one of the following properties (all starting with `[c.s.p].`): `name`, `excluded-sections`, `item-content-types`, `item-valid-from-property`, `item-valid-to-property`.

5.4.23 Topic Pages

Requirements

Topic pages are a popular feature on most websites. Usually, topic pages are assembled from existing content which has already been published in another context before. Thus, topic pages should not cause any extra effort for the editors, but be available completely automatic.

Solution

In *CoreMedia Blueprint* topic pages are based on tags. Each tag content can be rendered as a topic page, showing the assets which are tagged with this particular tag. That is, add for example the Professionals tag content to the *Subjects* field of the Metadata tab and you will get a link to the topic page. Clicking this link opens the topic page for the topic "Professionals" in the default topic context of the site. See section “A Topic Page is a Page” [232] for details about context.

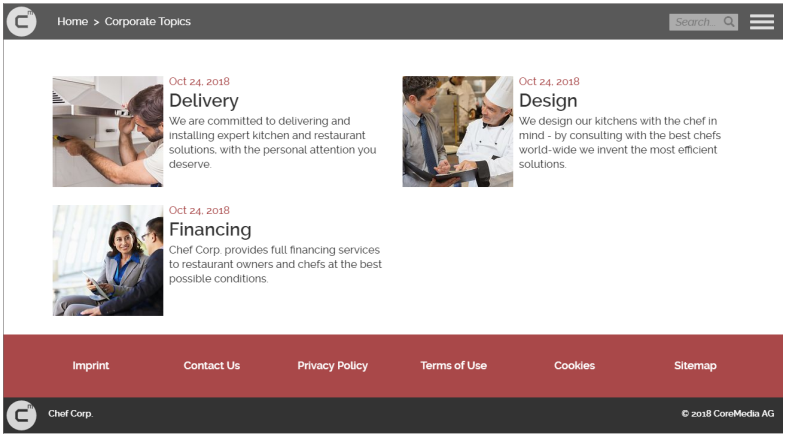

Figure 5.22. Generated topic page for topic "Professionals"

Configuration

The topic pages feature needs some configuration which is collected in a settings content. In the *Blueprint* example content this setting is located at `<Site Name>/Options/Settings/TopicPages` content of each site folder. This path must be configured as `topicpageConfigurationPath` for the

`topicpageContextFinder` Spring bean in `blueprint-contextstrategy.xml`. Topic page configuration is site specific. Relative paths will be concatenated with the root folder of the active site.

A Topic Page is a Page

Topic pages are based on the well known page concept. Just like any other asset, a tag content needs a context in which it is rendered as a topic page. Ordinary assets like articles have their explicit navigation contexts. For tag content there is a default context for each site. This default context is just another Page content. It must be configured as a `TopicPagePage` link property in the `TopicPages` settings content. This deviating context determination has two reasons:

- It spares your editors the tedious task of assigning a context to each tag
- It allows you to create site specific topic pages for global `com.coremedia.blueprint.studio.rest.taxonomies` tags

CAUTION

If you create your own default topic page set the `Hidden` flags (on the *Metadata* tab) of the page to true, and add the page as child to the site root. So it inherits all the JavaScript and CSS which is responsible for the design of the site.

Managed Topic Pages

The default topic page context allows you to generate a topic page for any tag content. This is convenient, but for some very popular topics you might want to provide a thoroughly styled and edited topic page, for example with an introductory text and image. *Studio* comes with a UI that executes the following steps for your custom topic page:

1. Creating a new page content item.
2. Adding the page as a child to the default context page in order to inherit its features and preserve the style of the site.
3. Setting the new page as context in the tag content item which represents the topic.

When the UI has created and linked the topic page, you only need to care for the layout and the accompanying content of the placements other than main:

1. Select the desired layout for your custom topic page.

2. The main placement is reserved for the content list. Fill the other placements with content of your choice (an introductory article or related topics, for instance).

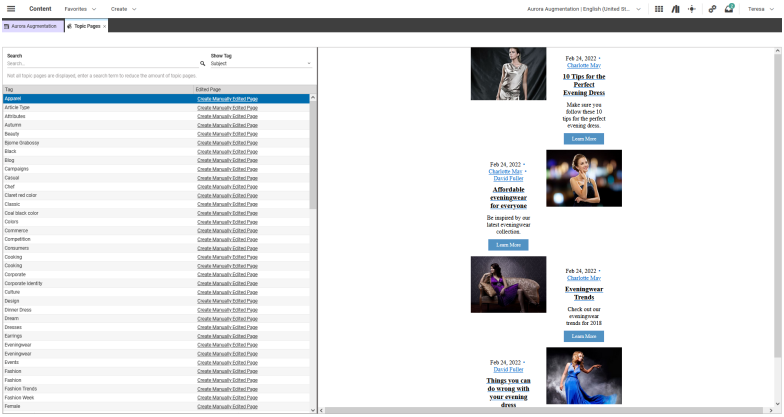

Figure 5.23. The topic pages administration in Studio

The *Studio* plugin provides a separate preview for the generated topic page too. This preview is more reliable than the one that is rendered for the page, since the preview is parameterized with the selected tag.

For global topics each site can have a specific custom topic page. All custom pages are linked in the topic's `contexts`, and at runtime the CAE determines the custom topic page which matches the context of the request or falls back to the default topic page of the particular site.

Configure target folder of new topic pages

When a custom topic page is created by using the Topic Page *Studio* extension, a new page content is created for the active site in folder `Content/Topic Pages`. The path is configurable in the site specific settings content `TopicPages`, located in the settings folder of the site. The property name for the corresponding string property is `folder`.

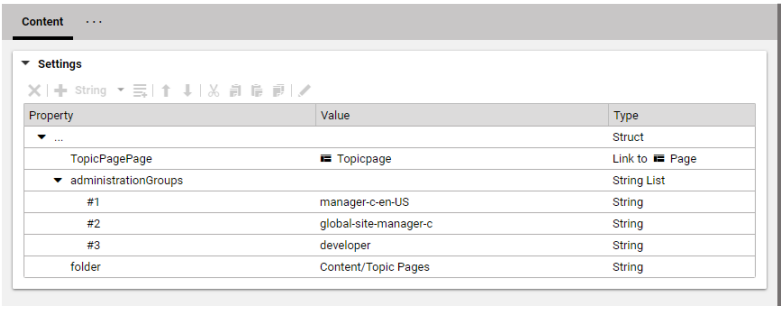

Figure 5.24. Settings content items for topic pages

Configure access to the topic page administration

You can configure the list of user groups that can access the topic pages manager in the `jangaroo.config.js` file of the `custom-topic-pages` extension. By default, the following groups are allowed to open the manager:

- global-manager
- taxonomy-manager
- topic-pages-manager
- developer

This list can not be customized during runtime. To add or remove access for users, ensure that they are a member of the corresponding groups.

URL format

Topic pages are search relevant, so they need SEO friendly URLs. The pattern is the same as for standard pages, and the URLs for topic pages are generated and resolved by the `PageHandler`. Only the sequence of segments is different from ordinary page URLs. It does not manifest a hierarchy but consists of exactly the segments `/<site>/<topicpage>/<topic>/<id>`. Topicpage is the segment of the topic page context, topic is the value property of the tag content, and id is the id of the tag content.

Disable Managed Topic Pages

The Managed Topic Pages are implemented as *CoreMedia Extension* called `custom-topic-pages`. Therefore, you can disable it like any other extension (see Section 4.1.5, “Project Extensions” [77] for details).

```
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>custom-topic-pages-bom</artifactId>
  <version>${project.version}</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
```

Search for Topic Pages

Topic pages can also be found by *CoreMedia Blueprint's* search. To get topic pages in the search result list, the topic page name must match the entered search query. Topics which are not used in the current site do not appear in the search results. Topic pages are displayed on top of the search results list with a customizable teaser image as shown in Figure 5.25, "A Search Result for a Topic Page" [235]. The capability to search for topic pages can be controlled by settings located in the search action of the particular site.

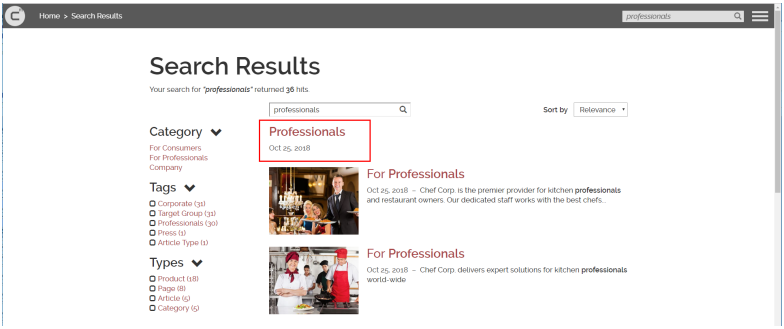

Figure 5.25. A Search Result for a Topic Page

Feeding Topic Pages

In the *Website Blueprint* topics are represented by `CMTaxonomy` and `CMLocTaxonomy` content. The *Blueprint CAE Feeder* is configured accordingly so they get indexed by the search engine.

Enable and Disable Topic Search

The *CoreMedia Blueprint* searches for topic pages by default. This feature can be enabled and disabled for each site independently. The folder `Options/Actions` contains an action "Search" which has a `StringListProperty` setting `search.topicsdoctypeselect`. The entries define all content type names which should be considered by a topic page search. In case of the *Website Blueprint* the content types `CMTaxonomy` and `CMLocTaxonomy` are required here. To avoid searching for topic pages, this setting has to be removed.

If you change the configuration of `search.topicsdoctypeselect` to a different non-empty list of content type names, then you should also add the setting `search.topicsindexfields` with a string list of search engine index fields that reference assigned tags of a content. These index fields are used to determine the tags that are used in the current site. If not set or empty, the setting defaults to the index fields `subjecttaxonomy` and `locationtaxonomy`, which reference `CMTaxonomy` and `CMLocTaxonomy` contents, respectively. For example, if you set `search.topicsdoctypeselect` to `CMTaxonomy` only, then should also set `search.topicsindexfields` to `subjecttaxonomy` only.

5.4.24 Search Landing Pages

NOTE

Feature is only supported in *eCommerce Blueprint*

Requirements

Using *CoreMedia Content Cloud* the user should have the possibility to define custom page layouts for search terms.

Solution

Search Landing Pages are used to apply a custom page layout for product searches that match specific search terms. This feature is used when CAE fragments should be included to search result pages of an eCommerce system. To provide a new search landing page, do the following:

1. Create a new folder with the name `Search Landing Pages` in one of your sites folders. The folder must be part of a site, global search landing pages are not provided.
2. Create a new page content item and add the matching keywords in the input field "HTML Keywords" (CMChannel property "keywords").
3. Add the newly created page content item as navigation child to the root content item. Ensure that the search landing page has checked the "Hidden in Sitemap" and "Hidden in Navigation" checkboxes.

When the search landing page is included to the commerce storefront, only the main placement of the page's page grid will be included as fragment.

5.4.25 Theme Importer

With the theme importer you can import themes into the content repository. It is the command line equivalent of uploading themes in Studio as described in the Frontend Developer Manual.

For basic information on the use of server utilities, refer to section "General usage" in *Content Server Manual*.

```
usage: cm import-themes -u <user> [other options] [-f <folder>] [-c] [-dm]
          <theme.zip> ...

available options:

 -?,--help                    Print usage information and quit.
 -c,--clean                   Delete existing theme before import in order
                              to get rid of obsolete code resources.
 -d,--domain <domain name>    domain for login (default=<builtin>)
 -dm,--development-mode       Development mode.  Creates a user (frontend
                              developer) specific copy of the theme.
 -f,--folder <arg>            Folder within CoreMedia where themes are
                              stored. Default is /Themes
 -g,--grpc-endpoint <addr>    gRPC endpoint address to connect to
 --http-base-uri <HTTP base URI> HTTP base URI of Content Server
                                  where HTTP servlets are provided
                                  (e.g. blob servlet)
 -p,--password <password>     password for login; you will be prompted for
                              password if not given
 -u,--user <user name>        user for login (required)
 --url <ior url>              url to connect to
```

Example 5.7. Usage of import-themes

The options have the following meaning:

Parameter	Description
`-c`	Delete existing theme before import in order to get rid of obsolete code resources. This option is only intended for the development workflow. It does not delete published themes.
`-dm`	Development mode. Creates a user (frontend developer) specific copy of the theme.
`-f`	Folder within CoreMedia where themes are stored. Default is `/Themes`

Table 5.18. Options of the import-themes tool

5.4.26 Tag Management

A Tag Management System is a tool to deploy analytics code fragments and others on your website dynamically during runtime. With the user interface of CoreMedia Studio it is easy to set up tag management systems depending on the configuration defined in the repository.

Configuration

Define a local or linked setting called "TagManagement" with type `Struct List` and add it to the root page of the site. Each struct in this list represents a Tag Management integration which defines its own `Markup` properties for "head", "bodyStart", and "bodyEnd".

The *chefcorp-theme* includes the `brick-tag-management` and the support for tag management systems out of the box.

If the *Google Tag Manager* is your preferred solution, refer to Google's documentation at Google Tag Manager Overview to retrieve the code snippets for your integration.

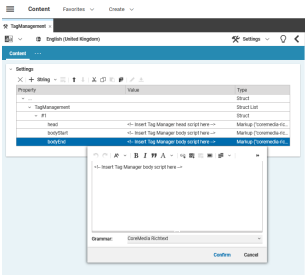

Figure 5.26. Tag Management Configuration

5.5 Localized Content Management

One of the primary challenges when engaging in a global market is to reach all customers in different countries.

The first most obvious task is to provide your website contents in different languages. But in addition you may also want to customize your advertised products to local holidays or meet the different legal requirements in different countries.

CoreMedia Content Cloud's Multi-Site concept assists you in meeting these requirements.

NOTE

Also have a look into the Multi-Site Manual. The manual describes different options to design your site hierarchy and gives some guidance to avoid common pitfalls when working with multi-site content.

5.5.1 Concept

There are many possible approaches to fulfill the requirements for providing multiple sites in different countries. *CoreMedia Content Cloud* offers a solution which you can customize to your needs and to the workflows you are used to.

The following chapter will present the basic ideas and concepts of *CoreMedia Content Cloud's* Multi-Site to you.

5.5.1.1 Terms

The multi-site concept and documentation is based on the following terms. You may skip this section for now and return to it later when these terms are referenced.

Derived Site	A derived site is a site, which receives updates from its master site. A derived site might itself take the role of a master site for other derived sites.
Home Page	The site's home page is the main entry point for all visitors of a site. Technically it is also the main entry point to calculate the default layout and the contents of a site.

Localization	In context of the multi-site concept, the main scope of sites within a web presence is to provide localized versions of content items. This does not only include the translation of content items but also the adaptation of content items to the local culture, customs and traditions. The latter may result in, for example, other images to reference or just different content items to be referenced for a given locale. In the context of *CoreMedia Blueprint*, the term *localization* often also serves as a generic term for the terms translation and synchronization. Thus, the term *localization workflow* refers to *translation workflows* as well as *synchronization workflows*.	
Locale	A locale is briefly a combination of a language and a country or region. Within a web presence, the site's locale must be unique. The locale is represented as IETF BCP 47 language tag (*Tags for Identifying Languages*).	*locale = language + country* *IETF BCP 47*
Master Site	A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites. This reflects the need that, for example, your localized Canadian site (which is in English) needs another localized variant in French.	
Root-Master Site	A root-master site is the top-level site in a multi-site hierarchy. The root-master site is the only site for a given web presence that is not derived from another site.	
Site	A site is a cohesive collection of web pages in a single locale, sometimes called localized site. Technically, a site consists of: • The site folder • The site indicator, • The site's home page and • Other contents of the site.	
Site Folder	All contents of a site are bundled in one dedicated folder. A typical example of a site folder is:	

`/Sites/MySite/Canada/French`

Site Indicator

A site indicator is the central configuration object for a site. It is an instance of the content type `CMSite`. It explicitly configures:

- The site's home page
- The site ID
- The site name
- The site's locale
- The master site
- The site manager groups.

It also implicitly defines the root of the site folder.

Site ID

The site ID needs to be unique among all sites. It can be used to reference a site reliably also outside the CMS, for example, in configuration files.

Site Manager Group

Members of a site manager group are typically responsible for one localized site. The recommendation is to have one dedicated group for each site with appropriate permissions applied for the site folder.

Responsible means that they take care of the contents of that site. This includes but is not limited to:

- adapting contents to local needs,
- accepting translation workflows, or
- triggering localization workflows, thus, synchronization or translation workflows to their site.

For the latter, the corresponding users need to be added to the translation manager role.

Global and Local Site Manager

While the Site Manager Groups are typically local to their site, there is another role that is eligible to manage all sites within a web presence. This role is called *global site manager*, while the other is referred to as *local site managers*.

Remark on permissions: While local site managers are typically managing only the contents

	in their site, they still need read access to the master site to be able to manage translation processes or to compare their local contents with the master site contents.
Site Name	The site name is the name of a master site and all derived sites. A derived site inherits the site name from its master site and must not change it.
Synchronization	Synchronization is a special type of localization process. It mostly means to provide one-to-one copies of content items for one site to a derived site with only minor adaptations to the derived content items. A typical example of two sites that are synchronized is that both share the same language but are in different countries. To synchronize content items, a built-in synchronization workflow is used, and the target derived site, that receives the synchronized content items, is referred to as *synchronization site*.
Translation	One element of the localization process is the translation of a content item from one locale to another. The term refers to the actual translation done by editors or translation agencies as well as, in the context of *CoreMedia Blueprint*, to the translation workflow that guides editors through the translation process. The target derived site, which receives the translation results, is referred to as *translation site*.
Translation Manager Role	Editors in the translation manager role are in charge of triggering localization workflows either from or to a site. Different to what the name suggests, the translation manager role also applies to the ability to start synchronization workflows.
Variants	The set of all content items related to each other via master references. This includes the top-level master content items themselves.

	In typical multi-site setups this definition can be simplified to all content items that share the same root-master content item directly or transitively via master references.
Web Presence	A web presence is a collection of all sites below the same root-master site — including the root-master site itself.

5.5.1.2 Sites Structure

CoreMedia Content Cloud organizes your sites for a given web presence in a strictly hierarchical, top-to-bottom structure starting with the root-master site. Connections between sibling nodes are unavailable and can only be expressed implicitly by a common ancestor.

The site hierarchy might be nested, thus a site derived from the master site again might have derivatives. You can trigger the localization process from your master site, directly derived sites will adapt and forward changes to their derived sites.

The examples below refer to the default configuration which comes with *CoreMedia Blueprint*. To adapt, for example, the auto-generated structure for when deriving new sites, you can adapt this and other settings in the `SiteModel` - see also Section 5.5.3.1, “Site Model and Sites Service” [252].

Multi-Site Folder Structure

All sites within a web presence share a dedicated root folder, that contains the root-master site as well as all its (nested) derived sites. Other web presences might be placed in parallel to this folder structure below the `/Sites/` folder.

```
/Sites/
  MySite/
    United States/
      English/ master
      Spanish/ derived from U.S. English
    Canada/
      English/ derived from U.S. English
      French/ derived from Canadian English
  MyOtherSite/ another master site structure
```

Example 5.8. Multi-Site Folder Structure Example

The folder structure of the root-master site and its derived sites is best kept equal to avoid the automatic recreation of removed or renamed folders during the localization workflows.

In addition to this, common aspects for all sites might be placed outside this folder structure. For details see Section 5.4.1, "Folder and User Rights Concept" [171].

Site Folder Structure

It is the location of the site indicator that defines the site folder. Thus, a folder containing a valid site indicator with all required properties set (like the site ID) is regarded as a site folder. The site indicator also references (as root document) the site's home page.

The site's home page is placed in `Navigation`, so that the key content item structure is like this:

```
MySite/
  United States/
    English/
      MySite [Site] site indicator
      Navigation/
        MySite site's home page
        ...
```

Example 5.9. Site Folder Structure Example

While the above describes the mandatory folder structure for a site, there are additional structures which adhere to the proposed separation of concerns in Section 5.4.1, "Folder and User Rights Concept" [171], thus within a site you can have several user roles taking care of different aspects of the site as there are:

- **Editorial content:** For example, articles, images, collections etc. This is the real content of a site that is rendered to the web page. They are located in folders `Editorial`, `Pictures` and `Videos`.
- **Navigation content:** Channels that span the navigation tree and provide context information, as well as their page grids (see also Section 5.4.2, "Navigation and Contexts" [173]). These contents are located in a folder named `Navigation`.
- **Technical content:** Site specific, technical content items, like actions, settings, view types, etc. They can be found in folder named `Options`.

Site Interdependence

Having a site derived from its master you will have two layers of interdependence:

1. The site indicator points to its master site indicator.
2. Each derived content item points to its master annotated by the version of the master when the derived content item retrieved its last update from the

master. This information is used in the update process when a new master version requires its derived contents to be localized again.

3. A site indicator inherits the site name from its master. If a site indicator has no master it has to define the site name, which will be used for all derived sites.

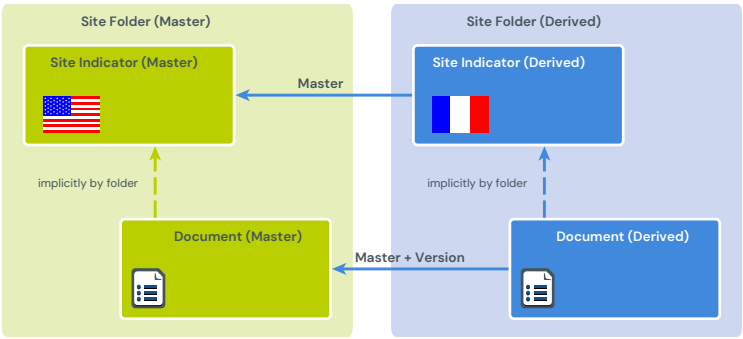

Figure 5.27. Multi-Site Interdependence

The `master` property is configured as weak link by default. Thus, you might publish derived sites before (or without) publishing the master site.

Modifying the Site Structure

Whenever possible, the structure of a site should not be changed after it has been set up initially. In particular, you should not:

- Change the ID of a site. If you do so, you must at least reindex its entire master site, if any. See Section 4.6.1, "Re-Indexing" in *Search Manual* for details on the reindexing procedure. However, the site ID might also be stored in other places that a simple reindexing will not update.
- Move a content to a different site. If you do so, you must at least update the master links of the affected contents to point into the master site of the new site.
- Change the locale of a site. If you do so, you must at least update the locale stored in each individual content of the site.
- Change the master site of a site. If you do so, you must at least update the master links of all contents in the site.

  For more information about options and challenges to restructure sites, see Section 4.10, "Changing the Site Hierarchy" in *Multi-Site Manual*.

Validate Sites

After significant changes of the site structure, you should run the `cm validate-multisite` tool to detect inconsistencies in the content. See Section 3.13.1.11, "Validate Multi-Site" in *Content Server Manual* for details.

5.5.2 Administration

Using *CoreMedia Content Cloud's* Multi-Site concept requires some administrative efforts which are described in this section.

5.5.2.1 Locales Administration

Each site is bound to a specific locale (see Locale [240]). In order to ensure a consistent usage of locale strings across multiple sites that might be managed in a single content repository, the entire list of available locales is maintained in a central content item of type `CMSettings`.

LocaleSettings content item for locale configuration

The content item `/Settings/Options/Settings/LocaleSettings` contains in the property `Settings` a String List property `availableLocales` which contains locale strings. Example 5.10, "XML of locale Struct" [246] shows the XML structure of the Struct:

```
<settings>
 <Struct xmlns="http://www.coremedia.com/2008/struct"
         xmlns:xlink="http://www.w3.org/1999/xlink">
   <StringListProperty Name="availableLocales">
     <String>de</String>
   </StringListProperty>
 </Struct>
</settings>
```

Example 5.10. XML of locale Struct

Please make sure, that the path to the `LocaleSettings` is configured in the Studio properties, as described in Section 9.24, "Available Locales" in *Studio Developer Manual*.

For providing a new locale, you can simply open the content item `LocaleSettings` and add a new entry to the list of locales. See Section 4.6.4, "Editing Struct Properties" in *Studio User Manual* for details on how to edit a struct property and add items to string lists. Figure 5.28, "Locales Administration in

CoreMedia Studio" [247] shows a Studio tab in which the `LocalesSettings` content item is being edited.

Supranational regions

Sometimes you might want to define locales for a supranational region such as Africa or Latin America. In this case you can add the language code followed by the UN M.49 area code as described in https://en.wikipedia.org/wiki/UN_M.49. For Spanish in Latin America and the Caribbean add, for example, "es-419".

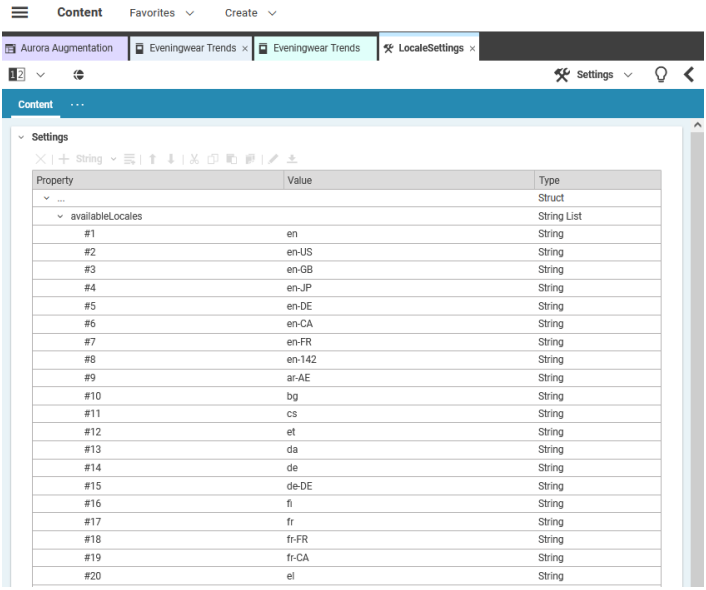

Figure 5.28. Locales Administration in CoreMedia Studio

5.5.2.2 Groups and Rights Administration

This chapter describes all groups and users, that have to be defined for localization. There are several explicit groups and one user, that can be configured in the `SiteModel`. See also Section 5.5.3.1, "Site Model and Sites Service" [252]. For an overview of predefined editorial users that come with *CoreMedia Blueprint* have a look at Reference - Predefined Users [439].

translation manager role

The translation manager role is defined once in the property `translationManagerRole` of the `SiteModel`. It is a required group for every user that needs to start a localization workflow and to derive a site.

global site manager

In case, you do not want to allow every translation manager to also derive sites, it is advisable to create an additional global site manager group, that has the right, to make modifications in the global sites folder.

site manager group

Members of a site manager group take care of the contents of one or more sites. They may for example accept translation workflows if they manage the corresponding target site of a workflow. Or they may start a localization workflow (be it synchronization or translation) from the master site. For the latter, they must also be member of the translation manager role group, which is described above.

Difference for Translation and Synchronization

Despite the required translation manager role, as stated above, a user must also be assigned as a site manager group member to some site.

For both, synchronization and translation, it is enough, if the user is denoted as manager of the master site. Both types of localization workflows offer each a different alternative to this:

- For translation workflows, users may alternatively also be set as site managers of all target sites of the workflow.
- For synchronization workflows, users may alternatively also be set as site managers in any of the master sites up in the hierarchy.

The site manager groups can be defined in the site indicator. The name of the corresponding property field is defined in the `siteManagerGroupProperty` of the `SiteModel`. Alternatives to this may be applied on application configuration level. By default, alternative members who may manage a site, must be member of the built-in group administratoren (group-identifier 0 (zero)). Also by default (other strategies may be defined) alternative members are taken into account, when none of the specified groups exist or if the corresponding property in the site indicators is empty. For corresponding configuration options, see Section 5.5.3.1, “Site Model and Sites Service” [252].

You may enter multiple groups separated by a comma by default. The separator is configured in `groupPropertyDelimitingRegEx` of the `SiteModel`.

There are two ways to set the site manager group:

- While deriving a new site in the sites window, you can set the group.
- Directly in the site manager group property of the site indicator.

translation workflow robot user

For technical reasons the actual changes during a localization workflow are performed as the translation workflow robot user as configured in the property `translationWorkflowRobotUser` of the `SiteModel`. The user needs

read and write access on the sites taking part in a localization workflow. As this user is only technical, access to the editor service should be restricted, which can be done in the file `jaas.conf` in the module `content-management-server-blueprint-config`. (For details see Section 3.12.1.1, "LoginModule Configuration in jaas.conf" in *Content Server Manual*). Additionally, the username should be added to the property `userchanges.excluded-user-names` of the *User Changes Application* to prevent superfluous generation of *My Edited Content*.

Overview of required users and groups for multi-site

Table 5.19, "Suggested Users and Groups for multi-site" [249] shows an example, how the configuration of user groups may look like in *CoreMedia Blueprint*.

Type	Name	Member of	Rights	Remark
group	global-site-manager	approver-role, publisher-role, translation-manager-role	• /Home (folder: RSF, content: RMDS) • /Settings (folder and content: R) • /System (folder and content: R) • /Sites (folder: RAPSF, content: RMDAPS)	
group	local-site-manager	approver-role, publisher-role, translation-manager-role	• `/Home` (folder: RSF, content: RMDS) • `/Settings` (folder and content: R) • `/System` (folder: RF, content: RMD) • `/Themes` (folder: RAPF, content: RMDAP) • `/Sites/<master-site-root-folder>` (folder and content: R)	
group	manager-<language-tag>	local-site-manager	• `/Sites/<site-root-folder>` (folder: RAPF, content: RMDAP)	Suggested pattern configured in `siteManagerGroupPattern` of the `SiteModel`

Type	Name	Member of	Rights	Remark
group	translation-manager-role			Configured in `translationManagerRole` of the `SiteModel`
group	translation-workflow-ro-bots		• / (folder and content: R) • `/Sites` (folder: RFAS, content: RMDAS)	Group for automatic multi-site localization actions. This group requires supervise permissions in order to assign rights to newly created sites (deriving sites, see Section 5.7.3, "Deriving Sites" [326]). Approve rights are required for localization processing, not for publishing to Master Live Servers. So, publish permissions are not required for this group.
user	translation-workflow-ro-bot	translation-workflow-ro-bots		Configured in `translationWorkflowRobotUser` of the `SiteModel`

Table 5.19. Suggested Users and Groups for multi-site

The rights abbreviations denote:

- R - read
- M - modify / edit
- D - delete
- A - approve
- P - publish
- F - folder
- S - supervise

For further information about the rights, please refer to Section 3.15.2, "User Rights Management" in *Content Server Manual*.

Definition while deriving site

When deriving a new site, a proposal for the name of the site manager group is generated from a predefined pattern. By default, the name starts with *manager* followed by the language tag of the selected target locale (see also Figure 5.29, “Derive Site: Setting site manager group” [251]). This pattern may be configured in the property `siteManagerGroupPattern` of the `SiteModel`.

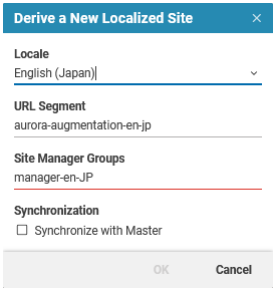

Figure 5.29. Derive Site: Setting site manager group

Adapting site manager group later on

If the site already exists, the names of site manager groups can be set or modified directly in the site indicator (see Figure 5.30, “Site Indicator: Setting site manager group” [251]).

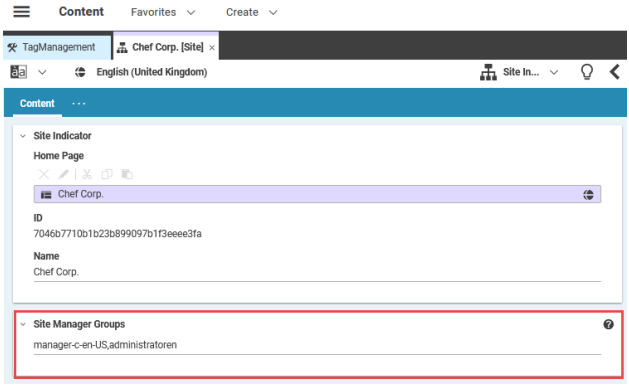

Figure 5.30. Site Indicator: Setting site manager group

If any of the given groups does not exist, the property field will be marked red and the creation of the site or the assignment of the group may not be performed, thus the groups need to have been created before. Read more about users, groups and administration in Section 3.15, "User Administration" in *Content Server Manual*.

5.5.3 Development

CoreMedia Content Cloud's Multi-Site concept contains an example implementation for translation and localization processes. As you might have different requirements, for example defined by a translation agency which does the translation for you, the Multi-Site feature is highly configurable. Read the following sections to learn about the configuration options.

5.5.3.1 Site Model and Sites Service

The site model, and the sites service are strongly connected with each other. While the site model consists of properties defining the site structure, the sites service uses this model to work with sites programmatically.

Sites Service

The sites service is designed to access the available sites and to determine the relation between sites and contents. The site model configures the behavior of the sites service.

Contrary to what the name suggests, the sites service is not a service operating in the background. It is an API to bundle access to the multi-site feature.

Site Model

The site model is the centralized configuration of the CoreMedia Multi-Site behavior. Its configuration is required in all applications including the Command Line Tools.

While the server applications are configurable via Spring `application.properties` as usual, the configuration for the Command Line Tools can be adjusted in the `commandline-tools-sitemodel.properties` file in the `properties/corem` directory of the `cms-tools-application` module.

Site Model Properties

The following table illustrates the configurable site model properties. To get to know more about the properties and patterns used, consult the Javadoc of `com.coremedia.cap.multisite.SiteModel`.

`sitemodel.site.indicator.documentType`	
Description	Specifies the content type of the site indicator content item. Each site must only have one instance of that content type.
Default Value	*`CMSite`*
`sitemodel.site.indicator.depth`	
Description	Defines the depth under the root of the site folder, where the site indicator content item resides. Note, that it is recommended to leave this to the default value of 0 (zero), as experience shows, that editors find it easier to understand the site structure, when the site indicator is located in the site root folder.
Default Value	0
`sitemodel.site.indicator.namePattern`	
Description	Name pattern, which will be used for the name of the site indicator content item when deriving a site. Only placeholder *`{0}`* is available for this property. For an overview of placeholders see Table 5.21, "Placeholders for Site Model Configuration" [257].
Default Value	*`{0} [Site]`*
`sitemodel.site.rootdocument.namePattern`	
Description	Defines the pattern for the site's home page content item name, used while deriving a site. Only placeholder *`{0}`* is available for this property. For an overview of placeholders see Table 5.21, "Placeholders for Site Model Configuration" [257].
Default Value	*`{0}`*
`sitemodel.site.manager.groupPattern`	

Description	Defines the pattern for responsible default site manager group name when deriving a site. For available placeholders see Table 5.21, "Placeholders for Site Model Configuration" [257]
Default Value	*manager-{4}*

`sitemodel.siteManagerGroupProperty`

Description	Defines the property of the site indicator content item holding the site manager group names.
Default Value	*siteManagerGroup*

`sitemodel.siteManagerGroupAlternatives`

Description	Defines groups to consider as an alternative (depending on the configured strategy, see `sitemodel.siteManagerGroupAlternativesStrategy`), that may take over managing a given site. Groups may be referenced by name (name-at-domain), by CoreMedia group-ID or by UUID.
Default Value	*coremedia:///cap/group/0*

`sitemodel.siteManagerGroupAlternativesStrategy`

Description	Defines the strategy when to consider alternative site manager groups as configured by `sitemodel.siteManagerGroupAlternatives`. Prior to describing them in more detail, the options to choose from are: `ALWAYS`, `IF_EMPTY`, `IF_EMPTY_OR_INVALID`, `NEVER`. More details on the strategies, when configured groups, and their respective members are considered to take over the role of managing a site, are as follows: `ALWAYS`: Will always also add alternative groups. `IF_EMPTY`: Will only consider alternative groups, when the site manager group property is empty. `IF_EMPTY_OR_INVALID`: Will only consider alternative groups, when the site manager group property is either empty or all referenced groups are invalid (i.e., they do not exist). `NEVER`: Will never add alternative groups.

Default Value	*IF_EMPTY_OR_INVALID*
sitemodel.groupPropertyDelimitingRegEx	
Description	Defines the separator (as regular expression) how to split group names into elements, as for example for the site-manager-groups.
Default Value	*comma, including trailing space characters*
sitemodel.translationManagerRole	
Description	Defines the group name denoting the role that permits a user to start a translation workflow.
Default Value	*translation-manager-role*
sitemodel.idProperty	
Description	Defines the property of the site indicator content item that contains the site id.
Default Value	*id*
sitemodel.nameProperty	
Description	Defines the property of the site indicator content item that contains the site name.
Default Value	*name*
sitemodel.localeProperty	
Description	Defines the property of translatable content, and the site indicator content item, which holds the locale of the content.
Default Value	*locale*
sitemodel.masterProperty	
Description	Defines the property of translatable content and the site indicator, which contains the link the master content item.

Default Value	*master*

sitemodel.masterVersionProperty

Description	Defines the property of translatable content, which contains the version the corresponding master content item.
Default Value	*masterVersion*

sitemodel.rootProperty

Description	Defines the property of the site indicator content item, which refers to the home page content item of this site.
Default Value	*root*

sitemodel.uriSegmentProperty

Description	Defines the property of the site's home page content type, which defines the root URI segment of the site.
Default Value	*segment*

sitemodel.uriSegmentPattern

Description	Defines the pattern for the default root URI segment when deriving a site. For available placeholders see Table 5.21, "Placeholders for Site Model Configuration" [257].
Default Value	*{0}-{4}*

sitemodel.rootFolderPathPattern

Description	Defines the pattern to determine the site folder for a new derived site. For available placeholders see Table 5.21, "Placeholders for Site Model Configuration" [257].
Default Value	*/Sites/{0}/{6}/{5}{7}*

sitemodel.rootFolderPathDefaultCountry

Description	Defines the folder name for the country folder, if the locale chosen while deriving a site defines no country explicitly.
Default Value	*NO_COUNTRY*
sitemodel.translationWorkflowRobotUser	
Description	Defines the username of the user responsible for creating derived content during a translation workflow. • The user should have read / write access on all localizable Sites. • The user should not be allowed to use the editor service (configuration via file `jaas.conf` in the module `content-management-server-blueprint-config`). • The *User Changes Application* should not generate superfluous My Edited Content for the user (configuration via property `userchanges.excluded-user-names`).
Default Value	*translation-workflow-robot*

Table 5.20. Properties of the Site Model

Site Model Placeholders

Placeholder	Description	Example
{0}	site name	*MySite*
{1}	site locale's language code	*en*
{2}	site locale's country code (defaults to language code, if not available)	*US*
{3}	site locale's variant (defaults to country or language code, if not available); using BCP 47 Extensions	*u-cu-usd*
{4}	site locale's IETF BCP 47 language tag	*en-US-u-cu-usd*

Placeholder	Description	Example
`{5}`	site locale's language display name (localized in U.S. English); only available for `sitemodel.rootFolderPathPattern`	*English*
`{6}`	site locale's country display name (localized in U.S. English); only available for `sitemodel.rootFolderPathPattern`	*United States*
`{7}`	site locale's variant with the prefix `variantPrefix` configured in site model's Spring context; defaults to empty String; only available for `sitemodel.rootFolderPathPattern`. See IANA Language Subtag Registry for valid registered variants.	*_arevela*

Table 5.21. Placeholders for Site Model Configuration

5.5.3.2 Content Type Model

While you might create your very own content type model, the following description is based on the assumption that you use the content type model of *CoreMedia Blueprint*. For a custom content type model you must meet certain requirements which are described at the end of this section.

Content Types

`CMLocalized` is the parent content type that defines all required properties for the multi-site feature. Any content item in any site must be of a type that is a child of `CMLocalized`. Any content items within a site violating this requirement, may trigger improper or unexpected behavior during operational tasks such as translating or synchronizing content items. For further information see Section 7.1, “Content Type Model” [430].

```
<DocType Name="CMLocalized" Parent="CMObject" Abstract="true">
  <StringProperty Name="locale" Length="64"/>
  <LinkListProperty Name="master" Max="1"
                    LinkType="CMLocalized"
                    extensions:weakLink="true"/>
  <IntProperty Name="masterVersion"/>
  ...
</DocType>
```

Example 5.11. CMLocalized

Weak Link Attribute

The contents of each site have to be published and withdrawn independently of their master. Therefore, the `weakLink` attribute of every master property must be set to true - see also Content Type Model - LinkListProperty in *Content Server Manual*.

Attributes for Multi-Site

The attributes `extensions:translatable` and `extensions:automerge`, which can be attached to all properties, affect the translation and synchronization behavior.

Both attributes have no effect when you derive a site. A derived site is a complete copy of the master site. Only links are changed, so that they point to content in the derived site and not to content in the master site.

The attribute `extensions:variantStable` can also be attached to all properties and affects which properties are treated as stable across variants.

Translatable Properties

The properties that have to be translated to derived sites can be marked as translatable in the content type model by attaching the `extensions:translatable` attribute to the property declaration - see also Content Type Model - Translatable Properties in *Content Server Manual*.

section “Translatable Predicate” [268] describes other ways to mark a property as translatable, for example to mark nested properties of a Struct property as translatable.

```
<DocType Name="CMTeasable" Parent="CMHasContexts" Abstract="true">
  <LinkListProperty Name="master" Max="1"
                    LinkType="CMTeasable"
                    Override="true"
                    extensions:weakLink="true"/>
  <StringProperty Name="teaserTitle" Length="512"
                  extensions:translatable="true"/>
  <XmlProperty Name="teaserText" Grammar="coremedia-richtext-1.0"
               extensions:translatable="true"/>
  <XmlProperty Name="detailText" Grammar="coremedia-richtext-1.0"
               extensions:translatable="true"/>
  ...
</DocType>
```

Example 5.12. CMTeasable

Automatically Merged Properties

Usually all property changes from the master content will be merged automatically to the derived content when a translation task is accepted. To disable

automatic merging for a property, set the *extensions:automerge* attribute to `false` as described in Content Type Model - Translatable Properties in *Content Server Manual*.

You can also disable updates for a specific derived content item with the checkbox *Disable updates from master* in the *Localization* tab in Studio (see Section 4.7.3.1, "Disable Updates From Master" in *Studio User Manual* for details).

Variant Stable Properties

Properties that should not differ between a baseline content item and any of its variants can be marked as stable by attaching the `extensions:variantStable` attribute to the property declaration - see also Content Type Model - Variant Stable Properties in *Content Server Manual*.

When a property is marked as stable on a parent type, all subtypes inherit that annotation. Alternatively, stable properties can be configured per content type via application properties.

Custom Content Type Models

Even if it is not recommended, you can use your own content type model with the Multi-Site feature of *CoreMedia Content Cloud*. Prerequisite is, that you can configure the Site Model mentioned before to meet the requirements of your own content type model. In addition, you probably need to adapt your content type model to fit the requirements of the multi-site concept.

Therefore, every content type, which may occur in a site must contain all properties, listed below.

- master
- masterVersion
- locale

Please adapt the configuration of each property to the properties of `CMLocalized` in the example above.

5.5.3.3 ServerImport and ServerExport

Both `serverimport` and `serverexport` have a special handling built in for the `master` and `masterVersion` properties. The export will store the localization state of a derived content item and on import efforts are taken to reestablish a comparable localization state.

Localization State in Server-Export XML

Within XML export files, the localization state is represented by the attribute `translationState`. Other, then the name suggests, this attribute applies to both: translation and synchronization.

For the concrete names of the `master` and `masterVersion` properties, the `SiteModel` has to be provided to the tools, which is done via *Spring* in the file `commandline-tools-context.xml` that is part of the `cms-tools-application` module in *CoreMedia Blueprint*.

Examples:

The examples assume that you export a content item and its master and import it afterward into a clean system. The table uses `#` (hash mark) to denote contents having the given latest version number.

Master (before) The master version before export. *none* means that no master link is set.

Version (before) Value of the master version property of the derived content item before export. *none* means that no version is specified yet which actually marks derived content items as not being up to date with its master content item.

State (before) The localization state of the derived content item before export.

Master (after) The master version after import (actually always `#1`).

Version (after) Value of the master version property of the derived content item after import.

State (after) The localization state of the derived content item after import.

Master (before)	Version (before)	State (before)	Master (after)	Version (after)	State (after)	Comment
#5	*5*	up-to-date	*#1*	*1*	up to date	The master and derived content item were up to date before export (derived content item is the most recent localiz-

Master (before)	Version (before)	State (before)	Master (after)	Version (after)	State (after)	Comment
						ation of its master). Thus, after import the same state is set.
#5	*4*	not up-to-date	*#1*	*0*	master version destroyed	The master and derived content item were not up to date before export. Thus, after import the value of the master version property is set to a special version number denoting that the derived content is not up-to-date. On API level this is regarded as if the referred master version got destroyed meanwhile. For the editor the content item will appear as being not up-to-date.
#5	*none*	not translated yet	*#1*	*none*	not translated yet	Derived content item was never localized from its master. Thus, the same state applies after import.
none	*5*	no master	*none*	*5*	no master	Corrupted content: No special logic is applied. The overall approach for import and export is defensive thus if the state was invalid before, the fallback is to use the default behavior from import and export keeping the values as is.

Table 5.22. Example for server export and import for multi-site

5.5.3.4 XLIFF Integration

Translation jobs can be represented using the XLIFF, the *XML Localization Interchange File Format*. XLIFF is an OASIS standard to interchange localizable data

for tools as for example used by translation agencies. An XLIFF file contains the source language content of translatable properties from one or more content items. It is then enriched by a translation agency to contain the translated content, too. *CoreMedia Content Cloud* supports XLIFF 1.2.

XLIFF Structure

An XLIFF file is structured into multiple translation units. While a string property is encoded as a single translation unit, a richtext property is split into semantically meaningful parts, comprising for example a paragraph or a list item. Translation units are then grouped, so that units belonging to a single property are readily apparent.

All properties of a single content item are included in a single file section according to the XLIFF standard. A custom attribute `cmxliff:target` allows the importer to identify the target content item that should receive the translation, as supported by the XLIFF standard. Translation tools must preserve this extension attribute when filling the target content into the XLIFF file.

For details on the standard and extended XLIFF schemas, see the corresponding Javadoc of `XliffSchemaConfiguration`. Refer to the 'See also' sections for links to the schemas.

The fragment in Example 5.13, “XLIFF fragment” [263] shows the start tag of a `<file>` element for translating from English to French, indicating the source document 222 and the target document 444.

```
<file
  xmlns:cmxliff=
     "http://www.coremedia.com/2013/xliff-extensions-1.0"
  original="coremedia:///cap/version/222/1"
  source-language="en"
  datatype="xml"
  target-language="fr"
  cmxliff:target="coremedia:///cap/content/444">
```

Example 5.13. XLIFF fragment

The elements to translate are grouped in `<trans-unit>` elements, which consists of a `<source>` element containing the original text and will later contain the translated text inside a `<target>` element.

XLIFF Export

In order to export XLIFF as part of a workflow action, the following actions need to be taken:

1. Pre-Processing: Translation Items: extract properties and partial properties (Structs) to be translated, and
2. Generating XLIFF Document: transform these properties into an XLIFF representation.

Pre-Processing: Translation Items

A translation item represents a content item during the translation process. It is meant to contain only those properties or partial properties (for Structs) which should be translated. Any filtering for example of empty properties (and deciding what *empty* actually means) is done here.

```
Map<Locale, List<TranslateItem>> getTranslationItemsByLocale(
        Collection<ContentObject> masterContentObjects,
        Collection<Content> derivedContents,
        Function<ContentObjectSiteAspect, Locale> localeMapper) {

    ContentToTranslateItemTransformer transformer =
            getSpringContext()
                    .getBean(ContentToTranslateItemTransformer.class);

    return transformer
            .transform(
                    masterContentObjects,
                    derivedContents,
                    localeMapper,
                    ITEM_PER_TARGET
            )
            .collect(
              Collectors.groupingBy(TranslateItem::getSingleTargetLocale)
            );
}
```

Example 5.14. Transforming to Translation Items

In Example 5.14, "Transforming to Translation Items" [264] you see a typical example used within some workflow action to generate a representation as translation items. It uses the Spring context of the workflow server (retrieved via `getSpringContext()` from `SpringAwareLongAction`) to retrieve a bean of type `ContentToTranslateItemTransformer`.

The `ContentToTranslateItemTransformer` is prepared for typical translation workflow scenarios, where you have a set of master content objects (contents or versions) to translate and a set of target contents which should receive the translation result.

As configuration options the transformation process takes a strategy to determine the language from the master content objects as well as from the derived contents and a flag how to group the results (see `TransformStrategy`). For XLIFF export the recommended flag is `ITEM_PER_TARGET`.

The result in the given example will be grouped by target locale, which allows combining all translation items into one XLIFF document for one common target locale.

```
static Locale preferSiteLocale(ContentObjectSiteAspect aspect) {
  Site site = aspect.getSite();
  if (site == null) {
    return aspect.getLocale();
  }
  return site.getLocale();
}
```

Example 5.15. Function to Determine Locales

In Example 5.15, "Function to Determine Locales" [265] you see a recommended function used as `localeMapper` passed to the method defined in Example 5.14, "Transforming to Translation Items" [264] to determine the locale (source as well as target) to set within XLIFF. The method prefers the site locale over the locale within a content item.

Generating XLIFF Document

Just as the `ContentToTranslateItemTransformer` the `XliffExporter` bean is available in Spring context.

```
Path exportToXliff(Locale sourceLocale,
                   Map.Entry<Locale, List<TranslateItem>> entry)
        throws IOException {

  XliffExporter xliffExporter =
          getSpringContext().getBean(XliffExporter.class);
  String targetLanguageTag = entry.getKey().toLanguageTag();
  List<TranslateItem> items = entry.getValue();

  // Provide some meaningful name.
  String baseName = ...;

  Path xliffPath =
    Files.createTempFile(
          baseName,
          "." + sourceLocale.toLanguageTag() + "2" +
          targetLanguageTag + ".xliff"
    ).toAbsolutePath();

  try (Writer xliffWriter =
          Files.newBufferedWriter(xliffPath, StandardCharsets.UTF_8)) {
    xliffExporter.exportXliff(
            items,
            xliffWriter,
            XliffExportOptions.xliffExportOptions()
                    .option(XliffExportOptions.EMPTY_IGNORE)
                    .option(XliffExportOptions.TARGET_SOURCE)
                    .build());
  }
```

```
  return xliffPath;
}
```

Example 5.16. Exporting XLIFF

In Example 5.16, "Exporting XLIFF" [265] you see how you may transform the translation items generated above into an XLIFF document which you may then upload to the translation service.

The parameter `sourceLocale` is just required to generate some meaningful, easy to debug filename. If your translation service preserves these filenames, it may be useful to generate a name, which can easily be identified later on. That is why the filename in the example contains information such as the source as well as the target language.

When invoking the `XliffExporter` you have several options to choose from, to control the output of XLIFF. In the given example empty trans-unit nodes should be ignored (they will not be part of the XLIFF export) and the target nodes will contain the same value as the source node, which is recommended by some translation services. For additional options have a look at `XliffExportOptions`.

XLIFF Import

In order to import XLIFF and apply the translation to the target content items, you will use the `XliffImporter`.

```
boolean importXliffFile(InputStream inputStream) {
  XliffImporter importer = getSpringContext()
          .getBean(XliffImporter.class);

  List<XliffImportResultItem> resultItems;

  try {
    resultItems = asRobotUser(() -> importer.importXliff(inputStream));
  } catch (CapXliffImportException e) {
    LOG.warn("Failed to import XLIFF.", e);
    return false;
  }

  List<XliffImportResultItem> majorIssues =
          XliffImportResults.getMajorIssues(resultItems);

  if (!majorIssues.isEmpty()) {
    LOG.warn("XLIFF has major issues: {}", majorIssues);
    return false;
  }

  return true;
}
```

Example 5.17. Importing XLIFF

Example 5.17, "Importing XLIFF" [266] shows how you may import a received XLIFF document within a workflow action. The example is used within an action to poll the results from the translation service and which will automatically apply XLIFF results as soon as they are available. The implementation assumes that it should repeat the XLIFF download until the download was successful.

To start get the `XliffImporter` bean from Spring context via `getSpringContext()` from `SpringAwareLongAction`.

The `XliffImporter` provides two ways to signal problems. While exceptions are typically related to for example I/O problems, internal problems such as missing translations, target content items which are gone, and so on, are reported as major issues. To help rating and analyzing the issues there is a utility class `XliffImportResults`, which will for example filter any relevant issues for you.

Note, that you may want to store issues in a workflow variable instead to present the information in *CoreMedia Studio* as part of the workflow detail panel.

If the workflow action runs in an automated task, it needs some user to update contents according to the translation result. It depends on your actual design of the action and the workflow process, which user should be taken. In the example, the translation workflow robot user is reused, which is part of the site model.

```
<T> T asRobotUser(Supplier<T> run) {
  User robotUser = getRobotUser();

  // Perform content operations in the name of the robot user.
  CapSession session = getCapSessionPool().acquireSession(robotUser);

  try {
    CapSession oldSession = session.activate();
    try {
      return run.get();
    } finally {
      oldSession.activate();
    }
  } finally {
    getCapSessionPool().releaseSession(session);
  }

}

User getRobotUser() {
  SiteModel siteModel = getSpringContext().getBean(SiteModel.class);
  String robotUserName = siteModel.getTranslationWorkflowRobotUser();
  UserRepository userRepository = getConnection().getUserRepository();

  User robotUser = userRepository.getUserByName(robotUserName);

  // Recommended: Add a check that the robotUser actually exists.

  return robotUser;
}
```

Example 5.18. Importing XLIFF

Example 5.18, "Importing XLIFF" [267] sketches a possible implementation. `asRobotUser` will switch the session to a user session using the `CapSessionPool` provided by `LongActionBase` to execute the given supplier.

`getRobotUser` uses the site model to get the artificial robot user to perform the import action.

XLIFF Customization

You may customize XLIFF handling at various extension points:

- you may modify strategies to identify translatable properties, or
- you may change representation in XLIFF export, or
- you may adapt XLIFF import for special property types.

Assumptions

In here you will see a small use case which leads you through all required adaptations to take to your system. The use case is based on the following assumptions:

- you introduced a markup property `xhtml` with XHTML grammar (`http://www.w3.org/1999/xhtml`) to your content type model,
- you do not support attributes for XHTML elements, and
- only markup properties named `xhtml` should be considered for translation (in other words: you do not use the translatable attribute inside your content type model).

In section "Handling Attributes" [278] this simple example is continued by adding the attributes, as this is an expert scenario which requires taking care of XLIFF validation.

Translatable Predicate

To add the property to the translation process, you have to mark it as translatable. There are several ways of doing so, where the default one is to mark the property as translatable within the content type model (using `extension:translatable=true`). As alternative to this you may add a custom bean of type `TranslatablePredicate` to your application context, which will then be ORed with all other existing beans of this type. To completely override the behavior, you need to override the bean named `translatablePredicate` in `TranslatablePredicateConfiguration`.

The Blueprint contains a default predicate that can be configured with property `translatable.predicate.property-path` and that makes it possible to mark only certain nested properties of Struct properties as translatable. See section "XLIFF Configuration" [284] for details.

Translation Items

Having marked the property `xhtml` as translatable, as described in section "Translatable Predicate" [268] and trying to export a content item containing the `xhtml` property, you will see a `CapTranslateItemException` as in Example 5.19, "Example for CapTranslateItemException" [269].

```
com.coremedia.translate.item.CapTranslateItemException:

Property 'xhtml' (type: MARKUP) of content type 'MyDocument' is configured
to be considered during translation but a corresponding transformer of type
'com.coremedia.translate.item.transform.TranslatePropertyTransformer'
cannot be found.

Please consider to add an appropriate transformer bean to your Spring context.
```

Example 5.19. Example for CapTranslateItemException

The exception tells you which steps to do next, in order to support the `xhtml` property during XLIFF export: You have to add a `TranslatePropertyTransformer` to your Spring context.

Some default transformers for standard CoreMedia property types are configured in `TranslateItemConfiguration`. One of them is the `AtomicRichtextTranslatePropertyTransformer` for CoreMedia RichText properties.

Just as the `AtomicRichtextTranslatePropertyTransformer` the transformer can be based on `AbstractAtomicMarkupTranslatePropertyTransformer` which ends up that you only have to specify how to match the grammar name and to add a predicate to decide if the property should be considered empty or not.

```
public class AtomicXhtmlTranslatePropertyTransformer
        extends AbstractAtomicMarkupTranslatePropertyTransformer {
  private static final String XHTML_GRAMMAR_NAME =
          "http://www.w3.org/1999/xhtml";

  private AtomicXhtmlTranslatePropertyTransformer(
          TranslatablePredicate translatablePredicate,
          Predicate<? super Markup> emptyPredicate) {
    super(
            translatablePredicate,
            AtomicXhtmlTranslatePropertyTransformer::isXhtml,
            emptyPredicate
    );
  }

  public AtomicXhtmlTranslatePropertyTransformer(
```

```
          TranslatablePredicate translatablePredicate) {
    this(translatablePredicate,
         AtomicXhtmlTranslatePropertyTransformer::isEmpty);
  }

  private static boolean isEmpty(@Nullable Markup value) {
    return !MarkupUtil.hasText(value, true);
  }

  private static boolean isXhtml(XmlGrammar grammar) {
    return XHTML_GRAMMAR_NAME.equals(grammar.getName());
  }
}
```

Example 5.20. TranslatePropertyTransformer for XHTML

Now you have a ready-to-use strategy to add your `xhtml` property to your translation items.

Property Hints

Transformed properties may hold meta information which are required for later processing. Typical scenarios are to tell the translation service how many characters are allowed within a String or to tell the XLIFF importer which grammar is used for the contained Markup.

For details have a look at `TranslateProperty` and `PropertyHint`.

XLIFF Export

As before, an exception provides further guidance how to continue as can be seen in Example 5.21, “Example for CapXliffExportException” [270].

```
com.coremedia.cap.translate.xliff.CapXliffExportException:

Missing XLIFF export handler for property
  'TranslateProperty[
          id = property:markup:xml,
          propertyHints =
              [GrammarNameHint[value = http://www.w3.org/1999/xhtml]],
          value = <html xml:lang="en" ...]'.

Please consider adding an appropriate handler of type
  'com.coremedia.cap.translate.xliff.handler.XliffPropertyExportHandler'
  to your Spring context.
```

Example 5.21. Example for CapXliffExportException

You now have to implement a strategy to transform XHTML into a representation which contains the texts to translate as XLIFF `<trans-unit>` elements. The strategy is created by implementing `XliffPropertyExportHandler`.

```
public class XliffXhtmlPropertyExportHandler
        implements XliffPropertyExportHandler {
  private static final String XHTML_GRAMMAR_NAME =
          "http://www.w3.org/1999/xhtml";

  @Override
  public boolean isApplicable(TranslateProperty property) {
    return property.getValue() instanceof Markup &&
            property.getGrammarName()
                    .map(XHTML_GRAMMAR_NAME::equals)
                    .orElse(false);
  }

  @Override
  public Optional<Group> toGroup(TranslateProperty property,
                                 Supplier<String> idProvider,
                                 XliffExportOptions xliffExportOptions) {
    return Optional.ofNullable(
            toGroup(property.getId(),
                    (Markup) property.getValue(),
                    idProvider,
                    xliffExportOptions
            )
    );
  }

  @Nullable
  private static Group toGroup(String id,
                               Markup value,
                               Supplier<String> idProvider,
                               XliffExportOptions xliffExportOptions) {
    if (!MarkupUtil.hasText(value, true) &&
            xliffExportOptions.isEmptyIgnore()) {
      // Don't add anything to translate for empty XHTML.
      return null;
    }

    Group markupGroup =
            new XhtmlToXliffConverter(idProvider, xliffExportOptions)
                    .apply(value);

    Group result = new Group();
    // Required: Set the property name. This is important for
    // the import process later on, to identify the property
    // to change.
    result.setResname(id);
    result.setDatatype(DatatypeValueList.XHTML.value());

    result.getGroupOrTransUnitOrBinUnit().add(markupGroup);

    return result;
  }

}
```

Example 5.22. PropertyExportHandler for XHTML

Example 5.22, "PropertyExportHandler for XHTML" [271] contains the base for such a handler. It especially creates an XLIFF group object, which contains the property name. For mapping XHTML to XLIFF the handler delegates to another class in this case, as this is the most complex task during XLIFF export.

```
public class XhtmlToXliffConverter implements Function<Markup, Group> {
  private final Supplier<String> transUnitIdProvider;
  private final XliffExportOptions xliffExportOptions;
```

```
public XhtmlToXliffConverter(Supplier<String> transUnitIdProvider,
                             XliffExportOptions ) {
  this.transUnitIdProvider = transUnitIdProvider;
  this.xliffExportOptions = xliffExportOptions;
}

@Override
public Group apply(Markup markup) {
  XhtmlContentHandler handler = new XhtmlContentHandler();
  markup.writeOn(handler);
  return handler.getResult();
}

private class XhtmlContentHandler extends DefaultHandler {
  // First element in stack is the currently modified element.
  private final Deque<Object> xliffStack = new LinkedList<>();
  private Group result;

  @Override
  public void startElement(String uri,
                           String localName,
                           String qName,
                           Attributes attributes) {
    String typeId = localName.toLowerCase(Locale.ROOT);
    Group elementWrapper = new Group();
    if (result == null) {
      result = elementWrapper;
    }
    elementWrapper.setType(getType(typeId));
    addElement(elementWrapper);
  }

  private void addElement(Object elementWrapper) {
    Object first = xliffStack.peekFirst();
    if (first instanceof ContentHolder) {
      ContentHolder contentHolder = (ContentHolder) first;
      contentHolder.getContent().add(elementWrapper);
    }
    xliffStack.push(elementWrapper);
  }

  @Override
  public void endElement(String uri,
                         String localName,
                         String qName) {
    // remove element from xliffStack

    while (!xliffStack.isEmpty()) {
      Object object = xliffStack.pop();
      if (object instanceof TypeHolder) {
        TypeHolder typeHolder = (TypeHolder) object;
        if (getType(localName).equals(typeHolder.getType())) {
          break;
        }
      }
    }
  }

  @Override
  public void characters(char[] ch,
                         int start,
                         int length) {
    String content = new String(ch, start, length);

    TransUnit transUnit = requireTransUnit();

    transUnit.getSource().getContent().add(content);
    if (xliffExportOptions.getTargetOption() ==
            XliffExportOptions.TargetOption.TARGET_SOURCE) {
      transUnit.getTarget().getContent().add(content);
```

```
      }
    }

    private TransUnit requireTransUnit() {
      Object first = xliffStack.peekFirst();
      TransUnit transUnit;
      if (first instanceof TransUnit) {
        transUnit = (TransUnit) first;
      } else {
        transUnit = new TransUnit();
        transUnit.setId(transUnitIdProvider.get());
        transUnit.setSource(new Source());
        addElement(transUnit);
        if (xliffExportOptions.getTargetOption() !=
                XliffExportOptions.TargetOption.TARGET_OMIT) {
          transUnit.setTarget(new Target());
        }
      }
      return transUnit;
    }

    private String getType(String typeId) {
      return "x-html-" + typeId.toLowerCase(Locale.ROOT);
    }

    private Group getResult() {
      // possibly check that the result is actually set
      return result;
    }
  }
}
```

Example 5.23. XhtmlToXliffConverter

Example 5.23, "XhtmlToXliffConverter" [271] is a very basic implementation of such a converter (it ignores for example any attributes). It contains the most relevant aspects, that you have to take into account while generating XLIFF structures.

Elements of XhtmlToXliffConverter

`transUnitIdProvider` Each `<trans-unit>` has to get a unique ID. This ID is supplied by the `transUnitIdProvider`.

`xliffExportOptions` The options will especially tell you, how to deal with target nodes. You may for example have to create empty target nodes in `<trans-unit>` elements or you have to copy the value contained in the source node to the target node.

`XhtmlContentHandler` Markup can be written to a SAX `ContentHandler`. Use this handler, for going through the elements and creating a parallel XLIFF structure. For convenience you use the implementation SAX `DefaultHandler`, so

	that you only have to implement the methods you are interested in.
startElement	For any element in XHTML, you need to create a representation in XLIFF. Regarding the type identifier you can use any String value - you need this value later on to create the corresponding XHTML element. Thus, it makes sense, that the ID contains a reference to the corresponding XHTML element.
addElement	The XLIFF JAXB Classes contain some convenience methods and interfaces such as `ContentHolder` to easily identify XLIFF elements which may have some content.
endElement	This method just updates the element stack, so that new elements get added at the expected location.
characters	Obviously, characters are those elements of your XHTML which you want to translate. Thus, you need to add the characters to a `<trans-unit>` node. Depending on the configured XLIFF export options, you also need to add them to the target node. Do not forget to set the `<trans-unit>` ID here.

Marshalling Failures

If marshalling the XLIFF structure fails, ensure that you check that your generated XLIFF structure is valid. The JAXB wrapper classes do not provide much support there. Thus, it is always recommended having your XLIFF 1.2 specification at hand.

If everything is implemented correctly, an XHTML property as in Example 5.24, “XHTML Example Input” [274] will be transformed to XLIFF as in Example 5.25, “XHTML as XLIFF Example Output” [275], which again will be part of the XLIFF file representing the content containing the property.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>abcdefg</title>
  </head>
  <body>
```

```
    <p>abcdefg</p>
  </body>
</html>
```

Example 5.24. XHTML Example Input

```
<group datatype="xhtml" resname="property:markup:xml">
  <group restype="x-html-html">
    <group restype="x-html-head">
      <group restype="x-html-title">
        <trans-unit id="12">
          <source>abcdefg</source>
          <target>abcdefg</target>
        </trans-unit>
      </group>
    </group>
    <group restype="x-html-body">
      <group restype="x-html-p">
        <trans-unit id="13">
          <source>abcdefg</source>
          <target>abcdefg</target>
        </trans-unit>
      </group>
    </group>
  </group>
</group>
```

Example 5.25. XHTML as XLIFF Example Output

XLIFF Import

Having implemented all the above, let's handle the XLIFF import now. Without further changes, the just exported property will be ignored on XLIFF import, and no change will be applied to the target content item.

Extending XLIFF Schema

XLIFF 1.2 specification may be extended by custom attributes. If you have enabled XLIFF schema validation for XLIFF importer, and if you generated extended attributes you may have to add additional XLIFF schema sources. For details have a look at section “Handling Attributes” [278].

In order to import the `xhtml` property, you have to implement an `XliffPropertyImportHandler` and add it as bean to the Spring context.

```
public class XliffXhtmlPropertyImportHandler
        implements XliffPropertyImportHandler {

  private static final String XHTML_GRAMMAR_NAME =
          "http://www.w3.org/1999/xhtml";

  @Override
  public boolean isAccepting(CapPropertyDescriptor property) {
```

```
    if (property.getType() != CapPropertyDescriptorType.MARKUP) {
      return false;
    }

    XmlGrammar grammar = ((MarkupPropertyDescriptor) property).getGrammar();

    return grammar != null && XHTML_GRAMMAR_NAME.equals(grammar.getName());
  }

  @Nullable
  @Override
  public Object convertXliffToProperty(
          CapPropertyDescriptor descriptor,
          Group group,
          @Nullable Object originalPropertyValue,
          String targetLocale,
          XliffImportResultCollector result) {

    Group markupGroup = (Group) group.getContent().get(0);
    Markup markup = new XliffToXhtmlConverter().apply(markupGroup);

    if (!descriptor.isValidValue(markup)) {
      result.addItem(XliffImportResultCode.INVALID_MARKUP);
    }

    return markup;
  }
}
```

Example 5.26. XliffXhtmlPropertyImportHandler

Example 5.26, "XliffXhtmlPropertyImportHandler" [275] shows a very simplistic implementation of an import handler. It decides if it is responsible for importing into the given property, and if it is, the method `convertXliffToProperty` will be called. Many of the arguments provide context information which may be used for further validation. You only use the `group` to parse the XLIFF and the `descriptor` to validate the resulting value. The `group` handed over is the group node you added before, which contains the reference to the property. Thus, the relevant content for you to parse is the first group contained in it.

As you can see, it is important that XLIFF export and import go hand in hand, to understand each other. Adapting XLIFF export almost always means to adapt the XLIFF import, too.

The `XliffXhtmlPropertyImportHandler` delegates further processing to a converter which will now generate the Markup as can be seen in Example 5.27, "XliffToXhtmlConverter" [276].

```
public class XliffToXhtmlConverter
        implements Function<Group, Markup> {
  private static final String XHTML_GRAMMAR_NAME =
          "http://www.w3.org/1999/xhtml";

  @Override
  public Markup apply(Group group) {
    Document document = MarkupFactory.newDocument(
            XHTML_GRAMMAR_NAME,
            elementName(group),
```

```
            null
    );
    Element documentElement = document.getDocumentElement();
    documentElement.setAttribute("xmlns", XHTML_GRAMMAR_NAME);

    processContents(group, documentElement);

    return MarkupFactory.fromDOM(document);
  }

  private static void processContents(ContentHolder contentHolder,
                                      Element parent) {
    for (Object content : contentHolder.getContent()) {
      if (content instanceof String) {
        processString(parent, (String) content);
      } else if (content instanceof Group) {
        processGroup(parent, (Group) content);
      } else if (content instanceof TransUnit) {
        TransUnit transUnit = (TransUnit) content;
        processContents(transUnit.getTarget(), parent);
      }
    }
  }

  private static void processGroup(Node parent, Group group) {
    Document document = parent.getOwnerDocument();
    Element node = document.createElement(elementName(group));
    parent.appendChild(node);
    processContents(group, node);
  }

  private static void processString(Node parent, String text) {
    Document document = parent.getOwnerDocument();
    Node node = document.createTextNode(text);
    parent.appendChild(node);
  }

  private static String elementName(TypeHolder group) {
    return group.getType().replace("x-html-", "");
  }
}
```

Example 5.27. XliffToXhtmlConverter

Methods of XliffXhtmlPropertyImportHandler

apply	Create a document from the main group which you may later transform to Markup.
processContents	Steps through the elements of a `ContentHolder` (again a convenience interface for XLIFF) and decides based on the (XLIFF) class type what to do. Strings will be the translated text, Groups represent nested elements and TransUnits contain the translated text in their target nodes.

The other methods just create the corresponding nodes for elements or texts.

Backward Compatibility

Especially for the import handler it is important to respect backward compatibility if necessary. The import handler may have to deal with XLIFF documents which were sent to a translation service weeks or month before.

Done

Having all this, you are done with this initial example and you got to know most of the API entry points you will need for customized XLIFF export. You learned about the initial filtering, the transformation to XLIFF and the transformation from XLIFF to your new translated property value.

Handling Attributes

In the previous sections, handling attributes was skipped in order to keep the example simple. But when it comes to attributes, you not only need to decide which of them should be translatable and which not, you also need to care of XLIFF validation, if you turned XLIFF validation on.

Start extending Example 5.23, "XhtmlToXliffConverter" [271] by a method to map arguments.

```
public class XhtmlToXliffConverter implements Function<Markup, Group> {

  /* ... */

  private class XhtmlContentHandler extends DefaultHandler {

    /* ... */

    @Override
    public void startElement(String uri,
                             String localName,
                             String qName,
                             Attributes attributes) {
      /* ... */
      mapAttributes(attributes, elementWrapper);
      addElement(elementWrapper);
    }

    private void mapAttributes(Attributes attributes, Group elementWrapper)
{
      for (int i = 0; i < attributes.getLength(); i++) {
        String localName = attributes.getLocalName(i);
        String attributeValue = attributes.getValue(i);

        switch (localName) {
          case "title":
          case "alt":
          case "summary":
            elementWrapper
                    .getContent()
```

```
                    .add(
                            createTransUnit(
                                    "x-html-attr-" + localName,
                                    attributeValue
                            )
                    );
            break;
          default:
            // Wraps the attribute to a custom namespace.
            QName xliffAttrQName =
                    new QName(
                            "http://www.mycompany.com/custom-xliff-1.0",
                            localName
                    );
            elementWrapper.getOtherAttributes().put(xliffAttrQName,
attributeValue);
        }
      }
    }

    private TransUnit createTransUnit(String resType, String value) {
      TransUnit transUnit = new TransUnit();
      transUnit.setId(transUnitIdProvider.get());
      transUnit.setType(resType);

      Source source = new Source();
      transUnit.setSource(source);
      source.getContent().add(value);

      if (xliffExportOptions.getTargetOption() !=
              XliffExportOptions.TargetOption.TARGET_OMIT) {
        Target target = new Target();
        transUnit.setTarget(target);
        if (xliffExportOptions.getTargetOption() ==
                XliffExportOptions.TargetOption.TARGET_SOURCE) {
          target.getContent().add(value);
        }
      }
      return transUnit;
    }

    /* ... */

  }
}
```

Example 5.28. Attribute Export

In Example 5.28, "Attribute Export" [278] a new method `mapAttributes` is added, which distinguishes between attributes to translate, and attributes not to translate.

Translatable Attributes: For translatable attributes you just add a `<trans-unit>` element which also specifies a `restype` attribute. This attribute has to be resolved later on to a property. The rest is very similar to the `characters` method in the initial example.

Non-Translatable Attributes: Non-translatable attributes, are represented as custom attributes within XLIFF, as XLIFF Schema allows adding such custom attributes at many places. In order to be prepared for XLIFF validation, CoreMedia recommends adding a namespace URI to the attribute, here `http://www.mycompany.com/custom-xliff-1.0`.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>abcdefg</title>
  </head>
  <body>
    <p class="some--class" title="some title">abcdefg</p>
  </body>
</html>
```

Example 5.29. XHTML Example Input (Attributes)

```
<group restype="x-html-p"
       ns4:class="some--class"
       xmlns:ns4="http://www.mycompany.com/custom-xliff-1.0">
  <trans-unit id="14" restype="x-html-attr-title">
    <source>some title</source>
    <target>some title</target>
  </trans-unit>
  <trans-unit id="15">
    <source>abcdefg</source>
    <target>abcdefg</target>
  </trans-unit>
</group>
```

Example 5.30. XHTML as XLIFF Example Output (Attributes)

Given an example XHTML like in Example 5.29, "XHTML Example Input (Attributes)" [280] the paragraph will be transformed as given in Example 5.30, "XHTML as XLIFF Example Output (Attributes)" [280]. As you can see, the class attribute is added with a custom namespace, and the title attribute is added as trans-unit at first place within the group.

```
[Fatal Error] cvc-complex-type.3.2.2: Attribute 'ns4:class' is not allowed
to appear in element 'group'. (23, 8)
```

Example 5.31. XLIFF Validation Error

XLIFF Validation: If XLIFF validation is turned on, the XLIFF import will fail to recognize the custom workspace and raises an error like in Example 5.31, "XLIFF Validation Error" [280]. What you need to do now, is to create a schema defining your new attributes and to add the schema to the Spring application context.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            targetNamespace="http://www.mycompany.com/custom-xliff-1.0"
            xml:lang="en">

  <xsd:attribute name="class" type="xsd:string"/>

</xsd:schema>
```

Example 5.32. Custom XLIFF XSD

```
@Bean
@Scope(BeanDefinition.SCOPE_SINGLETON)
public XliffSchemaSource<Source> customXliffSchema() {
  return () -> new StreamSource(
      getClass().getResourceAsStream("/custom-xliff-1.0.xsd")
  );
}
```

Example 5.33. Custom XLIFF XSD (Bean)

Example 5.32, "Custom XLIFF XSD" [280] adds the `class` attribute to the new custom namespace. Note, that you have to do this for every attribute, you want to support. In Example 5.33, "Custom XLIFF XSD (Bean)" [281] you see, how to add the new schema to the application context, so that it can be used for validation. Having this, you are prepared for the actual XLIFF import handling.

For more details on XLIFF Validation Schemes, see the corresponding Javadoc of `XliffSchemaSource` and the configuration class adding standard schema required for validation `XliffSchemaConfiguration`. Refer to the 'See also' sections for links to the schemas.

XLIFF Import: Now you are going to extend the `XliffToXhtmlConverter` as given in Example 5.27, "XliffToXhtmlConverter" [276].

```
public class XliffToXhtmlConverter
        implements Function<Group, Markup> {

  /* ... */

  private static void processContents(ContentHolder contentHolder,
                                      Element parent) {
    for (Object content : contentHolder.getContent()) {
      if (content instanceof String) {
        processString(parent, (String) content);
      } else if (content instanceof Group) {
        processGroup(parent, (Group) content);
      } else if (content instanceof TransUnit) {
        TransUnit transUnit = (TransUnit) content;
        String type = attributeName(transUnit);
        if (type.isEmpty()) {
          processContents(transUnit.getTarget(), parent);
        } else {
          processTranslatableAttribute(type, transUnit.getTarget(), parent);

        }
      }
    }
  }

  private static void processTranslatableAttribute(String type, Target target,
 Element parent) {
    StringBuilder value = new StringBuilder();
    for (Object content : target.getContent()) {
      value.append(content);
    }
    parent.setAttribute(type, value.toString());
  }

  private static String attributeName(TypeHolder group) {
    String type = group.getType();
    if (type == null || type.isEmpty()) {
```

```
      return "";
    }
    return type.replace("x-html-attr-", "");
  }

  /* ... */

}
```

Example 5.34. Importing Translatable Attributes

In Example 5.34, "Importing Translatable Attributes" [281] add importing the translatable attributes. Add a branch for `<trans-unit>` elements, to detect that they represent a translatable attribute, if the type is set.

```
public class XliffToXhtmlConverter
        implements Function<Group, Markup> {

  /* ... */

  private static void processGroup(Node parent, Group group) {
    Document document = parent.getOwnerDocument();
    Element node = document.createElement(elementName(group));
    processNonTranslatableAttributes(node, group);
    parent.appendChild(node);
    processContents(group, node);
  }

  private static void processNonTranslatableAttributes(Element node,
                                                       Group group) {
    Map<QName, String> otherAttributes = group.getOtherAttributes();
    for (Map.Entry<QName, String> entry : otherAttributes.entrySet()) {
      QName attributeQName = entry.getKey();
      String localPart = attributeQName.getLocalPart();
      node.setAttribute(localPart, entry.getValue());
    }
  }

  /* ... */

}
```

Example 5.35. Importing Non-Translatable Attributes

In Example 5.35, "Importing Non-Translatable Attributes" [282] the custom attributes handed over to the group element are parsed and set as corresponding attribute for the DOM element. Note, that in order to shorten the code, the namespace URI is not checked.

Now you are done, and can support custom attributes, translatable or non-translatable.

5.5.3.5 Translation Workflow

Translation Workflow Configuration

This section describes general configuration options for translation workflows.

> **NOTE**
>
> The necessary Spring configurations for new custom workflows in Studio are documented within Section 9.28, "Custom Workflows" in *Studio Developer Manual*.

Spring configuration for custom translation workflows

A new custom translation workflow requires a strategy for extracting derived contents from your customTranslation.xml workflow definition. Therefore, you need to introduce a bean definition from the class *com.coremedia.translate.workflow.DefaultTranslationWorkflowDerivedContentsStrategy* and adapt it to your custom workflow (see example below).

> **NOTE**
>
> There is already a default bean with the "id" *defaultTranslationWorkflowDerivedContentsStrategy*, that is defined for the *processDefinitionName* "Translation".

```
      <bean id="customTranslationWorkflowDerivedContentsStrategy"
class="com.coremedia.translate.workflow.DefaultTranslationWorkflowDerivedContentsStrategy">
    <description>
      A strategy for extracting derived contents from
      the custom translation.xml workflow definition.
    </description>
    <property name="processDefinitionName" value="customTranslation"/>
    <property name="derivedContentsVariable" value="derivedContents"/>
    <property name="masterContentObjectsVariable"
value="masterContentObjects"/>
</bean>
```

Example 5.36. `Example for a customTranslationWorkflowDerivedContentsStrategy`

The bean needs to be customized in the Workflow Server web application, for example with a Blueprint extension module with Maven property `coremedia.project.extension.for` set to `workflow-server`.

Translation Workflow Properties

studio.translation.showPullTranslationStartWindow	
Description	Use this property to show the workflow start dialog in *Studio* also for translations into the preferred site (pull translations). The pull translation always uses the first available translation workflow. If you have multiple workflow definitions, it might be necessary to select the appropriate workflow definition from the available workflows. Possible values: • *TRUE*: Shows the workflow start dialog in *Studio* also for translations into the preferred site (pull translations). • *FALSE*: The workflow is started automatically. No workflow start dialog is shown.
Default Value	*FALSE*
studio.workflow.translation.extendedWorkflow	
Description	Use this property to define the amount of dependent content, that will be shown in a translation workflow window.
Default Value	*100*

Table 5.23. Translation Workflow Properties

XLIFF Configuration

XLIFF Configuration Properties

The handling of empty translation units during XLIFF import can be configured using the following properties:

translatable.predicate.property-path	
Description	This property holds a map of paths of content type properties (as keys) to mark them as translatable, and thus they will be part of the XLIFF Export. The same property types are supported as for the `extensions:translatable`

content type model setting (see Chapter 4, *Developing a Content Type Model* in *Content Server Manual* for details).

While it is recommended to use `extensions:translatable` in favor of this configuration, the configuration provides support for nested Struct property value access, which is not possible within the content type model. Thus, the intended use case is to specify selected elements of Struct properties which should be part of the XLIFF file while the Struct property itself is not translatable.

A default configuration is shipped with *CoreMedia Blueprint* in module `com.coremedia.blueprint:multi-site`, which is by default a transitive dependency of Blueprint *Studio* and *Workflow Server*.

Note on map value `false`: As there is no concept to veto a decision that a property is considered translatable, entries with a map value of `false` are ignored.

Example:

```
translatable.predicate.property-path[MyType.aStruct.aProperty]=true
```

Example 5.37. Configuration Example for Translatable Property Paths

Syntax:

The first two elements of the property path define the content type, and the property which should be known as (partially) translatable. Following elements define sub-elements within the property. This is currently only supported for Structs. The content type must be the content type which also defines the property.

The expressions support separate properties either by periods or by strings in square brackets. Thus, `CMLinkable.localSettings` is the same as `CMLinkable['localSettings']`.

Limitations:

Using this mechanism to mark properties inside Structs as being translatable works for String and Markup properties. For the latter, only the grammar `coremedia-richtext-1.0` is fully supported.

Alternatively, you can add the bean `RichtextInStructTranslatablePredicate` to your application context to mark *all* richtext markup properties inside Structs as being translatable.

Default Value

Property paths marked as translatable by default in *CoreMedia Blueprint*:

- `CMLinkable.localSettings.callToActionCustomText`

	• `CMTeaser.targets.links.callToActionCustomText`
`translate.item.include-unchanged-properties`	
Description	Configure whether properties that have not changed since last translation are also included in items for translation. Possible values are: `true`, `false`.
Default Value	true
`translate.item.transform.failure.mode`	
Description	Configure the strictness of the XLIFF Export pre-processing regarding missing property transformers. If mode is `fail`, the XLIFF export will fail if a property is marked as translatable where no corresponding `TranslateProperty Transformer` has been found for. Possible values are: `fail`, `warn`, `none`.
Default Value	fail
`translate.xliff.export.handler.failure.mode`	
Description	Configure the strictness of the XLIFF Export regarding missing property export handlers. If mode is `fail`, the XLIFF export will fail if a property is marked as translatable where no corresponding `XliffPropertyExportHandler` has been found for. Possible values are: `fail`, `warn`, `none`.
Default Value	fail
`translate.xliff.import.emptyTransUnitMode`	
Description	Configure handling of empty trans-unit targets for XLIFF import. Possible values: • *`IGNORE`*: Empty targets are allowed. On import the empty translation unit will replace a possibly non-empty target and thus delete its contents. • *`FORBIDDEN`*: No empty targets are allowed. • *`IGNORE_WHITESPACE`*: Empty targets are only allowed where the matching source is empty or contains only whitespace characters

Default Value	*IGNORE_WHITESPACE*
translate.xliff.import.ignorableWhitespaceRegex	
Description	Configure the regular expression that determines which characters are considered *Ignorable spaces*. **Note:** This configuration is only used when `translate.xliff.import.emptyTransUnitMode` is set to *IGNORE_WHITESPACE*.
Default Value	[\\s\\p{Z}\\p{Cc}\\p{Cf}]*
translate.xliff.import.xliffValidationMode	
Description	Configure XLIFF validation behavior. Possible values: • *FAIL_ON_WARNING*: XLIFF validation will fail if any warnings or above occurred. • *FAIL_ON_ERROR*: XLIFF validation will fail if any errors or above occurred. • *FAIL_ON_FATAL_ERROR*: XLIFF validation will fail if any fatal errors occurred. • *DISABLED*: XLIFF validation is disabled.
Default Value	*FAIL_ON_WARNING*
translate.xliff.export.excludeContentName	
Description	Configure the flag that determines, whether the name of content in a translation workflow will be excluded in an XLIFF-Export. CAUTION Including content names into translation may harm your system. Thus, if you set this property to `false`, you should be aware that for example some settings content items referenced by name or content items like `_folderProperties` must not get their names translated.
Default Value	true

Table 5.24. XLIFF Properties

Translation Workflow Studio UI

The possibilities for UI configurations for new custom workflows in Studio are documented within Section 9.28, "Custom Workflows" in *Studio Developer Manual*.

5.6 Personalized Content Management

While Section 5.5, "Localized Content Management" [239] addresses the need to serve *different regions and languages*, Native Personalization addresses the need to serve *different users*. It allows you to present different content to different users of the same site, depending on various conditions.

NOTE

To use Native Personalization, you need to have an Engagement Cloud licence. If you do not have a licence, contact your Key Account Manager or the CoreMedia Support for details.

Native Personalization supports two use cases. First, you can assign users to *user segments* based on their preferences, traits, or past behavior, so that the site can choose the most suitable content for them. Second, during an *A/B test* Native Personalization can assign users randomly to a test case group or a control group, so that you can compare new and old content.

To this end, you can define *personalization variants* for content items. A variant stores both the variant's content and the condition under which it is chosen. In the following, this manual refers to personalization variants simply as *variants*.

NOTE

Do not confuse personalization variants with the localization variants used for multi-site content management (see Section 5.5, "Localized Content Management" [239]). For localization, CoreMedia clones all content items of a site to a derived site, where you can apply arbitrary changes. For personalization, you create variants only for specific content items with audience-specific content and, in general, keep the amount of changes small.

NOTE

Native Personalization determines and renders variants inside CoreMedia Content Cloud. This is different from the client-side Personalization Hub, which integrates third-party personalization providers such as Dynamic Yield. CoreMedia Engagement Cloud typically provides the user segments and A/B test cases used as conditions.

5.6.1 Concept

CoreMedia stores personalization data in ordinary content items, distinguishing baseline content items from variant content items and linking them to conditions. This section introduces the relevant terms and describes how variants, conditions and multi-site content interact.

5.6.1.1 Terms

Baseline	A content item that is shown to users for whom no variant is applicable, because none of the variants' conditions match. Baselines define the overall site structure and store the content that appears when no special conditions apply.
Variant	A content item that replaces a baseline if one of a set of conditions applies.
Condition	A condition defines whether a variant is shown to a user instead of the baseline. Conditions are based on either user segments or test cases.
User segment	A group of users that Native Personalization treats similarly when selecting variants, based on the users' preferences, traits, or past behavior (for example, an age range, a gender, shopping frequency, or a known common interest). In the context of variants, a user is always a website visitor.
Test	During a test, you evaluate the relative merit of two or more test cases in order to improve a site. Native Personalization assigns some or all users to test cases and presents them with the associated variants. One test case serves as the test control.
Test case	You can associate variants of multiple baselines with a test case, which governs the experience of users assigned to it.
Test control	Users in the test control group see the website as though the test were not active.
Stable property	A property that must not change between a baseline and its variants. Typically a property that defines the overall structure of a collection of content items (for example, a children property) or that associates content items with external entities (for example, a product code).

5.6.1.2 Baselines and Variants

At creation time, each content item becomes permanently either a baseline content item or a variant content item. A variant always refers, in its `base` property, to exactly one baseline that it can replace, and it identifies the segments and test cases that enable it. When a baseline is rendered, the delivery service detects the presence of variants and replaces the rendered baseline content as needed.

A variant always has the *same content type* as its baseline. This ensures that the same templates that render the baseline can also render the variant. A variant stores a copy of its baseline's properties with the necessary changes; as a consequence, you must propagate any change to the baseline to its variants explicitly.

Variants are *not nested*: every variant refers directly to its baseline, never to another variant. If certain user segments should override others, you express this by a variant `priority`. If a user matches the conditions of multiple variants, the site chooses the variant with the highest numerical priority. For example, a variant for the segment "vegan food interest" should get a higher priority than a variant for the segment "food interest".

Native Personalization ignores deleted variants for all intents and purposes. You must publish a variant for it to take effect in the delivery environment; publishing a baseline does not publish its variants automatically. Because links should never point to variants, you can publish and withdraw variants easily.

Variants reside close to their baselines in the folder hierarchy. A variant with the name *X* for a baseline *Y* in folder *Z* is placed by CoreMedia Studio at *Z*`/.variants/`*Y*`/`*X*. CoreMedia Studio maintains this location even during moves and renames. The same rules determine access rights for variants and baselines; because a variant resides close to its baseline, a user typically has the same rights on both.

5.6.1.3 Conditions

A variant specifies the conditions under which it can be shown. A condition can indicate the membership of the accessing user in a certain user segment or the assignment of the accessing user to a test case. You can specify multiple segments for a single variant; the variant becomes active as soon as the user is in at least one of the segments or in the test case.

Conditions are stored in a struct property `variantCondition`. A variant stores them either as strings or, alternatively, as links to content items that

represent the conditions. Condition strings follow a URI-style format `VENDOR:VENDORSPECIFIC`, for example `cmec:///p13n/segment/3239470C3C68260C33E4;cmecType=segment` for the Engagement Cloud integration. The struct can contain either a list of segments in the property `segments` or an A/B test in the property `abTestCase`. If you provide an A/B test case, you must also store the test control in the property `abTestControl` of the struct, and the test itself in the ordinary string property `variantAbTest`.

Sometimes you must represent conditions using dedicated content items instead of strings. This matters particularly if you must manage the set of available conditions in the CMS and cannot retrieve it from the variant provider easily. In such a case, link properties replace the string properties mentioned above. To convert these links from condition content items to condition strings, provide a strategy bean implementing the interface `ContentVariantPropertyLinkTranslator` in the Spring application context. The strategy bean defines the content type of the condition content items it can handle.

NOTE

A variant cannot specify a complex boolean condition based on segments or test cases. If you need more complex conditions, define them at the condition provider, typically CoreMedia Engagement Cloud.

5.6.1.4 Multi-Site Support

You can translate and synchronize variants into derived sites just like baselines (see Section 5.5, "Localized Content Management" [239]). The baseline of a derived variant is the derived content item of the original variant's baseline.

When you create a variant directly from a baseline, its `master` property points to that baseline. This indicates that content changes come from the baseline, not from another content item in a different site. There are, however, no automatic processes for synchronizing variants and their baseline at this point.

NOTE

Translation workflows do not include variants automatically. If you need a variant in a derived site, you must include it explicitly in a translation workflow.

5.6.2 Development

Typically you do not need to handle variants and baselines differently. Because they build on the same content type model, and because variants include all properties available for baselines, they support the same operations. The Variant Model describes the variant meta-model, and the Variant Service provides access to baselines and variants; the sections below describe both.

Component-specific development topics are documented in the respective manuals and only summarized in Section 5.6.2.5, "Further Development Topics" [297].

5.6.2.1 Variant Model

The class `ContentVariantModel` defines, similar to the `SiteModel`, the names of the content types and properties used for personalization variants. You rarely need to change the property names, unless they conflict with existing content type extensions.

Configuration key	Default	Description
`variant.model.baseProperty`	base	Name of the property that holds the baseline content reference.
`variant.model.variantConditionProperty`	variantCondition	Name of the struct property that holds the variant condition.
`variant.model.segmentsProperty`	segments	Name of the property that holds the user segments that should see this variant.
`variant.model.segmentLinksProperty`	segmentLinks	Name of the property that links to condition content items representing the segments.
`variant.model.abTestCaseProperty`	abTestCase	Name of the property that holds the A/B test case whose users should see this variant.
`variant.model.abTestCaseLinkProperty`	abTestCaseLink	Name of the property that holds a link to a condition content item representing the A/B test case.

Configuration key	Default	Description
`variant.model.abTestProperty`	variantAbTest	Name of the property that holds the A/B test id.
`variant.model.abTestLinkProperty`	variantAbTest-Link	Name of the property that holds a link to a condition content item representing the A/B test.
`variant.model.abTestControlProperty`	abTestControl	Name of the property that holds the A/B test control id.
`variant.model.abTestControlLinkProperty`	abTestCon-trolLink	Name of the property that holds a link to a condition content item representing the A/B test control.
`variant.model.priorityProperty`	priority	Name of the property that holds the variant priority.

Table 5.25. Content variant property names

You may need to mark additional content types added to the Blueprint as either supporting or not supporting variants. By default, the subtypes of `CMLocalized` support variants, with the exception of `CMFolderProperties`, `CMImage`, `CMSettings`, `CMResourceBundle`, `CMSite`, `CMSitemap`, `CMSymbol`, `CMTaxonomy`, and `CMTheme`. See `variant-blueprint.properties` for the default configuration.

Configuration key	Default	Description
`variant.model.type.`*`TYPENAME`*	Inherited from super type, or false if not defined for any super type	Whether content items of this type can have variants.

Table 5.26. Variant support per content type

You can also define *stable properties*, that is, properties that must not change between a baseline and its variants. A content type inherits the stable properties of its supertype and may define additional ones.

Configuration key	Default	Description
`variant.model.stableProperties`	None	A list of stable properties that apply to all content types.
`variant.model.stablePropertiesByType.`*`TYPENAME`*	None	A list of stable properties that apply to this type and all its subtypes.

Table 5.27. Stable property configuration

5.6.2.2 Variant Service

The interface `ContentVariantService` defines methods that let you access baseline and variant content items without having to deal with the intricacies of the content type model. The Spring application context provides an instance of this interface in the CAE, the Headless Server, the Studio Server, and the Content Feeder.

In particular, the service retrieves the variant descriptions for a given baseline content item. These descriptions use subclasses of the `ContentVariant` interface:

- A `SegmentedContentVariant` represents a variant that is shown to one or more user segments.
- A `TestCaseContentVariant` represents a variant that is shown to users assigned to a certain test case.
- A `TestControlContentVariant` indicates that a user belongs to a test control group. Unlike the other two descriptions, a test control does not come with a specific variant content item to render. Instead, the CAE should render the variant with the highest priority but count the view towards the test control.

You can also navigate from a variant back to its baseline and retrieve the properties of a variant, in particular the conditions that activate it.

5.6.2.3 Engagement Cloud Custom Events

During an *A/B test*, Engagement Cloud judges individual test cases by the frequency of certain events for user sessions that are assigned to each specific test case. An event can be the retrieval of a specific page, but it can also be a custom event that the web browser generates.

After you define a custom event template in Engagement Cloud, a JavaScript call sends such events:

```
bysideWebcare_event('checkOutPageReached', {});
```

where

- `checkOutPageReached` is the required event id defined in Engagement Cloud,
- `{}` is an event object that you can enrich with additional properties (if configured in Engagement Cloud).

You are free to send these events whenever you like, but avoid calling this method in an `onClick` handler for a hyperlink, because the browser might leave the current page before having sent the event to the Engagement Cloud server.

5.6.2.4 Command Line Tools

All command line tools handle baseline content items and variant content items uniformly. For example, you can dump them using **cm dump**, approve them using **cm approve**, and so on.

The output of **cm serverexport** contains a marker for variants in the attribute `variant` of the XML root node, which can be either true or false. The tool **cm serverimport** respects this flag when creating content items.

The tool **cm validate-multisite** adjusts its rules to the special semantics of baselines and variants. For example, it issues no warning if a test case variant has not been translated, because you typically perform tests in a single site and do not need to translate them.

The tool **cm cleanvariants** removes variants that have become unused because their baseline was deleted or destroyed. See Section "Clean Variants" in *Content Server Manual* for details.

5.6.2.5 Further Development Topics

The Variant Model and Variant Service form the shared foundation. How each component delivers and queries variants varies; the details are documented in the respective manuals:

- **Unified API** — Baselines and variants share the same document type model and support the same operations. Three tasks become more specific: analyzing content structures via referrers, accessing content items via observed properties, and navigating between sites in a multi-site setup. For these, the API adds baseline-only accessors (for example, `getBaselineReferrersWithDescriptor`, `getBaselineContentsWithValue`, and the `getAllVariants` variants of the `ContentSiteAspect` methods), and the `Content.isVariant` / `Content.isBaseline` flags. See the *Unified API Developer Manual* for details.
- **CAE** — The CAE uses a client-side JavaScript library to replace baseline fragments with variant fragments dynamically, using metadata `div` elements and the `p13n-dynamic-include` brick. The *Content Application Engine Developer Manual* describes the extension points (`BodyDetector`, `ContentVariantProcessor`, `VariantConditionPredicate`), the programming model, and the FreeMarker library.
- **Headless Server** — Each content object exposes a contentVariants field (and contentVariantsAndBaseline) that lists the possible variants in descending priority order, together with their conditionStrings and parsed conditions. The *Headless Server Developer Manual* describes the variant-selection algorithm and example GraphQL queries.
- **Search and Feeder** — The Content Feeder indexes variants like baseline content and adds variant metadata (for example, variantBase, variantSegments, variantAbTestCase) to Solr. The CAE Feeder does *not* process variants, so a website search finds baseline content only.
- **Studio, tools and configuration** — Studio maintains the variant folder location and enforces the variant meta-model through validators. The command line tools handle baselines and variants uniformly (see the variant marker in **cm serverexport**/**cm serverimport**), and **cm cleanvariants** removes left-over variants of deleted or destroyed baselines.

NOTE

When you apply bulk updates through the Unified API, remember that you must create variants at their designated location and move them along with their baseline. Studio ensures this automatically; Unified API clients must take care of it explicitly.

5.7 Workflow Management

In this chapter you will find a description of the predefined workflows as well as the workflow actions that are needed to customize existing workflows or define new ones.

Predefined workflows described in here:

- workflows covering the publication of resources, see Section 5.7.1, "Publication" [299],
- an example translation workflow, see Section 5.7.2, "Translation Workflow" [306],
- a fixed workflow for initially deriving a site from an existing site, see Section 5.7.3, "Deriving Sites" [326].

5.7.1 Publication

In this chapter you will find a description of publication workflows and a description of the publication semantics.

CoreMedia delivers the listed example workflows. But the workflow facilities are not restricted to those features. They can be tailored to fit all types of business processes.

5.7.1.1 Approval and Publication of Folders and Content Items

What is and what does a publication?

A publication synchronizes the state of the *Live Server* with the state of the *Content Management Server*. All actions such as setting up new versions, deleting, moving or renaming files, withdrawing content from the live site require a publication to make the changes appear on the *Live Server*.

CoreMedia makes a distinction between the publication of structural and of content changes:

- Content-related changes are changes in content item versions such as a newly inserted image, modified links, text.
- Structure-related changes are moving, renaming, withdrawing or deleting of resources. So it becomes possible to publish structural changes separately from latest and approved content item versions.

For every publication a number of changes is aggregated in a change set. This change set is normally composed in the course of a publication workflow. The

administrator and other users with appropriately configured editors can also execute a direct publication, which provides a simpler, although less flexible means of creating a change set.

Change Set in Direct Publications

When performing a direct publication, the change set is primarily based on the set of currently selected resources or on the single currently viewed resource. As the set of resources does not give enough information for all possible types of changes, three rules apply:

- You cannot publish movements and content changes separately. Whenever applicable, both kinds of changes are included in the change set.
- When a content item is marked for deletion or for withdrawal, new versions of that content item are not published.
- If the specific version to be published is not explicitly selected, the last approved resource version is included in the change set.

There are also some automated extension rules for the change set, which modify the set of to-be-published resources itself. These rules can be configured in detail. Ask your Administrator about the current settings.

- When new or modified content is published and links to an as yet unpublished resource, the unpublished resource is included in the change set. Depending on the configuration, also recursively linked content items can be included in the change set. Target content items that are linked via a weak link property are not included in the change set.
- When the deletion of a folder is published, all directly and indirectly contained resources are included in the change set.
- When the withdrawal of a folder is published, all directly and indirectly contained published resources are included in the change set.
- When the creation, movement, or renaming of a resource in an unpublished parent folder is published, that folder is included in the change set.

Preconditions

Preconditions for a successful publication

Preconditions for a successful publication are:

- all path information concerning the resource has to be approved too: if the resource is located in a folder never published before, this folder has to be published with the resource. So, add it to the change set or publish the folder before.
- withdrawals and deletions must be approved before publication.
- all content items linked to from a content item which is going to be published have to be already published or included in the change set. This is because

a publication that would cause dead links will not be performed. This rule does not apply for weak link properties.
- a content item which is going to be deleted must not be linked to from other content items or these content items have to be deleted during the same publication. This rule does not apply for weak link properties.

Status and action on the *Content Management Server*	Effect on the Live Server on publication
A version of the content item does not yet exist on the *Live Server*. The content item is not marked for deletion. You approve the version.	The approved version is copied to the *Live Server*.
The last approved version of a content item already exists on the *Live Server*. The content item is not marked for deletion. You start a new publication without any further preparation.	No effect on the *Live Server*.
The content item is published and is not marked for deletion. It therefore exists on both servers. You rename the content item and approve the change.	The content item is renamed.
The content item is published and is not marked for deletion. It therefore exists on both servers. You move the content item and approve the change.	The content item is moved.
The content item is published. It therefore exists on both servers. No links to this content item exist. You mark the content item for withdrawal and approve the change.	The content item is destroyed on the Live Server.
The content item is published. It therefore exists on both servers. No links to this content item exist. You mark the content item for deletion and approve the change.	The content item is destroyed on the *Live Server*. The content item is moved into the recycle bin on the *Content Management Server*.

Status and action on the *Content Management Server*	Effect on the Live Server on publication
The content item is published. It therefore exists on both servers. Links to this content item from other published content items exist. You mark the content item for deletion and approve the change.	The deletion cannot be published, since an invalid link would be created. A message is displayed in the publication window. Remove the link in the other content item and publish again.
The content item is published. It therefore exists on both servers. Weak links to this content item from other published content items exist. You mark the content item for deletion and approve the change.	The content item is destroyed on the *Live Server*. The content item is moved into the recycle bin on the *Content Management Server*.

Table 5.28. Publishing content item: actions and effects

Status and action on the *Content Management Server*	Effects on the Live Server on publication
The folder is published and is not marked for deletion. It therefore exists on both servers. You rename the folder and approve it.	The folder is renamed.
The folder is published and is not marked for deletion. It therefore exists on both servers. You move the folder and approve the change.	The folder is moved.
The folder is not published and not marked for deletion. You approve the folder.	The folder is created on the *Live Server*.
The folder is published. You mark it for withdrawal. When queried, you acknowledge the mark for withdrawal of all contained resources. You approve the change.	The folder is destroyed on the *Live Server*. The withdrawal can only succeed if all resources on the *Live Server* or *Content Management Server* that are contained in the folder, and all published resources that link to this folders content via a non-weak link property, are also contained in the change set.

Status and action on the *Content Management Server*	Effects on the Live Server on publication
The folder is published. You mark it for deletion. When queried, you acknowledge the mark for deletion of all contained resources. You approve the change.	The folder is destroyed on the *Live Server*. The folder is moved to the recycle bin on the *Content Management Server*. The deletion can only succeed if all resources on the *Live Server* or *Content Management Server* that are contained in the folder, and all published resources that link to this folders content via a non-weak link property, are also contained in the change set.

Table 5.29. Publishing folders: actions and effects

Special cases

Special cases

Please keep in mind that:

- Older versions cannot be published.
  Example: if a version No. 4 had already been published it is not possible to publish version No. 3 thereafter. To do so, create a version No. 5 from No. 3.
- During a deletion, a resource that has not been published yet is moved to the recycle bin immediately.

In addition, consult the previous tables for effects of a publication depending on the state of the resource. For all examples it is assumed that you have appropriate rights to perform the action.

Delete and withdraw resources

Withdrawing Publications and Deleting Resources

There is only one fundamental difference between withdrawal of publications and deletion: a withdrawal affects only the *Live Server*, whereas the deletion of a resource - folder or content item - causes the resource to be moved into the trash folder on the *Content Management Server*.

Before a withdrawal or deletion can be published as described before, a mark for withdrawal or for deletion must be applied using the appropriate menu entries or tool bar buttons. In the case of folders, the contained resources are affected, too. If you have marked a resource for deletion and withdrawal, then the deletion will be executed.

- When a folder is marked for deletion, all contained published resources are marked for deletion, too. Not published resources are immediately moved into the recycle bin without requiring you to start a publication.
- When a folder is marked for withdrawal, all contained published resources are marked for withdrawal, too.

- When a mark for withdrawal or deletion of a folder is revoked, this also affects all contained resources with the same mark.
- If you use direct publication and approve a folder that is marked for withdrawal deletion, that approval is implicitly extended to the contained resources that are also marked for withdrawal or deletion.
- Disapprovals extend to contained resources in the same way.

5.7.1.2 Predefined Publication Workflows

The predefined workflows for the approval and publication of resources are described in the following table. These workflows can be uploaded using `cm upload -n <filename>`. You can examine their definition and use them as examples for your own definitions, by downloading an uploaded definition using `cm download <ProcessName>`.

Workflow	Definition name
simple publication	Process `StudioSimplePublication` defined in `studio-simple-publication.xml`
2-step publication	Process `StudioTwoStepPublication` defined in `studio-two-step-publication.xml`

Table 5.30. Predefined publication workflow definitions

Publication workflow steps

The following table compares the working steps which are covered by the predefined workflows.

Step	simple publication	2-step publication
1.	A user creates the workflow with all necessary resources.	A user creates the workflow with all necessary resources.
2.	The resources are published (and implicitly approved) in one step, performed by the same user, who needs 'approve' and 'publish' rights.	A second user (needs 'approval' and 'publish' rights) can explicitly approve resources. In *Studio*, the second user may also modify the resources before
3.		Publication will be executed when finishing the task after all resources in the change set have been approved.

Step	simple publication	2-step publication
4.		*(If not, the workflow is returned to its 'composer')*

Table 5.31. Predefined publication workflow steps

5.7.1.3 Features of the Publication Workflows

The predefined publication workflows have some features in common, which are described in the following:

Users and Groups

In order to execute tasks within workflows, users have to be assigned to special groups. In the predefined publication workflows, these are the following:

1. *composer-role*: to be able to create (and start) a publication workflow and compose a change set
2. *approver-role*: to be able to approve the resources in the change set
3. *publisher-role*: to be able to publish the resources in the change set

Special groups can be defined and linked to the workflow via the `Grant` element in the workflow definition file.

Read more about users, groups and administration in the *Content Server Manual*.

Note that, when all eligible users for a task reject that task, the task is again offered to all eligible users. So if you are the only user for an *approver-role* group and you start a publication workflow, the second step of the workflow will be escalated. That is because you cannot be the composer and the approver of a resource - and there is no other user than you.

Basic Steps in a Publication Workflow

After a user has created one or more content items, these content items should be proofread, approved and published in a workflow:

1. The user (not necessarily the user who did the editing) starts a workflow. If he selects resources at starting time, these resources will be added to the change set and the compose task will be accepted automatically. Otherwise, he has to add the resources to the change set later.
2. The user completes the 'compose' task.

3. The task 'approve' is automatically offered to all appropriate users (members of the *approver-role* group, but not to the composer - even if he is a member of this group). Somebody accepts the task and approves the resources.

The user has the following options:

option A	option B	option C	option D
The user accepts the task, approves the resource(s)and finishes the task. All resources are approved.	The user accepts the task, does not approve all resource(s) and finishes the task	The user rejects the task.	The user accepts the task but delegates it to somebody else.
The task 'Publication' is offered to all members of the group *publisher-role*.	The change set is sent back to the user who completed the 'compose' task.	The task is offered all other members of the group approver-role.	The task is automatically accepted by this user.

Table 5.32. User options.

5.7.2 Translation Workflow

A translation workflow can be used to communicate changes in the project of a master site to the derived sites.

CoreMedia Blueprint provides one template translation workflow named *Translation* in the file `translation.xml` in the `wfs-tools-application` module. The workflow is built around an empty action, the `SendToTranslationServiceAction` in the `workflow-lib` module, which is supposed to implement the sending / receiving of contents to / from a translation agency. Without an implementation of this action, the workflow can still be used for manual in-house translation, possibly in conjunction with XLIFF download/upload.

5.7.2.1 Roles and Rights

The translation workflow process is based on two roles defined for *CoreMedia Content Cloud's* Multi-Site concept:

- The group translation-manager-role contains all users that are allowed to start a translation workflow.

The name of this group has to be configured in the property `translationManagerRole` of the `SiteModel` (see section "Site Model" [252]). After changing this property, you have to upload the workflows again, because uploading persists the current property value.

- The site manager groups define the users who may accept translation workflows for the content of a site.

  Groups and Rights Administration for Localized Content Management [247] describes how to set this property for every site.

5.7.2.2 Workflow Lifecycle

As described in Section 5.7.2.1, "Roles and Rights" [306], the translation managers start the translation workflow for a set of new or changed contents from the Control Room. Therefore, a new `Process` instance will be created for every site that has been selected as a translation target.

At first, the `Process` instances both run two `AutomatedTasks` that retrieve the manager group and collect / create the derived contents for the target site. For details see Section 5.7.2.4, "Predefined Translation Workflow Actions" [309].

The following `UserTask` called `Translate` is used to let the user choose a next step. This is done by selecting a next step in the radio group of the `workflowForm`. The selected value will then be set as value for the `translationAction` process variable. This variable is then used in a `Switch` task to choose the successor task.

These successor tasks are:

- `SendToTranslationService`: Send / retrieve content to / from translation agency (has to be implemented in the project)
- `Rollback`: Cancel the translation and rollback changes that may have been made to the target content. (E.g.: The `GetDerivedContentsAction` may have created content in the target site derived from the provided master content.)
- `Complete`: Update the `masterVersion` of the target content to indicate, that the translation is completed. This can be used, for example when the user translated the content manually.

While the `Rollback` and `Complete` tasks finish the process, the `SendToTranslationService` task has another `UserTask` successor called `Review`. This task simply gives the user an opportunity to check the content imported from the translation agency. For details on the Actions behind these tasks see Section 5.7.2.4, "Predefined Translation Workflow Actions" [309].

5.7.2.3 Configuration and Customization

The example translation workflow is meant to be configured to your needs. You might define multiple translation workflows, like translation via translation agency or manual translation performed by the site managers.

The only restriction is that every translation workflow needs a process variable `subject` of type `String`, which will be set by the framework.

To reliably track content that is *in translation*, the process definition specifies a `CleanInTranslationFinalAction` which resets the state both when a workflow process is completed or aborted. See Section "CleanInTranslationFinalAction" [325] for details.

It is recommended to use a `CleanInTranslationFinalAction` in every custom translation or localization process definition. However, old process definitions may not yet use that final action but still depend on a background job to regularly clean up the *in translation* state of content. To that end, you need to define, configure and regularly invoke an instance of the `com.coremedia.translate.workflow.CleanInTranslation` class. An example definition is included in the blueprint source in the `com.coremedia.blueprint.workflow.boot.BlueprintWorkflowServerAutoConfiguration` file, which you may have to adapt.

The scheduling for `CleanInTranslation` may be adapted in the `application.properties` using the following properties, each prefixed with `workflow.blueprint`:

`clean-in-translation.enabled`	Set to `false` to disable `CleanInTranslation` if all localization processes use a `CleanInTranslationFinalAction` in their definition. In that case, `CleanInTranslation` is not needed and can be disabled. Defaults to `true`.
`clean-in-translation.initial-delay`	Sets the initial delay, when to run cleanup the first after start of workflow-server. It defaults to 10 seconds.
`clean-in-translation.confidence-threshold`	Threshold for confidence we need to reach, before we are going to remove a merge-version of a derived content. 0 (zero) or below signals, that merge-versions shall be removed immediately when considered unused.

The confidence is increased on each scheduled run of `CleanInTranslation`. Thus, a threshold of 1 (one) will clean up a merge-version when it got detected twice as being unused. A threshold is recommended not to be chosen higher than 10.

A threshold greater than 0 (zero) is strongly recommended, as asynchronous updates may case false positive rating as unused.

The default threshold is 1 (one).

`clean-in-translation.fixed-delay`

Sets the time to wait after the previous run ended, to repeat cleanup. It defaults to 15 minutes.

As `CleanInTranslation` visibly clears the *in-translation* state of abnormally ended workflows, adjusting the delay is a trade-off between more correct content state reporting versus adjusting load on workflow server.

The delay is given as duration according to Converting Durations in Spring Boot's Core Features Reference. Thus, a value of `10s` is parsed as *10 seconds*, `20m` is parsed as *20 minutes*, and without any unit, it defaults to milliseconds.

Be aware, that changes in the process definition will probably lead to changes in the UI, too. If you want to change only small bits of the provided translation workflow like adding another user-selectable `translationAction` and `Task`, this can be done pretty easily through configuration of the `defaultTranslationWorkflowDetailForm` inside the `ControlRoomStudioPlugin`.

But if you want to use a workflow completely different to the one provided, be prepared to write your own implementations of the `workflowForms` and start panel used to display your workflow in *Studio*.

For details on customizing workflows see the Workflow Manual. For details on customizing the *Studio* UI for the translation workflows see Section "Translation Workflow Studio UI" [288].

5.7.2.4 Predefined Translation Workflow Actions

This section describes various actions that can be used to define a translation workflow.

- Section "GetDerivedContentsAction" [310] describes an action that computes, and if necessary creates derived contents from a given set of master contents.
- Section "CreateTranslationTreeData" [312] describes an action that computes the data for the TranslationTree in the Studio Client.
- Section "FilterDerivedContentsAction" [313] describes an action that filters previously computed derived contents.
- Section "GetSiteManagerGroupAction" [315] describes an action that determines a site manager group and stores it in a process variable. If the process variable is atomic, only the first given site manager group will be set. It is recommended to use an aggregation variable as target, though.
- Section "GetMultiSiteManagerGroupAction" [316] describes an action that resolves site manager groups for one or multiple target sites and combines them using a configurable merge strategy. It supports both atomic and aggregated site id input variables and is intended as a successor to GetSiteManagerGroupAction for multi-site translation workflows.
- Section "ExtractPerformerAction" [318] describes an action that identifies the user who executes that current task and stores a user object in a process variable.
- Section "AutoMergeTranslationAction" [318] describes an action that automatically updates properties of derived contents after changes in their master content.
- Section " AutoMergeSyncAction " [320] describes an action that automatically updates properties of derived contents after changes in their master content in the context of a synchronization workflow.
- Section "CompleteTranslationAction" [321] describes an action that finishes a manual translation process.
- Section "RollbackTranslationAction" [323] describes an action that rolls back a translation process, possibly deleting spurious content.
- Section "CleanInTranslationFinalAction" [325] describes a so-called final-action that is required for cleaning up the *in-translation* state of documents, unless already handled within CompleteTranslationAction [321] and/or RollbackTranslationAction [323].

GetDerivedContentsAction

The `GetDerivedContentsAction` retrieves or creates all derived contents for the given master content objects in the target site(s).

Derived contents will be marked as being in translation.

Contents are created in the name of the translation workflow robot user.

`targetSiteIdVariable`	
Required	yes
Description	The name of the variable that contains the id of the target site
`masterContentObjects`	
Required	yes
Description	The name of the variable that contains the list of content objects in the master site
`derivedContentsVariable`	
Required	no
Description	The name of the variable into which a list of all derived contents is stored
`createdContentsVariable`	
Required	no
Description	The name of the variable into which a list of all newly created contents is stored. If the workflow is subsequently aborted, these contents can be deleted by the action described in Section "RollbackTranslationAction" [323]
`placeholderPreparationStrategyVariable`	
Required	no
Description	The name of the variable holding the strategy to determine derived contents. The strategy can be either `missingReferencesPreparation` or `missingOrByTranslationStatePreparation`. While the first one only considers content that does not yet exist in the target site, the second one also considers content that is outdated in the target site.

Table 5.33. Attributes of GetDerivedContentsAction

```
<Variable name="siteId" type="String"/>
<AggregationVariable name="masterContentObjects" type="Resource"/>
<AggregationVariable name="derivedContents" type="Resource"/>
<AggregationVariable name="createdContents" type="Resource"/>
<Variable name="placeholderPreparationStrategy" type="String"/>
...

<AutomatedTask name="GetDerivedContents" successor="FollowUpAction">
  <Action class="com.coremedia.translate.workflow.GetDerivedContentsAction"
          masterContentObjectsVariable="masterContentObjects"
          derivedContentsVariable="derivedContents"
          createdContentsVariable="createdContents"

placeholderPreparationStrategyVariable="placeholderPreparationStrategy"
          targetSiteIdVariable="siteId"/>
</AutomatedTask>
```

Example 5.38. Usage of GetDerivedContentsAction

Referrer Transfer Augmentation

When computing derived contents, GetDerivedContentsAction automatically considers referrers that are structurally relevant, such as navigation channels and taxonomies. This ensures that structural dependencies are preserved in the target site. The set of referrer relations and the augmentation behavior are configurable. For details see Section 4.5, "Referrer Relations in Localization" in *Multi-Site Manual* and Section 3.6.2, "Shared Multi-Site Configuration" in *Deployment Manual* as well as in Section 3.4.3, "Shared Multi-Site Configuration" in *Deployment Manual*.

CreateTranslationTreeData

This Action will calculate the data for the `TranslationTree` Studio component. As a result two maps will be created and stored in the process:

- A map that groups the derived content by its locale
- A map that groups each derived content by its master version

`translationTreeDataVariable`

Required	yes
Description	The name of the process variable where the map of the derived contents, grouped by their locale is stored as a blob.

`premularConfigDataVariable`

Required	yes
Description	The name of the process variable where the map of the masterVersions, grouped by their derived content is stored as a blob.
`masterContentObjectsVariable`	
Required	yes
Description	The name of the variable that contains the list of content objects in the master site.
`derivedContentsVariable`	
Required	no
Description	The name of the variable into which a list of all derived contents is stored.

Table 5.34. Attributes of CreateTranslationTreeData

```
<AggregationVariable name="masterContentObjects" type="Resource"/>
<AggregationVariable name="derivedContents" type="Resource"/>
<Variable name="translationTreeData" type="Blob"/>
<Variable name="premularConfigData" type="Blob"/>
...

<AutomatedTask name="CreateTranslationTreeData"
successor="CheckIfSelfAssigned">
      <Action
class="com.coremedia.translate.workflow.CreateTranslationTreeDataAction"
              masterContentObjectsVariable="masterContentObjects"
              derivedContentsVariable="derivedContents"
              translationTreeDataVariable="translationTreeData"/>
</AutomatedTask>
```

Example 5.39. Usage of CreateTranslationTreeDataAction

FilterDerivedContentsAction

This action is supposed to follow up `GetDerivedContentsAction`. It filters the derived contents in two ways.

- It checks for each content in the given `derivedContents` whether the content's `masterVersion` is more recent than the corresponding version in the given `masterContentObjects`. In this case, the content is excluded from the `derivedContents`. If a `skippedContentsVariable` is given,

all of these excluded contents are stored under this variable of the corresponding process.

- For all remaining contents from `derivedContents`, it checks whether the content has its `ignoreUpdates` property set (see `ContentObjectSiteAspect#getIgnoreUpdates()`). If so, this content is also excluded from the `derivedContents` (but not stored under `skippedContentsVariable`).

masterContentObjects	
Required	yes
Description	The name of the variable that contains the list of content objects in the master site
derivedContentsVariable	
Required	yes
Description	The name of the variable into which a list of all derived contents is stored
skippedContentsVariable	
Required	no
Description	The name of the variable into which the list of skipped contents (because of outdated master content object) is stored.

Table 5.35. Attributes of FilterDerivedContentsAction

```
<Variable name="siteId" type="String"/>
<AggregationVariable name="masterContentObjects" type="Resource"/>
<AggregationVariable name="derivedContents" type="Resource"/>
<AggregationVariable name="createdContents" type="Resource"/>
<Variable name="placeholderPreparationStrategy" type="String"/>
<AggregationVariable name="skippedContents" type="Resource"/>
...

<AutomatedTask name="StartSyncFromParentToTarget"
successor="CheckIfDerivedContentsEmpty">
  <Action class="com.coremedia.translate.workflow.GetDerivedContentsAction"
          masterContentObjectsVariable="masterContentObjects"
          derivedContentsVariable="derivedContents"
          createdContentsVariable="createdContents"

placeholderPreparationStrategyVariable="placeholderPreparationStrategy"
          targetSiteIdVariable="siteId"/>
  <Action class="com.coremedia.translate.workflow.FilterDerivedContentsAction"
```

```
          masterContentObjectsVariable="masterContentObjects"
          derivedContentsVariable="derivedContents"
          skippedContentsVariable="skippedContents"/>
</AutomatedTask>
```

Example 5.40. Usage of FilterDerivedContentsAction

GetSiteManagerGroupAction

This action is used to determine the user groups that are responsible for managing the site. The names of these groups are defined in the property `siteManagerGroup` of every site indicator. As this property is not required, the group administratoren will be used per default.

Note, that the action transparently deals with atomic and aggregation variables as target variable. If the target variable is atomic, only the first group in `siteManagerGroup` will be taken into account.

siteVariable	
Required	yes
Description	The name of the variable that contains the id of the site
siteManagerGroupVariable	
Required	no
Description	The name of the variable into which the site manager groups are stored

Table 5.36. Attributes of GetSiteManagerGroupAction

```
<Variable name="siteId" type="String"/>
<Variable name="siteManagerGroup" type="Group"/>
...

<AutomatedTask name="GetTargetSiteManagerGroup" successor="FollowUpAction">
  <Action class="com.coremedia.translate.workflow.GetSiteManagerGroupAction"

          siteVariable="siteId"
          siteManagerGroupVariable="siteManagerGroup"/>
</AutomatedTask>
```

Example 5.41. Usage of GetSiteManagerGroupAction

GetMultiSiteManagerGroupAction

This action resolves site manager groups for one or multiple target sites and stores the result into a workflow process variable. It is designed as a successor to `GetSiteManagerGroupAction` (see Section "GetSiteManagerGroupAction" [315]) and adds support for aggregated site id input variables as well as configurable merge strategies for combining groups from multiple sites.

When a single target site is provided, the action behaves equivalently to `GetSiteManagerGroupAction`. When multiple target sites are provided, groups from each site are combined according to the configured `mergeStrategy`. The default strategy is `INTERSECTION`, which retains only groups that are common to all target sites, thereby avoiding unintentional permission broadening.

If the merged result is empty, an optional `fallbackGroups` list is consulted. If the result still remains empty after fallback processing, the action returns UNSUCCESSFUL.

`siteVariable`	
Required	yes
Description	The name of the variable that contains the id(s) of the target site(s). May reference an atomic `String` or an aggregated `StringList` variable.
`siteManagerGroupVariable`	
Required	yes
Description	The name of the variable into which the resolved manager group(s) are stored. Supports both atomic (`Group`) and aggregated (`GroupList`) target variables. When the target variable is atomic and multiple groups are resolved, only the first group is stored.
`mergeStrategy`	
Required	no (default: `INTERSECTION`)
Description	Strategy for combining groups resolved from multiple target sites. Accepted values: • `INTERSECTION` — retains only groups common to all target sites (default; most restrictive). Only considers directly assigned groups.

- `UNION` — combines all groups from all target sites (broadest permissions; explicit opt-in).
- `COMMON_ANCESTOR` — walks the group hierarchy to find the *lowest* shared parent groups across all target sites. Only the most specific common ancestors are returned; higher-level ancestors reachable from a more-specific common group are pruned. Unlike `INTERSECTION`, this strategy considers transitive group membership (advanced opt-in).

`fallbackGroups`	
Required	no
Description	A delimited string of fallback group names to use when the merge result is empty. Group names are split using the `fallbackGroupsDelimitingRegex` (default: comma followed by optional whitespace). Example: `translation-manager-role, editors`.

`fallbackGroupsDelimitingRegex`	
Required	no (default: `,\s*`)
Description	Regular expression used to split the `fallbackGroups` string into individual group names.

Table 5.37. Attributes of GetMultiSiteManagerGroupAction

```
<AggregationVariable name="targetSiteId" type="String"/>
<AggregationVariable name="targetSiteManagerGroup" type="Group"/>
...

<AutomatedTask name="GetTargetSiteManagerGroups" successor="FollowUpAction">

  <Action
class="com.coremedia.translate.workflow.GetMultiSiteManagerGroupAction"
          siteVariable="targetSiteId"
          siteManagerGroupVariable="targetSiteManagerGroup"
          mergeStrategy="INTERSECTION"
          fallbackGroups="translation-manager-role"/>
</AutomatedTask>
```

Example 5.42. Usage of GetMultiSiteManagerGroupAction

ExtractPerformerAction

To perform an `AutomatedTask` with the same performer used in a previous `UserTask`, you can store the performer of the `UserTask` to the given workflow variable.

performerVariable	
Required	no
Description	The name of the variable into which the performer of the current user task is stored

Table 5.38. Attributes of ExtractPerformerAction

```
<Variable name="performer" type="User"/>
...
<UserTask name="Translate" successor="FollowUpAction">
  ...
  <EntryAction class="com.coremedia.translate.workflow.ExtractPerformerAction"
               performerVariable="performer"/>
  ...
</UserTask>
```

Example 5.43. Usage of ExtractPerformerAction

AutoMergeTranslationAction

This action automatically updates properties of derived contents after changes in their master content since its last translation. See also the API documentation in `AutoMergeTranslationAction` and Content Type Model - Properties for Multi-Site [259] for the behavior of the automerge feature.

performerVariable	
Required	yes
Description	The name of the variable that contains the user in whose name this action performed. Typically, the user has been retrieved previously by the action described in Section "ExtractPerformerAction" [318].
derivedContentsVariable	

Required	yes
Description	The name of the variable that contains all translated content items.
`masterContentObjectsVariable`	
Required	yes
Description	The name of the variable that contains all master content objects.
`resultVariable`	
Required	yes
Description	The name of the result variable to store derived contents in, whose properties could not be updated automatically.
`autoMergePredicateFactoryName`	
Required	no
Description	The name of a custom Spring bean that implements interface `AutoMergePredicateFactory` and that is used to decide which content properties are updated automatically. If this attribute is not specified, the Spring bean with name `defaultAutoMergePredicateFactory` is used.
`autoMergeStructListMapKeyFactoryName`	
Required	no
Description	The name of a custom Spring bean that implements the interface `AutoMergeStructListMapKeyFactory` and that is used to select the merge algorithm for nested struct list properties. For some struct lists, like the placements of a page grid, a better merge algorithm can be used, which enables automatic updates of a derived content for more types of changes. To this end, the merge algorithm can use a selected property of the struct values to find corresponding values in master and derived contents. The default implementation `DefaultAutoMergeStructListMapKeyFactory` is configured in the Blueprint Spring application context for some standard properties like the page grid placements.

	If this attribute is not specified, the default implementation `DefaultAutoMergeStructListMapKeyFactory` is used, which can be configured in the Spring application context.
`translatablePredicateName`	
Required	no
Description	The name of a custom Spring bean that implements the interface `TranslatablePredicate` and that is used to decide if a property is translatable. If the value is an empty string, then an instance of `BySchemaAttributeTranslatablePredicate` will be used. Translatable properties are handled differently by the merge algorithm, most importantly there won't be warnings about merge conflicts, if the property has also changed in the derived content, because that's the expected state for translated properties. If this attribute is not specified, the Spring bean with name `translatablePredicate` is used.

Table 5.39. Attributes of AutoMergeTranslationAction

```
<AggregationVariable name="derivedContents" type="Resource"/>
<AggregationVariable name="masterContentObjects" type="Resource"/>
<AggregationVariable name="autoMergeConflicts" type="Resource"/>
<Variable name="performer" type="User"/>

...

<UserTask name="Translate" successor="FollowUpAction">
  ...
  <EntryAction
class="com.coremedia.translate.workflow.AutoMergeTranslationAction"
               derivedContentsVariable="derivedContents"
               masterContentObjectsVariable="masterContentObjects"
               resultVariable="autoMergeConflicts"
               performerVariable="performer"/>
  ...
</UserTask>
```

Example 5.44. Usage of AutoMergeTranslationAction

AutoMergeSyncAction

This action extends the AutoMergeTranslationAction and allows to configure a merge strategy. See Javadoc of `AutoMergeSyncAction` for details.

`mergeStrategyVariable`	
Required	no

Description	The name of the variable into which the merge strategy bean name is stored. The bean name refers to a bean in the Spring application context that is an implementation of `ThreeWayMerge`.

Table 5.40. Attributes of AutoMergeSyncAction

```
<AggregationVariable name="derivedContents" type="Resource"/>
<AggregationVariable name="masterContentObjects" type="Resource"/>
<AggregationVariable name="autoMergeConflicts" type="Resource"/>
<Variable name="performer" type="User"/>
<Variable name="mergeStrategy" type="String"/>

<AutomatedTask name="Synchronize">
  ...
  <Action
class="com.coremedia.translate.workflow.synchronization.AutoMergeSyncAction"


autoMergePredicateFactoryName="allMergeablePropertiesPredicateFactory"
          derivedContentsVariable="derivedContents"
          masterContentObjectsVariable="masterContentObjects"
          resultVariable="autoMergeConflicts"
          performerVariable="performer"
          escalateOnConflict="true"
          mergeStrategyVariable="mergeStrategy"/>
</AutomatedTask>
```

Example 5.45. Usage of AutoMergeSyncAction

A custom merge strategy for a synchronization workflow can be configured by either by implementing the interface `ThreeWayMerge` or `SyncThreeWayMerge` and adding the bean to the Spring application context. The interface SyncThreeWayMerge allows control about whether a property needs to be merged or updated.

The bean name can be passed to the synchronization workflow via the variable mergeStrategy. How to add a custom merge strategy to the studio client is described in Section 9.28.10, “Synchronization Workflow Specifics” in *Studio Developer Manual*.

CompleteTranslationAction

After successfully completing a translation workflow, the `masterVersion` of all translated contents will be set to the current version of their masters.

If not disabled, all target contents will be marked as no longer being in translation.

Must Set skipCleanInTranslation to True

If you configured CleanInTranslationFinalAction [325] within your localization process definition, it is important to set `skipCleanInTranslation` to `true` for this action.

If not doing so, your editors may end up with a false-negative *in-translation* state displayed in *CoreMedia Studio*.

performerVariable	
Required	yes
Description	The name of the variable that contains the user in whose name this action performed. Typically, the user has been retrieved previously by the action described in Section "ExtractPerformerAction" [318].
skipCleanInTranslation	
Required	no
Description	If using the default `false` the *in-translation* status is cleared implicitly by this action. It is recommended, though, to set this to `true` and activate the explicit CleanInTranslationFinalAction [325] instead.
derivedContentsVariable	
Required	yes
Description	The name of the variable that contains all translated content items.
masterContentObjectsVariable	
Required	yes
Description	The name of the variable that contains all master content objects.

Table 5.41. Attributes of CompleteTranslationAction

```
<Variable name="performer" type="User"/>
<AggregationVariable name="derivedContents" type="Resource"/>
...

<AutomatedTask name="Complete" successor="Finish">
  <Action
    class="com.coremedia.translate.workflow.CompleteTranslationAction"
    skipCleanInTranslation="true"
    derivedContentsVariable="derivedContents"
    masterContentObjectsVariable="masterContentObjects"
    performerVariable="performer"/>
</AutomatedTask>
```

Example 5.46. Usage of CompleteTranslationAction

```
<Variable name="performer" type="User"/>
<AggregationVariable name="derivedContents" type="Resource"/>
...

<AutomatedTask name="Complete" successor="Finish">
  <Action
    class="com.coremedia.translate.workflow.CompleteTranslationAction"
    derivedContentsVariable="derivedContents"
    masterContentObjectsVariable="masterContentObjects"
    performerVariable="performer"/>
</AutomatedTask>
```

Example 5.47. Usage of CompleteTranslationAction (implicit clean-in-translation)

RollbackTranslationAction

If the master content is not needed in the target site, the translation workflow can be aborted with the `RollbackTranslationAction`. In this case all content items and folders that were created by the Section "GetDerivedContent-sAction" [310] will be deleted.

If not disabled, all target contents will be marked as no longer being in translation.

Must Set skipCleanInTranslation to True

If you configured CleanInTranslationFinalAction [325] within your localization process definition, it is important to set `skipCleanInTranslation` to `true` for this action.

If not doing so, your editors may end up with a false-negative *in-translation* state displayed in *CoreMedia Studio*.

skipCleanInTranslation	
Required	no

Description	If using the default `false` the *in-translation* status is cleared implicitly by this action. It is recommended, though, to set this to `true` and activate the explicit CleanInTranslationFinalAction [325] instead.
`contentsVariable`	
Required	yes
Description	The name of the variable that contains all content items and folders that have to be deleted during while rolling back the translation
`derivedContentsVariable`	
Required	no
Description	The name of the variable that contains all translated content items. Defaults to "derivedContents". This variable name is ignored/irrelevant if `skipCleanIn Translation` is set to `true`.
`masterContentObjectsVariable`	
Required	no
Description	The name of the variable that contains all master content objects. Defaults to "masterContentObjects". This variable name is ignored/irrelevant if `skipClean InTranslation` is set to `true`.

Table 5.42. Attributes of RollbackTranslationAction

```
<AggregationVariable name="createdContents" type="Resource"/>
...

<AutomatedTask name="Rollback" successor="Finish">
  <Action
    class="com.coremedia.translate.workflow.RollbackTranslationAction"
    skipCleanInTranslation="true"
    contentsVariable="createdContents"/>
</AutomatedTask>
```

Example 5.48. Usage of RollbackTranslationAction

```
<AggregationVariable name="createdContents" type="Resource"/>
...

<AutomatedTask name="Rollback" successor="Finish">
```

```
  <Action
    class="com.coremedia.translate.workflow.RollbackTranslationAction"
    derivedContentsVariable="derivedContents"
    masterContentObjectsVariable="masterContentObjects"
    contentsVariable="createdContents"/>
</AutomatedTask>
```

Example 5.49. Usage of RollbackTranslationAction (implicit clean-in-translation)

CleanInTranslationFinalAction

As GetDerivedContentsAction [310] implicitly registers contents as being *in-translation*, we must ensure that, to the end, this registration is undone. `CleanInTranslationFinalAction` is the recommended approach of doing so, as it ensures that the registration is always undone, no matter if the localization process ends normally or abnormally by escalation.

Alternative approaches in CompleteTranslationAction [321] and RollbackTranslationAction [323] are considered deprecated, as they may leave inconsistent *in-translation* states, that have to be cleaned up by a scheduled task `CleanInTranslation`. As scheduling is typically done at a low rate, editors may be left with false positive *in-translation* warnings until that cleanup is triggered.

Must Set skipCleanInTranslation to True

If you configured `CleanInTranslationFinalAction` within your localization process definition, it is important to set `skipCleanInTranslation` to `true` for actions, that would clean up the *in-translation* state implicitly. These are: CompleteTranslationAction [321] and RollbackTranslationAction [323].

If not doing so, your editors may end up with a false-negative *in-translation* state displayed in *CoreMedia Studio*.

derivedContentsVariable	
Required	no
Description	The name of the variable that contains all translated content items. Defaults to "derivedContents".
masterContentObjectsVariable	
Required	no

Description	The name of the variable that contains all master content objects. Defaults to "masterContentObjects".

Table 5.43. Attributes of CleanInTranslationFinalAction

```
<AggregationVariable name="masterContentObjects" type="Resource"/>
<AggregationVariable name="derivedContents" type="Resource"/>

...

<FinalAction
  class="com.coremedia.translate.workflow.CleanInTranslationFinalAction"
  derivedContentsVariable="derivedContents"
  masterContentObjectsVariable="masterContentObjects"/>
```

Example 5.50. Usage of CleanInTranslationFinalAction

5.7.3 Deriving Sites

A predefined workflow exists to derive an entire site from an existing site. The derive-site workflow cannot be adapted and is available as a built-in workflow from the module `translate-workflow`. To upload the derive-site workflow, use **cm upload -n /com/coremedia/translate/workflow/derive-site.xml** on the command line.

Typically, the derive site workflow is started as a background process from the sites window of *CoreMedia Studio*. The workflow can be started by all members of the translation manager group, as configured in the property `translationManagerRole` of the `SiteModel` (see section "Site Model" [252]). After changing this property, you have to upload the workflow again, because uploading persists the current property value.

Translation Workflow Robot

Actions performed while deriving a new site are performed as `translation-workflow-robot`. As last step when deriving a site, this user will assign possibly missing rights to the chosen site-managers-groups. This requires supervise permissions to the `/Sites` folder.

For details see Groups and Rights Administration for Localized Content Management [247].

5.7.4 Synchronization Workflow

A predefined workflow exists to synchronize content of an existing site to derived synchronization sites. The synchronization workflow cannot be adapted and is available as a built-in workflow from the module `translate-workflow`.

To upload the synchronization workflow, use

```
cm upload -n /com/coremedia/translate/workflow/synchronization.xml
```

on the command line.

The workflow can be started by all members of the translation manager group, as configured in the property `translationManagerRole` of the `SiteModel` (see section “Site Model” [252]). After changing this property, you have to upload the workflow again, because uploading persists the current property value.

6. Editorial and Backend Functionality

CoreMedia Content Cloud enhances *CoreMedia CMS* with additional functionality that is described in the following sections:

- Section 6.1, "Studio Enhancements" [329] describes extensions to *CoreMedia Studio* as the unified editing platform. The editorial usage of the features is described in the Studio User Manual.
- Section 6.2, "CAE Enhancements" [357] describes extensions to the *Content Application Engine* the delivery module of *CoreMedia Content Cloud*.
- Section 6.3, "Elastic Social" [363] describes extensions to *CoreMedia Elastic Social* that are integrated in *CoreMedia Content Cloud*. The standard functionality of *Elastic Social* is described in the Elastic Social Manual.
- Section 6.4, "Third-Party Integration" [378] describes the integration of third-party components, such as Open Street Map, into *CoreMedia Content Cloud*.

These modules are integrated into *CoreMedia Content Cloud* and the example websites and add extended functionality to their default features.

6.1 Studio Enhancements

CoreMedia Blueprint enhances *CoreMedia Studio* with plugins for better usage. This ranges from improved content editors such as the image list editor, which shows a preview of a selected image, up to a complete taxonomy management.

- Document editors

  Content query form, see Section 6.1.1, “Content Query Form” [329].
- Library, see Section 6.1.5, “Library” [334].
- Bookmarks, see Section 6.1.6, “Bookmarks” [336].
- External preview, see Section 6.1.7, “External Preview” [336].
- Content creation, see Section 6.1.9, “Content Creation” [338].
- Create content from template, see Section 6.1.10, “Create from Template” [344].
- Site selection, see Section 6.1.12, “Site Selection” [347].
- Upload dialog, see Section 6.1.13, “Upload Files” [348].
- Upload content to Salesforce Marketing Cloud, see Section 6.1.15, “Uploading Content to Salesforce Marketing Cloud” [355].

6.1.1 Content Query Form

Rather than having to maintain a collection of content items manually, you might want to just specify a search rule that updates a list of content items dynamically as new content gets added to the system. The content query form provides a convenient interface to edit such rules.

For example, you can specify a rule that finds the latest five articles from your site's sports subsection, and displays them on a "latest sports news" section of your site's front page.

In the standard configuration of *Blueprint*, you can use the query form to filter for content items according to the following aspects:

- The content item's content type
- The channel the content item belongs to
- The content item's modification date
- Whether the content item is tagged with a given location or subject tag
- Whether the content is tagged with a tag determined from the context (see Section 4.4.1.5, “Creating Content Queries” in *Studio User Manual* for more details.

Furthermore, you can order the result set by different criteria, and you can specify a maximum number of hits in order to ensure proper layout on a column-based page design, for example.

Support for dynamic content queries is bundled in the *Studio* plugin, and the main component to use is `ContentQueryForm.ts`. You can use the editor as shown in the following example.

```
Config(ContentQueryForm, {
  bindTo: config.bindTo,
  itemId: "contentQueryForm",
  forceReadOnlyValueExpression: config.forceReadOnlyValueExpression,
  queryPropertyName: "localSettings",
  documentTypesPropertyName: "documenttype",
  sortingPropertyName: "order",
  plugins: [
    Config(VerticalSpacingPlugin, {
      modifier: SpacingBEMEntities.VERTICAL_SPACING_MODIFIER_200,
    })
  ],
  conditions: [
    Config(ModificationDateConditionEditor, {
      bindTo: config.bindTo,
      propertyName: "freshness",
      group: "attributes",
      documentTypes: ["CMArticle", "CMVideo", "CMPicture", "CMGallery",
"CMChannel"],
      forceReadOnlyValueExpression: config.forceReadOnlyValueExpression,
      sortable: true,
      timeSlots: [
        {
          name: "sameDay",
         text: QueryEditor_properties.DCQE_text_modification_date_same_day,

          expression: "TODAY",
        },
        {
          name: "sevenDays",
         text: QueryEditor_properties.DCQE_text_modification_date_seven_days,

          expression: "7 DAYS TO NOW",
        },
        {
          name: "thirtyDays",
         text: QueryEditor_properties.DCQE_text_modification_date_thirty_days,

          expression: "30 DAYS TO NOW",
        },
      ]
    }),
    // ...
  ],
})
```

Example 6.1. Using the content query form

In the example, the editor is configured to allow only for a single condition (a content item's modification date). You may combine the existing condition editors - there are predefined conditions for context, date ranges, and taxonomy links - or even write your own condition editors by extending `ConditionEditorBase.ts`. Each condition editor provides the user interface for editing the respective condition, and must persist the actual search query fragment in a string

property that will be written to the respective struct property. Also, all condition editors support the configuration of a list of content types that this condition may apply to. See the API documentation for the package `com.coremedia.blueprint.base.queryeditor.conditions` for details.

When rendering the result of a search query in your CAE application, you can use `SettingsStructToSearchQueryConverter.java` to convert the search component that the editor stores in the struct property to an actual search query. See `CMQueryListImpl.java` for an example.

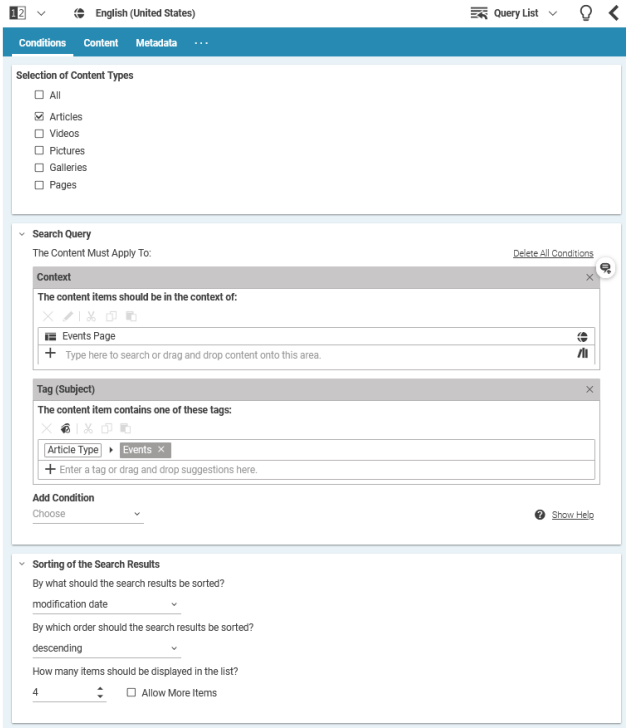

Figure 6.1. Content Query Form

6.1.2 Call-to-Action Button

If you use teasers in your website, you want to animate the users to a specific action. To make this more explicit, *Brand Blueprint* renders a button on a teaser

with a configurable text (see Figure 6.3, “Call-to-Action button in banner view” [332]). By default, this text reads "Learn more".

Figure 6.2. Call-to-Action-Button editor

You can either use the default text, define a content specific text or render no button.

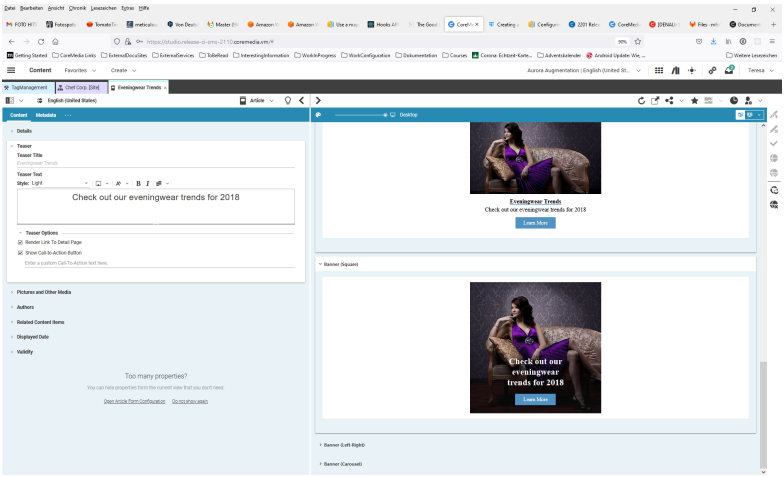

Figure 6.3. Call-to-Action button in banner view

6.1.3 Media Player Configuration

CoreMedia CMS offers the possibility to configure player settings of certain media files in a site. Player settings for Video and Audio contents can be configured in the *Video Options* and *Audio Options* panels of Video and Audio content forms.

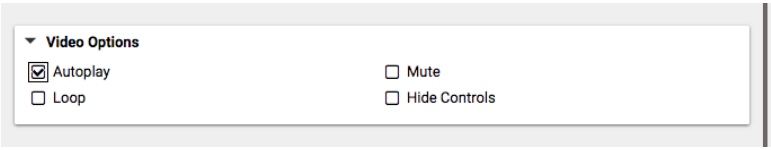

Figure 6.4. Video Options panel in the `DocumentForm` of a Video content

NOTE

Please note, that this configuration can be overwritten in the theme's FreeMarker templates. E.g., a setting a player's loop configuration in *CoreMedia Studio* will have no impact for media in a hero teaser if the template for hero teasers sets the loop option explicitly.

Configuration of media files

The following options can be enabled by checking the corresponding checkboxes in the content item form of the content item. The configuration will be saved in the content's local settings struct:

Video player settings

- Autoplay
- Mute
- Loop
- Hide Controls

Audio player settings

- Autoplay
- Loop

6.1.4 Displayed Date

When you change already published content, you have to publish this change. Of course, the publication changes the publication date of the content. However, you may want that this content always shows the date of the initial publication (or any other fixed date). To do so, you can set a custom displayed date. *Studio* contains an editor for a displayed date for all `CMLinkable` types.

This data can also be used, to sort results of a `Query List` (see Section 6.1.1, “Content Query Form” [329]).

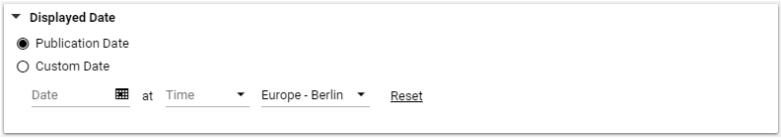

Figure 6.5. Displayed Date editor

You can either choose that the date of the last publication is used or that a fixed date is shown.

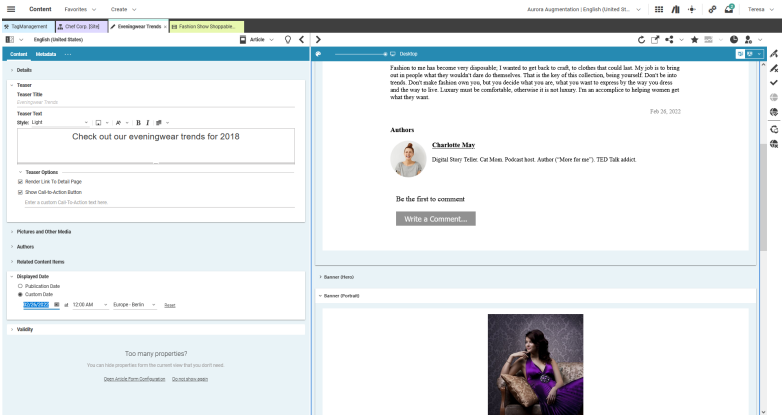

Figure 6.6. Setting a Custom Date

6.1.5 Library

The library plugin uses the extension points of the *Studio* library to extend some basic features of it and to add some new ones.

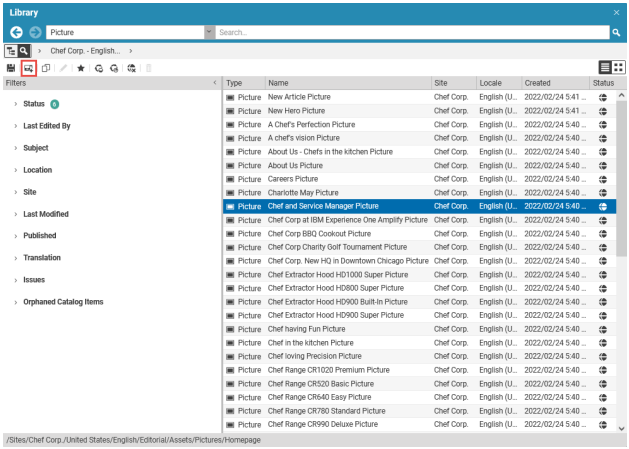

Figure 6.7. Image Gallery Creation Button

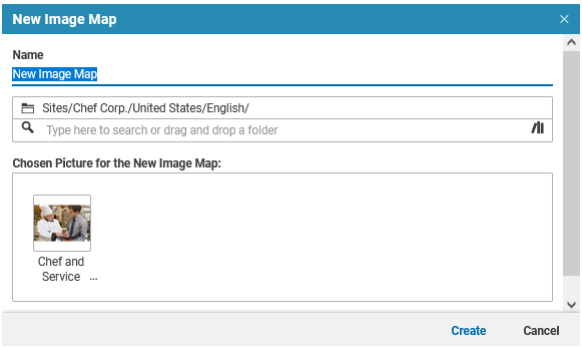

Figure 6.8. Image Gallery Creation Dialog

The image gallery creation dialog allows the user to create a new gallery content item from an image selection. The images selected in the library are shown as thumbnails in the dialog when the 'Create Image Gallery' button is pressed. After the creation of the gallery, these images are automatically assigned to the list property of the content item.

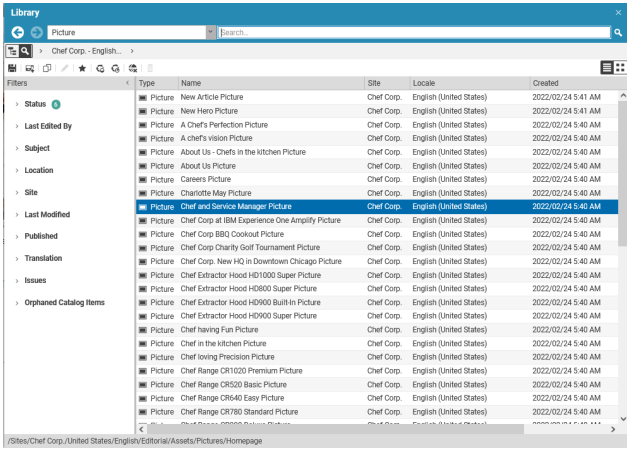

Figure 6.9. Library List View

The library plugin uses the library list view extension point to show some additional columns in the list view/search results. Additional columns are a site column, where the site name of a content item is displayed and a preview column, where images are shown as thumbnails. If the content item itself is not an image item, a referenced image is shown, such as the first picture of a gallery.

6.1.6 Bookmarks

The user can add and remove bookmarks using the bookmark action available on the preview toolbar, the library toolbar or the library list view's context menu.

Figure 6.10. Bookmarks

6.1.7 External Preview

The external preview is a Studio utility that allows you to use one or more additional displays for Studio's preview based editing. When working with *CoreMedia Studio*, the external preview can be started by clicking on the 'open external preview' button that is located on the toolbar of the preview.

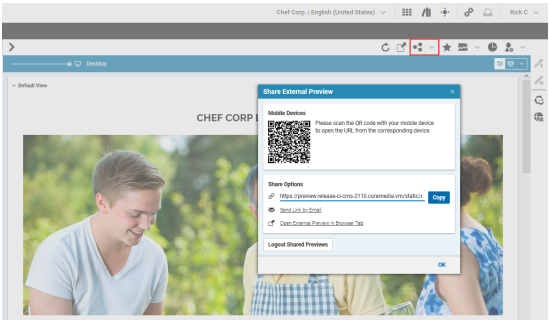

Figure 6.11. External Preview

The dialog shows the options of the external preview. It can be invoked on any browser and device, including tablets to see how the content item would look like on this device.

6.1.8 Settings for Studio

In order to use content-based settings not only for Content Application Engine usage but also for Studio, a new utility class `StudioConfigurationUtil` was introduced. Now you can, for example, configure paths used for the *Create Content* dialog (see Section 6.1.9, “Content Creation” [338]) in `CMSettings` content items.

The `StudioConfigurationUtil` class searches for bundles located at `<SITE_ROOT_FOLDER>/Options/Settings`, and falls back to `/Settings/Options/Settings` if no site-specific configuration bundle is found there. Bundle content items can be placed anywhere below these paths, and must be of type `CMSettings`.

You can use the `#getConfiguration(bundle, configuration, context)` method, where `bundle` is the name of the `CMSettings` content item, and `configuration` is a path to a respective struct property. Optionally, you can also specify a `context`. The latter can be either a `Content` or a `Site`. If you provide `Content`, the site this content item belongs to is resolved, otherwise, the given site is used as the lookup context. If you omit the `context`, the current user's preferred site is used.

The utility class is fully dependency tracked, which means that you should wrap a `FunctionValueExpression` around returned values and bind the UI components that depend on the setting to this expression.

6.1.9 Content Creation

CoreMedia Blueprint provides additional buttons and actions to create new content besides the regular content creation action in the library. The user can click on the **Create** menu on the Header toolbar to open a selection of content items to create. The action is also available for link lists and several dialogs.

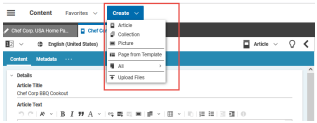

Figure 6.12. Create content menu on the Header toolbar

The user selects a content to create from the **Create** menu of the Header toolbar. Afterwards, a dialog opens where (at least) the content name and folder can be set.

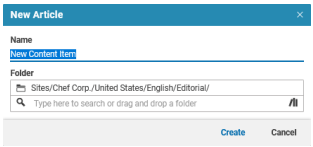

Figure 6.13. Create content dialog

The user can decide if the content should be opened in a tab afterwards. The checkbox for this is enabled by default. The *Name* and *Folder* properties are the mandatory fields of the dialog. Depending on the content type the dialog shows different property editors, for example for `Page` content items, the additional field *Navigation Parent* is configured so that the user can select the navigation parent of the new page.

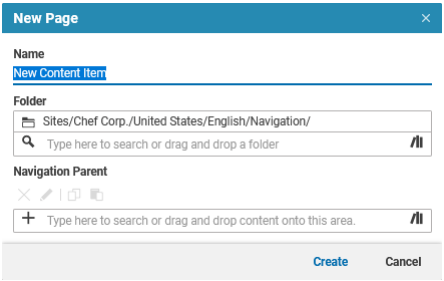

Figure 6.14. Create content dialog for pages

The dialog can be extended in several ways and plugged into existing components using the predefined menu item or button components which will invoke the dialog. Also, the dialog provides a plugin mechanism for new property editors and allows you to customize the post-processing after the content creation, depending on the type of the created content. The following "How To" sections describe how to configure and customize the dialog.

How to add a **Create** menu item to the Header toolbar

There are already some entries defined for this menu, most of them configured in the class `BlueprintFormsStudioPlugin.ts`. The menu can be extended using the `quickCreateMenuItem`:

```
<bp:newContentMenu>
    <plugins>
      <ui:addItemsPlugin>
    <ui:items>
  <bpb-components:quickCreateMenuItem contentType="MyDocumentType"/>
    ...
```

Separators can be added by:

```
<menuseparator cls="fav-menu-separator"/>
```

How to add a 'New Content' menu item to link list

There are two ways to add the content creation dialog to link lists. First is using the `QuickCreateToolbarButton` class and apply it to an existing link list using the `additionalToolbarItems` plugin. This will add one button to the toolbar of the link list to create a specific content type, for example creating a new child for the `CMChannel` content hierarchy:

```
<bp:extendedLinkListPropertyField bindTo="{config.bindTo}"
                                  propertyName="children">
    <bp:additionalToolbarItems>
    <tbseparator/>
    <bpb-components:quickCreateToolbarButton contentType="CMChannel" />
  </bp:additionalToolbarItems>
</bp:extendedLinkListPropertyField>
```

Example 6.2. Add content creation dialog to link list with `quickCreateLinkListMenu`

Figure 6.15. New content dialog as button on a link list toolbar

The second variant is that you apply a complete dropdown menu with several content types in it. By default, these content types are configured in the file

`QuickCreateSettings.properties` that is part of the `blueprint-base` and overwritten with the file `NewContentSettingsStudioPlugin.properties` (see `BlueprintFormsStudioPlugin.ts`). The file contains a property `default_link_list_contentTypes` which contains the content types to display in a comma separated value format. This default can be overwritten by adding the `contentTypes` attribute to the `quickCreateLinklistMenu` element when the dropdown elements are declared in exml. The attribute value can have a comma separated format to support multiple content types too:

```
<bp:extendedLinkListPropertyField bindTo="{config.bindTo}"
  propertyName="header">
  <bp:additionalToolbarItems>
    <tbseparator/>
    <bpb-components:quickCreateLinklistMenu bindTo="{config.bindTo}"
                              contentTypes="CMArticle,CMTeaser,..."
                              propertyName="children" />
    </bp:additionalToolbarItems>
</bp:extendedLinkListPropertyField>
```

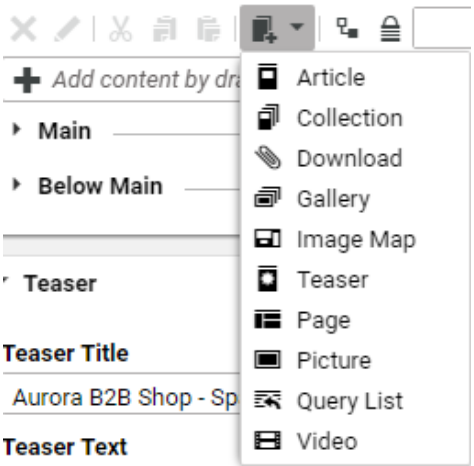

Figure 6.16. New content dialog menu on a link list toolbar

How to link new content to a link list

When the dialog is added to the toolbar of a link list by using the button component of the menu, the newly created content is automatically linked to the list. The dialog checks during the post-processing if the parameters *propertyName*" and *bindTo* have been passed to it and will link the new content to the existing ones. The dialog always assumes that if these two parameters have been passed, the corresponding property is a link list, so using other properties with other types here will raise an error here.

How to add an event handler to the button or menu item

Both components, the `quickCreateLinkListMenu` and the `quickCreateToolbarButton` provide a configuration parameter called `onSuccess`. The method passed there will be executed after a successful content creation and must provide the signature:

```
method(content:Content, data:ProcessingData, callback:Function)
```

The `ProcessingData` instance "data" contains all the data entered by the user for the mandatory and optional properties of the dialog. The object is a `Bean` instance, so the values can be accessed by using `data.get(<KEY>)` calls. Since the new content dialog has already applied all dialog properties to the content, the retrieved new content instance already contains all inputted data.

CAUTION

Ensure that the callback handler is called once the post-processing is finished. Otherwise, the post-processing of the content can not terminate correctly and steps may be missing.

How to add a content property to the new content dialog

A new property editor that should be mapped to a standard content property can be defined in the file `NewContentSettingsStudioPlugin.properties`. The configuration entry supports a comma separated format in order to apply multiple property fields to the dialog. For example when the configuration entry `item_CMArticle=title,segment` is added to the properties file, each time the dialog is opened for a `CMArticle` document the String properties "title" and "segment" are editable in the dialog and will be applied to the new content.

CAUTION

Currently only text fields are supported, so do not configure a content property here that has a different format than "String".

How to add an event handler for a specific content type

The new content dialog allows you to apply a content type depending success handlers that are executed for every execution of the dialog. The success handler must implement the following signature:

```
method(content:Content, data:ProcessingData, callback:Function)
```

and is applied to the dialog by invoking:

```
QuickCreate.addSuccessHandler(<CONTENT_TYPE>, <METHOD>);
```

CAUTION

Unlike the onSuccess handler described in the previous section, these types of event handlers will be executed for every content creation of a specific type, no matter how and where the new content dialog is invoked from.

How to add a custom property to the new content dialog

Sometimes it is necessary to configure a value for the dialog that is not a content property. Instead, the value should be processed in the success handler. The dialog allows you to apply new editors to the dialog that are mapped to a specific field in the `ProcessingData` instance.

To apply a custom editor a corresponding factory method has to be implemented that will create the editor every time the dialog is created. This factory method is applied to the dialog then by invoking:

```
QuickCreate.addQuickCreateDialogProperty(<CONTENT_TYPE>,
  <CUSTOM_PROPERTY>,
  function (data:ProcessingData, properties:Object):Component {
  ...
  //for example return new CustomEditor(customEditor{properties});
  });
```

The `ProcessingData` instance is a bean, so it can be used to create `ValueExpressions` that are passed as parameters to the component. The predefined parameters are already applied to the `properties` object that is passed to the factory method. Additional properties can be added to this object, like the `emptyText` of an input field.

CAUTION

Make sure that the name of the custom property is unique and does not match an existing property of the given content type.

Since the new editor is shown for each dialog creation of the specific type, a success handler must be applied to the dialog too that processes the value:

```
QuickCreate.addSuccessHandler(<CONTENT_TYPE>,
    <myPostProcessingHandler>);
```

The processing handler must implement the same method signature like the ones defined for menu items or buttons:

```
method(content:Content, data:ProcessingData, callback:Function)
```

The custom property can be access in the handler by invoking:

```
data.get(<CUSTOM_PROPERTY>)
```

NOTE

The post-processing of the dialog will execute the following steps:

1. create the new content
2. apply values to property fields (default processing)
3. invoke success handlers for custom processing (methods that have been applied through `QuickCreate.addSuccessHandler`)
4. invoke success handler configured for the button or menu items (methods that have been applied by declaring a value for the `onSuccess` attribute)
5. link content to a link list if parameters are defined
6. open created content
7. open additional content in background

Where do I find some examples?

Check the class `CMChannelExtension.ts`. The class adds a `successHandler` for the creation of new `CMChannel` documents that is used to apply a value for the `title` property. Additionally, the newly created `CMChannel` content item is also linked to a parent (if available) that may have been provided

by the `NavigationLinkFieldWrapper` component that also has been added to the dialog.

6.1.10 Create from Template

As described in Section 6.1.9, “Content Creation” [338] when you create a Page content item in the *Create content* menu or from a link list, you will get a new and empty content item. If you want, on the other hand, create a Page with predefined content, or even a complete navigation hierarchy, you can use the**Create content → Create from Template**menu item. This will open a dialog where you can choose your Page from predefined templates.

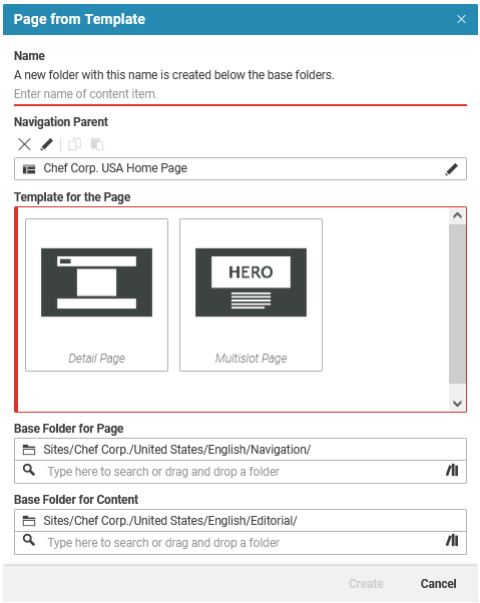

Figure 6.17. Create from template dialog

As with the standard **Create Page** dialog you can choose a name, the destination folder for the page and the navigation parent. The *Create from Template* dialog adds a template chooser from which you can select the template and a new folder chooser (*Base Folder for Content*) where you can select a destination folder for the editorial content. The folder defined in the *Base Folder for Page* chooser must not contain a folder with the name entered above.

The suggested target paths for editorial content and content used to model the navigation are taken from a content-based setting from the bundle `Content Creation` (see Section 6.1.8, "Settings for Studio" [337] for an explanation of the content-based settings mechanism). You can modify the settings *`paths.editorial`* and *`paths.navigation`* to match your specific content tree.

Location of new template folders

By default, templates will be looked up in the following folders:

- Global: `/Settings/Options/Settings/Templates/CMChannel/`
- Site specific: `Options/Settings/Templates/CMChannel/`
- User's home folder: `{USER_HOME}/Templates/CMChannel/`

The lookup path is configurable in the *Studio* properties file `CreateFromTemplateStudioPluginSettings.properties` by changing the property `pagegrid_template_paths`. Additional entries can be added in a comma separated format.

CAUTION

Keep care when you configure a template path outside the site hierarchy or when you use the global templates location. It is possible that the preconfigured layout of a global template may not be available for the active site. Therefore, the page grid extending mechanism won't work anymore, since the page grid editor can't find the layout definitions of other pages.

How to add a new template folder

Descriptor content

Template folders must have a specific format to be detected as template folders. Each template is defined in a separate folder inside the `Templates/CMChannel` folder. The folder must contain a `CMSymbol` content item named "Descriptor" that might contain an additional icon and description for the template. The icon is used as a preview in the template chooser and the description will be shown as the template name in the template chooser.

Each template folder must contain exactly one page content item at root level, otherwise the folder will be ignored. If the template consists of several pages, the sub pages should be placed within a subfolder of the template. Editorial content (Article, Images ...) that is contained in these folders and is linked by Page templates will be copied to the destination, defined in the *Create from Template* dialog.

Localization

If the name and the description should be internationalized, create an additional `Descriptor` content item next to the original descriptor and append the locale to the content name, "Descriptor_de" for the German version, for instance.

6.1.11 Open Street Map

Open Street Map is a project that creates and provides free geographic data and mapping. *CoreMedia Blueprint* supports an Open Street Map integration scenario:

- The OpenStreetMap property field (`OSMPropertyField.ts`) offers a convenient method to visually edit geographic coordinates. It displays a map segment, and users can just drag a marker to the location they want to point out. Internally, a pair of geographic coordinates (longitude/latitude) is stored in a string property field

OpenStreetMap Support is bundled in a *Blueprint* extension. Note that in order for the integration to work properly, the machine hosting the *CoreMedia Studio* web application needs to have Internet access. On startup, a connectivity check is performed, and when the machine cannot reach the OSM servers, the extension is automatically turned off.

If you have changed the default group id of the Blueprint, the property `osm.groupId` of `OSMStudioPlugin.properties` has to be adapted accordingly. Alternatively, an absolute URL for another marker icon can be specified. In that case, the `osm.groupId` should remain empty or should be removed completely.

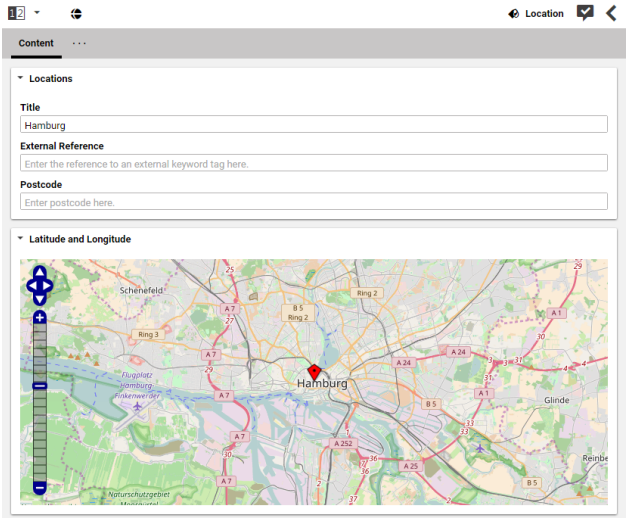

Figure 6.18. OpenStreetMap Property Editor

The OpenStreetMap property editor gives the editor the possibility to update the geographic location just by dragging a marker.

6.1.12 Site Selection

Since *CoreMedia Blueprint* provides multisite editing, a default working site can be configured in the settings dialog. If you select from *Preferred Site* for example **`'Chef Corp. - German (Germany)'`** and then create a new article, it will be moved to a folder like this `/Sites/Chef Corp./Germany/...`.

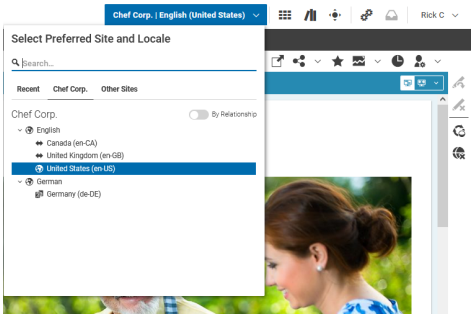

Figure 6.19. The site selector on the Header bar

6.1.13 Upload Files

Upload dialog

You can invoke the upload files dialog from the new content menu or the library. The dialog shows a drop area and the folder combo box shows where the uploaded documents will be imported to. You can drag and drop files from the desktop or the file system explorer onto the drag area. After the drop, the files are enlisted with a preview (if supported by the OS), a name text field and a mime type field. The mime type is automatically determined by the OS. After pressing the confirmation button the files are uploaded and corresponding content items are created and checked-in. You may choose to open the content items automatically after the upload is finished.

Drag and drop of files

Besides the upload dialog, you can simply drag and drop files into a folder of the Library or into a link list. Studio will automatically create the content items based on the MIME type of the file.

The upload of Word documents is a special case. If the Word document contains images, Studio will create articles for the text content as well as pictures for the images in the Word document. The article links automatically to the pictures. *CoreMedia Blueprint* contains a prototype class `WordUploadInterceptor` in the `Validators` extension. The class defines the conversion of Word documents to rich text and images. Use the class to add your own conversion logic.

NOTE

The `WordUploadInterceptor` class is only a prototype that does not support all Word documents and Word formats. Most likely, you have to adapt it to your requirements.

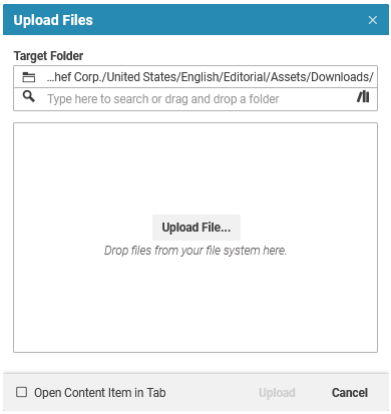

Figure 6.20. The upload files dialog

How to configure the upload settings

The upload settings are stored in the settings content item `UploadSettings` in folder `/Settings/Options/Settings`. An example for this configuration is shown below.

> **NOTE**
>
> Note that the fields `maxFileSize` and `maxFileSizeMB` from the `Upload Settings` are applied for all upload types. This includes the bulk upload and blob properties fields of content items. All other properties are only relevant for the bulk upload. For additional blob properties of the server side, please refer to the Chapter 4, *Common Concepts* in *Unified API Developer Manual*.

```
<Struct xmlns="http://www.coremedia.com/2008/struct"
xmlns:xlink="http://www.w3.org/1999/xlink">
      <StringProperty Name="defaultContentType">CMDownload</StringProperty>
      <StringProperty Name="defaultBlobPropertyName">data</StringProperty>
      <IntProperty Name="timeout">300000</IntProperty>
      <IntProperty Name="maxFileSizeMB">64</IntProperty>
      <IntProperty Name="previewMaxFileSizeMB">32</IntProperty>
      <BooleanProperty Name="previewDisabled">false</BooleanProperty>
      <BooleanProperty Name="autoCheckin">true</BooleanProperty>
      <StructProperty Name="mimeTypeMappings">
        <Struct>
          <StringProperty Name="image">CMPicture</StringProperty>
          <StringProperty Name="application">CMDownload</StringProperty>
          <StringProperty Name="audio">CMAudio</StringProperty>
          <StringProperty Name="video">CMVideo</StringProperty>
          <StringProperty Name="text">CMDownload</StringProperty>
          <StringProperty Name="text/css">CMCSS</StringProperty>
       <StringProperty Name="text/javascript">CMJavaScript</StringProperty>

          <StringProperty Name="text/html">CMHTML</StringProperty>
```

```
        </Struct>
      </StructProperty>
      <StructProperty Name="mimeTypeToMarkupPropertyMappings">
        <Struct>
          <StringProperty Name="text/css">code</StringProperty>
          <StringProperty Name="text/javascript">code</StringProperty>
          <StringProperty Name="text/html">data</StringProperty>
        </Struct>
      </StructProperty>
    </Struct>
```

For a detailed description about the elements and attributes see table below.

autoCheckin	
Format	String
Description	If set to *true*, the uploaded contents are checked in after being created.
defaultContentType	
Format	String
Description	The default content type to create if the mime type of a file has no corresponding mime type mapping.
defaultBlobPropertyName	
Format	String
Description	The default blob property name to which the file blob is written to.
previewMaxFileSizeMB	
Format	Number
Description	Files that are dropped to the upload dialog and are larger than this value won't have a preview. This is used to avoid browser crashed for big file and defaults to 32MB.
previewDisabled	
Format	Boolean
Description	Boolean flag to disable the preview of upload items completely, defaults to 'false'.

mimeTypeMappings	
Format	Struct
Description	Depending on the mime type the content type to generate is mapped here. Here the primary type or the whole mime type can be specified.
mimeTypeToMarkupPropertyMappings	
Format	Struct
Description	Depending on the mime type the markup property name to which the file is written.
mimeTypeToBlobPropertyMappings	
Format	Struct
Description	Depending on the mime type the blob property name to which the file is written.
timeout	
Format	Integer
Description	The timeout in milliseconds for uploads, default value is 300000.
maxFileSizeMB	
Format	Integer
Description	The maximum allowed file size to upload in megabytes, default value is 64. Starting with version 12.2404 this setting supersedes `maxFileSize` to allow for uploads greater than 2GB.

Table 6.1. Upload Settings

How to intercept the content's properties before creation

There is an example of a Content Write Interceptor contained in the `Validators` extension:

```
<bean id="pictureUploadInterceptor"
    class="com.coremedia.rest.cap.intercept.PictureUploadInterceptor">
    <property name="priority" value="-1"/> <!-- Ensure that this interceptor
 is executed before other blob interceptors -->
    <property name="type" value="CMPicture"/>
    <property name="imageProperty" value="data"/>
    <property name="widthProperty" value="width"/>
    <property name="heightProperty" value="height"/>

    <!-- uploadLimit: max image size (width * height) in pixels. Images are
 not uploaded if too big to prevent
         OutOfMemoryExceptions. -->
    <property name="uploadLimit" value="100000000"/>

    <!-- maxDimension: max width and height in pixels of stored images in
the database. Images are scaled down
         if too big. -->
    <property name="maxDimension" value="4000"/>
    <property name="blobTransformer" ref="blobTransformer"/>
    <property name="extractor" ref="imageDimensionsExtractor"/>
    </bean>
```

It is a Content Write Interceptor for the `CMPicture` content type which scales an uploaded image blob to a configurable max dimension and writes the image dimensions to the width and height String property of the image content item. It also rejects images that exceeds a total pixel size specified in the `uploadLimit` property to avoid the JVM from running out of memory. See the Studio Developer Manual for Content Write Interceptor. The interceptor class itself is now part of the core. You can find other interceptor sources files in the `Validators` extension, for example, the `WordUploadInterceptor.java` file.

6.1.14 Studio Preview Slider

Introduction

CoreMedia Studio's preview features a `slider` tool. The slider tool was build to let the user choose between devices with different resolutions in order to let the preview perform a responsive transformation of the page in the preview window. This means, that the preview will show the page as if it was to be viewed on a device with a different resolution than a "conventional" desktop display (that is a mobile device for instance).

Figure 6.21. The slider of the Studio Preview

Configuration of preview CAE

In order to enable the responsive slider functionality, you have to enable the use of `metadata tags` within the FreeMarker templates. These tags are used for communication between the CAE and *CoreMedia Studio* in order to exchange meta information about the previewed page. (See `CoreMedia Studio Developer Manual` for more details about metadata tags). The following listing illustrates the enabled setting within the file `cae-preview-app/src/main/resources/application.properties`:

```
cae.preview.metadata-enabled=true
```

> **NOTE**
>
> The settings for the responsive slider can also be configured for each theme individually. Therefore, they are assigned in the theme's settings Json files. For more information see Frontend Developer Manual.

Integration of metadata tags in FreeMarker templates

The following list illustrates the use of metadata tags in the `Page.body.ftl` template.

```
<#ftl strip_whitespace=true>

  <#-- responsive design slider information for studio -->
    <#assign sliderMetadata={
            "cm_preferredWidth": 1281,
            "cm_responsiveDevices": {
    <#-- list of the devices.
    naming and icons see: BlueprintDeviceTypes.properties
    the default icons are in studio-core, but you can define
    your own style-classes in slider-icons.css.
    -->
    <#-- e.g. iphone4 -->
      "mobile_portrait": {
      "width": 320,
      "height": 480,
      "order": 1,
      "isDefault": true
      },
    <#-- e.g. iphone4 -->
      "mobile_landscape": {
      "width": 480,
      "height": 320,
      "order": 2
      },
    <#-- e.g. nexus7 -->
      "tablet_portrait": {
      "width": 600,
      "height": 800,
      "order": 3
      },
```

```
    <#-- e.g. ipad -->
      "hybrid_app_portrait": {
      "width": 768,
      "height": 1024,
      "order": 4
      },
    <#-- e.g. nexus7 -->
      "tablet_landscape": {
      "width": 960,
      "height": 540,
      "order": 5
      },
    <#-- e.g. ipad -->
      "hybrid_app_landscape": {
      "width": 1024,
      "height": 768,
      "order": 6
      }
    }
  }
/>
```

To introduce new devices with even different resolutions, simply extend the content of the file appropriately.

Configuration in Studio

The configuration in Studio has to be made in the appropriate bundle files. The following listing shows the content of the file `apps/studio-client/modules/studio/blueprint-forms/src/main/joo/com/coremedia/blueprint/studio/BlueprintDeviceTypes.properties`.

```
      Device_mobile_portrait_icon=Resource(key='channel_mobile_portrait',
bundle='com.coremedia.icons.CoreIcons')
      Device_mobile_landscape_icon=Resource(key='channel_mobile_landscape',
 bundle='com.coremedia.icons.CoreIcons')
      Device_tablet_portrait_icon=Resource(key='channel_tablet_portrait',
bundle='com.coremedia.icons.CoreIcons')
      Device_tablet_landscape_icon=Resource(key='channel_tablet_landscape',
 bundle='com.coremedia.icons.CoreIcons')
      Device_notebook_icon=Resource(key='channel_notebook',
bundle='com.coremedia.icons.CoreIcons')
      Device_desktop_icon=Resource(key='channel_desktop',
bundle='com.coremedia.icons.CoreIcons')
     Device_hybrid_app_portrait_icon=Resource(key='channel_tablet_portrait',
 bundle='com.coremedia.icons.CoreIcons')
     Device_hybrid_app_landscape_icon=Resource(key='channel_tablet_landscape',
 bundle='com.coremedia.icons.CoreIcons')

      Device_mobile_portrait_text=Mobile
      Device_mobile_landscape_text=Mobile
      Device_tablet_portrait_text=Tablet
      Device_tablet_landscape_text=Tablet
      Device_notebook_text=Notebook
      Device_desktop_text=Desktop
      Device_hybrid_app_portrait_text=Hybrid App
      Device_hybrid_app_landscape_text=Hybrid App
      Device_desktopMode_text=Desktop
```

The configuration, which is relatively straightforward, consists of two parts. The top part of the configuration deals with the appropriate icons, that will be dis-

played for the according device type in the slider. The bottom part defines the text, that will be shown next to the slider. This configuration can be extended to introduce new device types with new device icons. For configuring the device icons, perform the following step:

- Declare a new class for the configured icon name in the file `apps/studio-client/modules/studio/blueprint-forms/src/main/resources/META-INF/resources/joo/resources/css/slider-icons.css`.

6.1.15 Uploading Content to Salesforce Marketing Cloud

Salesforce Marketing Cloud (SFMC) is a customer relationship management (CRM) tool by Salesforce. *CoreMedia Blueprint* supports pushing CoreMedia content to the SFMC. To enable this feature you have to configure the following settings for SFMC.

- `sfmc-credentials-clientId`: The client ID of the client credential
- `sfmc-credentials-customerId`: The subdomain of the client
- `sfmc-credentials-clientSecret`: The second part of the client credential. The passwords can be encrypted by using the tool `cm encrypt passwords` as described in Section 3.13.2.7, "Encryptpasswords" in *Content Server Manual*.
- `sfmc-push-translations`: Whether to push into SFMC the master language and all translations of the configured text properties, or only the master language. This property can be `true` or `false`

The settings can be configured globally in the application context of the Studio application or in the settings content item under the path `/Settings/Options/Settings/Marketing/Salesforce Marketing Cloud`. The SFMC settings can be also configured site-specific in the site settings.

The client credentials have to be set up in the Salesforce Marketing Cloud settings as a new "Installed Package". The new package needs a component for an API Integration (server-to-server) with the following permissions.

- documents_and_images_read
- documents_and_images_write
- saved_content_write
- journeys_read

- list_and_subscribers_read
- data_extensions_read
- data_extensions_write

In addition, the uploadable content properties for a given content type have to be configured. This can be done globally in the same settings content item as above. To that end add a struct `sfmc-uploadableProperties` to the `settings` property of the content item. The credentials can be also configured site-specific in the site settings.

sfmc-push-translations	true	String
▾ sfmc-uploadableProperties		Struct
CMArticle	title,detailText,pictures	String
CMTeaser	teaserTitle,teaserText,pictures	String
CMPicture	data	String

Figure 6.22. SFMC Uploadable Properties Setting

Currently, CoreMedia supports the string, richtext and blob image property. In the configuration example above the string property `title` and the richtext property `detailText` of the content item type `CMArticle` and the image blob property `data` of the content type `CMPicture` are configured as uploadable content properties.

6.2 CAE Enhancements

This section describes enhancements of the *Content Application Engine*.

- Section 6.2.1, "Using Dynamic Fragments in HTML Responses" [357] describes how context dependent HTML snippets can easily be used in a *Content Application Engine* application.
- Section 6.2.2, "Image Cropping in CAE" [361] describes how you can use cropped images in the CAE.
- Section 6.2.3, "RSS Feeds" [362] describes how you can generate an RSS feed from content.

6.2.1 Using Dynamic Fragments in HTML Responses

Basic concept

Fragments of responses generated by the *Content Application Engine* may depend on a context, for example session data or the time of day. If fragments of a response may not be valid for every request, and responses are cached by reverse proxies (like Varnish or a CDN), it's necessary to exclude those parts from the response and load them separately using techniques like AHAH / Ajax or ESI.

To load the fragments, a link scheme and a matching handler handling the bean's type are needed.

CAE Implementation

In order to support loading of fragments in a generic and almost transparent way, beans are wrapped in a (`com.coremedia.blueprint.cae.view.DynamicInclude`) bean when they are included in the view layer. Whether the bean is wrapped or not is decided using `Predicate<RenderNode>` implementations that are called with the current `RenderNode`. A `RenderNode` represents the current "self" object and the view it's supposed to be rendered in. If any of the available predicates evaluate to true, the bean and view is wrapped as described above.

```
    public class DynamicPredicate implements DynamicIncludePredicate {

  //only use DynamicInclude if view matches.
  private static final String VIEW_NAME="myView";

  public boolean apply(RenderNode input) {

    if (input == null) {
      return false;
    } else if (input.getBean() instanceof MyBean
          && VIEW_NAME.equals(input.getView())) {
      return true;
    }

    return false;
  }
}
```

Example 6.3. Predicate Example

The predicate has to be added to a predefined Spring bean in order to be evaluated:

```
    @Customize("dynamicIncludePredicates")
@Bean(autowireCandidate = false)
public List<DynamicIncludePredicate> addMyDynamicPredicates() {
  return List.of(new MyDynamicPredicate());
}
```

Example 6.4. Predicate Customizer Example

Render fragment placeholder

After wrapping the bean, the `DynamicInclude` is then rendered by the *Content Application Engine*.

`DynamicInclude` beans are rendered just as other beans by the *Content Application Engine*. By default, the view `DynamicInclude.ftl` is used to render the beans. It will either add a placeholder DOM element that can be used to load the fragment using AHAH, or an `<esi:include>` tag, depending on whether there is a reverse proxy telling the CAE to do so using the `Surrogate-Capability` header. This is described in the Edge Architecture Specification.

Links to dynamic fragments

In order to generate a link for either AHAH or ESI, a separate link scheme must be created for each bean type that should be included dynamically.

If the fragment depends on the context (for example, Cookies, session or the time of day), the link scheme must have the prefix `/dynamic/` (see `UriConstants$Prefixes`) so that a preconfigured interceptor will set all Cache headers necessary that downstream proxies never cache those fragments. Matching Apache and Varnish rewrite rules are provided by *CoreMedia Blueprint*.

```
      @Link(type = MyBean.class,
      view = "fragment",
      uri = "/dynamicfragment/mybean")
public UriComponents buildFragmentLink(Cart cart,
        UriTemplate uriPattern,
        Map<String, Object> linkParameters,
        HttpServletRequest request) {

    UriComponentsBuilder result =fromPath(uriPattern.toString());
    //parameter "targetView" needs to be added
    result.queryParam("targetView",linkParameters.get("targetView"));
    return result.build();
}
```

Example 6.5. Dynamic Include Link Scheme Example

Handling dynamic fragments

These links have to be handled by using a handler. The handler has to use the `RequestParam "targetView"` to be able to construct a `ModelAndView` matching the values as originally intended in the `include` including the original bean.

```
      @RequestMapping(value="/dynamicfragment/{mybean}")
public ModelAndView handleFragmentRequest(
       @PathVariable("mybean") String mybean,
       @RequestParam(value = "targetView") String view) {

  Object myBean = resolve(mybean);

  //do not create Page, return bean directly (!)
  ModelAndView modelWithView = createModelWithView(myBean, view);
  return modelWithView;
}
```

Example 6.6. Dynamic Include Handler Example

Preserve view parameters for dynamic fragments

When including fragments dynamically expect the same behaviour as for server side includes. This means that the view parameters which may include all kinds of objects need to be passed to subsequent templates.

To preserve these parameters a hashed string representation of the parameters will be appended by the `IncludeParamsAppendingLinkTransformer` as `includeParams` query parameter to the asynchronous call. When receiving the call, the `IncludeParamsFilter` will retrieve the view parameters back from the query parameter.

NOTE

Custom URI paths, considered by the `IncludeParamsAppendingLinkTransformer` may be configured via `cae.link-transformer.uri-paths` property.

A server side secret for the hash generation has to be configured via `cae.hashing.secret` property.

Have a look at Table 3.1, "Configuration Properties with Prefix cae" in *Deployment Manual* for further information.

WARNING

If the server side secret for the hash generation is not configured, the CAE generates a secret and prints it to the log. You may copy the secret value to your config. If the secret changes then hashes change and may break HTTP caching.

The following types of view parameters are supported for dynamic fragments:

- primitives (`boolean`,`int`, `float`,`long`)
- `String`
- `ContentBeans`
- `Content`
- Maps and Collections of the above types.

Additional custom types may be configured via the `cae.link-transformer.serializer-classes` property.

CAUTION

For every custom type an `IdScheme` registered at the `IdProvider` is presumed.

6.2.2 Image Cropping in CAE

As described in Section 9.5.3, "Image Cropping and Image Transformation" in *Studio Developer Manual*, there are predefined crops, which can be applied to image rendering in the CAE. *CoreMedia Blueprint* comes with four predefined cropping definitions.

- portrait_ratio3x4
- portrait_ratio1x1
- landscape_ratio4x3
- landscape_ratio16x9

The necessary settings for the image will be set by *Studio* once you open the image in *Studio*. To render images correctly even if they were not imported through *Studio*, the CAE provides a default cropping configuration for those images, which don't have the settings explicitly set. Please refer to the Javadoc of `com.coremedia.cap.transform.Transformation` for all configuration possibilities. New Spring bean definitions of this class will be automatically injected to the `TransformImageService` that is responsible for all variant definitions.

Site Specific Image Variants

For the CAE, the class `TransformImageService` is responsible for loading site specific cropping information. The feature can be enabled by changing/adding the Spring property `imagetransformation.dynamic-variants` to `true`.

The `TransformImageService` requires a lookup of the Struct that contains the information about the image variants. Therefore, it must be injected with an instance of `VariantsStructResolver` which resolves the global and site specific image variants. The implementation of this interface is part of the `shared` module `image-transformation`, since the lookup is content type specific and therefore can not be part of the core.

For example the Corporate site comes with additional predefined cropping definitions.

- portrait_ratio20x31
- portrait_ratio3x4
- portrait_ratio1x1
- landscape_ratio4x3
- landscape_ratio16x9
- landscape_ratio5x2

- landscape_ratio4x1

6.2.3 RSS Feeds

The CAE supports the rendering of RSS feeds for all content beans that implement the interface `FeedSource`. The interface is currently implemented by the classes `CMNavigation` and `CMCollection`. Feeds can be generated by invoking URLs with the following pattern:
`/service/rss/[SITE_URL_SEGMENT]/[CONTENT_ID]/feed.rss for the RSS feed with content id [CONTENT_ID]`

The programmed view `FeedView` collects all content beans that should be part of the feed and generates the RSS XML that is returned to the browser. The default implementation of the *CoreMedia Blueprint* returns the contents of the `items` list for `CMCollection` beans and the content of the `main` section for `CMNavigation` beans.

The conversion from a content bean to a feed entry is implemented through `FeedItemDataProviders`. The programmed view `FeedSource` contains a list of `FeedItemDataProvider` instances. If a content bean is applicable to a `FeedItemDataProvider`, the content bean is passed to it and the RSS entry with all required data is returned.

CoreMedia Blueprint comes with the following `FeedItemDataProviders`:

- `TeasableFeedItemDataProvider`
- `PictureFeedItemDataProvider`

The amount of items that should be part of an RSS feed can be limited by setting the `String` struct property 'RSS.limit' in a settings content item that is part of the invoked context.

6.3 Elastic Social

CoreMedia Elastic Social is integrated into *CoreMedia Blueprint*. It includes the following features:

- Comments and Reviews

  Comments and reviews are supported for any kind of editorial CMS content items, for example articles and products. It is possible to configure for a context if writing comments or reviews is enabled and if it is allowed for anonymous or registered users. A review includes 5-star ratings with title and text.

  Elastic Social provides aggregations like "Most Commented" or "Top Reviewed" content in a defined time interval for a certain context.

- User Profiles

  User profiles can be created using a registration flow and can be managed in the *CAE* by the user or in the *Studio* plugin "User Management".

  A user profile is activated by a user via a link in a registration confirmation email.

- Moderation

  In the moderation of *Elastic Social* comments, reviews and user profiles can be edited, approved or rejected. In case of rejecting, a preconfigured template-based email can be sent directly or be modified by the moderator before sending it. A prioritization for comments, reviews or user profiles can be set. For all items that have to be moderated, premoderation, post-moderation or no moderation can be configured.

- Ratings

  Rating is supported for any kind of editorial CMS content item, like articles. Ratings are provided via a five star model. *Elastic Social* calculates average ratings for the star rating model and aggregates "Top Rated" and "Most Rated" content items per channel for a certain time span and context for a channel.

- Registration

  A user can register by creating a community user from scratch.

- Authentication

  The authentication is handled by *Elastic Social*.

- Password Reset

Password reset is available for registered users who authenticate directly with *Elastic Social*.

- User Management

  The *Elastic Social* user management in *Studio* includes a search for community users. The user management allows editing, searching, approving, blocking, ignoring and deleting users, as well as resending registration confirmation emails.

- All Contributions

  In the *All Contributions* section in *Studio* a list of all comments and reviews can be displayed. The list can be filtered by user, type, status or search term. Selected comments/reviews can then be edited, remoderated and marked for later editorial use.

- Display custom information in *Studio*

  Custom information about users, comments or reviews can easily be integrated into the *Studio* moderation and user management via extension points.

- Emails

  An email for a specific event can be sent by implementing the corresponding listener. Email templates can be created and edited in *Studio*.

6.3.1 Configuring Elastic Social

This section describes the configuration of the *Elastic Social* plugin.

Context settings for Elastic Social are defined in the following contexts:

- Root channel: Application context settings can only be defined in the root channel and can not be overwritten
- Every Channel: Channel context settings can be defined in every channel and are inherited or can be overwritten by child channels

Root Channel

The following context settings are defined for the root channel and can not be overwritten:

`tenant`	
Type	String property
Description	The tenant

Example	elastic
Default Value	
Required	true
`userModerationType`	
Type	String Property
Description	Moderation type for users
Example	*`PRE_MODERATION`*, *`POST_MODERATION`*, *`NONE`*
Default Value	NONE
Required	false
`recaptchaForRegistrationRequired`	
Type	Boolean property
Description	Enable/disable captcha for user registration
Example	true, false
Default Value	false
Required	false

Table 6.2. Root Channel Context Settings

The context setting tenant is needed to define which tenant is used for a site.

```
<?xml version="1.0" encoding="UTF-8"?>
<Struct xmlns="http://www.coremedia.com/2008/struct"
        xmlns:xlink="http://www.w3.org/1999/xlink">
  <StructProperty Name="elasticSocial">
    <Struct>
      <StringProperty Name="tenant">
        elastic
      </StringProperty>
      <StringProperty Name="userModerationType">
        POST_MODERATION
      </StringProperty>
```

```
      <BooleanProperty Name="recaptchaForRegistrationRequired">
        true
      </BooleanProperty>
    </Struct>
 </StructProperty>
</Struct>
```

Example 6.7. Root Channel Context Settings

```
<?xml version="1.0" encoding="UTF-8"?>
<Struct xmlns="http://www.coremedia.com/2008/struct"
        xmlns:xlink="http://www.w3.org/1999/xlink">
  <StructProperty Name="elasticSocial">
    <Struct>
      <StringProperty Name="tenant">
        elastic
      </StringProperty>
      <StringProperty Name="userModerationType">
        POST_MODERATION
      </StringProperty>
    </Struct>
 </StructProperty>
</Struct>
```

Example 6.8. Root Channel Context Settings

Every Channel

The following context settings can be defined per channel and are inherited or can be overwritten by child channels:

Name	Type	Description	Example	Default value
`enabled`	Boolean Property	Enable/disable feedback for the channel. If disabled, all other settings are ignored	true, false	false
`commentType`	String Property	Disable commenting generally by settings this property to DISABLED. Enable reading comments by setting this property to READONLY. Enable only registered users to write comments by settings the property to REGISTERED. Enable all users (re-	DISABLED, READONLY, REGISTERED, ANONYMOUS	DISABLED

Name	Type	Description	Example	Default value
		gistered and anonymous) to write comments by settings the property to ANONYMOUS. This property is only available if enabled is true.		
`reviewType`	String Property	Disable reviewing generally by settings this property to DISABLED. Enable reading reviews by setting this property to READONLY. Enable only registered users to write reviews by settings the property to REGISTERED. Enable all users (registered and anonymous) to write reviews by settings the property to ANONYMOUS. This property is only available if enabled is true.	DISABLED, READONLY, REGISTERED, ANONYMOUS	DISABLED
`recaptchaForReviewRequired`	Boolean Property	Enable reCAPTCHA for Reviews and Ratings.	true, false	false
`commentModerationType`	String Property	Moderation Type for comments.	*PRE_MODERATION*, *POST_MODERATION*, *NONE*	NONE
`reviewModerationType`	String Property	Moderation Type for reviews.	*PRE_MODERATION*, *POST_MODERATION*, *NONE*	NONE

Name	Type	Description	Example	Default value
`reviewDocumentTypes`	String List Property	Optional whitelist of technical content type identifiers for reviews. Do not set this configuration if reviews should be available for all subtypes of CMTeasable	*`CMArticle`*, *`CMTeasable`*, etc.	
`commentDocumentTypes`	String List Property	Optional whitelist of technical content type identifiers for comments. Do not set this configuration if comments should be available for all subtypes of CMTeasable	*`CMArticle`*, *`CMTeasable`*, etc.	
`likeDocumentTypes`	String List Property	Optional whitelist of technical content type identifiers for likes. Do not set this configuration if likes should be available for all subtypes of CMTeasable	*`CMArticle`*, *`CMTeasable`*, etc.	
`ratingDocumentTypes`	String List Property	Optional whitelist of technical content type identifiers for ratings. Do not set this configuration if ratings should be available for all subtypes of CMTeasable	*`CMArticle`*, *`CMTeasable`*, etc.	
`defaultNumberOfReviews`	Integer Property	Default number of reviews to be displayed initially. If 0, all reviews are displayed.	3	0

Name	Type	Description	Example	Default value
`maxImageFileSize`	Integer Property	Maximum size of uploaded images (in bytes).	512000	512000
`userImageHeight`	Integer Property	Height of user image in px.	150	150
`userImageWidth`	Integer Property	Width of user image in px.	200	200
`userImageThumbnailHeight`	Integer Property	Height of user thumbnail image in px.	48	48
`userImageThumbnailWidth`	Integer Property	Width of user thumbnail image in px.	48	48
`userImageCommentThumbnailHeight`	Integer Property	Height of user thumbnail image in px, displayed for a comment.	48	48
`recaptchaPublicKey`	String Property	ID of your registered reCAPTCHA app	ABCD123...	
`recaptchaPrivateKey`	String Property	Secret authentication key of your registered reCAPTCHA app	ABCD123...	

Name	Type	Description	Example	Default value
`filterCategories`	LinkListProperty	Configures filter options for the comment moderation list. You can add navigation and taxonomy documents.		

Context Settings for Every Channel

Table 6.3. Context Settings for Every Channel

```
<?xml version="1.0" encoding="UTF-8"?>
<Struct xmlns="http://www.coremedia.com/2008/struct"
        xmlns:xlink="http://www.w3.org/1999/xlink">
  <StructProperty Name="elasticSocial">
    <Struct>
      <BooleanProperty Name="enabled">
        true
      </BooleanProperty>
      <StringProperty Name="commentType">
        ANONYMOUS
      </StringProperty>
      <StringProperty Name="reviewType">
        REGISTERED
      </StringProperty>
      <StringProperty Name="commentModerationType">
        PRE_MODERATION
      </StringProperty>
      <StringProperty Name="reviewModerationType">
        PRE_MODERATION
      </StringProperty>
    </Struct>
  </StructProperty>
</Struct>
```

Example 6.9. Context Settings for Every Channel

6.3.2 Displaying Custom Information in Studio

You can show additional information inside the moderation tab and user management window of *CoreMedia Studio* by extending the *Studio* web application (server side) and modifying the `ElasticSocialStudioPlugin.ts` (client side).

Server Side: REST JsonCustomizer

Provide a `JsonCustomizer` to the *Studio* web application that adds the additional information to the data that is transferred from the REST backend to the *Studio* app for users:

```
@Named
public class MyCommunityUserJsonCustomizer implements
JsonCustomizer<CommunityUser> {
  public void customize(CommunityUser communityUser, Map<String, Object>
serializedObject) {
    serializedObject.put("additional", communityUser.getProperty("information",
 String.class));
  }
}
```

or for comments:

```
@Named
public class MyCommentJsonCustomizer implements JsonCustomizer<Comment> {
  public void customize(Comment comment, Map<String, Object> serializedObject)
 {
    serializedObject.put("additional", comment.getProperty("information",
String.class));
  }
}
```

Client Side (1): Display Custom Properties

Three extension points are provided for displaying custom properties for comments or users.

1. Extend the `CommentExtensionTabPanel` to add components for comments that are displayed above the approve and reject buttons inside the moderation/archive tab (use `activeContributionAdministration` in the expression for the `ElasticPluginLabel` in order to reference the active contribution administration, depending on whether the moderation or the archive tab is active):

```
rules: [
  ...
  Config(CommentExtensionTabPanel, {
    plugins: [
      Config(AddItemsPlugin, {
        items: [
          Config(Panel, {
            title: "additionalInformation",
            items: [
              Config(ElasticPluginLabel, {
                fieldLabel: "additional",
                expression:
"activeContributionAdministration.displayed.additional"
              }),
            ]
```

```
          }),
        ],
      }),
    ],
  }),
  ...
]
```

2. Extend the `UserProfileExtensionTabPanel` to add components for user profiles that are displayed above the approve and reject buttons inside the moderation tab:

```
rules: [
  ...
  Config(UserProfileExtensionTabPanel, {
    plugins: [
      Config(AddItemsPlugin, {
        items: [
          Config(Panel, {
            title: "additionalProfileInformation",
            items: [
              Config(ElasticPluginLabel, {
                fieldLabel: "additional",
                expression:
"activeContributionAdministration.displayed.additional"
              }),
            ]
          }),
        ],
      }),
    ],
  }),
  ...
],
```

3. Extend the `CustomUserInformationContainer` to add components that are displayed below the user meta information panel inside the user management view:

```
rules: [
  ...
  Config(CustomUserInformationContainer, {
    plugins: [
      Config(AddItemsPlugin, {
        items: [
          Config(Container, {
            items: [
              Config(ElasticPluginLabel, {
                fieldLabel: "additional",
                expression: "userAdministration.edited.additional"
              }),
            ]
          }),
        ],
      }),
    ],
  }),
```

```
  ...
],
```

Client Side (2): Edit Custom Properties

For all three extensions points described above it is also possible to not just display but to edit/moderate custom properties. Instead of `ElasticPluginLabel` just use `ElasticPluginPropertyField`. This provides a text field for editing the property. Number or Boolean fields are not provided but can be constructed analogously. When you construct your own property field it is important to register the corresponding property as being moderated. This can either be done directly by your property field (c.f. `ElasticPluginPropertyFieldBase`) or you use the `RegisterModeratedPropertiesPlugin` for this purpose.

6.3.3 Adding Custom Filters for Moderation View

The list of moderated items of the *Moderation View* includes a filter section (see chapter *Using Elastic Social* of the *CoreMedia Studio User Manual*). By default, this section encompasses a filter for showing/hiding comments and users and for filtering comments in terms of comment categories.

It is possible to add further filters. You have to add your custom `FilterPanel` to the container `ModeratedItemsSearchFilters` via the `AddItemsPlugin`.

Each `FilterPanel` has to implement the method `buildQuery()`. For the case of moderation list filters, it has to return a string denoting comment/user properties and their desired values for filtering. Comment properties have to be prefixed with `"comments_"`. User properties have to be prefixed with `"users_"`.

For instance, if your filter returned `"comments_authorName=Nick"`, only comments written by an author named *Nick* would show up. You can combine multiple property-value pairs by separating them with `"&"`

Note that you probably have to provide appropriate indexes for your database in order to prevent your custom filters to have a negative effect on query performance.

6.3.4 Emails

Sending emails is supported by *Elastic Social* and can easily be incorporated for common use cases in a project. *Elastic Social* provides listeners which can be implemented to send emails (see *Elastic Social* documentation).

In *CoreMedia Blueprint*, an example implementation and the corresponding mail templates are provided. For example the `SendRegistrationMailListener` is a provided listener to send emails with a link to confirm a registration.

The `MailTemplateService` allows you to generate and send emails with a template name and parameters.

In the provided implementation in *Blueprint* the template name references a content item in the CMS with content type `CMMail` (email Template).
The parameters define variables which can be used in the mail templates. Locale specific mail templates are used if a locale specific variant is available (locale specific suffixed name).

Per default all properties of a `CommunityUser` (the model for a user) are available as variable in a mail template. For example you can use `$givenName` to include the given name of a user (if you use *FreeMarker* for templating as *CoreMedia Blueprint* does). Additional parameters must be provided programmatically by passing them as map `additionalParameters` to the `MailTemplateService`.

In *CoreMedia Blueprint*, the following mail templates for the user and moderation processes are already provided with the example content. For each mail template, the template name and additional parameters are described.

If you want to use different additional parameters, redefine the variable in the mail template and pass the corresponding parameter in the `additionalParameters` map. All properties of the `CommunityUser` can be used in the templates without changing the code.

Use case	Template Name	Additional Parameters
Registration	registration	*`baseUrl`* (registration activation link)
User activated (premoderation)	`userActivated`	N/A
Reset Password	`passwordreset`	*`baseUrl`* (reset password link)
Profile Change	`profileChanged`	N/A

Use case	Template Name	Additional Parameters
Comment Replied Mail Templates	commentReplied	*replyText*, *replyAuthorName*, *replyDate*, *commentText*, *commentDate*, *commentUrl*

Table 6.4. Mail Templates

Configuration

To enable email dispatch, the following configuration is needed:

- At least one application node needs to be configured as worker node. For more information see configuration of *taskqueues.worker-node* in the *Elastic Social* Manual. In *Blueprint*, the *elastic-worker-app* is configured as worker node.
- The application context needs to be set up with implementations of specific beans (`JavaMailSender` and `MailTemplateService`), more information is available in the *Elastic Social* Manual.
- The `mailSender` defined in *Blueprint* can be configured with the properties:

  elastic.social.mail.smtp.server, default 'localhost'

  elastic.social.mail.smtp.port, default 25

  elastic.social.mail.protocol, default 'smtp'

  elastic.social.mail.username, default '<empty>'

  elastic.social.mail.password, default '<empty>'

6.3.5 Resend Registration Confirmation Mail from *Studio*

Process: A registration confirmation mail with an activation link can be resent from *CoreMedia Studio* for users with state *REGISTRATION_REQUESTED*. The resend registration confirmation mail contains a link to the registration flow of *Blueprint*.

The following configuration is needed in a properties file of the Studio web application to use this functionality:

- `es.cae.http.host`

6.3.6 Curated transfer

Contributions can be transformed into content objects for further use. In *CoreMedia Blueprint* the `CuratedTransferExtensionPoint` must be configured to define the type of content:

```
rules: [
  ...
  Config(CuratedTransferExtensionPoint, {
    plugins: [
      Config(AddItemsPlugin, {
        items: [
          Config(Separator, {
            itemId: ElasticSocialStudioPlugin.
                    CURATED_TRANSFER_EXTENSION_POINT_SEP_FIRST_ITEM_ID
          }),
          Config(IconButton, {
            itemId: "createArticleBtn",
            scale: "medium",
            ui: ButtonSkin.WORKAREA.getSkin(),
            tooltip: ...,
            text: ...,
            iconCls: ...,
            baseAction: new OpenQuickCreateAction({
              contentType: "CMArticle",
              skipInitializers: true,
              onSuccess: CuratedUtil.postCreateArticleFromComments,
            }),
          }),
        ],
      }),
    ],
  ...
]
```

The content property can be configured in `CuratedTransferResource.java`:

```
    private static final String CONTENT_PROPERTY_TO_COPY_TO = "detailText";
```

6.3.8 reCAPTCHA

reCAPTCHA (see http://www.google.com/recaptcha for more information) is used to verify real user interaction for anonymous commenting and for registration.

Note: If reCAPTCHA is not configured, anonymous commenting is not possible!

Configure reCAPTCHA with the following settings:

- `recaptchaPublicKey`
- `recaptchaPrivateKey`

reCAPTCHA can be configured for the registration process with:

- `recaptchaForRegistrationRequired`

6.3.9 Sign Cookie

The `signCookie.privateKey` and `signCookie.publicKey` properties are used to configure a RSA key pair that is used to recognize returning unknown visitors via a signed token. The token is created and verified by `com.coremedia.blueprint.elastic.social.cae.guid.GuidCookieHandler`.

Private and public key must be set via Spring Settings for all deployed blueprint CAE instances.

The recommended way to create a key pair is to use external tools like OpenSSL. To generate a key pair with OpenSSL follow these steps:

- Generate Private Key: `openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out private-key.pem`

  Show Private Key: `openssl pkey -in private-key.pem -text`

  Generate corresponding Public Key: `openssl pkey -in private-key.pem -out public-key.pem -pubout`

  Show Public Key: `openssl pkey -in public-key.pem -pubin -text`

Another possible way to generate a key pair is to create a custom JVM based tool using the Java standard library classes `java.security.KeyFactory` and `java.security.KeyPairGenerator`.

A less recommended way is to generate the key pair via CAE code or application:

- Execute the main method of the class RSAKeyPair, the newly generated RSA key pair is logged on level WARN.
- Use the actuator endpoint of the CAE: /actuator/rsakeypair

A newly generated RSA key pair must be added to the application properties of all existing CAEs.

If no or invalid key values are configured, the application generates temporary keys that are not persisted and only valid for the current CAE application.

6.4 Third-Party Integration

CoreMedia Blueprint comes with default integrations of third-party software.

CAUTION

CoreMedia ships various third-party tool example integrations - however any licensing or privacy compliance for the use of these tools remains the responsibility of the customer implementing and operating the product. Please review with your legal counsel.

6.4.1 Open Street Map Integration

NOTE

This extension is discontinued since 1907.1. If you want to embed Open Street Map on your website, please see https://wiki.openstreetmap.org/wiki/Using_OpenStreetMap

The Open Street Map project creates and distributes free geographic data. *CoreMedia Blueprint* is prepared to include the project to display the location of location based taxonomies, but map integration are not included in the default templates.

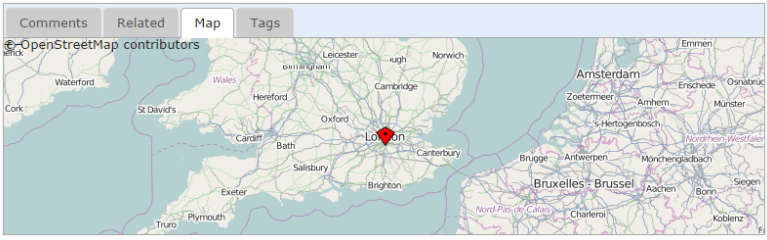

Figure 6.23. Example for an Open Street Map integration in a website

In order to use Open Street Map on your site, you have to create a settings content item and link it to the root channel of your site. The JavaScript for Open Street Map will be loaded using an aspect that is only enabled if the corresponding settings property is set. The available settings for Open Street Map are shown in the table below and must be configured to enable the map in the CAE. A

template renders a map segment according to geographic coordinates stored in the string property `latitudeLongitude` of a linked location content, and pinpoints the matching location with a marker image (see `CMTeasable.map.ftl` for a usage example).

Setting	Struct Type	Mandatory	Description
detail.show.map	Boolean Property	not	If true, the Open Street Map aspect will be enabled.
map.zoom	Int Property	no	The map zoom factor to use.

Table 6.5. Settings for Open Street Map Integration

6.4.2 Personalization Hub

The CoreMedia Personalization Hub allows to connect third-party personalization solutions with CoreMedia CMCC. These solutions decide on the user client software (mostly the browser) which content to show. The rules are defined in a personalization system.

Personalization consists of a core extension (`p13n-core`) and multiple adapter extensions. While the `p13n-core` is mandatory, the adapter is chosen for the personalization system which is integrated. This mentioned, the following extensions currently exist:

- The `p13n-core` extension which offers the basis functionality used by the adapter extensions. This extension must always be installed.
- The `p13n-adapter-monetate` extension which connects with Monetate.
- The `p13n-adapter-generic` extension which connects with Evergage and Dynamic Yield.

You must always install `p13n-core` as the base extension required by the other ones.

Based on the capability of the provider the following use cases are supported:

- Optimization and Testing: Run experiences with split traffic allocation (A/B/n tests) and targeting
- Personalization: Run experiences with machine learning and targeting
- Segmentation: Use targeting capabilities of the provider to serve custom content experiences to different user segments

For all these use cases, content editors in *CoreMedia Studio* can easily create and manage content and connect them to the provider for optimized and personalized delivery.

`p13n-core` Extension

The CoreMedia `p13n-core` extension provides the base functionality for integrating third party web personalization, optimization, and testing providers with CoreMedia.

`p13n-adapter-monetate` Extension

Monetate is a well known provider of personalization, targeting and optimization solutions for websites.

The `p13n-adapter-monetate` extension for *CoreMedia Blueprint* enables you to use Monetate's technology to personalize, target and test content in an easy way to build a whole new customer experience.

`p13n-adapter-generic` Extension

The `p13n-adapter-generic` adapter is a generic means to connect third party web personalization, optimization and targeting solutions that don't provide dedicated client APIs to query metadata state.

The current implementation has been tested with the following systems:

- Dynamic Yield
- Evergage

Architecture

For all supported personalization providers, a client-side integration approach is used:

The provider's JavaScript tag is rendered into the head of the generated HTML output. The script tag loads and calls additional JavaScript code which evaluates the current request and determines the actions to run on this page, for example which variant to show from what experience. When actions related to CoreMedia experiences are run, custom CoreMedia JavaScript callbacks are triggered, loading personalized content items from the *CAE* via AJAX, replacing default content items on the page. Using this approach offers two main benefits:

In addition to CMS content actions, all capabilities of the provider can be used for use cases that are currently not supported by CoreMedia. The generated CoreMedia pages are not dependent on the personalization actions and can be cached.

Content Placement and Rendering Restrictions

Personalization bases on a few assumptions that affect the way you can place content and render content:

- Experiences and segments target single teasable items only - not collections. This means it is not possible to replace an item with a collection of items and vice versa.
- A baseline item must always exist, therefore, it is not possible to add content to the variants/segments while the baseline remains empty. Moreover, for all variants/segments a content item must be defined, which means it is not possible to add a baseline content item and leave some or all the variant/segment content empty.
- The design of segments follows the idea that a user is part of only one segment. There is no ruleset to combine different segments to target content.
- Different experiences/segmentation documents cannot target the same baseline slot.

6.4.2.1 Installing Personalization Hub

Using CoreMedia Personalization requires the installation of at least two extensions (see Section 4.1.5, “Project Extensions” [77] for more details about extensions):

- The `p13n-core` extension at https://github.com/coremedia-contributions/p13n-core.
- At least, one of the adapters `p13n-adapter-monetate` (https://github.com/coremedia-contributions/p13n-adapter-monetate) or `p13n-adapter-generic` (https://github.com/coremedia-contributions/p13n-adapter-generic) to connect with a third-party system.

You will find the extensions in the following repositories:

- https://github.com/coremedia-contributions/p13n-core

  The core extensions that must be installed.
- https://github.com/coremedia-contributions/p13n-adapter-monetate

This adapter connects to Monetate and provides a direct integration using the Kibo metadata API.

- https://github.com/coremedia-contributions/p13n-adapter-generic

  This adapter provides a generic integration base for providers without a client API. Experiences must be mirrored in the CoreMedia system with a special content type. Built-in support is included for Dynamic Yield and Evergage

You can either add the extension repositories as Git submodules, the recommended approach, or copy them into your Blueprint workspace in the `modules/extensions` folder. Copying the extensions requires more work when upgrading the Blueprint workspace to newer releases. In the following section you will learn how to add them as submodules.

Adding Extension as Submodule

If you plan to customize the extensions, first create a fork of the repositories and add your forks as submodules. Otherwise, you can simply add the CoreMedia repositories as submodules as shown in the following example.

Add the core and adapter extensions to the Blueprint workspace as Git submodules as follows (this example uses the `p13n-adapter-monetate` adapter). To do this, open a terminal window and run the following commands:

```
cd /<blueprint-root-dir>

mkdir -p modules/extensions

git submodule add https://github.com/coremedia-contributions/p13n-core.git
modules/extensions/p13n-core

git submodule add
https://github.com/coremedia-contributions/p13n-adapter-monetate.git
modules/extensions/p13n-adapter-monetate

git submodule init
```

Example 6.10. Adding submodules

After the submodules are added, go to each submodule directory and checkout the branch matching your Blueprint version.

```
cd /<blueprint-root-dir>/modules/extensions/p13n-core
git checkout -b <branch-name>
cd ../p13n-adapter-monetate
git checkout <branch-name>
```

Example 6.11. Checkout branch in submodule

Then commit the changes to the submodules:

```
cd /<blueprint-root-dir>/modules/extensions
git add p13n-core
git add p13n-adapter-monetate
git commit -m 'Add personalization submodules'
```

Example 6.12. Commit changes to submodules

Now, you have to activate the extensions.

Activating the Extensions

Run the extensions tool (see Section 4.1.5, "Project Extensions" [77] for more details about the extensions tool) in `workspace-configuration/extensions` to activate the extensions like this (here, the core and the `p13n-adapter-monetate` extensions are activated):

```
mvn extensions:sync
mvn extensions:sync -Denable=p13n-core,p13n-adapter-monetate
```

Example 6.13. Activate extensions

Now you are done with the installation and activation of the extensions. In order to work with the extensions, you have to configure them, as described in the next sections.

Updating the Extensions

If you are upgrading your Blueprint workspace to another major release, an update of the personalization extension may be required. Check the extension's repositories for the available branches and select the branch matching your Blueprint release version or the highest release version lower than your Blueprint version.

Update the extension's submodules by opening a terminal window and running the following commands (the example shows the steps for the `p13n-adapter-monetate` adapter):

```
cd /<blueprint-root-dir>/modules/extensions
cd p13n-core
git fetch
git checkout <branch-name>
cd ..
git add p13n-core

cd p13n-adapter-monetate
git fetch
git checkout <branch-name>
cd ..
git add p13n-adapter-monetate/
```

```
git commit -m 'Update personalization submodules to release xxxx.x'
```

Example 6.14. Updating an extension

6.4.2.2 Personalization Hub Configuration and Operation

This section describes, how you configure the Personalization Hub. Getting the Personalization Hub working contains the following steps:

1. Install the `p13n-core` extension and at least one adapter extension. See Section 6.4.2.1, "Installing Personalization Hub" [381] for details.
2. Configure the `p13n-core` extension which is the basis for the adapter extensions. See Section "Configuring the p13n-core Extension" [384] for details.
3. Configure the installed adapter extensions. See Section "Monetate - p13n-adapter-monetate Extension" [397] and Section "Evergage & Dynamic Yield - p13n-adapter-generic Extension" [386] for details.

Configuring the p13n-core Extension

The `p13n-core` extension adds, for example, the following features:

- Content types
- API definition
- Studio forms and preview
- Studio backend implementation
- *CAE* core logic

In order to use the `p13n-core` extension you have to extend your frontend theme with a brick

Frontend Integration

The frontend theme can be extended with a brick that acts on the callbacks received from the personalization provider. The brick is responsible for loading variant fragments and displaying the baseline content if no variant is triggered. The provider's script tag is independently included in the adapter *CAE* extensions.

Together with a small piece of JavaScript proxy code injected into the HTML head by the *CAE* extension, the brick provides the callback functions called from the third-party service if a variant or segment is to be displayed and replaces the default content fragment with the variant loaded via AJAX.

The frontend code adds the object `cm_p13n` to the `window` object providing the following methods:

- `pushVariant(variantId:String)`

  Activates the variant with the given ID.
- `pushSegment(segmentName:String)`

  Activates the segment with the given name.
- `exchangeVariant(variantId:String, contentId:String)`

  Replaces a variant when using the preview icon of a variant in the content item tab.
- `completed(providerId)`

  Called when the third-party service is finished and displays the baseline for all fragments where no variant or segment is active.

In addition, an event listener is installed for integration with the Studio preview.

Adding the brick to the theme

Add the brick `p13n-dynamic-include` as dependency to your theme (`package.json`).

```
pnpm add @coremedia/brick-p13n-dynamic-include
```

Fragment Caching

The personalization resolves everything in the client's browser. Thus, the fragments included via AJAX do not require any user specific server side processing and are cacheable for a limited time.

CAE Caching

To enable the required HTTP caching headers add the following line to the live *CAE*'s `application.properties` file and adjust the cache time to your need:

```
cae.cache-control.for-url-pattern[/dynamic/fragment/experience/item/**].max-age=5me
```

Caching in a commerce-led Scenario

If you are running personalization in a commerce-led scenario, the cache headers from the *CAE* might not be passed through the eCommerce delivery system. To counter this, add static rules to your CDN, allowing caching and storing of fragments with the path `<prefix>/dynamic/fragment/experience/item/**` on the client's side.

Evergage & Dynamic Yield - p13n-adapter-generic Extension

The generic adapter allows you to connect with Evergage and Dynamic Yield.

The generic adapter comes with an additional prerequisite. To generate IDs for the content representations of segments and experiences UUIDs must be available in the system and every segment and experience representation must have a UUID in the Content Management Server as well as in the Master Live Server. UUIDs are fully available since CoreMedia version 2210. For details on migrating and transferring content UUIDs, see the documentation page Section 3.13.2.4, "Content UUID Migration and Transfer" in *Content Server Manual*.

Configuring the `p13n-adapter-generic` requires the following steps:

- Enabling Evergage or Dynamic Yield Connection in Studio (see Section "Connecting Evergage and Dynamic Yield with Studio" [386].
- Creating CMExperienceDefinition content items in Studio to mirror the experiences defined in Evergage and Dynamic Yield. See Section "Creating Experience Definitions in Studio" [388].
- Creating experiences in Evergage and Dynamic Yield. See Section "Creating Experiences for Testing or Personalization" [390] and Section "Creating Experiences for Dynamic Yield" [394].

Connecting Evergage and Dynamic Yield with Studio

For both integrations, you have to create a `Settings` content item in your site and link it from the root page of the site. The `Settings` items need to have the following properties:

Evergage Settings Content Item

The properties need to have the following values:

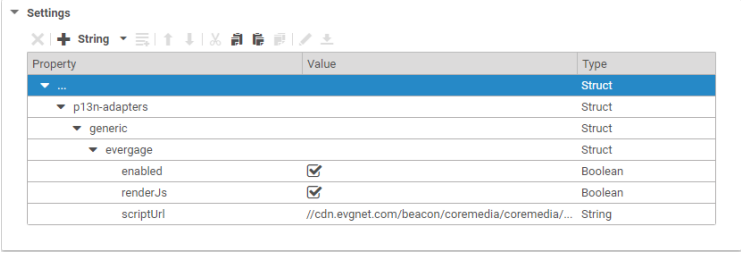

Figure 6.24. Evergage settings item

enabled Required. Enables the Evergage integration for this site.

renderJS Optional. Instructs the *CAE* to include the Evergage script link in its head section. Disable in a commerce-led scenario where the shop frontend already includes the script. Defaults to true if missing.

scriptUrl Required. The URL of the Evergage script. Can be obtained from the JavaScript integration page in the Evergage portal (**Web** → JavaScript Integration → **Synchronous**).

Dynamic Yield Settings Content Item

The properties need to have the following values:

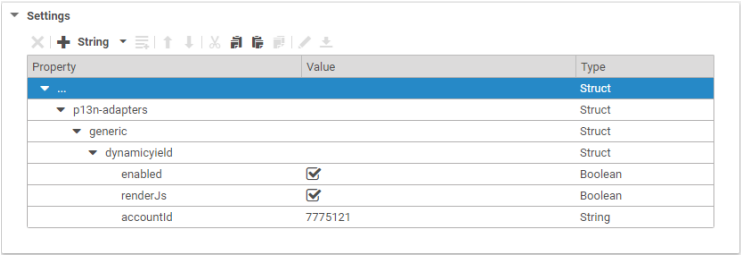

Figure 6.25. Dynamic Yield settings item

enabled Required. Enables the Dynamic Yield integration for this site.

renderJS Optional. Instructs the *CAE* to include the Dynamic Yield script in its head section. Disable in a commerce-led scenario where the shop frontend already includes the scripts. Defaults to true if missing.

scriptUrl Required. The account ID as displayed on the general settings page in the Dynamic Yield portal.

Creating Experience Definitions in Studio

In order to integrate personalization providers into Studio you need to mirror the experiences in Studio with special configuration content item of type `CMExperienceDefinition`.

1. For each site and experience, create `CMExperienceDefinition` content items in the folder `<Site Root>/Options/Personalization/Experiences/<ProviderID>`. Replace *<ProviderId>* with the key of the provider configuration in the `Settings` content item (see Section "Connecting Evergage and Dynamic Yield with Studio" [386]). By default, this is "evergage" and "dynamicyield", respectively.

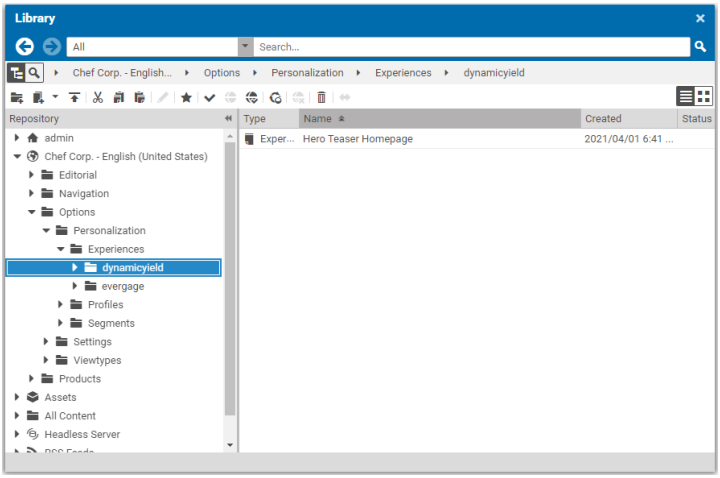

Figure 6.26. Create CMExperienceDefinitions in Studio

The name of the content item is not important, but it is recommended to use the name of the experience, defined in the personalization provider software.

2. In the content item, set a unique name and description and add all variants defined in the third-party system. The concrete name is not important, but it is recommended to use the name of the experience, defined in the personalization provider software.

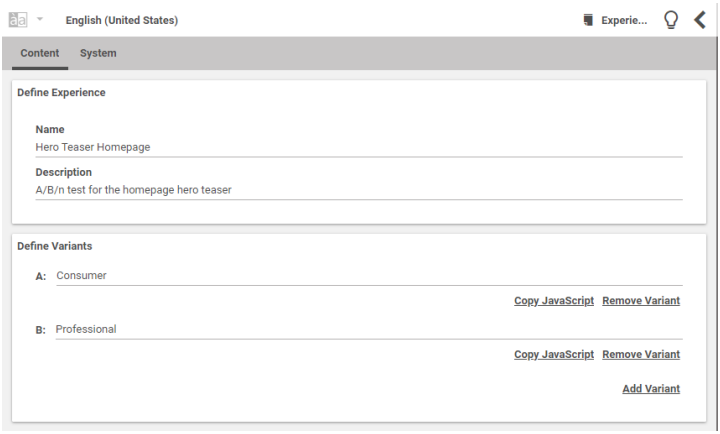

Figure 6.27. Configure experience definition in Studio

Once the definition has been created, it can be selected in experience content items.

Creating Experiences for Evergage

Evergage uses different naming conventions than the CoreMedia personalization extension and the other supported personalization providers. The following table shows the mapping of Evergage terms to CoreMedia terms:

Evergage	CoreMedia
Campaign	Experience
Experience	Variant

Table 6.6. Evergage naming

Prerequisites

- Installation of the `p13n-core` extension and the `p13n-adapter-generic` extension as described in Section 6.4.2.1, "Installing Personalization Hub" [381]
- Configuration of the `p13n-core` Extension as described in Section "Configuring the p13n-core Extension" [384].

- Configuration of the `p13n-adapter-generic` as described in Section "Connecting Evergage and Dynamic Yield with Studio" [386]

Creating Experiences for Testing or Personalization

1. Create a new web campaign in the Evergage portal. Add the required number of experiences (one for each variation) and switch to the setup menu.
2. Set the global campaign settings like user targeting, goal and metric.
3. Switch to the experiences panel. Set the test mode and traffic allocation, name the experiences, and set each experience to type *Personalize*.

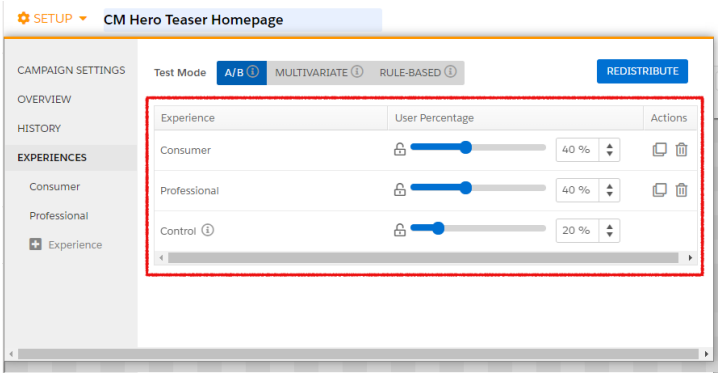

Figure 6.28. Evergage Experiences panel

4. Add the JavaScript snippets to connect the variations to CoreMedia. For this open the mirrored experience definition (see Section "Creating Experience Definitions in Studio" [388]) in *CoreMedia Studio*. For each variation copy the Javascript snippet to the clipboard, change back to the Experience JavaScript tab and paste the JavaScript.

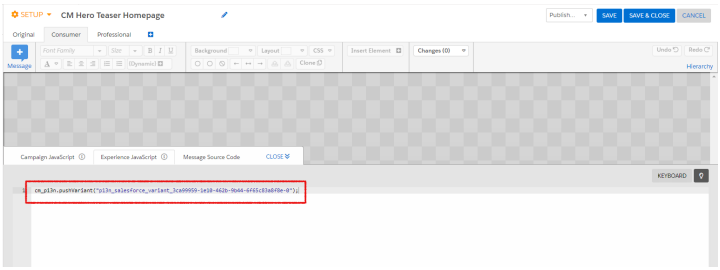

Figure 6.29. Adding JavaScript code to the variants

Creating Experiences for Segmentation

The segmentation use case allocates the customer base to a set of segments.

Creating Segments

For the segmentation use case you have to create an `External Segment` content item for each segment you want to use in the `<Site Root>/Options/Personalization/Segments/evergage` folder of the site. For the content name use lower- and upper-case letters, numbers, minus and underscore, no special characters are allowed.

1. Create a new web campaign and add an experience for each segment.
2. Open the setup menu and switch to the experiences panel.
3. Set the test mode to *Rule-Based* and set traffic allocation for *Control* to 0%. Afterwards edit each experience: Set its name and add a targeting rule with the fitting segment.

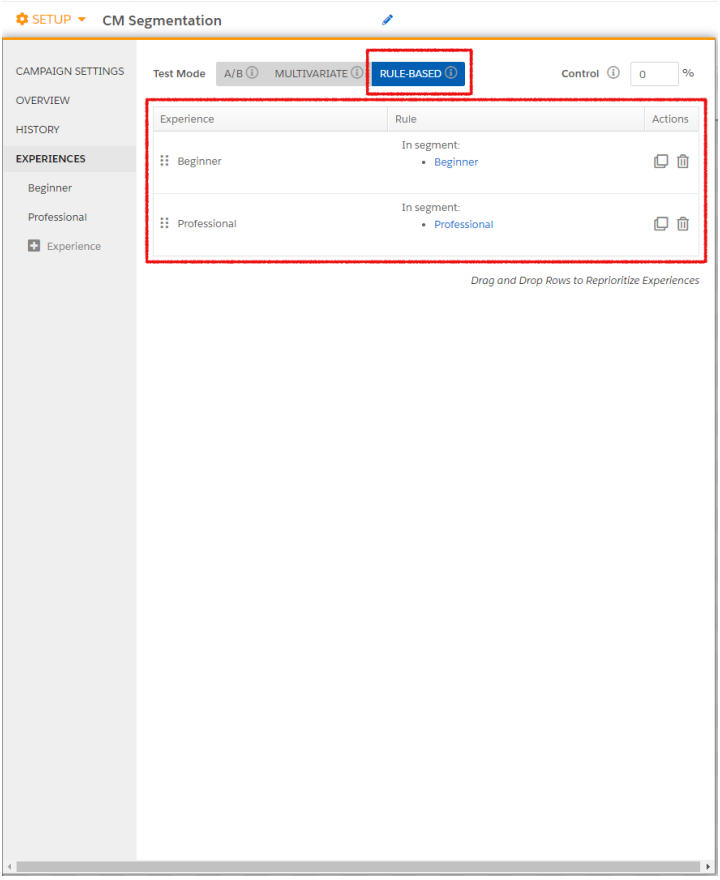

Figure 6.30. Defining Segment experiences in Evergage

4. Add the JavaScript code for connecting the segments to CoreMedia. Add the following code for each experience, substituting the parameter *segment_name* with the name defined in CoreMedia (see Section "Creating Experiences for Segmentation" [391]).

```
cm_p13n.pushSegment("<segment_name>");
```

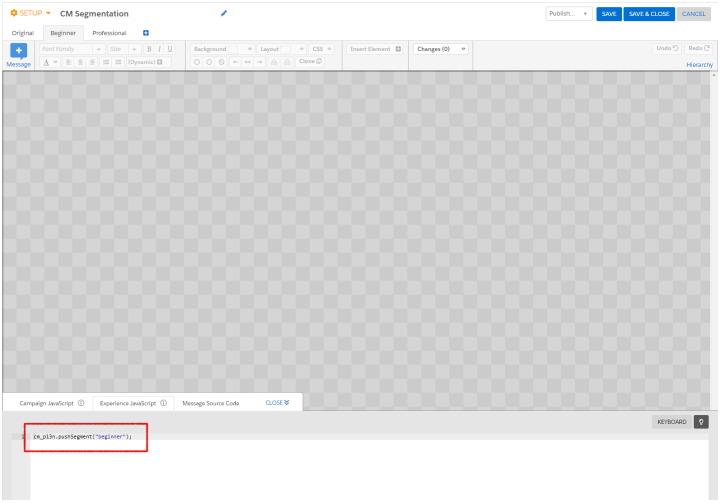

Figure 6.31. Evergage add JavaScript to experience

Creating Experience for Content Masking

The CoreMedia JavaScript frontend integration code - triggered by the Evergage JavaScript - dynamically loads content via AJAX from the *CAE* backend and injects it into the already displayed page. To minimize visually disturbing effects, such as elements flickering, popping up or moving around, during these page updates it is essential for the CoreMedia code to get notified when Evergage has finished processing all decisions. Therefore, a special campaign must be added, which must always be executed last by giving it the lowest priority:

1. Create a new campaign, named CM Final.
2. Set its priority to a lower value than all the other CoreMedia campaigns.
3. Set its *Test Mode* to **A/B**.
4. Add a single experience named Final and allocate 100% traffic to it.
5. Set the experience's JavaScript code to: `cm_p13n.completed('evergage');`.

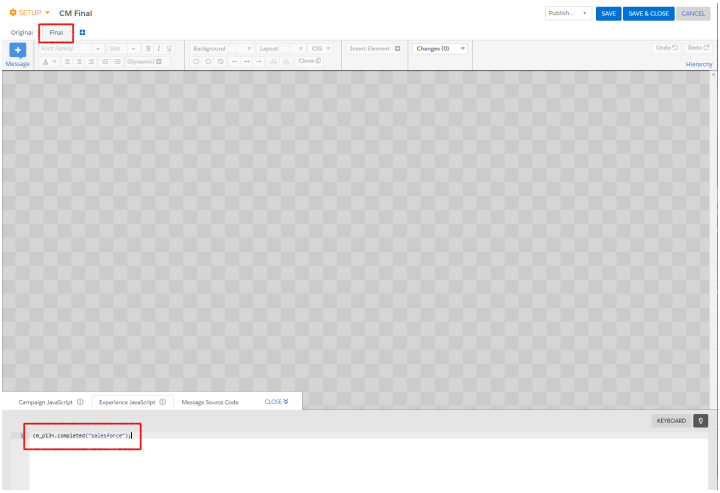

Figure 6.32. Create final experience for Evergage

NOTE

Do not set any targeting or other rules, the experience must run on every page load.

Creating Experiences for Dynamic Yield

Creating Experiences for Testing or Personalization

1. In the Dynamic Yield portal create a new Custom Code campaign using the following settings:
   - *Trigger*: Page Load
   - *Frequency*: Once per pageview
2. Add a single experience.

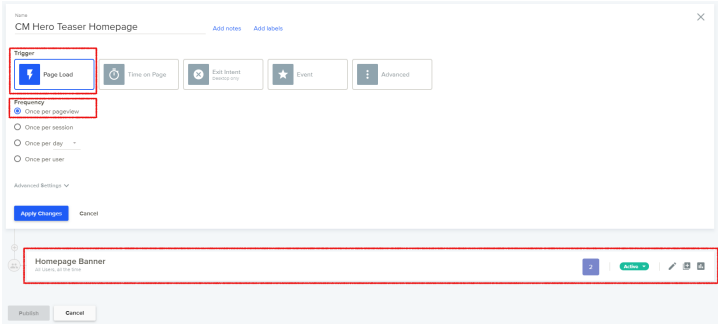

Figure 6.33. Create campaign in Dynamic Yield

3. Open the experience for editing and add the desired targeting. Switch to the *Variations* tab. Choose static or dynamic traffic allocation and select a primary metric. Then add the variations and choose their individual traffic allocation.

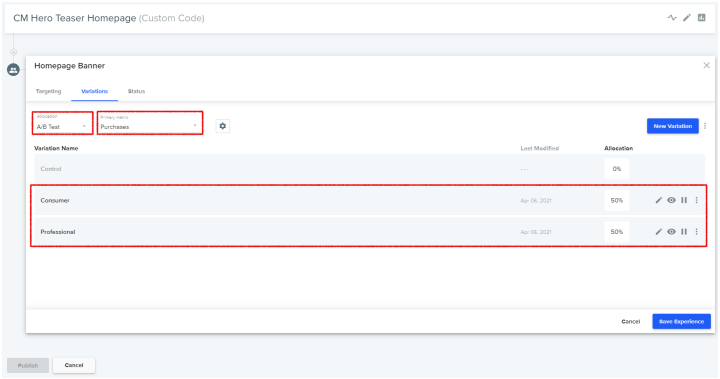

Figure 6.34. Edit Dynamic Yield experience

4. Add the JavaScript code to connect the variations to *CoreMedia Content Cloud*. Open the mirrored experience definitions in *CoreMedia Studio* (see Section "Creating Experience Definitions in Studio" [388]). For each variation copy the Javascript snippet to the clipboard, change back to the variation's JavaScript tab and paste the code. Save the variation.

Creating Experiences for Segmentation

The segmentation use case allocates the customer base to a set of segments.

1. In the Dynamic Yield portal create a new *Custom Code* campaign using the following settings:

- *Trigger*: Page Load
- *Frequency*: Once per pageview

2. For each segment add a new experience. If a user can be a member of multiple segments, be sure to prioritize the segments by ordering them accordingly.

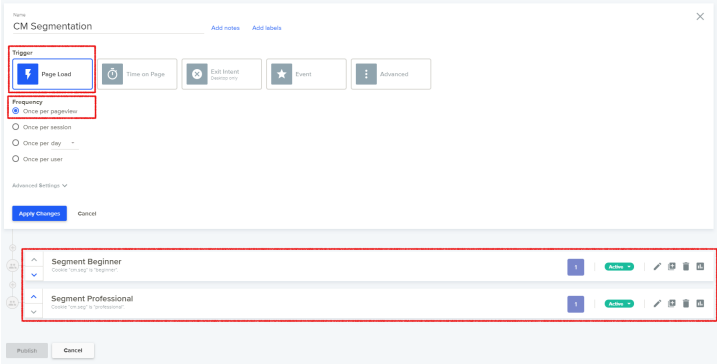

Figure 6.35. Creating campaign for Dynamic Yield segmentation

3. Set up the targeting for each experience and set the traffic *Allocation* to **`A/B Test`**. Add a single variation with 100% traffic *Allocation*.

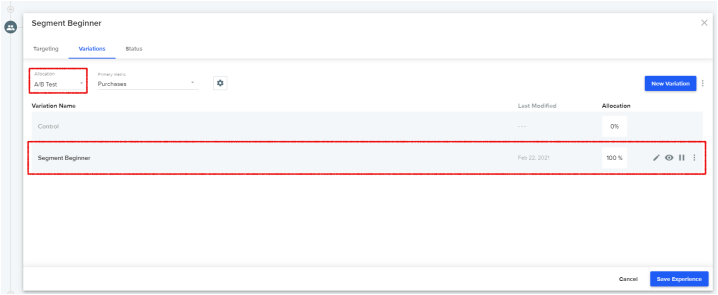

Figure 6.36. Dynamic Yield configure experience for segmentation

4. Add the JavaScript code for connecting the segment to CoreMedia. Add the following code, substituting the parameter *`segment_name`* with the name defined in CoreMedia (see Section "Creating Experiences for Segmentation" [391]).

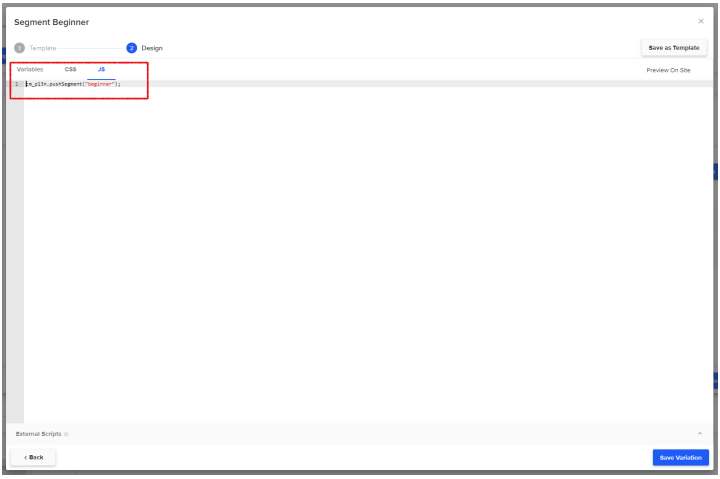

Figure 6.37. Dynamic Yield add JavaScript to segment

Monetate - p13n-adapter-monetate Extension

The `p13n-adapter-monetate` extension connects the CoreMedia system with the Monetate personalization platform.

The monetate adapter comes with an additional prerequisite. To generate IDs for the content representations of segments UUIDs must be available in the system and every segment representation must have a UUID in the Content Management Server as well as in the Master Live Server. UUIDs are fully available since CoreMedia version 2210. For details on migrating and transferring content UUIDs, see the documentation page Section 3.13.2.4, "Content UUID Migration and Transfer" in *Content Server Manual*.

Configuring the `p13n-adapter-monetate` extension requires the following steps:

1. Enabling the Monetate connection in Studio (see Section "Connecting *CoreMedia Content Cloud* with Monetate" [398] for details).
2. Creating experiences for different use cases (see Section "Creating Experiences in Monetate" [400] for details).

Connecting *CoreMedia Content Cloud* with Monetate

To connect *CoreMedia Content Cloud* with the Monetate system, you have to connect the *CAE* and *Studio Server* with Monetate and configure each site that should use the Monetate connection.

Configuring the Connection

To connect *CoreMedia Content Cloud* with the Monetate system, proceed as follows:

1. Create an RSA key pair, for example, with `openssl`.
2. Open the Monetate portal and navigate to **Account Settings** → Sites → **APIKeys**. and create a new API user.
3. Add the newly generated public key to the API user.
4. Configure the connection in the `monetate-connector.properties` file in `p13n-adapter-monetate-lib/src/main/resources/META-INF`. There, you will find a description of the properties. An example configuration would look as follows:

```
p13n.adapters.monetate.connectors[0].id=coremedia
p13n.adapters.monetate.connectors[0].retailer=coremedia
p13n.adapters.monetate.connectors[0].key-string=MIIBrzBJBgkqhkiG...
p13n.adapters.monetate.connectors[0].api-user=api-0815-connector
p13n.adapters.monetate.connectors[0].tag-uri-template=//se.monetate.net/js/2/{channel}/entry.js
p13n.adapters.monetate.connectors[0].analytics-uri-template=https://marketer.monetate.net/
            control/{channel}/analytics/{experienceId}/realtime
p13n.adapters.monetate.connectors[0].edit-uri-template=https://marketer.monetate.net/control/
          {channel}/experience/{experienceId}
p13n.adapters.monetate.connectors[0].filterExpressions[0]=(.*)\\$
p13n.adapters.monetate.connectors[0].filterExpressions[1]=(.*)#
p13n.adapters.monetate.connectors[0].filterMode=exclude
```

Example 6.15. Example Monetate configuration

Excluding Experiences from Studio

In Section “Creating Experiences in Monetate” [400] you will learn how to create experiences in Monetate. However, not all experiences in the Monetate system are applicable for use with a CoreMedia Monetate Experience content item, for example, experiences used for segmentation, the 'Final' experience (see Section “Creating Experience for Content Masking” [406]), which only serves a technical function, and experiences that don't target content (changing layout, for example). These experiences should be hidden from the Studio user and be not selectable in the experience drop-down list.

For this use case, you can apply the properties `monetate.connectors[n].filterMode` and `monetate.connectors[n].filterExpressions[n]` as shown in section "Configuring the Connection" [398]

Configuring the Site

To connect a site with the Monetate system, you have to create a `Settings` content item and link to it from the root page of the site. In the `Settings` content item set the following properties:

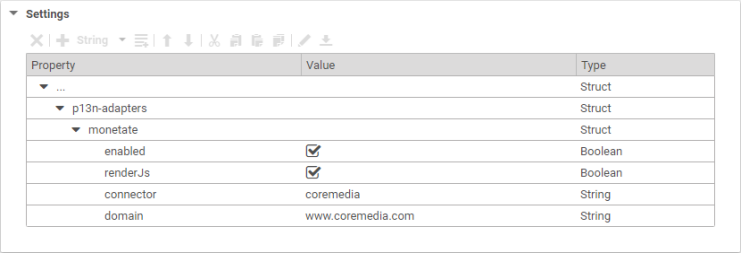

Figure 6.38. Properties for Monetate connection

The properties have the following meaning:

Property	Required	Default	Description
enabled	x		Enables the Monetate integration for this site.
renderJS		true	Instructs the *CAE* to include the Monetate tag in its head section. Disable in a commerce-led scenario where the shop frontend already includes the Monetate scripts.
connector	x		The ID of the backend connector. See the property `p13n.adapters.monetate.connectors[0].id` in section "Configuring the Connection" [398].
domain	x		The domain name. Corresponds to a site in your Monetate account.

Table 6.7. Monetate properties for site connection

Creating Experiences in Monetate

In order to use Monetate experiences and segments in *CoreMedia Studio*, you need to define experiences that use the CoreMedia JavaScript action.

You can define three types of experiences:

- Experiences for personalization and testing (see Section "Creating Experiences for Personalization and Testing" [402]).
- Experiences for segmentation (see Section "Creating Experiences for Segmentation" [404])
- A technically required experience to minimize visually disturbing effects in the website (see Section "Creating Experience for Content Masking" [406])

Prerequisites

- Installation of the `p13n-core` extension and the `p13n-adapter-monetate` extension as described in Section 6.4.2.1, "Installing Personalization Hub" [381]
- Configuration of the `p13n-core` Extension as described in Section "Configuring the p13n-core Extension" [384].
- Set up of the Monetate integration as described in Section "Connecting *CoreMedia Content Cloud* with Monetate" [398]

Setting Up a JavaScript Action

In order to use experiences from Monetate in CoreMedia Sites you first have to create an action in Monetate.

1. Go into the Monetate Action Builder, select **Components** → **Actions** and click **[Create Action]** . The live website opens up and the Monetate overlay is enabled:

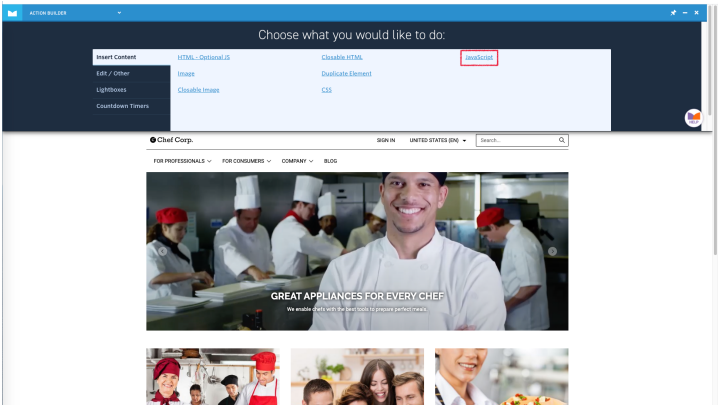

Figure 6.39. Create Action in Monetate

2. Select *JavaScript* and in the following screen set *Only run once?* (1) to *Yes*. Click the arrow (2) to go to the next screen.

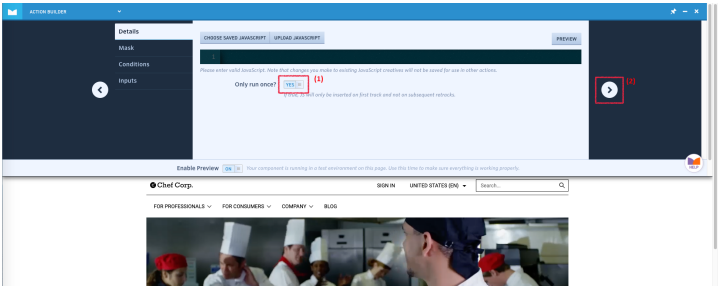

Figure 6.40. Monetate action screen

3. Fill the form fields as shown in the screenshot and click **[Create & Exit]** to create the action and to return to the Monetate portal.

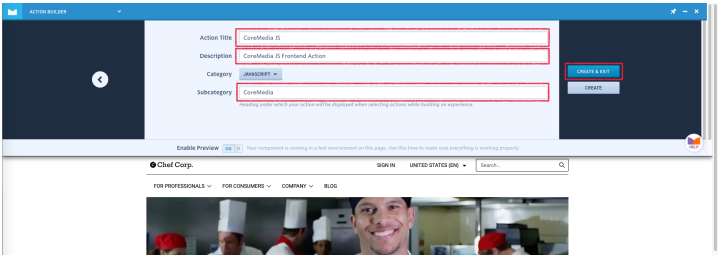

Figure 6.41. Monetate enter data for new action

Creating Experiences for Personalization and Testing

1. Follow the default steps for creating a Monetate experience. You can select any type of goal (WHY). Apply targeting rules as required (WHO). Then create the variants (WHAT). For each variant add a single action — the CoreMedia JavaScript action created in Section "Setting Up a JavaScript Action" [400].

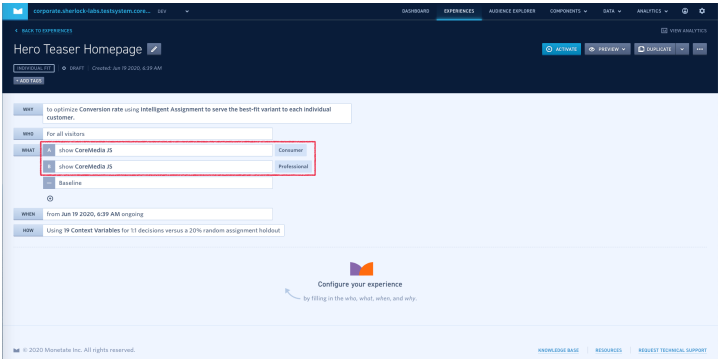

Figure 6.42. Create experience in Monetate

2. Configure each action with the required JavaScript code using one of the following two methods:
   - Look at the URL in your browser. It should look similar to `https://marketer.monetate.net/control/<ACCOUNT>/p/<DOMAIN>/experience/1275940#c1517122:what,a3489594`. Note the number after #c, in this example 1517122. Copy `cm_p13n.pushVariant("<NUMBER>")` to the clipboard where <NUMBER> is replaced by the number above.
   - Change to *CoreMedia Studio*, select your preferred site and open the dashboard. If the dashboard does not contain the experience widget, add

it to the dashboard. Select the newly created experience from the widget's list and open it in a new tab (if the experience is missing, use the reload button to update the list and check the widget's filter settings). In the variants panel each variant has a link button to copy the required JavaScript code the clipboard:

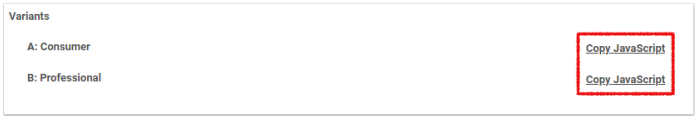

Figure 6.43. Copy JavaScript code from Studio

Click the **[Copy JavaScript]** button.

Go back to the Monetate portal, select the variant action and paste the copied code into the JavaScript code field.

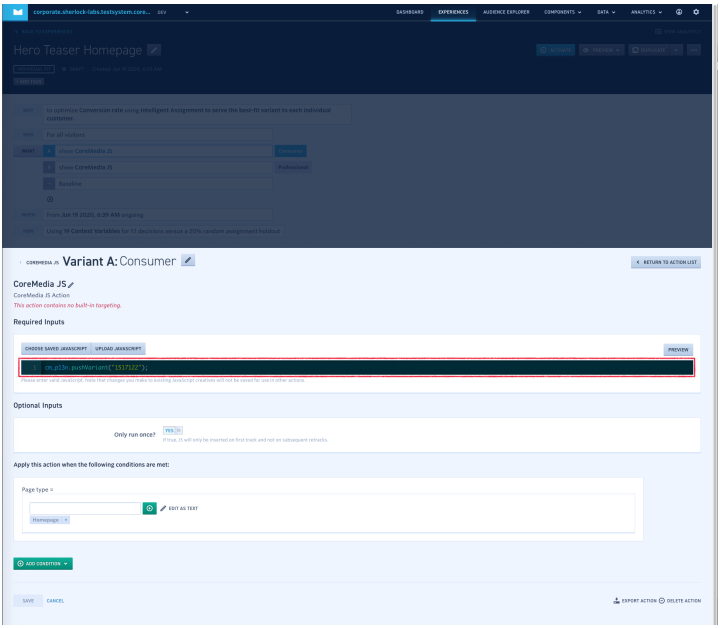

Figure 6.44. Insert JavaScript code into Monetate action

3. If the experience should not be placed on every page, add conditions restricting the action to a single page or set of pages using the **[Add Condition]** button.

Creating Experiences for Segmentation

The supported segmentation use case divides the customer base into disjoint sets of segments. For each segment, you want to use, you have to create a `P13NSegment` in *CoreMedia Studio*. You have two ways of applying an experience to a segment:

- Each segment can be used as a direct target rule of an experience.
- The Monetate AI decides about the segment with a single individual fit experience.

For the first case you create an experience for each segment, while for the second case you only create one experience with all segments.

Create Segments in *CoreMedia Studio*

For the segmentation use case you have to create an `External Segment` content item for each segment you want to use in the `<Site Root>/Options/Personalization/Segments/monetate` folder of the site. For the content name use lower- and upper-case letters, numbers, minus and underscore, no special characters are allowed.

Direct Target Rule

1. Define an experience with the goal *to serve everyone the same experience*. Apply the required targeting rules and add a single variant with the CoreMedia JavaScript action.

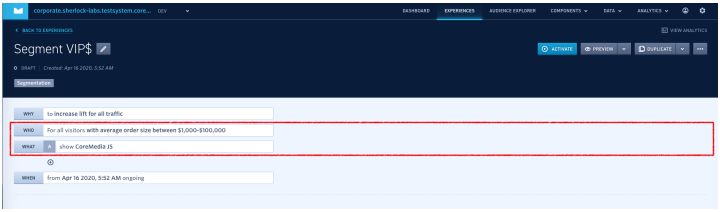

Figure 6.45. Define Monetate experience

2. Select the variant action and paste the following code into the JavaScript code field:

```
cm_p13n.pushSegment("<segment_name>");
```

Replace **<segment_name>** with the name of the corresponding Studio content item, which you have created in section "Create Segments in *Core-Media Studio*" [404].

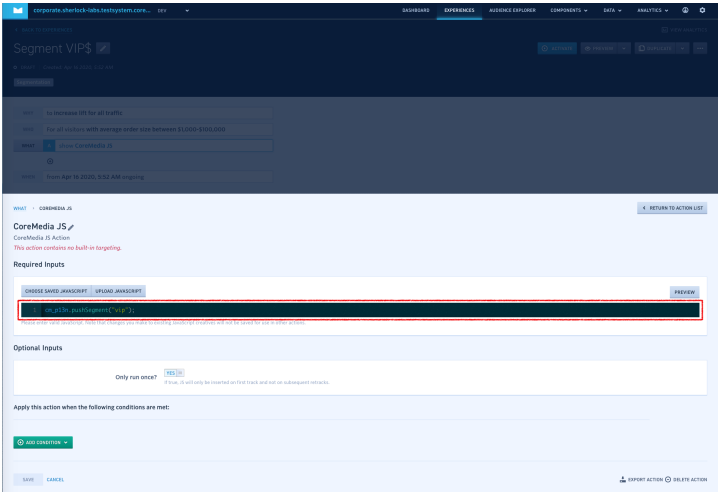

Figure 6.46. Add code to variants

3. Continue with experiences for other segments.

AI Optimized Content Selection

1. Create a single experience with goal *To optimize my goal with Machine Learning. Individual Fit.* For each segment add a variant.

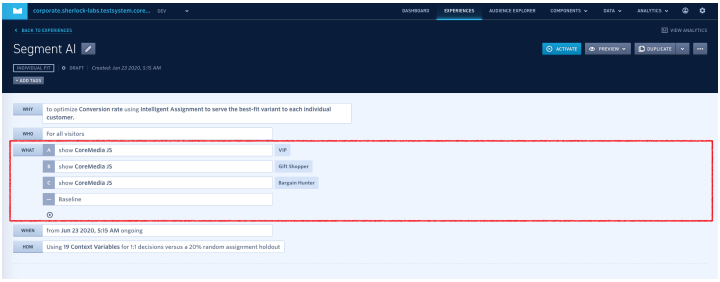

Figure 6.47. Add an experience for AI powered segmentation

For each variant add a single CoreMedia action and paste the following code into the JavaScript code field:

```
cm_p13n.pushSegment("<segment_name>");
```

Replace **<segment_name>** with the name of the corresponding Studio content item, which you have created in section "Create Segments in *CoreMedia Studio*" [404].

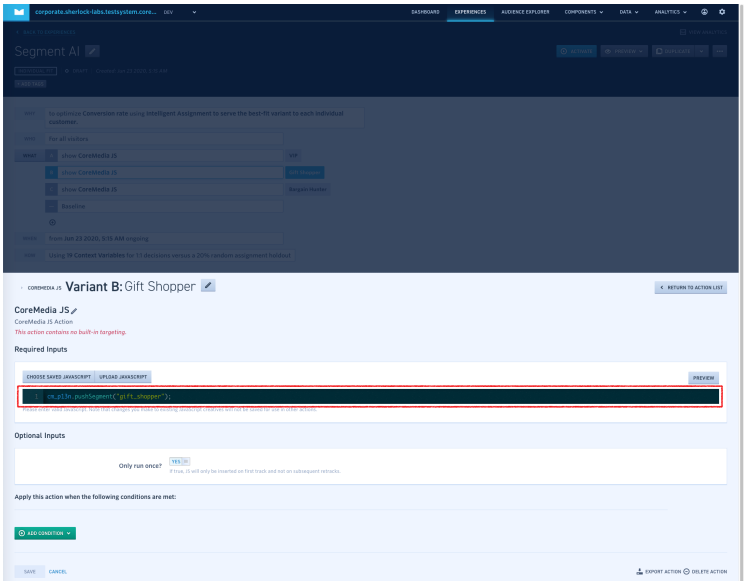

Figure 6.48. Experience for AI powered content selection

Creating Experience for Content Masking

The CoreMedia JavaScript frontend integration code - triggered by the Monetate Tag - dynamically loads content from the *CAE* backend via AJAX and injects it into the already displayed page. To minimize visually disturbing effects, such as elements flickering, popping up or moving around, during these page updates it is essential for the CoreMedia code to get notified when Monetate has finished processing all decisions. Hence, a special experience needs to be added, which must always be executed last by placing it at the end of the list of experiences:

1. Create a new experience, for example named *Final*.
2. Select *to serve everyone the same experience* as the goal. Don't add any targeting, let it apply to all visitors.

3. Add the CoreMedia JavaScript action to its single variant and insert the following code: `cm_p13n.completed('monetate');`.
4. Save and return to the experience overview.
5. Place the experience at the bottom of the list by giving it the highest priority value.

Figure 6.49. Place Final experience

6.4.3 Analytics Connectors Integration

CoreMedia Analytics Connectors demonstrates how to integrate third-party analytics services into *CoreMedia CMS*.

Brand Blueprint integrates the service **Google Analytics** into the website.

The integration extends the delivery side (*CAE*) with tracking of page impressions and *CoreMedia Studio* with a configurable link to the analytics provider in the main side bar.

If your project is based on *CoreMedia Blueprint*, you will be able to use the integration out of the box. Otherwise, it serves as an example of how to integrate external analytics services into your CoreMedia project.

6.4.3.1 Tracking

Tracking user actions on a website is typically implemented by adding calls to vendor specific JavaScript functions to the pages of the site. These functions populate a data structure which is eventually sent to the analytics service via an HTTP request for an invisible image, also known as tracking pixel.

The data being sent to the analytics service includes data about the following topics:

- Content being displayed on the page

In addition, each analytics service requires some way of identifying the account the incoming data is to be written to.

The tracking configuration is stored in the content repository within the `settings` property of navigation content items and can be inherited.

The configuration options for tracking are described further down in Section 6.4.3.5, "Studio Integration" [410].

6.4.3.2 Assembling Tracking Information

CoreMedia Analytics Connectors provides a `ViewHookEventListener` and an `AnalyticsProvider` implementation for each integrated third-party service provider. The `ViewHookEventListener` reacts on a `ViewHookEvent` of type `head` for content of type `Page`. If sufficient configuration is available, it renders the corresponding provider's JavasSript into the `head` section of the `Page`. The provider specific `AnalyticsProvider` implementation provides access to the basic configuration that is necessary to establish a tracking connection to that particular service. Most importantly, the `AnalyticsProvider` implementation can check if any required properties are missing and suppresses rendering of any output for that service if it is not properly configured.

Note that the integration of analytics extensions is controlled by the CoreMedia Extension Tool. Per default, all analytics extensions are active but can be turned off by using this tool (see Section 4.1.5, "Project Extensions" [77]).

See the Javadoc for more details on which properties the analytics listeners and interceptors provide for tracking and Section 6.4.3.5, "Studio Integration" [410] for details on how to configure them.

Consult the Content Application Developer Manual for information on how to register interceptors and `ViewHookEventListener`s.

6.4.3.3 Views

As explained in the previous section, *CAE* `ViewHookEventListener`s are used to make data to be tracked (and the tracking configuration itself) accessible when rendering a view. This data is used in views to build the tracking calls. *CAE* `ViewHookEventListener`s react on rendering of `com.coremedia.blueprint.common.contentbeans.Page` beans to add code into the head of a page. The analytics integration uses the `head` view that serves the following purposes:

- Includes third-party tracking libraries
- Sets up vendor-specific JavaScript data structures required for tracking
- Includes the asynchronous calls to the JavaScript tracking code

`GoogleAnalytics.head.ftl` includes JavaScript libraries specific for an external service and converts the tracking data into JavaScript objects used by the final tracking calls. The FTL checks if the service provider that it implements is enabled for the current page, that is, if it is properly configured and not explicitly disabled.

Page view tracking calls use the property `enabled` to check whether the third-party service is enabled. Setting this property explicitly to "false", disables the service provider, while setting it to "true" will only enable further processing of the provider's configuration (which might ultimately enable that service if the configuration is complete).

6.4.3.4 JavaScript Code

Tracking is performed by calling vendor-specific JavaScript functions. *CoreMedia Analytics Connectors* offers a thin layer around these.

Page view tracking calls are fired when a page is loaded. Corresponding calls are either included by the vendor specific JavaScript files to include, or have to be included in the implementation's `asHead` FTL of its analytics implementation.

The following subsections describe the JavaScript variables that are set in the `head` views of the FTLs described in the previous section.

Google Analytics

The Google Analytics integration provides an abstraction layer implemented in a JavaScript file `alx-integration-googleanalytics.js` which contains constructors for objects holding common data and functions to track page views and events:

- `GaAccountData` for the web property id and domain names,
- `GaPageviewData` for data related to a page view, such as the URL and the content id, and
- `GaEventData` for data related to an observed event, such as event category and label.

These data objects are supplied to the implemented functions

- `gaTrackPageview` for tracking page views, and
- `gaTrackEvent` for tracking events.

See JavaScript inline documentation for details.

6.4.3.5 Studio Integration

CoreMedia Analytics Connectors can be configured per site and per page. The settings for this can be configured using the struct editor for the property field `localSettings`. Each tracking provider is configured in a separate `StructProperty`. An example of the tracking configuration is shown below.

```
<StringProperty Name="analyticsProvider">googleAnalytics</StringProperty>
<StructProperty Name="googleAnalytics">
  <Struct>
    <StringProperty Name="measurementId">G-XXXXXXXX</StringProperty>
    <BooleanProperty Name="disabled">false</BooleanProperty>
  </Struct>
</StructProperty>
```

NOTE

Note that tracking can be temporarily disabled for any service provider (even for a particular page) by adding a Boolean property `disabled` with value `true` to the provider's struct property.

NOTE

In the following section the property names to be used in the generic struct editor of *CoreMedia Studio* are described.

Google Analytics

The integration of Google Analytics allows you to configure tracking and configure the link to Google Analytics in *CoreMedia Studio*'s main side bar.

Technical Variable Name	Description/Value	Required
`measurementId`	The Google Analytics measurement ID to track to. The Measurement ID has the format `G-XXXXXXXX`	true
`disableAdvertisingFeaturesPlugin`	Disables the Google Analytics Signals Plugin used for advertising features. If you do not need Google Analytics' advertising features, the plugin should be disabled. The plugin might create additional cookies if it is not disabled. See https://support.google.com/analytics/answer/9445345 for more information.	false

Technical Variable Name	Description/Value	Required
`disabled`	Disables tracking, default: false.	false

Table 6.8. Google Analytics Tracking Configuration Options

Technical Variable Name	Description/Value	Required
`homeUrl`	The URL to the Google Analytics login page. Should be https://analytics.google.com.	true

Table 6.9. Google Analytics Studio Configuration Options

Example for the Studio integration:

```
<StructProperty Name="googleAnalytics">
  <Struct>
    <StringProperty
Name="homeUrl">https://analytics.google.com</StringProperty>
  </Struct>
</StructProperty>
```

6.4.4 Salesforce Marketing Cloud Integration

NOTE

In order to use this integration, you need a license for Marketing Automation Hub and Salesforce Marketing Cloud Connector.

Additionally, you need to license the SFMC integration from CoreMedia.

The integration consists of a extension for the *CoreMedia Blueprint*: `sfmc`. The extension can be requested from the CoreMedia Support Team.

To activate the integration in the Blueprint Workspace you need to enable the SFMC extensions. To that end run the following Maven command in your workspace:

```
$ cd $CM_BLUEPRINT_HOME/workspace-configuration/extensions
$ mvn extensions:sync -Denable=sfmc
```

The blueprint workspace contains a docker compose YAML file `global/deployment/docker/compose/sfmc.yml` which you need to include in your docker compose setup.

Salesforce Marketing Cloud (SFMC) is a customer relationship management (CRM) tool by Salesforce. *CoreMedia Content Cloud* offers you an integration with the following features:

- Upload of content from the CoreMedia system into the SFMC system as assets. See Section 6.1.15, "Uploading Content to Salesforce Marketing Cloud" [355] for the configuration and Section 4.7.9, "Uploading Content to Salesforce Marketing Cloud" in *Studio User Manual* for the usage.
- SFMC Journey based personalization. See ??? for a description of the personalization condition.
- Service API to push data into SFMC data extensions. See the CoreMedia API Javadoc for the class `com.coremedia.blueprint.base.sfmc.libservices.dataextensions.SFMCDataExtensionService`

6.4.5 Pendo Integration

CoreMedia integrates the Pendo service into *CoreMedia Studio* to gather usage data for improving Studio and to offer in-app user guides for a better user experience.

The plugin is enabled by default, but *CoreMedia Studio* users can disable data tracking on their own. You can still exclude the plugin from your CoreMedia installation like any other plugin. Have a look into Section 4.1.6, "Application Plugins" [82].

Section 2.15, "In-App Guide Portal" in *Studio User Manual* and Section 3.8, "Usage Tracking" in *Studio User Manual* describe how to use in-app guides and tracking and how your privacy is guaranteed.

6.5 Advanced Asset Management

CoreMedia Advanced Asset Management consists of two parts:

- An Asset management component with new content types where you can manage your digital assets and licenses.
- An Asset management component which connects to a commerce system to manage assets for products and product variants of the commerce system.

Managing Assets

CoreMedia Asset Management allows you to store and manage your digital assets (for example, high resolution pictures of products) and corresponding licenses in the CoreMedia system. You can customize the storage of assets and the set of available asset types and rendition formats.

A rendition is a derivative of the raw asset, suitable for use in output channels, possibly with some further automated processing. A rendition might be, for example, a cropped and contrast adjusted image in a standardized file format whereas the original file might be stored in the proprietary format of the image editing software in use.

Enhancing Commerce Pages

From such assets, you can create common content items, such as `Picture` or `Download` which you can use to enrich products and product variants (products for short) in the commerce system.

- CMS images and even individual image crops can be used as product images.
- CMS videos can be used as product videos. They will be displayed together with the product images in a gallery.
- CMS content of type *Download* can be offered as additional content that can be downloaded for a product. Any type of binaries are supported, like PDF documents, ZIP archives or office documents.

Such product assets can be edited with *CoreMedia Studio* and will then be delivered by the CMS to enrich, for example, a product detail page.

This section describes the necessary configuration steps for either configuring and deploying *CoreMedia Asset Management* or for removing the contributing modules from the *CoreMedia Blueprint* workspace.

6.5.1 Product Asset Widget

eCommerce Connector specific feature

To present CMS assets on product detail pages you can add the *CoreMedia Product Asset Widget*. For *HCL Commerce*, you replace the default *HCL Commerce Full Image Widget* by the that displays images in an attractive gallery. This makes it particularly easy to present multiple product images and videos for a product.

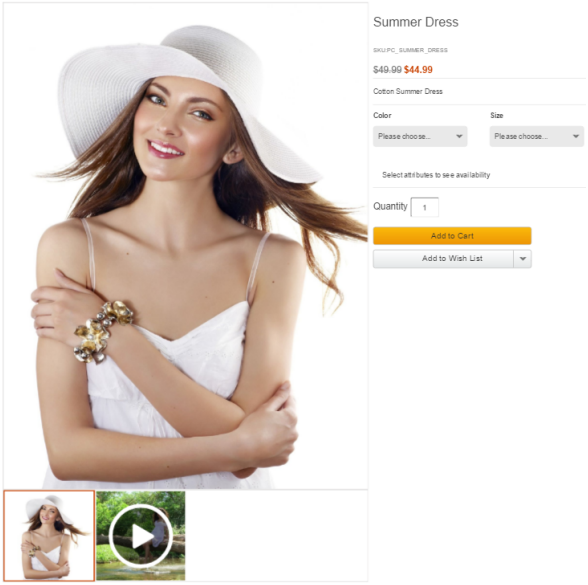

Figure 6.50. Product image gallery in HCL Commerce delivered by the CMS

The *CoreMedia Product Asset Widget* can also be used to display a list of download links that are associated with the product. The download links are shown together with the product image gallery as *Additional Downloads* or in a separate slot on the product detail page.

See Section 3.10, "Deploying the CoreMedia Widgets" in *Connector for HCL Commerce Manual* for *HCL Commerce* or Section 6.4.4, "Finding CMS Content for Product Detail Pages" in *Connector for SAP Commerce Cloud Manual* for SAP Commerce to get the information on how to deploy the *CoreMedia Product*

Asset Widget. For Salesforce you will find the description in the documentation of the commerce Workspace.

Assign Products to CMS Assets

CoreMedia Content Cloud allows you to manage assets in the CoreMedia system that will be used for products and SKUs in the commerce system.

To achieve this *Picture*, *Video* and *Download* content items can be linked with products. That means one picture, video or download can be (re)used for many products. All images and videos that link to the same product act together as a gallery of images and videos of the same product.

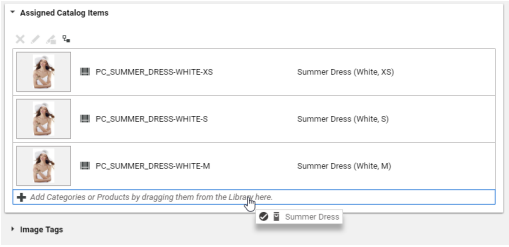

Figure 6.51. Assign a product to a picture

The same applies to downloads. All *Download* content items that link to the same product appear together in an *Available Downloads* list on the product detail page (if the option was used in the *CoreMedia Product Asset Widget*). The order of the images or downloads in the list is determined by the name (in alphabetic order).

You don't have to assign every existing SKU to an asset content item, for example an image, in order to achieve that for each SKU, the same image is delivered. If a SKU is not directly assigned the CMS searches for all asset content items that are assigned to the master product of the SKU or uses the default image for the site (in case of an image).

See Section 6.2.3.9, "Replacing Commerce Images in Products and SKUs with CMS Images" in *Studio User Manual* to learn how to assign products to images using the *CoreMedia Studio*.

6.5.2 Replaced Product and Category Images

Connector for HCL Commerce specific feature

In addition to the *Product Asset Widget* you can replace images directly by replacing the URL in the *HCL Commerce* system with a CoreMedia URL. The linking of product or category images from *HCL Commerce* to the CoreMedia *CAE* is done via Image URLs that you can add to the *Display* tab of the product or category definition.

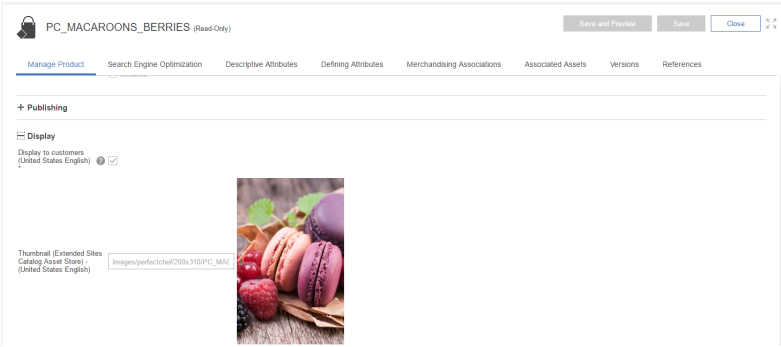

Figure 6.52. Define Product Image URLs in Management Center

NOTE

Regardless of the usage of the *CoreMedia Product Asset Widget*, once the image URLs of a product are pointing to the CMS all occurrences of these product images (for example, on catalog overview pages) will be delivered by the CMS. If multiple images are assigned to one product, then the first image is taken (in alphabetical order).

The Image URL has the following format:

```
For a product:

    http://[cmsHost]/blueprint/servlet/catalogimage/product/
    [storeId]/<Locale>/<Mapping>/<PartNumber>.jpg

    respectively in a multi-catalog scenario
```

```
    http://[cmsHost]/blueprint/servlet/catalogimage/product/
    [storeId]/<Locale>/[catalogId]/<Mapping>/<PartNumber>.jpg

For a category:

    http://[cmsHost]/blueprint/servlet/catalogimage/category/
    [storeId]/<Locale>/<Mapping>/<CategoryID>.jpg

    respectively in a multi-catalog scenario

    http://[cmsHost]/blueprint/servlet/catalogimage/category/
    [storeId]/<Locale>/[catalogId]/<Mapping>/<PartNumber>.jpg
```

Where the path segments have the following meaning:

Segment Name	Example	Description
`[cmsHost]`	[cmsHost]	The URL prefix of the server that can deliver CMS images. Typically, you will enter here the literal string [cmsHost] so the system can map it to a concrete URL prefix. Since the images are delivered from different servers depending on which side you are (preview or live) the hostname can alter between the systems. The placeholder [cmsHost] will then be replaced by a URL prefix containing the live host, provided the request comes from the live side. See also the *HCL Commerce* documentation "Configuration properties for content management system integration".
`[storeId]`	[storeId]	The ID of the *HCL Commerce* store for which the image is requested. An *HCL Commerce* store is configured for a specific site in the CoreMedia system. Typically, you will enter here the literal string [storeId] so the system can map it to a concrete store ID.
Locale	en_US	The locale of the store.
`[catalogId]`	[catalogId]	The ID of the *HCL Commerce* catalog for which the image is requested. This is required only in a multi-catalog scenario. Typically, you will enter here the literal string [catalogId] so the system can map it to a concrete catalog ID.
Mapping	thumbnail	The mapping between an image in the *HCL Commerce* product and the named image variant that is taken from the CoreMedia system.

Segment Name	Example	Description
PartNumber/Category-ID	GFR033_3301/PC_ToDrink	The product or SKU part number or category ID.

Table 6.10. Path segments in the image URL

Delivery of Images

The URL is resolved from the catalog picture handlers. The handlers map the "Named image format" segment to a cropped variant of a picture (see Section 5.4.14, "Images" [208] for details of crops). *CoreMedia Blueprint* comes with the following definition:

```
<bean id="productCatalogPictureHandler"
class="com.coremedia.livecontext.asset.ProductCatalogPictureHandler"
      parent="catalogPictureHandlerBase">
  ...
  <property name="pictureFormats">
    <map>
     <entry value="portrait_ratio20x31/200/310">
       <key>
         <util:constant static-field=

"com.coremedia.livecontext.asset.CatalogPictureHandlerBase.FORMAT_KEY_THUMBNAIL"/>

       </key>
     </entry>
     <entry value="portrait_ratio20x31/646/1000">
       <key>
         <util:constant static-field=

"com.coremedia.livecontext.asset.CatalogPictureHandlerBase.FORMAT_KEY_FULL"/>

       </key>
     </entry>
    </map>
  </property>
</bean>

<bean id="categoryCatalogPictureHandler"
class="com.coremedia.livecontext.asset.CategoryCatalogPictureHandler"
      parent="catalogPictureHandlerBase">
...
</bean>
```

That is, a URL with a segment thumbnail maps to an image variant portrait_ratio20x31 with the width "200" and the height "310" and a URL with segment full maps to the same image variant portrait_ratio20x31 but with width "646" and height "1000". These are the values required by the HCL Aurora Starter Store.

You can customize the configuration via a Spring configuration as described in Section 6.5.4.1, "Mapping of Custom Picture Formats" [421].

6.5.3 Extract Image Data During Upload

If your pictures files are enriched with the product codes as XMP/IPTC "artwork or object in the picture", the system automatically tries to extract data during the upload. How the data is used depends on the content item to which you upload the image.

- Upload to a `Picture`: The product codes are extracted and the system tries to add a reference to the product in the eCommerce repository with this product code.
- Upload to a `Picture Asset`: The product codes are extracted and are added to the `Picture Asset`.

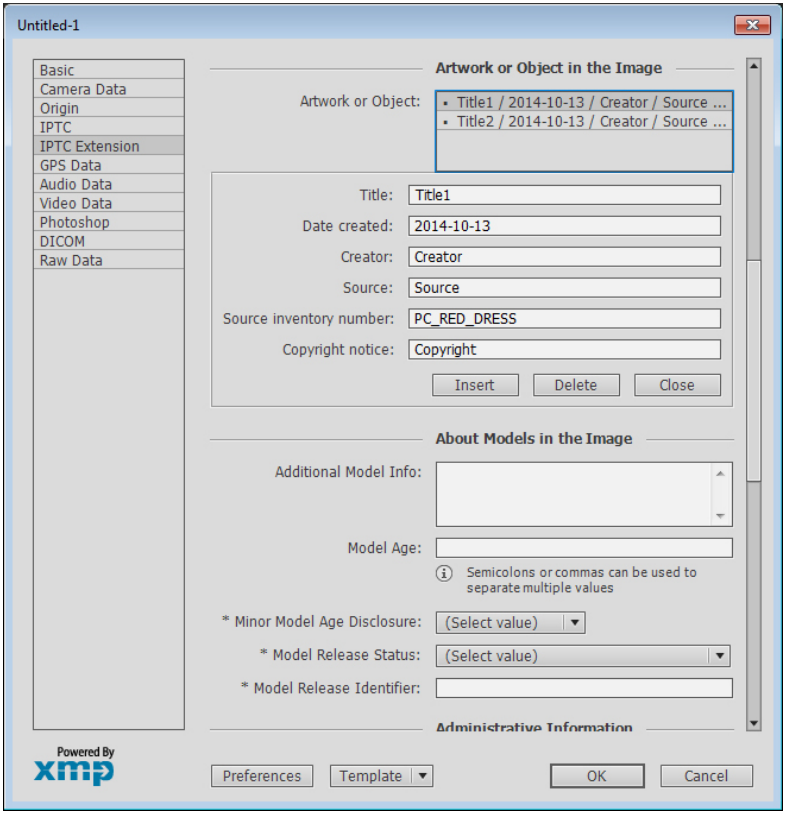

Figure 6.53. Screenshot from Adobe Photoshop for a Picture containing XMP Data

Upload to a Picture content item

While uploading the pictures via *CoreMedia Studio* into a `Picture` item, the system automatically extracts the product codes and adds references to the assigned products. At this process the product references contained in the original image data will be remembered. You have the option to reset to the original imported data after you have changed the assignments manually.

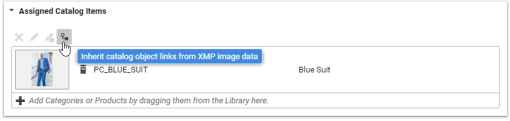

Figure 6.54. Picture linked to XMP Product Reference

After an initial import, the status of the *Assigned Products* section is set to "inherited". All associated product references are shown as "read only" and can only be edited if the *Switch off inheritance* button is pressed.

Each reimport of the same image data (with an update of the blob) leads to an update of the associated product references unless the references have been changed manually. In general, the rule applies, that no data will be overwritten that have been changed manually.

6.5.4 Configuring Asset Management

In the following it is described how you can adapt *CoreMedia Asset Management* to your specific needs:

- Define which crops of an image are used in shop pages.
- Define from which CAEs the commerce system gets images.
- Define content types for your own assets.
- Define publication behavior for renditions of your assets.
- Define where large blobs should be stored.
- Define appropriate rights in the CoreMedia system for your asset content.

6.5.4.1 Mapping of Custom Picture Formats

eCommerce Connector specific feature

You can manage pictures in *CoreMedia Content Cloud* that are used in commerce products and SKUs pages. You can use Spring configuration, to map URL path segments to specific crops.

CoreMedia Blueprint comes with a predefined mapping defined in the `catalogPictureHandler` bean. If you want to define your own mapping you can overwrite the default setting as follows:

```
@Customize(value = "catalogPictureHandler.pictureFormats", mode = REPLACE)
@Bean(autowireCandidate = false)
public Map<String, String> customizeCatalogPictureHandler() {
  return Map.of(
    "customFormat1", "custom_crop1/300/410",
    "customFormat2", "custom_crop2/700/1200");
}
```

The *key* attribute in the `entry` tag is the identifier that is used in the request URL while value is the name of the crop of the image that will be used followed by the size of the image as "/width/height/ in pixel. The definition of crops is explained in Section 5.4.14, “Images” [208]

6.5.4.2 Placeholder Resolution for Asset URLs

Connector for HCL Commerce specific feature

In the *HCL Commerce* system you can use a placeholder in image URLs which is resolved through a database lookup in theSTORECONFtable. See the HCL documentation for more details at https://help.hcltechsw.com/commerce/8.0.0/developer/refs/rwccmsresolvecontenttag.html .

For example:

```
http://[cmsHost]:<CAEPort>/blueprint/servlet/catalogimage/product/
    [storeId]/<Locale>/<Mapping>/<PartNumber>.jpg
```

The placeholders in the example above are *[cmsHost]* and *[storeId]*.

To resolve *[cmsHost]* - see the HCL documentation for `ResolveContentURLCmdImpl` for more information. If you want to connect preview and live *CAE* to one *Management Center* you can define different values for `wc.resolveContentURL.cmsHost` and `wc.resolveContentURL.cmsPreviewHost` in theSTORECONFtable.

If you use one extended sites catalog for multiple shops you can specify a *[storeId]* placeholder in your image URLs, which are dynamically resolved at runtime.

In a development setup you may share one *HCL Commerce* instance for preview and live delivery.

In order to identify the *CAE* (preview or live) from which the image should be delivered, depending on the shop URL, for example, `shop-helios.docker.localhost` versus `shop-preview-helios.docker.localhost` a proxy server can add a request header `X-FragmentHost` which contains the value preview or live.

If you want to activate *[cmsHost]* resolution for a shared *HCL Commerce* preview/live environment, perform the following steps:

1. Register and map the `FragmentHostFilter` servlet to `workspace/Stores/WebContent/WEB-INF/web.xml` of the *HCL Commerce* to extract the `X-FragmentHost` header information from the request.

```
...
<filter>
  <filter-name>FragmentHostFilter</filter-name>

<filter-class>com.coremedia.livecontext.servlet.FragmentHostFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>FragmentHostFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...
```

2. Register the `CoreMediaResolveContentURLCmdImpl` in the *HCL Commerce*. This command resolves `[cmsHost]` placeholder in image URLs depending on a preview or live switch for the current request. It resolves `[storeId]` placeholder as well. To register the command perform the following SQL statement:

```
insert into cmdreg (storeent_id, interfacename, classname)
      values (0,'com.ibm.commerce.content.commands.ResolveContentURLCmd',


'com.coremedia.commerce.content.commands.CoreMediaResolveContentURLCmdImpl');
```

   Refer to the HCL documentation for more details about registering custom command implementations in the command registry

   To resolve `[storeId]` in Management Center, you have to register and map the `ImageFilter` servlet to `workspace/LOBTools/WebContent/WEB-INF/web.xml` of the *HCL Commerce*.

```
...
<filter>
  <filter-name>ImageFilter</filter-name>
  <filter-class>com.coremedia.livecontext.servlet.ImageFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>ImageFilter</filter-name>
  <url-pattern>/LoadImage</url-pattern>
</filter-mapping>
...
```

6.5.4.3 Content Types

Abstract content type AMAsset

CoreMedia Advanced Asset Management stores its data in the content repository in content items. *CoreMedia Content Cloud* contains the abstract root content type `AMAsset` (see Chapter 4, *Developing a Content Type Model* in *Content Server Manual* for a description of content types) as a starting point for assets. `AMAsset` defines a property `original` to store the raw editable form of the asset and another property `thumbnail` to store a thumbnail view. The thumbnail property can be used for a uniform preview of assets. If there is no sensible thumbnail for an asset, it can be left empty.

Concrete content types AMPictureAsset and AMDocumentAsset

Concrete content types for specific assets, such as pictures or documents, need to extend the abstract content type `AMAsset`. Most probably, you will add more properties for different renditions of the asset. Names of rendition properties must be alphanumeric strings. By default, `AMPictureAsset` and `AMDocumentAsset` are provided as a non-abstract asset type, defining rendition properties for web delivery and for printing.

Defining your own asset types

You can modify existing asset types or define additional asset types in the file `asset-management-plugin-doctypes.xml` in the `am-server` module. For each asset type, you need an appropriate form in *CoreMedia Studio*. *CoreMedia Blueprint* already defines suitable *Studio* forms for the `AMPictureAsset` and `AMDocumentAsset`. Change this form when you adapt the `AMPictureAsset` or `AMDocumentType` content type and add further forms for your own asset types.

Store large blobs in the file system

When you add further rendition properties that hold very large blobs, modify the blob store configuration as described in Section 3.3, "Configuring Blob Storage" in *Content Server Manual*. Small renditions up to a few megabytes can be stored in the *Content Server* database and do not need additional configuration.

To prevent large blobs like the original rendition from being published, you can exclude them from publication process. For more information read Section 6.5.4.4, "Configure Rendition Publication" [424].

6.5.4.4 Configure Rendition Publication

Certain renditions can be excluded from publication. To do so the `am-server-component` comes with an `AssetPublishInterceptor` which reads the `metadata` property of assets to determine if a given rendition should be published or not.

The `AssetPublishInterceptor` bean is added to the *Content Server* and to the corresponding command-line tools. The following properties control the behavior of the interceptor:

`assetMetadataProperty` The `Struct` property which contains the information whether to publish a rendition or not at path `renditions.<rendition-name>.show`. When the Boolean property `show` is `true`, the rendition blob will be published. Otherwise, the blob will not be available on the master server.

`interceptingSubtypes` Boolean flag to control whether also subtypes of `type` should be intercepted or not.

removeDefault	The default value to control whether a rendition blob should be removed from publication or not. If unset the default is to remove blobs if nothing else is specified in either the metadata struct or in the removal overrides.
removeOverride	Overrides any setting or default for a given rendition. It contains a map from rendition name to removal flag. Thus, if you want the rendition `thumbnail` to be published in any case add an entry with key `thumbnail` and value false.
type	The content type the interceptor applies to. For subtype processing set the flag `interceptingSubtypes` accordingly.

Example 6.16, "Rendition Publication Configuration" [425] shows a possible configuration of the `AssetPublishInterceptor`.

```
<beans ...>

  <util:map id="removeOverride"
            key-type="java.lang.String"
            value-type="java.lang.Boolean">
    <entry key="thumbnail" value="false"/>
  </util:map>

  <bean id="assetPublishInterceptor"
        class=
    "com.coremedia.blueprint.assets.server.AssetPublishInterceptor">
    <property name="type" value="AMAsset"/>
    <property name="interceptingSubtypes" value="true"/>
    <property name="assetMetadataProperty" value="metadata"/>
    <property name="removeDefault" value="true"/>
    <property name="removeOverride" ref="removeOverride"/>
  </bean>
</beans>
```

Example 6.16. Rendition Publication Configuration

6.5.4.5 Blob Storage

Blobs of renditions can be stored in the database or in the file system. In general, content in the CoreMedia CMS is stored in a database, but for large blobs, the file system might be better suited for storage, because databases are not always optimized for this use case.

Blueprint default storage behavior

When you start the *Content Management Server* using `mvn spring-boot:run` in module `content-management-server-app` of the development installation, all blobs are stored in the database.

Configuration of blob storage

Blob storage is configured in the *Content Server*'s Spring application context, see Section 3.3, "Configuring Blob Storage" in *Content Server Manual* for details. With the property `am.blobstore.rootdir`, you define the root directory for file system storage.

NOTE

Keep in mind, that storing a blob in the file system might double the required space, when you use the rendition in another content item, for example, in a `Picture`.

This is because, when you store a blob in the database and the same blob is used in different content items, then all the content items link to this blob. On the other hand, when you have stored a blob in the file system and this blob is used in another content item that does not define file system storage, then a copy of the blob will be created in the database.

6.5.4.6 Rights

Assets in the form of `AMAsset` content items are placed in the `/Assets` folder by default. Define rights rules for the content repository in such a way that only authorized users can create and change assets and that assets can only be placed in the folder `/Assets`. Note that access rights for the root content type `Document_` automatically imply rights on assets.

Studio

The asset management extension of *CoreMedia Studio* is defined in the modules `am-studio` and `am-studio-component`.

In `am-studio` you can find the form definition for picture forms in the file `AMPictureAssetForm.ts`. Update this file if you change the set of renditions. Create additional form when you add further asset types. Localizations of asset types and rendition names can be added to the resource bundle `AMDocument Types`.

The module `am-studio-component` contains configuration information for the Studio REST backend. In the file `component-am-studio.xml` you can find the configuration of two write interceptors which update the asset metadata as renditions are uploaded using Studio.

6.5.4.7 Asset Download Portal

CoreMedia Advanced Asset Management comes with an asset download portal. You can configure the behavior of the portal in the `Asset Management Configuration` content item in *Studio* as shown in Figure 6.55, “Configuration of the download portal” [427].

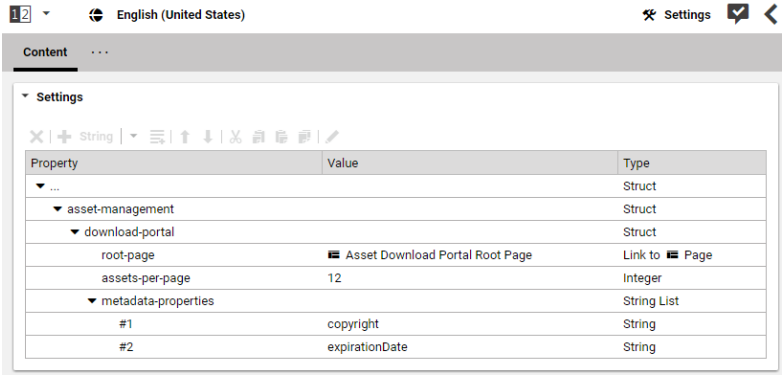

Figure 6.55. Configuration of the download portal

The properties in the `download-portal` struct have the following meaning:

root-page — A `Page` content item which defines the context of the download portal. The root page contains the *AM Download Portal Placeholder* in the *Main* placement.

assets-per-page — The number of assets that are shown in one page.

metadata-properties — The properties from the asset's metadata that are shown in the detail view of an asset.

The hierarchy of the assets in the download portal is determined by the Asset Download Portal taxonomy. That is, an asset content item is shown on the download portal, when it contains an asset category tag and a downloadable property.

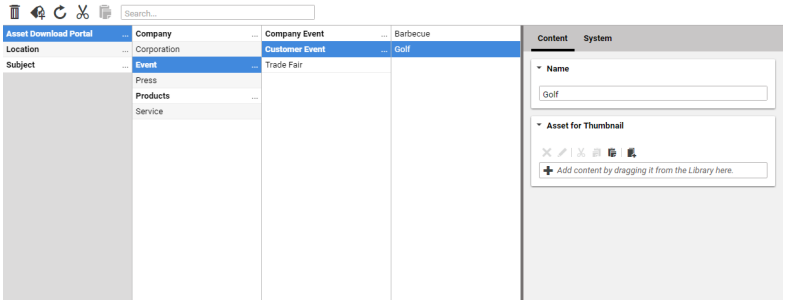

Figure 6.56. Taxonomy for assets

7. Reference

This chapter contains detailed information about some of *CoreMedia Blueprint's* features:

- Section 7.1, "Content Type Model" [430] shows UML diagrams of *CoreMedia Blueprint* content types.
- Section 7.2, "Link Format" [433] lists the controller, link schemes and link post-processors of *CoreMedia Blueprint*.
- Reference - Predefined Users [439] shows the predefined users that are available in the system for log in to *Studio*.
- Section 7.4, "Database Users" [445] shows the database users that are needed by the CoreMedia server components.
- Section 7.5, "Cookies" [447] lists all cookies delivered by *CoreMedia Content Cloud*.

7.1 Content Type Model

This section shows the content types of *CoreMedia Blueprint* as UML diagrams. Since the content type model exists of more than forty items it is split into the following diagrams:

- Figure 7.1, "CoreMedia Blueprint Content Type Model - CMLocalized" [430] shows the content types inheriting from CMLocalized.
- Figure 7.2, "CoreMedia Blueprint Content Type Model - CMNavigation" [431] shows the content types inheriting from CMNavigation.
- Figure 7.3, "CoreMedia Blueprint Content Type Model - CMHasContexts" [431] shows the content types inheriting from CMHasContexts.
- Figure 7.4, "CoreMedia Blueprint Content Type Model - CMMedia" [431] shows the content types inheriting from CMMedia.
- Figure 7.5, "CoreMedia Blueprint Content Type Model - CMCollection" [432] shows the content types inheriting from CMCollection.

The following diagrams contain most of the content types. The colors have the following meaning:

- Blue items are part of the basis Blueprint content items
- Yellow items are part of the eCommerce integration
- Red items are part of the Elastic Social Integration

You can download the complete diagram as a `graphml` file from the online documentation page below *Other Documentation* named *CoreMedia Content Cloud Content Type Diagram*:

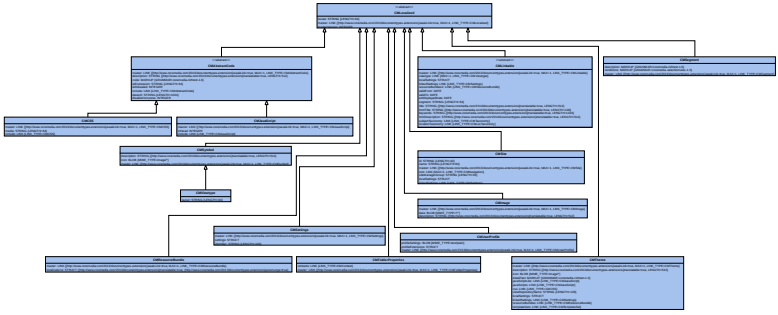

Figure 7.1. CoreMedia Blueprint Content Type Model - CMLocalized

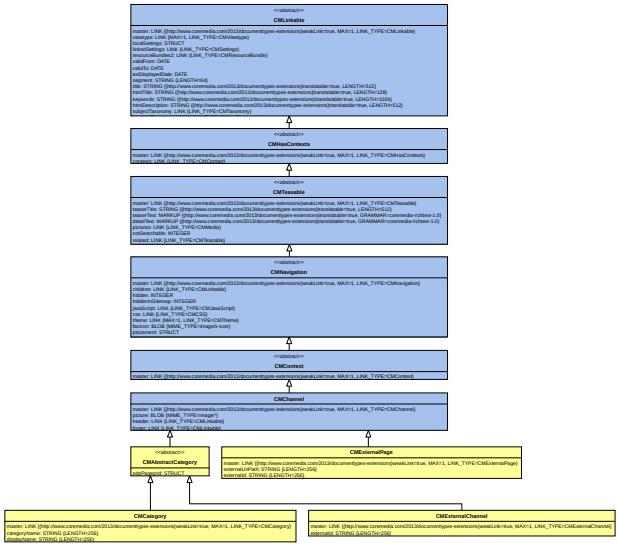

Figure 7.2. CoreMedia Blueprint Content Type Model - CMNavigation

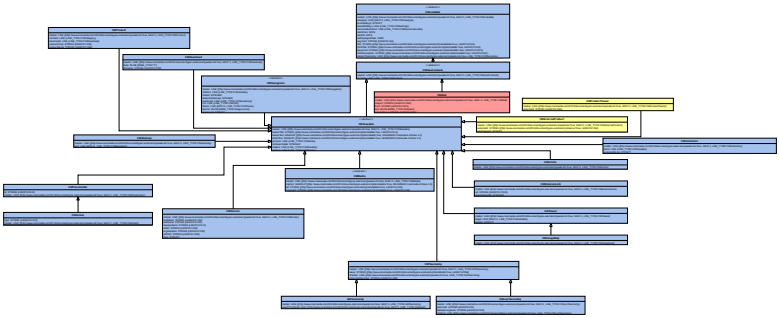

Figure 7.3. CoreMedia Blueprint Content Type Model - CMHasContexts

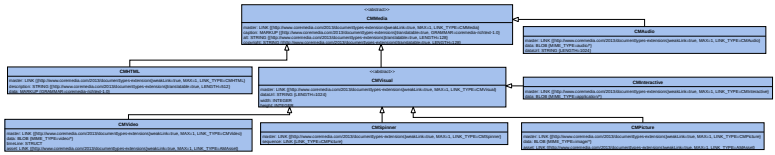

Figure 7.4. CoreMedia Blueprint Content Type Model - CMMedia

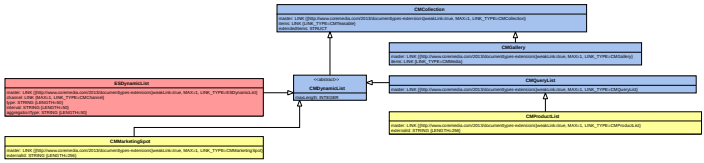

Figure 7.5. CoreMedia Blueprint Content Type Model - CMCollection

7.2 Link Format

The following table summarizes most of the corresponding link schemes and controllers of *CoreMedia Blueprint*. See the Javadoc of the respective classes for further details.

CapBlobHandler	
Description	Controller and link scheme for Blobs like Images in CSS, Images that do not have any scaling information or CMDownload content items.
Class	`com.coremedia.blueprint.cae.handlers.CapBlobHandler`
Example	`/blob/1784/4fb7741a1080d02953ac7d79c76c955c/media-data.ico` for a CSS background image

Table 7.1. `CapBlobHandler`

CodeResourceHandler	
Description	Controller and link scheme for CSS and JavaScript stored in the CMS.
Class	`com.coremedia.blueprint.cae.handlers.CodeResourceHandler`
Example	`/code/1214/5/responsive-css.css` for a CSS

Table 7.2. `CodeHandler`

ExternalLinkHandler	
Description	A Link scheme for external links stored in the CMS.
Class	`com.coremedia.blueprint.cae.handlers.ExternalLinkHandler`

Example	`http://www.coremedia.com`

Table 7.3. `ExternalLinkHandler`

PageActionHandler	
Description	Controller and link scheme for `CMAction` beans which are for example used to perform a search.
Class	`com.coremedia.blueprint.cae.handlers.PageActionHandler`
Example	`/action/corporate/4420/action/search` for performing a search

Table 7.4. `PageActionHandler`

PageHandler	
Description	Controller and link scheme for pages.
Class	`com.coremedia.blueprint.cae.handlers.PageHandler`
Example	`/corporate/for-professionals/services` for a service page.

Table 7.5. `PageHandler`

PageRssHandler	
Description	Controller and link scheme for handling RSS
Class	`com.coremedia.blueprint.cae.handlers.PageRssHandler`
Format	`/service/rss/[SITE_URL_SEGMENT]/[CONTENT_ID]/feed.rss for RSS feed with content id [CONTENT_ID]`

Table 7.6. `PageRssHandler`

PreviewHandler	

Description	Controller and link scheme previewing content in CoreMedia Studio.
Class	`com.coremedia.blueprint.cae.handlers.PreviewHandler`
Example	`/preview?id=coremedia:///cap/content/3048%26view=fragmentPreview` for preview content as a fragment

Table 7.7. `PreviewHandler`

StaticUrlHandler

Description	Controller and link scheme for generating static URLs based on Strings
Class	`com.coremedia.blueprint.cae.handlers.StaticUrlHandler`
Example	`/elastic/social/ratings` for a ES Post Controller

Table 7.8. `StaticUrlHandler`

TransformedBlobHandler

Description	Controller and link scheme for transformed blobs
Class	`com.coremedia.blueprint.cae.handlers.TransformedBlobHandler`
Example	`/image/3126/landscape_ratio4x3/349/261/971b670685dff69cfd28e55177d886db/Pi/mombasa-image-image.jpg`

Table 7.9. `TransformedBlobHandler`

Link Post Processors

While link schemes are responsible for the path and possibly the parameters of a resource's URL, they are not aware of deployment aspects like domains, hosts, ports, servlet contexts, rewrite rules and the like. The Blueprint uses Link Post Processors to format links according to the particular environment.

The following link post processors are applied in `com.coremedia.blueprint.base.links.impl.BaseUriPrepender` and `com.coremedia.blueprint.base.links.impl.LinkAbsolutizer`.

- `prependBaseUri`

  Prepends the "base URI" (web application and mapped servlet, for example `/blueprint/servlet`) to ALL (annotation based) links. This is required when the CAE web application is served directly by a web container with no prior URL rewriting.

- `makeAbsoluteUri`

  Adds a prefix that makes the URI absolute. There are several cases in which URLs must be made absolute:

  - a cross-site link: a URI pointing to a resource in a site that is served on a different domain
  - externalized URIs: a URI should be send by mail or become part of an RSS feed

  The prefixes for absolute URLs are specific for each site, therefore they are maintained in each site's settings in a struct named `absoluteUrlPrefixes`. The prefixes are different for the live and the preview CAE and must be maintained independently. A typical `absoluteUrlPrefixes` struct looks like this:

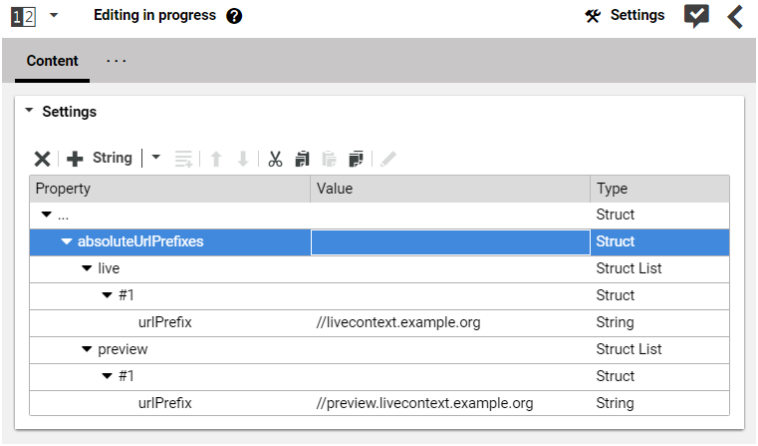

Figure 7.6. A basic absoluteUrlPrefixes Struct

NOTE

The URL prefixes must be at least a scheme-relative URL (beginning with "`//`").

The Blueprint features an application property `link.urlPrefixType` that determines which `absoluteUrlPrefixes` entry is effective in a particular application. You will find `link.urlPrefixType` set appropriately in the `application.properties` of all components that use the `bpbase-links-impl` module, for example, for the `cae-live-webapp`:

```
# The live webapp builds live URLs
link.urlPrefixType=live
```

Example 7.1. Configuration of URL prefix type

While the standard Blueprint distinguishes only between preview and live URL prefixes, projects may add additional `absoluteUrlPrefixes` entries of arbitrary names for special URL prefixes and applications.

Why are struct lists needed after all, if they have only one entry? The above example is valid, but it does not show all configuration options. There are some optional features, and the equivalent complete struct would look like this:

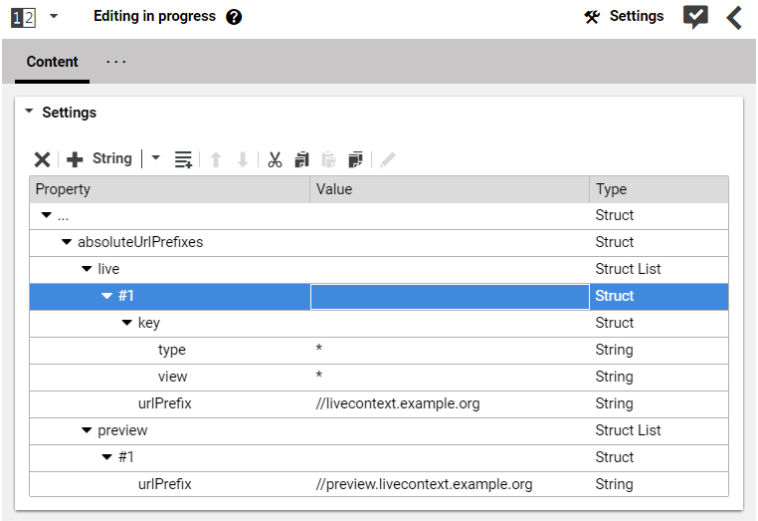

Figure 7.7. A complete absoluteUrlPrefixes Struct

You can declare special URL prefix rules for certain bean types or views, and you can specify an order for ambiguous rules. The default Blueprint does not make use of these options, but they reflect the format of the `key` field of the old `siteMappings` entries, so that you do not lose any features when upgrading to this mechanism.

When you start over with a fresh Blueprint and look at the `SiteSettings` content items of our example content, the configuration looks yet a little different:

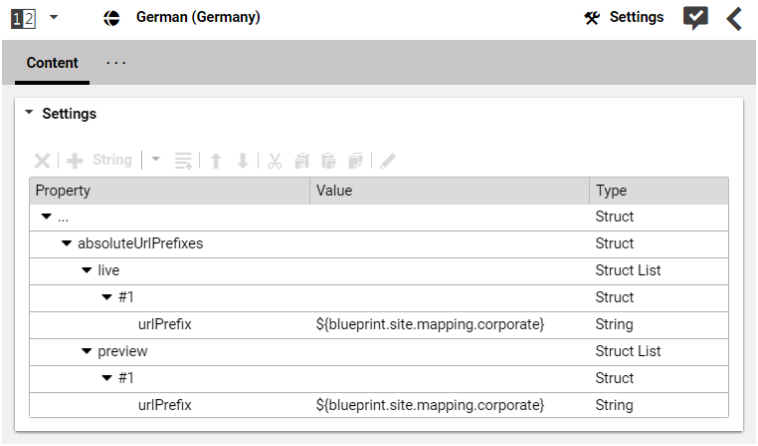

Figure 7.8. An initial absoluteUrlPrefixes Struct

CoreMedia supports a variety of deployments for various use cases like local development, production, quickrun, to name just a few. The appropriate URL prefixes in these setups vary from `//localhost:40080` to `//my.real.public.domain`. So there is no reasonable default to be hardcoded in the example content. Therefore, application properties in the values of `urlPrefix` are supported and use the well known `blueprint.site.mapping.*` properties which are declared in the CAEs' `application.properties` files. Initial content deployment is the only reason why these properties still exist, so you should not bother to maintain them for new sites in production repositories, but maintain the URL prefixes only in the content.

Force Scheme

In order to force a certain scheme (for example http, https, ftp) for a URL, two (cm) parameters must be set for the link: `absolute`: `true` and `scheme`: `<scheme-name>`.

7.3 Predefined Users

CoreMedia Blueprint provides some default users and groups that represent typical roles in an editorial staff. There are technical users with repository wide permissions and editorial users whose permissions are predominantly limited to a particular site or web presence (aside from a few exceptions like home folder access). The editorial users and groups are only available if you activate the particular extension. Depending on your specific processes and roles, the default groups may be a more or less useful starting point for a production systems. The users, however, are meant as examples only. You are supposed to replace them with users that match your actual staff. The password of all default users is the same as the name.

In the *Blueprint* workspace you will find some `test-data/users` directories (one global and some in the extensions). The XML files in those directories declare the default users, groups and rules. They can be imported with the restoreusers command line tool. For the initial setup of your systems, you can adapt those files to your needs. The `test-data/content` sets provide home folders with suitable editor preferences content items for the users.

The following tables show the most important default users and groups in detail.

Global

Group name	Description
staff	Root group, essential common read permissions, home folder access
administratoren	All possible permissions
developer	All possible permissions but user authorization
global-manager	Editorial permissions for global themes and settings
composer-role, approver-role, publisher-role	Publication workflow roles
translation-workflow-robots	Permissions for creating derive content

Table 7.10. Global groups

While some of the global groups contain users directly, most of them serve only as parent groups for the site-specific groups.

User name	Group	Description
Adam	administratoren	Administrator: IT operations, configuration, user authorization, workflow maintenance, recovery, performance analysis
Teresa	administratoren	Online Marketing Manager: Analytics analysis, campaign management, supervision
Dave	developer	Developer: Feature development, template development, performance tuning
Amy	asset-manager	Asset Manager: managing digital assets
translation-workflow-robot	translation-workflow-robots	Technical User: Used to derive sites. The user is defined via the property `sitemodel.translationWorkflowRobotUser`.

Table 7.11. Global users

Since user and group names are unique within one repository, they differ for the members of the various web presences of Blueprint. The following users and groups reflect the eCommerce web presence. The roles of the Brand web presence are basically the same, and use similar names that you will easily recognize.

eCommerce

Group name	Description
global-site-manager-sf	All permissions for a web presence
local-site-manager-sf	Editorial permissions for a site, read rights for the master site
online-editor-sf-en-US	Finegrained permissions for his particular tasks

Group name	Description
media-editor-sf-en-US	Finegrained permissions for his particular tasks on media objects

Table 7.12. Site specific groups for Salesforce Commerce

User name	Group	Description
Sally	global-site-manager-sf	Global site manager: organization of internal processes
Peter SF, Pierre SF, Pietro SF, Yoshi SF, Yang SF (for their respective regions)	manager-sf-en-GB, manager-sf-fr-FR, manager-sf-it-IT, manager-sf-ja-JP, manager-sf-zh-CN	Local content manager: management of dynamic content, targeting rules, A-B-testing, topic pages for their particular regions
George SF	online-editor-sf-en-US	Online editor: writing articles, creating slideshows, editing images, tagging contents
Mark SF	media-editor-sf-en-US	Media editor: creating media objects, creating slideshows, video and audio objects, editing images, tagging media content

Table 7.13. Site specific users for Salesforce Commerce

Group name	Description
global-site-manager-h-b2c	All permissions for a web presence
local-site-manager-h-b2c	Editorial permissions for a site, read rights for the master site
online-editor-h-b2c-en-GB, online-editor-h-b2c-de-DE	Finegrained permissions for his particular tasks

Group name	Description
media-editor-h-b2c-en-GB, media-editor-h-b2c-de-DE	Finegrained permissions for his particular tasks on media objects

Table 7.14. Site specific groups for SAP Commerce

User name	Group	Description
Rick H, Harry	global-site-manager-h-b2c	Global site manager: organization of internal processes
Peter H, Pierre H, Piet H, Pedro H, Yoshi H (for their respective regions)	manager-h-b2c-en-GB, manager-h-b2c-fr-FR, manager-h-b2c-de-DE, manager-h-b2c-es-ES, manager-h-b2c-ja-JP	Local content manager: management of dynamic content, targeting rules, A-B-testing, topic pages for their particular regions
George H, Georg H	online-editor-h-b2c-en-GB, online-editor-h-b2c-de-DE	Online editor: writing articles, creating slideshows, editing images, tagging contents
Mark H, Amy, Markus H	media-editor-h-b2c-en-GB, media-editor-h-b2c-en-GB, media-editor-h-b2c-de-DE	Media editor: creating media objects, creating slideshows, video and audio objects, editing images, tagging media content

Table 7.15. Site specific users for SAP Commerce

Group name	Description
global-site-manager, global-site-manager-b2b, global-site-manager-calista	All permissions for a web presence
local-site-manager, local-site-manager-b2b, local-site-manager-calista	Editorial permissions for a site, read rights for the master site
online-editor-en-US, online-editor-de-DE, online-	Finegrained permissions for his particular tasks

Group name	Description
editor-b2b-en-US, online-editor-b2b-de-DE	
media-editor-en-US, media-editor-de-DE, media-editor-b2b-en-US, media-editor-b2b-de-DE	Finegrained permissions for his particular tasks on media objects

Table 7.16. Site specific groups for HCL Commerce

User name	Group	Description
Rick, Rick Cal	global-site-manager, global-site-manager-calista	Global site manager: organization of internal processes
Peter, Piet, Pierre, Pedro, Yoshi (for their respective regions)	manager-en-US, manager-de-DE, manager-fr-FR, manager-es-ES, manager-ja-JP	Local content manager: management of dynamic content, targeting rules, A-B-testing, topic pages for their particular regions
Peter Cal, Piet Cal (for their respective regions)	manager-calista-en-US, manager-calista-de-DE	Local content manager: management of dynamic content, targeting rules, A-B-testing, topic pages for their particular regions
George, Georg	online-editor-en-US, online-editor-de-DE	Online editor: writing articles, creating slideshows, editing images, tagging contents
Mark, Markus	media-editor-en-US, media-editor-de-DE	Media editor: creating media objects, creating slideshows, video and audio objects, editing images, tagging media content

Table 7.17. Site specific users for HCL Commerce

Brand web presence

Group name	Description
global-site-manager-c	All permissions for a web presence
manager-c-en-US	Editorial permissions for a site, read rights for the master site

Group name	Description
online-editor-c-en-US	Finegrained permissions for his particular tasks

Table 7.18. Site specific groups Brand web presence

User name	Group	Description
Rick C	global-site-manager-c	Global site manager: organization of internal processes
Peter C, Piet C, Pedro C, Pierre C (for their respective regions)	manager-c-en-US, manager-c-de-DE, manager-c-es-ES, manager-c-fr-FR	Local content manager: management of dynamic content, targeting rules, A-B-testing, topic pages for their particular regions
George C, Marc C	online-editor-c-en-US	Online editor: writing articles, creating slideshows, editing images, tagging contents

Table 7.19. Site specific users Brand web presence

7.4 Database Users

The following tables show the database users that are required for CoreMedia applications and features. For MySQL you will find scripts in the workspace creating these users.

Table 7.20, " Application-Specific Database Users " [445] refers to the central, application-specific database users. Along with that come database users bound to certain features like *Editorial Comments* as shown in Table 7.21, " Feature-Specific Database Users " [445]. Unlike the application-specific database users, these may be accessed by several applications in parallel. Note, that these database users (thus, especially their respective names, passwords) must be configured in all bound applications.

Application	Username	Description
Content Management Server	cm_management	This database user will manage the content of the *Content Management Server*. This database will require most of the space, since content is versioned.
Master Live Server	cm_master	This database user will manage the content of the *Master Live Server*. Up to two versions of each published content will be stored.
Replication Live Server	cm_replication	This database user will manage the content of the *Replication Live Server*. Up to two version of each published content will be stored.
CAE Feeder for preview	cm_mcaefeeder	This database user will persist data for the *CAE Feeder* working in the management environment. Content is not versioned.
CAE Feeder for live site	cm_caefeeder	This database user will persist data for the delivery environment. Content is not versioned.

Table 7.20. Application-Specific Database Users

Username	Description	Applications(s)
cm_caplist	This database user will persist data for the features *My Edited*	*Studio Server*; *User Changes Application*; *Workflow Server*

Username	Description	Applications(s)
	Content and *Workflow Lists* if SQL persistence is enabled for CapLists. For details see Section 3.8, "My Edited Content and Workflow Lists Properties" in *Deployment Manual*.	
cm_editorial_comments	This database user will persist data for the feature *Editorial Comments*.	*Studio Server*
cm_notifications	This database user will persist data for the *Notifications* based on SQL persistence layer. For details see Notifications SQL Persistence Configuration for CoreMedia Studio in *Deployment Manual*.	*Studio Server*; *User Changes Application*
cm_projects	This database user will persist data for the *Projects/To-Dos* based on SQL persistence layer. For details see Section 3.4.12, "Projects/To-Dos SQL Persistence Configuration" in *Deployment Manual*.	*Studio Server*

Table 7.21. Feature-Specific Database Users

7.5 Cookies

Several customer facing modules of *CoreMedia Content Cloud* use cookies to fulfill their tasks.

Blueprint delivery CAE

The Blueprint delivery CAE is configured to not write any cookies. However, session cookies CM_SESSIONID and JSESSIONID are written, when a website visitor logs into the Blueprint delivery CAE. The name of these cookies may vary, depending on the deployment scenario.

Elastic Social

Elastic Social writes only one cookie:

guid A globally unique ID to identify the user's web browser

eCommerce

When you use eCommerce, the Commerce Server writes cookies. For *HCL Commerce* the cookies are documented at https://help.hcltechsw.com/commerce/8.0.0/admin/concepts/cse_cookies.html.

Glossary

Blob	Binary Large Object or short blob, a property type for binary objects, such as graphics.
CaaS	Content as a Service or short caas, a synonym for the CoreMedia Headless Server.
CAE Feeder	Content applications often require search functionality not only for single content items but for content beans. The *CAE Feeder* makes content beans searchable by sending their data to the *Search Engine*, which adds it to the index.
Content Application Engine (CAE)	The *Content Application Engine* (*CAE*) is a framework for developing content applications with *CoreMedia CMS*. While it focuses on web applications, the core frameworks remain usable in other environments such as standalone clients, portal containers or web service implementations. The CAE uses the Spring Framework for application setup and web request processing.
Content Bean	A content bean defines a business oriented access layer to the content, that is managed in *CoreMedia CMS* and third-party systems. Technically, a content bean is a Java object that encapsulates access to any content, either to CoreMedia CMS content items or to any other kind of third-party systems. Various CoreMedia components like the CAE Feeder or the data view cache are built on this layer. For these components the content beans act as a facade that hides the underlying technology.
Content Delivery Environment	The *Content Delivery Environment* is the environment in which the content is delivered to the end-user. It may contain any of the following modules: • *CoreMedia Master Live Server* • *CoreMedia Replication Live Server* • *CoreMedia Content Application Engine* • *CoreMedia Search Engine* • *Elastic Social*

	• *CoreMedia Native Personalization*
Content Feeder	The *Content Feeder* is a separate web application that feeds content items of the CoreMedia repository into the *CoreMedia Search Engine*. Editors can use the *Search Engine* to make a full text search for these fed items.
Content item	In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.
Content Management Environment	The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules: • *CoreMedia Content Management Server* • *CoreMedia Workflow Server* • *CoreMedia Studio* • *CoreMedia Search Engine* • *CoreMedia Native Personalization* • *CoreMedia Preview CAE*
Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.
Content Repository	*CoreMedia CMS* manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.
Content Server	*Content Server* is the umbrella term for all servers that directly access the CoreMedia repository: *Content Servers* are web applications running in a servlet container. • *Content Management Server* • *Master Live Server* • *Replication Live Server*
Content type	A content type describes the properties of a certain type of content. Such properties are for example title, text content, author, ...
Contributions	Contributions are tools or extensions that can be used to improve the work with *CoreMedia CMS*. They are written by CoreMedia developers - be it clients, partners or Core-

	Media employees. CoreMedia contributions are hosted on Github at https://github.com/coremedia-contributions.
Control Room	*Control Room* is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
CORBA (Common Object Request Broker Architecture)	The term *CORBA* refers to a language- and platform-independent distributed object standard which enables interoperation between heterogenous applications over a network. It was created and is currently controlled by the Object Management Group (OMG), a standards consortium for distributed object-oriented systems. CORBA programs communicate using the standard IIOP protocol.
CoreMedia Studio	*CoreMedia Studio* is the working environment for business specialists. Its functionality covers all the stages in a web-based editing process, from content creation and management to preview, test and publication. As a modern web application, *CoreMedia Studio* is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.
Dead Link	A link, whose target does not exist.
Derived Site	A derived site is a site, which receives localizations from its master site. A derived site might itself take the role of a master site for other derived sites.
DTD	A Document Type Definition is a formal context-free grammar for describing the structure of XML entities. The particular DTD of a given Entity can be deduced by looking at the document prolog: `<!DOCTYPE coremedia SYSTEM "http://www.coremedia.com/dtd/coremedia.dtd"` There're two ways to indicate the DTD: Either by Public or by System Identifier. The System Identifier is just that: a URL to the DTD. The Public Identifier is an SGML Legacy Concept.
Elastic Social	*CoreMedia Elastic Social* is a component of *CoreMedia CMS* that lets users engage with your website. It supports features like comments, rating, likings on your website. *Elastic Social* is integrated into *CoreMedia Studio* so editors can moderate user generated content from their common

	workplace. *Elastic Social* bases on NoSQL technology and offers nearly unlimited scalability.
EXML	EXML is an XML dialect used in former CoreMedia Studio version for the declarative development of complex Ext JS components. EXML is Jangaroo 2's equivalent to Apache Flex (formerly Adobe Flex) MXML and compiles down to ActionScript. Starting with release 1701 / Jangaroo 4, standard MXML syntax is used instead of EXML.
Folder	A folder is a resource in the CoreMedia system which can contain other resources. Conceptually, a folder corresponds to a directory in a file system.
FTL	FTL (FreeMarker Template Language) is a Java-based template technology for generating dynamic HTML pages.
gRPC	gRPC is an open source high performance Remote Procedure Call (RPC) framework.
Headless Server	CoreMedia Headless Server is a CoreMedia component introduced with CoreMedia Content Cloud which allows access to CoreMedia content as JSON through a GraphQL endpoint. The generic API allows customers to use CoreMedia CMS for headless use cases, for example delivery of pure content to Native Mobile Applications, Smartwatches/Wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of *Best current practice* (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese, or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.
IOR (Interoperable Object Reference)	A CORBA term, *Interoperable Object Reference* refers to the name with which a CORBA object can be referenced.
Jangaroo	*Jangaroo* is a JavaScript framework developed by CoreMedia that supports TypeScript (formerly MXML/ActionScript) as an input language which is compiled down to JavaScript compatible with Ext JS. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net. Jangaroo 4 is the Action-

	Script/MXML/Maven based version for CMCC 10. Since CMCC 11 (2110), Jangaroo uses TypeScript and is implemented as a *Node.js* and *npm* based set of tools.
Java Management Extensions (JMX)	The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
Locale	Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server	The *Master Live Server* is the heart of the *Content Delivery Environment*. It receives the published content from the *Content Management Server* and makes it available to the *CAE*. If you are using the *CoreMedia Multi-Master Management Extension* you may use multiple *Master Live Server* in a CoreMedia system.
Master Site	A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites.
MIME	With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multimedia emails and of web documents is standardised.
MXML	MXML is an XML dialect used by Apache Flex (formerly Adobe Flex) for the declarative specification of UI components and other objects. Up to CMCC 10 (2107), CoreMedia Studio used the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 7. Starting with CMCC 11 (2110), a new, Node.js and npm based version of Jangaroo is used that supports standard TypeScript syntax instead of MXML/ActionScript, still compiling to Ext JS 7 JavaScript.
OCI (Open Container Initiative)	The Open Container Initiative (OCI) is a Linux Foundation project that defines open industry standards for container formats and runtimes. OCI specifications ensure compatibility and interoperability between container tools, engines, and orchestration platforms like Docker and Kubernetes.

ORAS (OCI Registry As Storage)	ORAS (OCI Registry As Storage) is a tool and specification that extends OCI registries to store and distribute OCI artifacts beyond container images. It provides a standardized way for developers to push and pull arbitrary content types to and from container registries, enabling these registries to function as general artifact stores.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	With projects you can group content and manage and edit it collaboratively, setting due dates and defining to-dos. Projects are created in the Control Room and managed in project tabs.
Property	In relation to CoreMedia, properties have two different meanings: In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type. In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Replication Live Server	The aim of the *Replication Live Server* is to distribute load on different servers and to improve the robustness of the *Content Delivery Environment*. The *Replication Live Server* is a complete Content Server installation. Its content is an replicated image of the content of a *Master Live Server*. The *Replication Live Server* updates its database due to change events from the *Master Live Server*. You can connect an arbitrary number of *Replication Live Servers* to the *Master Live Server*.
Resource	A folder or a content item in the CoreMedia system.
ResourceURI	A ResourceUri uniquely identifies a page which has been or will be created by the *Active Delivery Server*. The ResourceUri consists of five components: Resource ID, Template ID, Version number, Property names and a number of key/value pairs as additional parameters.
Responsive Design	Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.

Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site. A typical site also has a master site it is derived from.
Site Folder	All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.
Site Indicator	A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.
Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Template	In CoreMedia, FreeMarker templates used for displaying content are known as Templates. OR In *Blueprint* a template is a predeveloped content structure for pages. Defined by typically an administrative user a content editor can use this template to quickly create a complete new page including, for example, navigation, predefined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes Application	The *User Changes Application* is a *Content Repository* listener, which collects all content, modified by *Studio* users. This content can then be managed in the *Control Room*, as a part of projects and workflows.
Variants	The set of all content items in a multi-site hierarchy related to each other via master references. This includes the top-level master content items themselves.
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.
Weak Links	In general *CoreMedia CMS* always guarantees link consistency. But links can be declared with the *weak* attribute, so that they are not checked during publication or withdrawal.

	Caution! Weak links may cause dead links in the live environment.
Workflow	A workflow is the defined series of tasks within an organization to produce a final outcome. Sophisticated applications allow you to define different workflows for different types of jobs. So, for example, in a publishing setting, a document might be automatically routed from writer to editor to proofreader to production. At each stage in the workflow, one individual or group is responsible for a specific task. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process.
Workflow Server	The *CoreMedia Workflow Server* is part of the Content Management Environment. It comes with predefined workflows for publication but also executes freely definable workflows.
XLIFF	XLIFF is an XML-based format, standardized by OASIS for the exchange of localizable data. An XLIFF file contains not only the text to be translated but also metadata about the text. For example, the source and target language. *CoreMedia Studio* allows you to export content items in the XLIFF format and to import the files again after translation.

Index

A

access rights, 171
actions, 204
    webflow, 207
AddOnManager, 97
Amazon EC2, 32
    (see also Amazon Linux)
Amazon Linux
    supported environments reference, **32**
Apache Tomcat (see Tomcat)
application
    architecture, 62
    plugins, **82** (see plugin)
approver-role, 305
assembling tracking information, 408
Asset Management, 413
    AMAsset, 423
    Asset Widget, 414
    blob storage, 425
    catalogPictureHandler, 418
    content types, 423
    download portal, 427
    metadata, 419
Asset Widget, 414
assets, 154, 171

B

BeansForPlugins, 95
blueprint
    Brand Blueprint, 29
    eCommerce Blueprint, 29
    removal, 105
Brand Blueprint, 29

C

CAE, 138
    Maven configuration, 140
CAE Feeder, 141
classify, 161
CleanInTranslation, 325
client code, 172, 201
    merging, 203
    performance, 203
    preview, 202
CMChannel, 173
CMCollection, 190
CMJavaScript, 201
CMLocalized, 258
CMSCSS, 201
CMTaxonomy, 161
    properties, 162
CMTeasable, 197, 259
CMViewtype, 191
common content
    types, 155
Component
    Artifact, 65
composer-role, 305
configuration
    User Changes Application, 248, 257
content, 154
    media, 160
content assets
    properties, 157
content lists, 190
content type
    CMLocalized, 258
    CMTeasable, 259
content type model, **430**
    extensions:automerge, 259
    extensions:translatable, 259
    extensions:variantStable, 260
    extensions:weakLink, 258
Content Types
    extending, 133
content types, 171
    assets, 171
    client code, 172
    navigation and page structure, 171
    technical content types, 172
content visibility, 214
ContentToTranslateItemTransformer, 264
context, 174
    determine, 174
CoreMedia Blueprint
    folder structure, 172

CoreMedia modules, 328
coremedia-application, 59
coremedia-plugin-maven-plugin, **84**
country
  locale, 240
create content
  add menu item, 339
create from template
  dialog, 344
  new template folder, 345
  template locations, 345
customizer, 62
  append bean, 63
  replace bean, 63

D

data privacy
  personal data, 141
Data Privacy
  personal data, 378
  third party, 378
data tracking, 412
Database
  Configuration, 117
  Users, 117
database
  users, 445
derived site, **239**
document type model (see content type model)
Documentation, 32
dynamic templating, 198
  add template to page, 199
  upload templates, 199

E

eCommerce Blueprint, 29
Elastic Social
  CAPTCHA, 376
  configuration, 364
  curated transfer, 376
  custom information, 370
  emails, 374
  features, 363
  mail templates, 374
end user interactions, 204
Extension, 77, 106, 114-115
  Analytics Connectors, 115
  content types, 134
  dependencies, 107
  Elastic Social, 113
Extension Tool, 77
Extensions, 76
extensions:automerge, 259
extensions:translatable, 259
extensions:variantStable, 260
extensions:weakLink, 258
External URLs, 213

F

FreeMarker, 198

G

global site manager, 247
Google Analytics
  tracking configuration, 410
group, 305
  approver-role, 305
  composer-role, 305
  publisher-role, 305
guides, 412

H

Hardware Prerequisites, 35-36
home page, **239**, 240-241, 253, 256
Hybris Commerce (see SAP Hybris Commerce)

I

IBM WebSphere Commerce, **37**
IBM WebSphere Commerce 7 Feature Pack 7
  supported environments reference, 37
IBM WebSphere Commerce 7 Feature Pack 8
  supported environments reference, 37
IBM WebSphere Commerce 8 Mod Pack 1
  supported environments reference, 37
IBM WebSphere Commerce 8 Mod Pack 4
  supported environments reference, 37
IETF BCP 47, 240
Images, 208
  configure sizes, 209
  default JPEG quality, 211
  High Resolution/Retina, 211
  linkMimeTypeMapping, 211
  MIME Type Mapping, 211
import-themes, 237

J

JavaScript code, 409

K

keywords, 161

L

language
  locale, 240
layout
  localization, 188
library
  Image Gallery, 335
link
  weak, 245, 258, 300-302
link format, 433
local site manager, **241**
locale, **240**, 246
  IETF BCP 47, 240
LocaleSettings, **246**, 246-247
localization, **240**
localized site, 240

M

mail templates, 374
MailTemplateService, 374
master site, **240**
  root-master site, **240**
Maven
  changing groupId, 116
  coremedia-application, 76
  Extension, 106
  settings.xml, 33
media content, 160
MongoDB
  supported environments reference, **32**
multi-site, 239
  administration, 246
  CMLocalized, 258
  CMTeasable, 259
  content types, **258**
  derived site, **239**, 243
  global site manager, 247
  groups, 247
  local site manager, **241**
  master site, **240**, 243
  permissions, 247
  root-master site, **240**
  site, **240**
  SiteModel, **252**, 306
  SitesService, **252**
  structure, 243
  translation manager role, **242**, 247, 306-307
  variants, **242**
MySQL
  supported environments reference, **32**

N

navigation, 173
navigation and page structure, 171

O

Open Street Map, 346, 378

P

page grid, 178
  configure new layout, 182
  editor, 179
  incompatible changes, 181
  inheriting placements, 179
  layout locations, 182
  lock placements, 179
  predefined layout, 181
Pendo, 412
personal data, 141, 378
placement, 178
placement editor, 181
placements
  localization, 188
plugin
  add-on, 97
  AddOnManager, 97
  application, **82**
  application context, 87
    bean, 87, 95
    property, 89
  bundle, 101
  class loader, 89
  coremedia-plugin-maven-plugin, 84
  dependency, 89, 94
  descriptor, 82, 101
  distribution, 101
  extension, 85

extension point, **86**
resource, **87**
files
Java, 83
Java, **83**
application context, 83, 87
class loader, 83, 89
coremedia-plugin-maven-plugin, 84
dependencies, 83
extension points, **83**
files, 83
Maven, 84
plugin.properties, 83-84
structure, 83
usage, 93
plugin-descriptors.json, 103
plugins.json, 103
plugins.required-plugins, 93
property, 83
add-on-for, 83
configuration-class, 83
dependencies, 83
id, 83
independent, 83
provider, 83
version, 83
structure
Java, 83
Studio, **98**
limitations, **99**
setup, **98**
workspace, **100**
tooling
Maven, 84
plugins (see plugin)
actuator, 94
plugins.required-plugins, 93
predefined user, 439
predefined workflows, 304
publisher-role, 305

Q

quick start, **38**

R

rights concept, 171
robots.txt, 216
example configuration, 218
RobotsHandler, 217

S

Salesforce Marketing Cloud
uploading content, 355
Salesforce Marketing Cloud integration, 412
SAP Hybris Commerce, **37**
search, 224
search landing pages, 236
keywords, 236
ServerExport, **260**
ServerImport, **260**
settings
linked, 176
local, 176
settings.xml, 33
SignCookie
RSA Key, 377
site, **240**
derived site, **239**, 243
global site manager, 247
home page, **239**, 240-241, 244, 253, 256
interdependence, **244**
local site manager, **241**
locale, **240**, 246
LocaleSettings, **246**, 246-247
localized site, 240
master site, **240**, 243
multi-site, 239
root-master site, **240**
site folder, **240**, 244, 256
site id, **241**, 255
site indicator, 240, **241**, 244, 248, 251, 253-256
site manager group, **241**, 248, 253-254, 306-307
site name, **242**, 255
SiteModel, 248, 251-252, **252**, 306
SitesService, **252**
translation manager role, **242**, 247, 255, 306-307
translation workflow robot user, 248
variants, **242**
site manager group, **241**, 248, 253-254, 306-307
sitemap, 215, 220
maximum number of URLs, 221
SiteModel, **252**, 306
SitesService, **252**
Software Prerequisites, 36
Solr, 225
SpringAwareLongAction, 264

Studio, 136
bookmarks, 336
create content, 338
create from template, 344
external preview, 336
library, 334
Open Street Map, 346
plugin, 98, 136
application plugin, 98
EditorPlugin, 98
StudioPlugin, 98
types, 98
plugins, 329
query form, 329
settings, 337
site selection, 347
upload content to Salesforce Marketing Cloud, 355
upload files, 348
Studio enhancements, 329
Studio integration, 410
suggestion strategy, 166
supported environments, **32**
Amazon Linux, 32
databases, 32
directory services, 32
HCL Commerce 7, 37
HCL Commerce 8, 37
HCL Commerce 9, 37
Java, 32
MongoDB, 32
MySQL, 32
operating systems, 32
servlet container, 32
Tomcat, 32
web browsers, 32
synchronization, **242**

T

tag management
tag manager, 238
taxonomies, 161
as conditions for dynamic lists, 161
hierarchical organization, 161
implement new suggestion strategy, 166
location, 162
related content, 161
site specific, 167
taxonomy editor, 163
taxonomy resolver strategy, 165
teaser management, 196
technical content types, 172
Test System Setup
preconfigured quick start, **37**
third party, 378
(see also Data Privacy)
Tomcat
supported environments reference, **32**
topic pages, 231
configuration, 231
managed, 232
taxonomy, 231
tracking
configuration, 407
identifying, 407
sent data, 407
translation, 240, **242**
configuration, 283
customization, 288
Studio, 288
UI, **288**
workflow, **306**, 309
workflow action, 309
XLIFF, **262**, 263, 284
Configuration, 284
Translation Item, **264**
translation manager, **242**, 247, 255, 306-307
translation manager role, **242**, 247, 255, 306-307
translation workflow robot user, 248

U

upload files
configuration, 349
URLs, 211
User Changes Application, 22
configuration, 248, 257
users
predefined, 439

V

validFrom, 214
Vanity URLs, 212
variants, **242**
view repositories, 200
view type selector, 192
view types, 191
localization, 189, 193

views, 408
viewType, 159
visibility, 214

W

weak link, 245, 258, 300-302
web presence, **243**
Webflow actions, 207
website
  navigation, 173
  page assembly, 177
  settings, 175
  structure, 177
website search, 224
WebSphere Commerce (see IBM WebSphere Commerce)
workflow
  action, 309
  publication, 304
  translation, 306, 309
workflow action, 309
  AutoMergeSyncAction, 320-321
  AutoMergeTranslationAction, 318, 320
  CleanInTranslationFinalAction, **325**, 326
  CompleteTranslationAction, 321, 323
  CreateTranslationTreeData, 312
  CreateTranslationTreeDataAction, 313
  ExtractPerformerAction, 318
  FilterDerivedContentsAction, 315
  GetDerivedContentsAction, 310, 312-313
  GetMultiSiteManagerGroupAction, 316-317
  GetSiteManagerGroupAction, 315
  RollbackTranslationAction, 323-324
Workspace
  Configuration, 116
  Structure, 75

X

XLIFF, **262**, 263, 284
  Configuration, 284
  properties, 284
  translatableExpressions, 284
  Translation Item, **264**
  translation unit, 263
  XliffExporter, 265
  XliffExportOptions, 266
  XliffImporter, 266
XliffExporter, 265
XliffExportOptions, 266
XliffImporter, 266
XML Localization Interchange File Format, **262**, 263