CoreMedia Digital Experience Platform 8
//Version 7.5.45-10

CoreMedia Studio Manual

CoreMedia Studio Manual

Copyright CoreMedia AG © 2015

CoreMedia AG

Ludwig-Erhard-Straße 18

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 AG.

Germany

Alle Rechte vorbehalten. CoreMedia und weitere im Text erwähnte CoreMedia Produkte sowie die entsprechenden Logos sind Marken oder eingetragene Marken der CoreMedia AG 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 AG reproduziert oder vervielfältigt werden. Unberührt hiervon bleiben die gesetzlich erlaubten Nutzungsarten nach dem UrhG.

Licenses and Trademarks

All trademarks acknowledged.
07.Mar 2017

1. Introduction 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 5
1.3.3. Documentation 6
1.3.4. CoreMedia Training 8
1.3.5. CoreMedia Support 9
1.4. Change Chapter 12
2. Overview 13
2.1. Architecture 14
2.2. Technologies 16
3. Deployment 18
3.1. Connecting to the Repository 19
3.2. Control Room Configuration 20
3.3. Basic Preview Configuration 21
3.4. Advanced Preview Configuration 22
3.5. Development Setup 23
4. Quick Start 24
5. Concepts and Technology 26
5.1. Ext JS Primer 27
5.1.1. Components 28
5.1.2. Declarative UI (Ext JSON) 29
5.1.3. Component Plugins 30
5.1.4. Actions 31
5.2. Ext JS with ActionScript and EXML 32
5.3. Client-side Model 37
5.3.1. Beans 38
5.3.2. Remote Beans 40
5.3.3. Issues 41
5.3.4. Operation Results 42
5.3.5. Model Beans for Custom Components 43
5.3.6. Value Expressions 44
5.4. Remote CoreMedia Objects 51
5.4.1. Connection and Services 51
5.4.2. Content 52
5.4.3. Workflow 53
5.4.4. Structs 54
5.4.5. Types and Property Descriptors 56
5.4.6. Concurrency 56

5.5. Studio Component IoC ... 57
5.5.1. Motivation ... 57
5.5.2. Inversion of Control ... 57
5.5.3. Annotations of Context Consumer and Context Provider ... 58
5.6. Web Application Structure ... 60
5.7. Localization ... 61
5.8. Multi-Site and Localization Management ... 63
5.9. Further Reading ... 64
6. Using the Development Environment ... 65
6.1. Configuring Connections ... 66
6.2. Build Process ... 67
6.3. IDE Support ... 69
6.4. Debugging ... 70
6.4.1. Browser Developer Tools ... 70
6.4.2. Ext JS `debug.js` ... 72
6.4.3. Illuminations ... 73
6.4.4. Tracing Memory Leaks ... 75
7. Customizing CoreMedia Studio ... 83
7.1. Studio Plugins ... 84
7.2. Localizing Labels ... 94
7.3. Document Type Model ... 97
7.3.1. Localizing Types and Fields ... 97
7.3.2. Defining Content Type Icons ... 98
7.3.3. Customizing Document Forms ... 101
7.3.4. Image Cropping and Image Transformation ... 108
7.3.5. Enabling Image Map Editing ... 111
7.3.6. Disabling Preview for Specific Document Types ... 112
7.3.7. Configuring Translation Support ... 112
7.3.8. Excluding Document Types from the Library ... 113
7.3.9. Client-side initialization of new Documents ... 114
7.4. Customizing Property Fields ... 115
7.4.1. Conventions for Property Fields ... 115
7.4.2. Standard Component StringPropertyField ... 116
7.4.3. Compound Field ... 119
7.4.4. Complex Setups ... 122
7.4.5. Customizing RichText Property Fields ... 122
7.5. Upgrading the CKEditor ... 133

7.5.1. Upgrading RichTextArea Plugins from CKEditor 3 to 4 133
7.5.2. Migrating Richtext Editor Dialogs 134
7.5.3. CKEditor plugins available 135
7.6. Coupling Studio and Embedded Preview 138
7.6.1. Built-in Processing of Content and Property Metadata 138
7.6.2. Using the Preview Metadata Service 138
7.7. Storing Preferences 142
7.8. Customizing Studio using Component IoC 143
7.8.1. Content Actions 143
7.8.2. Example: Add a disapprove button to the actions toolbar 143
7.8.3. Studio Component Map 144
7.9. Customizing Central Toolbars 145
7.9.1. Adding buttons to the Favorites Toolbar 145
7.9.2. Providing default Search Folders 146
7.9.3. Adding a Button with a Custom Action 148
7.9.4. Adding a Button to the Apps Menu 149
7.9.5. Adding Disapprove Buttons 150
7.10. Inheritance of Property Values 151
7.11. Customizing the Library Window 152
7.11.1. Defining List View Columns in Repository Mode 152
7.11.2. Defining Additional Data Fields for List Views 153
7.11.3. Defining List View Columns in Search Mode 154
7.11.4. Configuring the Thumbnail View 155
7.11.5. Adding Search Filters 155
7.11.6. Make Columns Sortable in Search and Repository View 158
7.12. Work Area Tabs 160
7.12.1. Configuring a Work Area Tab 160
7.12.2. Configure an Action to Open a Work Area Tab 160
7.12.3. Configure a Singleton Work Area Tab 161
7.12.4. Storing the State of a Work Area Tab 161
7.12.5. Customizing the Start up Behavior 162
7.12.6. Customizing the Work Area Tab Context Menu 164

7.13. Dashboard 166
7.13.1. Concepts 166
7.13.2. Defining the Dashboard 167
7.13.3. Predefined Widget Types 169
7.13.4. Adding Custom Widget Types 170
7.14. Configuring MIME Types 175
7.15. Server-Side Content Processing 176
7.15.1. Validators 176
7.15.2. Intercepting Write Requests 180
7.15.3. Immediate Validation 183
7.15.4. Post-processing Write Requests 184
7.16. Available Locales 186
7.17. Notifications 187
7.17.1. Configure Notifications 187
7.17.2. Adding Custom Notifications 187
7.17.3. Creating Notifications (Server Side) 188
7.17.4. Displaying Notifications (Client Side) 188
8. Security 191
8.1. Preview Integration 192
8.2. Content Security Policy 193
8.3. Single Sign On Integration 196
8.4. Auto Logout 201
8.5. Logging 202
Glossary 205
Index 212

List of Figures

2.1. Architecture of CoreMedia Studio 14
2.2. Runtime components 15
5.1. Ext JSON 28
5.2. EXML compared to Ext JSON 28
6.1. Studio project within the Project workspace in IntelliJ Idea 69
6.2. Firebug: console 72
6.3. Ext component tree 72
6.4. Illuminations: objects 74
6.5. Illuminations: methods 74
6.6. Illuminations: highlighting 75
6.7. Illuminations: inspect 75
6.8. Google Chrome's Developer Tools Support Comparing Heap Snapshots 81
7.1. plugin structure 85
7.2. Document form with content and metadata properties 102
7.3. Document form with a collapsible form panel 107
7.4. Premular and Actions Toolbar 144
7.5. Collection View 144
7.6. Apps Menu 150
7.7. Dashboard UML overview 168

List of Tables

1.1. Typographic conventions 3
1.2. Pictographs 3
1.3. CoreMedia manuals 6
1.4. Log files check list 10
1.5. Changes 12
7.1. Content Type Icons 98
7.2. Property Fields 104
7.3. `ImageEditorPropertyField` Configuration Settings 108
7.4. CKEditor plugins loaded by default 135
7.5. CM richtext plugins loaded by default 136
7.6. Predefined widget types 169
7.7. Selected predefined validators available in CoreMedia Studio 176

List of Examples

5.1. Ext JSON 27
5.2. Viewport definition as Ext JSON 29
5.3. Plugin usage in Ext JSON 31
5.4. Component instantiation using Ext JSON 35
5.5. Component instantiation using typed setters 35
5.6. Component instantiation using typed wrapper 35
5.7. Updating multiple bean properties 39
5.8. Model bean factory method 43
5.9. Model bean access 44
5.10. Adding a listener and initializing 46
5.11. Creating a property path expression 47
5.12. The valueExpression EXML element 48
5.13. Creating a function value expression 48
5.14. Creating a value expression from a private function 48
5.15. Creating a value expression from a static function 49
5.16. Manual dependency tracking 49
5.17. The bindPropertyPlugin EXML element 49
5.18. Property paths into struct 55
5.19. Adding struct properties 55
6.1. Detecting public API violations 68
7.1. Adding a plugin rule to customize the actions toolbar 88
7.2. Adding a separator and a button with a custom action to a toolbar 89
7.3. Adding a plugin rule to customize all LinkList property field toolbars 90
7.4. Using `<ui:nestedRulesPlugin>` to customize a sub-component using its container's API 90
7.5. Using `<ui:nestedRulesPlugin>` to customize a sub-component 91
7.6. Registering a plugin 92
7.7. Loading an external script 93
7.8. Loading an external style sheet 93
7.9. Adding a search button 95
7.10. Example property file 95
7.11. Overriding properties 96
7.12. Localizing document types 97
7.13. Article form 102
7.14. Configuring the Image Editor 108

7.15. Configuring the variants 109
7.16. Configuring an Image Map Editor 111
7.17. Configuring a validator for image maps 111
7.18. Defining document types without preview 112
7.19. Blueprint source language document resolver 112
7.20. Configuring a source language document resolver 113
7.21. Defining excluded document types 113
7.22. Defining excluded document types in EXML 114
7.23. Defining a content initializer 114
7.24. Custom property field 116
7.25. Using a base class method 122
7.26. Inline images in richtext 123
7.27. Configuring the rich text symbol mapping 127
7.28. Customizing the rich text editor toolbar 129
7.29. Adding a custom icon to the rich text editor toolbar 129
7.30. Adding resource path to `pom.xml` 129
7.31. Customizing the CKEditor 130
7.32. Adding a search for documents to be published 146
7.33. Adding a custom search folder 147
7.34. Creating a custom action 148
7.35. Creating a custom action configuration class 149
7.36. Using a custom action 149
7.37. Adding disapprove action using enableDisapprovePlu-
gin 150
7.38. Configuring Property Inheritance 151
7.39. Defining list view columns in the repository mode 152
7.40. Defining list view fields 153
7.41. Defining list view columns in the search mode 154
7.42. Configuring the thumbnail view 155
7.43. Two additional attributes for sorting. 158
7.44. Optional `extendOrderBy` Attribute for sort by more than one
column. 159
7.45. Optional `sortDirection` Attribute to enable only one sort
direction. 159
7.46. `defaultSortColumn` Attribute to configure one column as the
default for sorting. 159
7.47. Adding a button to open a tab 160
7.48. Adding a button to open a browser tab 161
7.49. Base class for browser tab 162
7.50. Dashboard Configuration 167
7.51. Fixed Search widget Configuration 169

7.52. Simple Search Widget Configuration 170
7.53. Simple Search Widget Type 171
7.54. Simple Search Widget Component 171
7.55. Simple Search widget Type with Editor Component 172
7.56. Simple Search widget Editor Component 173
7.57. widget State Class for Simple Search widget 174
7.58. Configuring MIME types 175
7.59. Implementing a property validator 177
7.60. Configuring a property validator 178
7.61. Implementing a content validator 178
7.62. Configuring a content validator 179
7.63. Configuring validator messages 179
7.64. Defining a Write Interceptor 182
7.65. Configuring a Write Interceptor 182
7.66. Configuring Immediate Validation 183
8.1. Import base context 196
8.2. Spring Security context 197
8.3. Delegating entry point 198
8.4. Logout filter 198
8.5. User finder 200
8.6. Enable user finder 200
8.7. Example Output 202
8.8. Marker Hierarchy 202
8.9. Configure Access Log 202
8.10. Configure Security Log 203
8.11. Configure Default Log 203
8.12. Configure Logger 204
8.13. Suppress Security Logging 204

1. Introduction

This manual describes the configuration of and development with *CoreMedia Studio*. You will learn, for example, how to add your own Favorites, how to change or add labels, or how to customize forms.

- Chapter 2, *Overview* [13] gives a short overview of *CoreMedia Studio*.
- Chapter 3, *Deployment* [18] describes how to deploy *CoreMedia Studio* into different servlet containers.
- Chapter 4, *Quick Start* [24] describes how to set up a development workspace that is ready for *CoreMedia Studio* development.
- Chapter 5, *Concepts and Technology* [26] gives an overview of the concepts and technologies used by *CoreMedia Studio*. It is not a prerequisite for the following chapters, but will give you valuable insight into the underlying concepts.
- Chapter 6, *Using the Development Environment* [65] introduces the build tools and processes that are recommended for the development of *CoreMedia Studio*.
- Chapter 7, *Customizing CoreMedia Studio* [83] explains specific customizations of *CoreMedia Studio*.

Since version 1.3, the *CoreMedia Studio* API is marked *final*, meaning that changes and extensions to the API are guaranteed to be backwards compatible. Any changes to the API are however described in the release notes, and it is recommended to consult these when upgrading to a newer version, so that you can benefit from added functionality or more convenient or powerful ways to make use of certain features.

1.1 Audience

This manual is intended for developers who want to customize *CoreMedia Studio*. You should know the basics of *CoreMedia CMS*. Knowledge about the *Unified API* is particularly helpful. You should also have a solid understanding of Maven, Action-Script, JavaScript and Ext JS.

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:

Table 1.1. Typographic conventions

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`
Mention of other manuals	Square Brackets	See the [Studio Developer Manual] for more information.

In addition, these symbols can mark single paragraphs:

Table 1.2. Pictographs

Pictograph	Description
	Tip: This denotes a best practice or a recommendation.
	Warning: Please pay special attention to the text.

Pictograph	Description
	Danger: The violation of these rules causes severe damage.

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.

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" [5] describes where to find the download of the software.
- Section 1.3.3, "Documentation" [6] 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" [8] describes CoreMedia training. This includes the training calendar,the curriculum and certification information.
- Section 1.3.5, "CoreMedia Support" [9] 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" [9]) 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:

http://releases.coremedia.com/dxp8

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

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 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 CoreMedia Digital Experience Platform 8 Developer Manual.

License files

You need license files to run the CoreMedia system. Contact the support (see Section 1.3.5, “CoreMedia Support” [9]) to get your licences.

1.3.3 Documentation

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

http://documentation.coremedia.com/dxp8

The manuals have the following content and use cases:

Table 1.3. CoreMedia manuals

Manual	Audience	Content
CoreMedia Utilized Open-Source Software	Developers, architects, administrators	This manual lists the third-party software used by CoreMedia and lists, when required, the licence texts.
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.
Studio User Manual, English	Editors	This manual describes the usage of *CoreMedia Studio* for editorial and administrative work. It also describes the usage of the *Adaptive Personalization* and *Elastic Social* GUI that are integrated into *Studio*.

Manual	Audience	Content
LiveContext for IBM WebSphere Manual	Developers, architects, administrators	This manual gives an overview over the structure and features of CoreMedia LiveContext. It describes the integration with the IBM WebSphere Commerce system, 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.
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 or the usage of the watchdog component.
Adaptive Personalization Manual	Developers, architects, administrators	This manual describes the configuration of and development with *Adaptive Personalization*, the CoreMedia module for personalized websites. You will learn how to configure the GUI used in *CoreMedia Studio*, how to use predefined contexts and how to develop your own extensions.
Analytics Connectors Manual	Developers, architects, administrators	This manual describes how you can connect your CoreMedia website with external analytic services, such as Google Analytics.
Content Application Developer Manual	Developers, architects	This manual describes concepts and development of the *Content Application Engine (CAE)*. You will learn how to write JSP or 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
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.
Importer Manual	Developers, architects	This manual describes the structure of the internal CoreMedia XML format used for storing data, how you set up an *Importer* application and how you define the transformations that convert your content into CoreMedia content.
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*.
Site Manager Developer Manual	Developers, architects, administrators	This manual describes the configuration and customization of *Site Manager*, the Java based stand-alone application for administrative tasks. You will learn how to configure the *Site Manager* with property files and XML files and how to develop your own extensions using the *Site Manager API*.
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 development environment and how to customize *Studio* to your needs.
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.
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.

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 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:

http://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:

http://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. The location where component log output is written is specified in its `< appName>-logback.xml` file.

Which Log File?

Mostly at least two CoreMedia components are involved in errors. In most cases, the *Content Server* log files in `coremedia.log` files together with the log file from the client. If you are able locate the problem exactly, solving the problem becomes much easier.

Where do I Find the Log Files?

By default, log files can be found in the CoreMedia component's installation directory in `/var/logs` or for web applications in the `logs/` directory of the servlet container . See the "Logging" chapter of the [Operations Basics Manual] for details.

Table 1.4. Log files check list

Component	Problem	Log files
CoreMedia Studio	general	`CoreMedia-Studio.log` `coremedia.log`

CoreMedia Editor	general	`editor.log`
		`coremedia.log`
		`workflowserver.log`
		`capclient.properties`

	check-in/check-out	`editor.log`
		`coremedia.log`
		`workflowserver.log`
		`capclient.properties`

	publication or preview	`coremedia.log` (*Content Management Server*) `coremedia.log` (Master Live Server)

Component	Problem	Log files
		workflowserver.log capclient.properties
	import	importer.log coremedia.log capclient.properties
	workflow	editor.log workflow.log coremedia.log capclient.properties
	spell check	editor.log MS Office version details coremedia.log
	licenses	coremedia.log (*Content Management Server*) coremedia.log (Master Live Server)
Server and client	communication errors	editor.log coremedia.log (*Content Management Server*) coremedia.log (Master Live Server) *.jpif files
	preview not running	coremedia.log (content server) preview.log
	website not running	coremedia.log (*Content Management Server*) coremedia.log (Master Live Server) *coremedia.log* (*Replication Live Server*) Blueprint.log capclient.properties license.zip
Server	not starting	coremedia.log (*Content Management Server*) coremedia.log (Master Live Server) coremedia.log (*Replication Live Server*) capclient.properties license.zip

1.4 Change Chapter

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

Table 1.5. Changes

Section	Version	Description
Chapter 8, *Security* [191]	7.5.24	Add chapter about security and Content Security Policy in particular.
Removed in 7.5.34	7.5.26	Added new section about NotificationService configuration.
Section 7.4.5, "Customizing RichText Property Fields" [122]	7.5.34	Removed section about NotificationsService configuration, added broader section about in-memory replacement of several MongoDB-based features to the CoreMedia Manual
Section 3.2, "Control Room Configuration" [20]	7.5.41	Removed properties `controlroom.jdbc.driver`, `controlroom.jdbc.url`, `controlroom.jdbc.user` and `controlroom.jdbc.password`. *Control Room* no longer supports *IBM DB2* for persisting collaboration data. See [CoreMedia DXP 8 Manual], Section "In-Memory Replacement for MongoDB-Based Services".
???	7.1.12	Added description of th `onlyIf` Plugin

2. Overview

CoreMedia Studio is a web application that is in the center of your web activities. It gives you complete control over context's determinants and lets you easily create compelling and engaging content experiences. Technically, *CoreMedia Studio* is a single-page Ajax application, using a REST based network protocol for communication.

2.1 Architecture

Figure 2.1, "Architecture of CoreMedia Studio" [14] shows the architecture of *CoreMedia Studio*. The top-level layer comprises content editing applications such as the *CoreMedia Studio* core application and its plugins. *CoreMedia Blueprint* defines several plugins, showcasing *Studio*'s various extension points.

Editing applications are built on a layer of editing components that deal with CoreMedia content objects. Editing components are built on the UI Toolkit layer which provides generic components for building rich internet applications. On this layer, components can be implemented in ActionScript 3 or declared in EXML and then compiled down to Ext JS. UI components separate layout, model and functionality according to the MVC paradigm. Models that are backed by server-side data are implemented as client-side beans that fetch the requested values via REST. UI components offer localization support. The lower level layers comprise the REST API of the *CoreMedia CMS*.

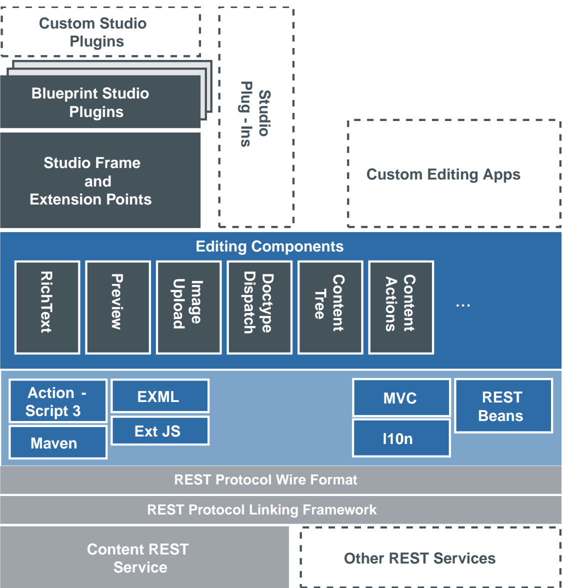


Figure 2.1. Architecture of CoreMedia Studio

As shown below, the *CoreMedia Studio* web application serves static and dynamic resources. The static resources are those that define the client-side UI structure (HTML and JavaScript) and the client-side layout (CSS and images). The dynamic resources can be accessed via the Content REST Service. When you start *CoreMedia Studio* from your browser, it loads the static resources and initializes the Ext JS UI component tree, Studio plugins and model beans. Model beans issue Ajax requests

to access the Content REST Service, which is the interface to the CoreMedia back-end systems, and load data from the returned JSON objects.

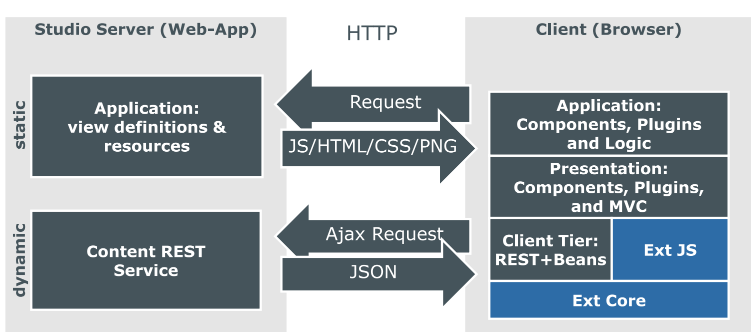


Figure 2.2. Runtime components

2.2 Technologies

This section gives you a brief overview of *CoreMedia Studio*'s underlying technologies. These are the ActionScript to JavaScript framework *Jangaroo*, including its declarative language EXML, the JavaScript UI framework Ext JS, and the CKEditor for richtext editing.

Jangaroo

Jangaroo is CoreMedia's open source JavaScript framework, supporting ActionScript as a source language, which is then compiled down to JavaScript.

A detailed description of the Jangaroo compiler `jooc` is given on the Jangaroo website https://github.com/CoreMedia/jangaroo-tools/wiki/Compiler. The compiler is integrated into the Maven build process of the CoreMedia CMS distribution, so in a Maven-based project you should never need to invoke the compiler directly.

EXML

EXML is an XML dialect developed by CoreMedia that supports the declarative development of complex Ext JS components. It is Jangaroo's equivalent to Adobe Flex MXML. Jangaroo compiles EXML down to ActionScript.

The rationale behind this is to benefit from static typing (provided by XML) when developing dynamically typed Ext JS components. To this end, Jangaroo provides an IntelliJ IDEA plugin supporting documentation lookup of EXML tags, navigation between EXML components, and compilation of EXML to ActionScript. See Section 5.2, “Ext JS with ActionScript and EXML” [32] for details.

Ext JS

Ext JS is a cross-browser rich internet application framework developed by Sencha Inc. It offers JavaScript UI widgets and client side MVC. To this end, Ext JS provides components, actions and data abstractions. Components can be customized by plugins. Component trees are described in JSON notation. Ext JS defines the JavaScript properties `xtype` and `ptype` to distinguish between components and plugins.

In short, Ext JS has the following features:

- clean object-oriented design,
- hierarchical component architecture (component tree),
- large UI library with mature widgets, especially mature business components (`Store` abstraction, `DataGrid`),
- built-in layout management,

- good drag and drop support with sophisticated visual feedback,
- declarative UI description language (JSON).

Ext JS also provides a rich set of utility functions to deal with components or plain JavaScript objects and functions. The complete Ext JS documentation can be found on http://www.sencha.com/learn/Learn_About_the_Ext_JavaScript_Library.

The *CoreMedia Studio* builds on Ext JS 3: http://www.sencha.com/products/extjs3/.

CKEditor

The CKEditor is a browser based open source WYSIWYG text editor (http://ckeditor.com/). Common editing features found on desktop editing applications like Microsoft Word and OpenOffice are brought to the web browser by using CKEditor.

The CKEditor is the default editor for `richTextPropertyField` in a document form. Thereby the CKEditor is encapsulated by the wrapper `richTextArea`, making it possible to use the CKEditor with the same look and feel as the rest of the Ext JS based *CoreMedia Studio*. The wrapper takes the editing area of the CKEditor and adds Ext JS based dialogs.

The CKEditor can be extended by custom plugins. *CoreMedia Studio* comes with several extra plugins supporting CoreMedia richtext specific formats and operations. Likewise, you can add more plugins to integrate your own functionality. See Section 7.4.5, “Customizing RichText Property Fields” [122] for more on this topic.

3. Deployment

In this chapter you will get to know how to deploy *CoreMedia Studio* to different servlet containers.

Perform all configurations of *CoreMedia Studio* described in this chapter in the module `studio-webapp` of *CoreMedia Blueprint* workspace before building or later on during deployment of *Studio*.

3.1 Connecting to the Repository

CoreMedia Studio needs to know the URL of the *Content Server* to connect to and the URL of the preview server. To this end, adjust the `repository.url` property in `WEB-INF/application.properties` of the *Studio* web application and let it point to your *Content Management Server*.

```
repository.url=http://<Host>:<Port>/coremedia/ior
```

Alternatively, you may configure the URL to connect to by modifying the `contentserver.*` properties in the same file.

```
contentserver.host=localhost
contentserver.port=44441
```

CoreMedia Studio offers connectivity to the *CoreMedia Workflow Server*. Therefore, a *Workflow Server* has to run when starting *CoreMedia Studio*. If this is not desired, you should set the property `repository.workflow` in the file `WEB-INF/application.properties` to `false`.

```
repository.workflow=false
```

Studio supports "Simple Publication" and "Two Step Publication" publication workflows. To use these workflows, upload the workflow definitions `studio-simple-publication.xml` and `studio-two-step-publication.xml` to the *Workflow Server* with the `cm upload` tool. See section "Predefined Publication Workflows" of the [CoreMedia Digital Experience Platform 8 Developer Manual] for more information on these workflows.

3.2 Control Room Configuration

The Control Room consists of the following components:

- Control Room Plugin is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
- User Changes web application is a repository listener, which collects content modified by a user working with *Studio*. To this end, the modified content can be managed in the Control Room plugin as projects, shared and used in workflows, for example.
- Extensions of the *Workflow Server* - Control Room comes with adapted workflow definitions that among other things persist finished workflows.

> Perform all configurations concerning the *User Changes* web application in the module `user-changes-webapp` in *CoreMedia Blueprint* before building or later on during deployment of the *User Changes* web application.

The Control Room stores and manages contents sets and finished workflows, commonly specified as collaboration data, by connecting to a *MongoDB*. Therefore, a *MongoDB* installation is necessary for utilizing *CoreMedia Studio* with the Control Room.

- **Deploying Control Room with MongoDB Database**

  See [CoreMedia Operations Basics] on how to deploy Control Room with MongoDB.

- **Saving Control Room data in memory**

  See [CoreMedia DXP 8 Manual], Section "In-Memory Replacement for MongoDB-Based Services".

3.3 Basic Preview Configuration

Since the preview *CAE* web application and *CoreMedia Studio* communicate via an internal messaging system, they can be deployed either in the same servlet container or in separate ones. In both scenarios the configuration of *Studio* is done via the `studio.previewUrlPrefix` key of the `WEB-INF/application.properties` property file.

Please note that running the preview in the same container under the same origin (the origin includes protocol, host, port) is not recommended in a production environment. Security and performance is increased significantly when the two applications are deployed independently from each other.

If *CoreMedia Studio* and the preview are deployed together in one servlet container, the property `studio.previewUrlPrefix` is set to the path of the preview's Spring servlet. For a project based on the standard *CoreMedia DXP 8*, this would be:

```
studio.previewUrlPrefix=/blueprint/servlet
```

If *Studio* and the preview are deployed independently, the aforementioned property must be set to the absolute URL of the preview web application. In a *Blueprint* related project, this could be:

```
studio.previewUrlPrefix=http://localhost:40081/blueprint/servlet
```

3.4 Advanced Preview Configuration

In case of a separate deployment, security can be improved even further by configuring a whitelist of valid Studio URLs in the preview *CAE* web application. This is done via the `pbe.studioUrlWhitelist` property in the `WEB-INF/application.properties` file of the preview *CAE* web application. If left empty, all URLs are considered valid.

In the opposite direction, it is possible to configure a whitelist of valid preview URLs in *Studio* (including protocol, host and port). This is done via the `studio.previewUrlWhitelist` property in the `WEB-INF/application.properties` file of the Studio web application. If left empty, the *only* valid preview URL is the one that is determined based on the `studio.previewUrlPrefix` property (that is, the given preview URL or the Studio URL itself if a relative preview URL prefix is given). When configuring valid preview URLs it is possible to use wildcards as in the following example:

```
studio.previewUrlWhitelist=https://host1:port1, https://host2:port2,
 http://localhost*, *company.com
```

Note, that once a preview URL whitelist is configured, *CoreMedia Studio* has no chance to set a target origin in outgoing messages anymore. Be aware that this is a minor security drawback.

In case of a separate deployment, enabling Elastic Social tenants in the embedded preview requires including a placeholder in the aforementioned `studio.previewUrlPrefix` key of the property file `WEB-INF/application.properties`. The CoreMedia Studio then replaces the token with the current tenant. In a *CoreMedia Blueprint* related project, this could be:

```
studio.previewUrlPrefix=http://{0}.localhost:40081/blueprint/servlet
```

3.5 Development Setup

During development, it may be convenient to specify the property `contentserver.host` and optionally the property `contentserver.port` for connecting to the *Content Server* as system properties on the command line when starting the Studio servlet container.

4. Quick Start

This chapter presents the basic steps to set up a *CoreMedia Studio* development environment quickly.

Setting Up the Workspace

CoreMedia Digital Experience Platform 8 comes with a fully preconfigured, Maven-based development workspace. Details on how to get and set up your development environment are described in the [CoreMedia Digital Experience Platform 8 Developer Manual] You will find guidance for the following topics:

1. Required third-party software, such as Maven.
2. Getting *CoreMedia Project*.
3. Installing *CoreMedia Project*.
4. Configuring all components.
5. Building the workspace.
6. Starting the components.

The recommended development setup is to use the `studio` module in the workspace, which is placed under `modules/studio`.

Setting Up the IDE

Once you have set up the workspace, you may configure your IDE as described in Section 6.3, "IDE Support" [69]. If you are using IntelliJ IDEA, this means that you need to get the plugin `Jangaroo 0.9`, which you can install via IDEA's plugin manager. There are other Jangaroo plugins available in that dialog ("Jangaroo Language", "Jangaroo EXML", and "Jangaroo"), which are intended for older releases and must not be activated together with the current plugin `Jangaroo 0.9`.

Building

A detailed description on how to build the *CoreMedia Studio* module can be found in Chapter 6, *Using the Development Environment* [65]. If you are using IntelliJ IDEA and the IDE is set up correctly, you can build the whole project via Maven from within the IDE. If you prefer building from the command line, you can do it by using standard Maven commands like

```
mvn clean install -DskipTests
```

The *CoreMedia Studio* application can then be launched by changing into the `modules/studio/studio-webapp` directory and using the following command:

```
mvn tomcat7:run
```

More details on how to build and start *CoreMedia Studio*, as well as how to run tests with it, are described in Section 6.2, "Build Process" [67].

Debugging

Firebug is the recommended JavaScript debugger. To facilitate debugging, single class JavaScript files of the *CoreMedia Studio* components can be loaded by attaching

```
#joo.debug
```

to the *CoreMedia Studio* URL. An Ext JS debugger allowing component inspection can be invoked by executing the following JavaScript statement:

```
Ext.log('')
```

Refer to Section 6.4, "Debugging" [70] for more details on how to debug.

If you have finished these steps you are ready to customize *CoreMedia Studio* as described in Chapter 7, *Customizing CoreMedia Studio* [83].

5. Concepts and Technology

This chapter describes the basic concepts and technologies on a more detailed level than in the Overview chapter. It is not a prerequisite for the subsequent chapters, but it will give you valuable insight into the underlying concepts.

5.1 Ext JS Primer

Ext JS is a JavaScript library for building interactive web applications. It provides a set of UI widgets like panels, input fields or toolbars and cross-browser abstractions (Ext core).

CoreMedia Studio uses Ext JS on the client side. With plain Ext JS, widgets are defined in JSON format as displayed in the following example:

Example 5.1. Ext JSON

```
{
 xtype: "panel",
 title: "Teaser Properties",
 items: [
  {
    xtype:
     "com.coremedia.cms.editor.sdk.config.stringPropertyField",
    itemId: "linktextEditor"
    propertyName: "linktext"
  },
  {
    xtype:
     "com.coremedia.cms.editor.sdk.config.richTextPropertyField",
    propertyName: "teaserText",
    anchor: "98%",
    height: 300
  }
 ],
 defaults: {
  xtype: "com.coremedia.cms.editor.sdk.config.propertyField",
  beanValueExpresion: config.beanValueExpresion
 }
}
```

The above code example defines a component of `xtype` "panel" with two *property editors* for editing a string and a richtext property, respectively. The `xtype` of the surrounding panel, like that of all Ext JS components, is a simple string without a namespace prefix. The `xtype` of a plain Ext JS component is, in most cases, the name of the component class, in all lowercase characters.

The property editors shown above are *CoreMedia Studio* components, that are based on plain Ext JS components, but add Studio-specific functionality. Their `xtype` is by convention the same as the name of the component class, but using a lowercase first character after a module specific prefix. See Section 5.2, “Ext JS with ActionScript and EXML” [32] for details.

The optional `itemId` property can be understood as a per-container id which identifies the component uniquely within its container. Note that `itemId`s are not to be confused with DOM element ids or Ext JS component ids which are unique within the entire application.

Figure 5.1. Ext JSON

xtype specifies type of component to build

predefined Ext xtypes are non-qualified and lower case

CoreMedia UI-Toolkit / Editor SDK xtypes are qualified and lower case

items property defines container's children (component tree)

itemId is a unique key within one items list

```
{
  xtype: "panel",
  title: "Teaser Properties",
  items: [
    {
      xtype: "com.coremedia.cms.editor.sdk.config ↵
           .stringPropertyField",
      itemId: "linktextEditor"
      propertyName: "linktext"
    },
    {
      xtype: "com.coremedia.cms.editor.sdk.config ↵
           .richTextPropertyField",
      propertyName: "teaserText",
      anchor: "98%",
      height: 300
    }
  ],
  defaults: {
    xtype: "com.coremedia.cms.editor.sdk.config ↵
           .propertyField",
    beanValueExpression: config.beanValueExpression
  }
}
```

defaults are applied to all child (items) definitions

When developing *CoreMedia Studio* extensions, you don't need to use the Ext JSON format directly. Instead, you're encouraged to specify widgets using the much more convenient and powerful EXML notation. The example below shows the corresponding EXML and JSON specifications:

Figure 5.2. EXML compared to Ext JSON

Ext JSON

```
{
  xtype: "panel",
  title: "Teaser Properties",
  items: [
    {
      xtype: "com.coremedia.cms.editor.sdk.config ↵
           .stringPropertyField",
      itemId: "linktextEditor"
      propertyName: "linktext"
    },
    {
      xtype: "com.coremedia.cms.editor.sdk.config ↵
           .richTextPropertyField",
      propertyName: "teaserText",
      anchor: "98%",
      height: 300
    }
  ],
  defaults: {
    xtype: "com.coremedia.cms.editor.sdk.config ↵
           .propertyField",
    beanValueExpression: config.beanValueExpression
  }
}
```

EXML

```
<?xml version="1.0" encoding="UTF-8"?>
<exml:component
  xmlns:exml="http://www.jangaroo.net/exml/0.8"
  xmlns="exml:ext.config"
  xmlns:editor="exml:com.coremedia.cms.editor.sdk.config">
 <panel title="Teaser Properties">
  <items>
   <editor:stringPropertyField
     itemId="linkTextEditor"
     propertyName="linkText"/>
   <editor:richTextPropertyField
     propertyName="teaserText"
     anchor="98%"
     height="300"
   />
  </items>
  <defaults>
   <editor:propertyField beanValueExpression=
     "{config.beanValueExpression}"/>
  </defaults>
 </panel>
</exml:component>
```

The following sections describe Ext JS components, plugins, and actions in more detail.

Ext JS-specific examples of advanced components are available on the Official Ext JS examples page. The full Ext JS API documentation is also available at Sencha.com.

5.1.1 Components

Ext JS defines three basic types of components

- `ext.Component`
- `ext.Container`
- `ext.ViewPort`

The base class for Ext JS UI controls is `ext.Component`. Components are registered with the `ext.ComponentMgr` at construction time. They can be referenced at any time by id using the `Ext.getCmp` utility function. Component classes are required to define a static property named "xtype" that is used by the component manager to determine the runtime type of a component given in JSON notation. Note that when you use EXML to declare your components, the *Jangaroo* framework will take care of that for you.

Components are nested in containers of class `ext.Container` which is a subclass of `ext.Component`. Containers manage the lifecycle (that is, control creation, rendering and destruction) of their child components.

The top-level component of *Studio*'s component tree is `ext.Viewport`, which represents the viewable application area of the browser.

The API documentation of Ext JS is available at sencha.com. Specifically, the documentation of `Ext.Component` provides a list of component types available in Ext JS. It is also worth looking into the API documentation of `ComponentMgr`, `Element`, and the `Ext` utility class.

5.1.2 Declarative UI (Ext JSON)

Ext JS builds on common JSON notation to describe the application's component tree declaratively. The root of an Ext JS component tree is a viewport component. Its constructor takes a JSON object that declares the UI's component structure, and initializes it. *CoreMedia Studio*'s top-level component tree is shown below:

Example 5.2. Viewport definition as Ext JSON

```
{
  id: com.coremedia.cms.editor.sdk.desktop.EditorMainView.ID,
  ctCls: "main-view",
  layout: {
    align: "stretch",
    type: "hbox"
  },
  items: [
    {
      id: "favorites-toolbar",
      itemId: "favorites-toolbar",
      width: 100,
      xtype:
com.coremedia.cms.editor.sdk.config.favoritesToolbar.xtype
    },
    {
      id: "desktop",
      flex: 1.0,
      xtype: com.coremedia.cms.editor.sdk.config.desktop.xtype
    },
```

```
        {
          width: 34,
          id: "actions-toolbar",
         xtype: com.coremedia.cms.editor.sdk.config.actionsToolbar.xtype

        }
    ]
}
```

This object defines the layout and the basic items of the application (or more specifically, the application's view port). The main items are the `favoritesToolbar`, the `desktop` and the `actionsToolbar`. When this configuration object is loaded, instances of the child components identified by the following `xtypes` are instantiated.

- `com.coremedia.cms.editor.sdk.config.favoritesToolbar.xtype`
- `com.coremedia.cms.editor.sdk.config.desktop.xtype`
- `com.coremedia.cms.editor.sdk.config.actionsToolbar.xtype`

The configuration of these components is specified by the sibling attributes of the respective `xtype` attribute (the favorites toolbar, for example, has width 100), and is merged with the component's default configuration.

5.1.3 Component Plugins

In general, the recommended strategy for extending Ext JS components is to use the component plugin mechanism, rather than subclassing. Reusable functionality should be separated out into component plugins, and can then be used by components of completely different types, without requiring them to inherit from a common base class.

Plugins are configured in a component's `plugins` property. A plugin must provide an `init` method accepting the component it is plugged into as parameter. This method is called by the component when the component is initialized.

If a plugin defines a `ptype` property, its type can be registered at the `ComponentMgr`.

```
ext.ComponentMgr.registerPlugin(pytpe, class)
```

Once registered, plugins of the given type can be instantiated using

```
ext.ComponentMgr.createPlugin(pluginConfig, defaultType)
```

The following code defines a `field` component and adds the plugins `ImmediateChangeEventsPlugin` and `BindPropertyPlugin`.

```
{
  xtype: 'field',
  name: 'properties.' + config.propertyName,
  plugins: [
    {ptype:
com.coremedia.ui.config.immediateChangeEventsPlugin.ptype},
    {
      bindTo: config.bindTo.extendBy('properties',
        config.propertyName),
      bidirectional: true,
      ptype: com.coremedia.ui.config.bindPropertyPlugin.ptype
    }
  ]
}
```

Example 5.3. Plugin usage in Ext JSON

Refer to http://www.sencha.com/blog/advanced-plugin-development-with-ext-js/ for further details on Ext JS plugins.

5.1.4 Actions

Actions combine some functional parts of your application with UI details to be attached to a component. Buttons, for example, are commonly associated with an action. The difference between designing an action and attaching a mere event handler to a component is that an action combines the handler code with UI details such as a name or a button icon, which simplifies reuse. *CoreMedia Studio* defines actions that work on content objects, for example for creating new content objects or publishing contents.

Actions are most commonly used in conjunction with buttons or (context) menu items. In general, you should avoid invoking Actions directly - use the corresponding API method instead. For example, assume you want to publish some content programmatically. You should not invoke `PublishAction` in this case; instead, use the API method `PublicationService#publish(content, callback)`.

5.2 Ext JS with ActionScript and EXML

While the *CoreMedia Studio* code you see at runtime is all JavaScript, *CoreMedia Studio* is completely written in *ActionScript* and *EXML*, an XML format to describe components declaratively. CoreMedia calls this combination of tools and approach *Ext AS* (where obviously, "ActionScript" replaces the "JavaScript" in *Ext JS*.

Ext AS is designed to provide a statically typed way to implement Ext JS applications. EXML is used to declaratively describe Ext UI components (or component trees), validated through a W3C standards compliant XML Schema. During the build process, EXML files are compiled down to ActionScript 3, which in a second step are then compiled further to JavaScript. For localization, property files can be converted to ActionScript classes, too, so that you can access a localization key as if it was a constant defined in a class.

While it is possible to extend *CoreMedia Studio* with components written in JavaScript, it is recommended to use Ext AS. With the Jangaroo project, CoreMedia offers Open Source tools and libraries that provide complete support for this development approach. All public *CoreMedia Studio* APIs are available as ActionScript 3 ASDoc and source stubs, so that you can set up your IDE to provide code completion, validation, and documentation lookup.

This section states the rationale for using Ext AS, gives you a rough overview of the approach and tools, and contains references to the detailed online documentation, which is part of the Jangaroo open source project.

Ext AS: the Typed Version of Ext JS

In contrast to JavaScript and JSON, ActionScript and EXML are *typed* languages. While originally, typed languages were chosen to find errors early at compile time, the more important advantage today is that much better tools can be built to ease and speed up development. In a good IDE, errors and possible mistakes are detected as you type, and the IDE even makes suggestions as to what to type next, how to resolve errors, and lets you look up documentation easily. Using a typed language is important for the IDE to be able to derive what the code is referring to. With an untyped language, only limited IDE support is possible, and the IDE has use more or less imprecise heuristics, and will in many cases make ambiguous (or even erroneous) suggestions.

Source File Types and Compilers

CoreMedia Studio is an Ext AS application and as such uses four different kinds of source files:

- EXML files to specify reusable UI components declaratively
- Property files for localized texts and labels

- ActionScript files for all other application code
- JavaScript files for bootstrapping code and CKEditor extensions

Consequently, the Jangaroo tool set contains three compilers:

- EXML to ActionScript
- Property files to ActionScript
- ActionScript to JavaScript

The first two compilers are chained with the last one, resulting in pure JavaScript output. CoreMedia chooses to let the additional compilers generate ActionScript, not JavaScript directly, as the generated ActionScript classes are better suited for access from ActionScript code, and integrate seamlessly in Jangaroo's lazy class initialization and automatic class loading.

Fitting into the Maven build process, all compilers are usually invoked through Maven, but there are also plugins for IntelliJ IDEA that extend IDEA's incremental build process and invoke the compilers directly, resulting in a much faster turnaround. Currently, CoreMedia strongly recommends using IntelliJ IDEA 10.x for Jangaroo development for highest productivity.

Online Jangaroo Documentation

Since CoreMedia is not primarily a manufacturer of development tools, all these tools are released as open source under an Apache 2 license. Consequently, the tools are not documented here, but on the Jangaroo Website and in Jangaroo's Wiki.

Since the *CoreMedia Project* workspace uses Maven, you can ignore all references to direct compiler command line interfaces or Ant. When starting with Jangaroo development, it is recommended to work through the documentation in the following order:

1. Start with the Jangaroo Tutorial to get familiar with writing, building, and starting a Jangaroo application.
2. Continue with Developing Jangaroo Applications with IntelliJ IDEA. This adds two aspects: on the one hand, the example project, like *CoreMedia Studio*, uses a multi-module setup, on the other hand, working with Jangaroo in IntelliJ IDEA is explained in detail. Please consider the multi-module example even if you use another IDE!
3. In parallel, you can start getting acquainted with Ext JS (see Section 5.1, “Ext JS Primer” [27]).

4. Now you are ready to face Ext AS, including EXML, which is documented as Ext AS: Creating Ext JS Applications with ActionScript and EXML.
5. Integrating Ext AS and especially EXML in *IntelliJ IDEA* requires some additional explanation; there is an IDEA plugin *Jangaroo EXML* that you're highly encouraged to install to get optimal EXML support. All about Ext AS and IDEA is documented as Developing Ext AS Applications with IntelliJ IDEA.

If you have questions about any Jangaroo tool, please post in the Jangaroo user group. You can also write an email to info@jangaroo.net.

If the question or problem is *Studio* related, please contact CoreMedia support.

ActionScript Documentation

Being integrated in our ActionScript programming model, the documentation of all Ext JS components and public API components of *CoreMedia Studio* is accessible through the ASDoc (ActionScript Documentation) linked from the Studio's most recent release page, which is available at the CoreMedia download section or from our documentation site at http://documentation.coremedia.com.

In the ASDoc, you will find *two* ActionScript classes per component. One class represents the component itself. This component class describes the type of the component at runtime, for example when registering event listeners or when updating the state of the component. For Ext JS components, the name and package of each class matches the official Ext JS documentation by Sencha, except that the top-level package is `ext` instead of `Ext`.

A second class defines the component's configuration time API, that is, when you create a JSON configuration object or an EXML component definition. Configuration classes are by convention placed in a package ending in `config`. For *Ext JS* components, all configuration classes are located in the package `ext.config` and are named like the `xtype` (or `ptype` for plugins) of the component. For Studio components, the name of the configuration class is identical to that of the component class, but with a lowercase initial character, and the package is chosen based on the module in which the component is defined:

- module `ui-components`: package `com.coremedia.ui.config`;
- module `editor-components`: package `com.coremedia.cms.editor.sdk.config`.

By convention, the inheritance hierarchy of configuration classes matches the hierarchy of component classes.

Configuring Components

Each configuration class defines the configuration attributes of that class. You can use instances of the configuration classes for configuring Ext JS components in a type-safe way, although it is still possible to write component configurations as a plain JSON object. The code fragments

Example 5.4. Component instantiation using Ext JSON

```
ComponentMgr.create({
  xtype:
    "com.coremedia.cms.editor.sdk.config.stringPropertyField",
  itemId: "linktextEditor",
  propertyName: "linktext"
}, null);
```

and

Example 5.5. Component instantiation using typed setters

```
var stringPropertyFieldConfig:stringPropertyField =
  new stringPropertyField();
stringPropertyFieldConfig.itemId = "linktextEditor";
stringPropertyFieldConfig.propertyName = "linktext";
ComponentMgr.create(stringPropertyFieldConfig, null);
```

and

Example 5.6. Component instantiation using typed wrapper

```
ComponentMgr.create(new stringPropertyField({
  itemId: "linktextEditor",
  propertyName: "linktext"
}), null);
```

are equivalent. Choose a programming style that suits you. Note however that the most convenient (and thus recommended) way to write component configurations is to use EXML rather than ActionScript.

The last example shows how a configuration class itself can be initialized untyped, while still allowing typed accesses later. Note that an instance creation performed in this way is not the same as a type cast: The `xtype` attribute of the configuration object is set implicitly when the constructor is run.

When developing with EXML, you don't have to deal with the ActionScript code manually: The EXML compiler automatically generates code equivalent to the third variant shown above. In this case, the reduced type checking is offset by the checks at XML level during development.

EXML files are described in more detail in the Jangaroo tools wiki. The namespaces to use in EXML files in the context of *CoreMedia Studio* are:

- `exml:com.coremedia.ui.config` for the reusable components of the *CoreMedia UI toolkit* and

- `exml:com.coremedia.cms.editor.sdk.config` for the actual *CoreMedia Studio* components.

5.3 Client-side Model

MVC pattern

The *CoreMedia Studio* user interface is implemented following the Model-view-controller (MVC) pattern. The widgets provided by Ext JS are considered the view, whereas Ext JS actions take the role of controllers. To deal with the model layer efficiently, the Studio framework provides the key concepts of *beans* and *value expressions*.

Beans

A bean is an object that aggregates a number of properties, where property values may be arbitrary JavaScript objects, including arrays or even other beans. Beans are capable of sending events when one of their properties changes, making it possible to update the view components dynamically when a bean changes.

Simple and complex wiring

While wiring up a UI component property to a plain bean property is mostly straightforward and can be as simple as connecting a button label to a simple string bean property, you will inevitably run into situations where you need to "compute" a UI component property based on complex model state that might span different bean properties, or even completely separate beans.

value expressions

Both the simple and the complex case can be conveniently solved using *value expressions*, which can encapsulate the computation of mutable values on the bean level. A frequently used value expression takes a start bean and follows property references from beans to beans to arrive at a target bean or value. Value expressions, too, generate events whenever their value changes, and you can attach event listeners to them to dynamically update the UI.

Using Ext JS plugins

While it is possible to hand code the view response to model changes, you are encouraged to make use of the *Studio SDK's* predefined Ext JS plugins. Plugins are available for setting UI component properties, selections, displayed values, and so on. All of these plugins transfer state between a value expression and an Ext JS component, sometimes in both directions ("bidirectional").

Uniform access layer

For experienced Ext JS developers, it may seem strange that an explicit model in the form of beans is used, instead of widget-internal state as an implicit model. However, the chosen approach allows for a more consistent representation of the model. By wrapping remote data sources as beans, a uniform access layer throughout *CoreMedia Studio* is achieved. In other words, from a developer's perspective, it is transparent whether model state is wired up to remote (server-side) or local (client-side) data. This also means that as a developer, you don't need to manually write code to make Ajax calls in order to update server-side data - you make sure that your model is properly wired up to your UI, and the framework takes care of the details for you.

For details about the ActionScript classes mentioned in the following sections, refer to the ActionScript documentation as found on the Studio release page, available at the CoreMedia download section.

5.3.1 Beans

Beans are objects with an arbitrary number of properties. Properties can be updated, generating events for each change. The name "bean" originates from the concept of Java Beans, which are also characterized by their properties and event handling capabilities. Unlike Java beans, the Studio beans do not enforce a strict typing and naming policy, whereby each property must be represented by individual getter and setter functions. Instead, untyped generic methods for getting and setting properties are provided. Specific bean implementations are allowed to add typed accessors, but are not required to do so.

Remote beans

All beans implement the interface `com.coremedia.ui.data.Bean`. Remote beans, which encapsulate server-side state, conform to the more specific interface `com.coremedia.ui.data.RemoteBean`. Refer to Section 5.3.2, "Remote Beans" [40] for more details about these concepts. At first, the more generic `Bean` interface is described, which is agnostic of a potential backing by a remote store.

Properties

Retrieving bean properties

Individual properties of any bean can be retrieved using the `get(propertyName)` method, which receives the name of the property as an argument. Arbitrary objects and primitive values are allowed as property values. The set of property names is not limited, but it is good practice to document the properties and their semantics for any given bean. If non-string values are used as property names, they will be converted to a string.

Beans may reference other beans. For example, the `Content` bean contains a property `properties` that contains a bean with schema-specific properties, whereas the `Content` bean itself contains the predefined content metadata, such as creation and publication date, which are defined implicitly for all CoreMedia content objects.

Updating properties

By calling `set(propertyName, value):Boolean`, a property value can be updated. The method returns `true` if (and only if) the bean was actually changed. Generally, the new value is considered to equal the old value if the `===` operator considers them equal. There are a number of exceptions, though:

- Arrays are equal if they are of the same length and if all elements are equal according to the bean semantics. That is, arrays are treated as values and not as modifiable objects with state.
- `Date` and `Calendar` values are equal if they denote the same date and time, with time zone information taken into account.
- Blobs as stored in the CMS are equal if they contain the same content with the same content type. As long as the blobs are not fully loaded from the server, a conservative heuristic is used that considers the blobs equal if it is known that they will ultimately represent the same value when loaded.

By using the method `updateProperties(newValues)`, you can set multiple properties at once. The argument object must contain one ActionScript property per bean property to be set. Bean properties not mentioned in the argument object are left unchanged. Consider the following example:

Example 5.7. Updating multiple bean properties

```
bean.updateProperties({
    a: 1,
    b: ["a", "b"],
    c: anotherBean
  });
```

The above code sets the three properties `a`, `b`, and `c` simultaneously, but the property `d` keeps its previous value if it was set. Apart from convenience, the main difference compared to three calls like `bean.set("a", 1)` is that events will be sent only after all properties have been updated. This can be useful when you want to update a bean atomically.

Calling `toObject()` on a bean will return a snapshot of the current bean state in the form of an object that contains one ActionScript property per bean property.

Events

Register and remove property event listener

Property event listeners for a single property are registered with `addPropertyChangeListener(propertyName, listener)` and removed with `removePropertyChangeListener(propertyName, listener)`. The listener argument must be a function that receives a simple argument of type `com.coremedia.ui.data.PropertyChangeEvent`. This event object contains information about the bean, the changed property and the old and the new value.

Listener for all property events

A listener function registered with `addValueChangeListener(listener)` receives events for all properties of the respective bean. When multiple properties are updated, the listener receives one call per updated property. Such listeners can be removed by calling `removeValueChangeListener(listener)`.

For beans, events are dispatched synchronously, before the update call returns.

Bean State

Beans, especially remote beans, may enter different states. The possible states are enumerated in the class `com.coremedia.ui.data.BeanState`. The method `getState()` provides the current state of the bean. State changes are also reported to all listeners. The event object provides the old and the new bean state.

The possible states are:

- `UNKNOWN`: The bean is still being set up.
- `NON_EXISTENT`: The bean represents an entity that does not exist. Typically, the entity existed at one time in the past, but has been destroyed.

- `UNREADABLE`: The bean represents an entity that exists, but authorization to access it is missing.
- `READABLE`: The bean can be accessed without restrictions.

Local beans are always in state `READABLE`.

Singleton Bean

The interface `IEditorContext`, whose default instance can be accessed as the package field `com.coremedia.cms.editor.sdk.editorContext`, provides the method `getApplicationContext()`, which returns a singleton local bean. This bean is provided as a starting point for navigating to other singletons and for sharing system-wide state. Individual APIs document the properties of the singleton bean that are set by that API. Be careful when adding custom properties and avoid name clashes.

5.3.2 Remote Beans

A remote bean encapsulates the state of a server-side object in the client-side application. Its properties are loaded on demand - most commonly by invoking the `RemoteBean#load` or `RemoteBean#invalidate` methods, respectively.

The SDK provides more specialized subclasses of remote beans, for example beans of type `Content`, which represents CoreMedia CMS documents and folders.

Bean values may change when the remote bean is invalidated and reloaded. Note however that currently, there is no active event mechanism that invalidates client-side beans immediately after the data they represent changes on the server.

In the interface `com.coremedia.ui.data.RemoteBean`, the method `getUri()` provides access to the URI from which its state is loaded. Its sibling method `getUriPath()` returns a URI path relative to the base URI of the remote service from which the bean is loaded. The latter value provides a more concise and still unique identification of the remote bean. There is only ever one remote bean for each URI path.

Asynchronous HTTP request

By calling `load(Function)`, the bean is instructed to load its properties, using an asynchronous HTTP request. Note that this is transparent to the developer, that is, you never need to manually construct an XHR, for example by invoking *Ext JS*'s `Ajax#request` method.

Once the call has returned, an optional callback function is invoked, indicating the new state of the bean. A remote bean is also loaded as soon as any of its properties are read. However, the bean will report properties as `undefined` initially and fire an event as soon as the property is updated to a different value after loading.

To reload the bean state, call the method `invalidate(Function)`, which takes an optional callback function which is invoked after all properties have been reloaded.

Listen to events until property is ready to use

Please note that computed bean properties may still be `undefined` when the callback functions are invoked. For example, the `Content` bean contains a property `path` that requires all the content's parents to be loaded recursively. Although the `Content` bean itself might be completely loaded, the `path` property remains `undefined` until all the content's parents have finished loading. Listen to the change events for the computed property to find out when the property is ready or use a `ValueExpression`. See Section 5.3.6, "Value Expressions" [44] for details.

Update properties on server

When properties of a remote bean are set, they are eventually written back to the server. The remote bean may bundle any number of writes before making its update request. At least all updates made in the same JavaScript execution without an intervening `window.setTimeout()` are bundled in one write. You can call the method `flush(Function)` to ensure that a callback function is invoked after the update call for all previously updated properties has completed, either successfully or with an error. The callback function can determine the success status of a flush call by its single argument, a `FlushResult` object. This object also carries a reference to the flushed bean and, in the case of an error, to a `RemoteError` object indicating the source of the problem.

Remote beans may be unreadable or even nonexistent, which is indicated by the method `getState()`. A bean's state can be observed by usual property change listeners (see previous section), since bean state changes trigger property change events and report the current state (see `com.coremedia.ui.data.PropertyChangeEvent#newState`). Working with remote beans generally requires more attention to error conditions than working with local beans.

5.3.3 Issues

CoreMedia Studio has built-in support for server-side validation of content objects. You can leverage the validation framework for your own (non CMS) data resources, but for content objects managed in the *CoreMedia Content Server*, the framework already offers convenient support (see Section 5.4.2, "Content" [52] for a general description of the Studio Content API.)

Server-side validation always works on values already saved (persisted) - in other words, a validator will never prevent the user from saving data, so that the risk of data loss is minimal. You can however set up *Studio* to prevent the user from approving or checking in documents that have validation issues with severity `ERROR` (see Section "Tying Document Validation to Editor Actions" [180] for details on how to configure this).

Getting issues from the server

The client can ask the server to compute *issues* of an entity (most commonly Content), where they become accessible as a `com.coremedia.ui.data.valida`

`tion.Issues` object. Once received, the client can do things like highlight a property field that contains an invalid value, or open a dialog. *Studio* offers built-in support for marking standard property fields invalid, and offers the user a convenient interface to step through and correct detected validation issues in one go.

The issues object provides access to individual `Issue` objects through a number of methods:

- `getAll()` returns all issues of the entity in a single array.
- `getByProperty()` returns a sub bean whose properties match the properties of the entity. Each property contains an array of issues that affect exactly that property.
- `getGlobal()` returns an array of issues that do not affect a specific property, but that describe the state of the entity as a whole. A common example for this is a validator that checks for the correct folder path of a document - you could set up a validator to raise a `WARNING` when a document is created in a wrong folder, for example.

An issue links back to its entity by means of the `entity` property. The `severity` property indicates a level of `"INFO"`, `"WARN"`, and `"ERROR"`. You can freely define the severity level for any validator.

The property `property` stores the name of the property whose value causes the issue. If `null`, this indicates a global issue that affects the entity as a whole, rather than one of its properties. In the property `code`, each issue stores a string identifier indicating the type of issue detected. Applications are expected to localize this identifier as needed. Depending on the code, the array property `arguments` might store additional data in a specific layout.

Error codes and further information

The issue code identifiers depend on the type of entity that has been validated. In fact, each server-side validator may introduce its own code and you have to refer to the documentation of the validators for details. Some validators allow you to configure the error code that they report. In custom validators, you can also pass on additional ("runtime") information describing the error in more detail, and use this additional information to present user-friendly descriptions of the problem in the UI. See Section 7.15.1, “Validators” [176] for details.

5.3.4 Operation Results

Callback functions

Complex remote operations typically allow you to specify a callback function. The callback function is called after the operation has completed, either successfully or unsuccessfully. This allows you to postpone subsequent steps until a remote resource is in a defined state again.

Callback functions often receive an *OperationResult* argument. Such objects indicate in their `success` attribute whether the attempted operation was successful. In

the case of errors, the attribute `error` points to a *RemoteError* object further detailing the problems. Individual operations may return richer result objects. For example, the previous section already mentioned the `FlushResult`, which also references the modified bean in the `remoteBean` property.

5.3.5 Model Beans for Custom Components

When creating complex GUI components, it is good practice to provide an abstract model in the form of a bean to back the view. It is often helpful to provide an ActionScript base class (`MyComponentBase` below) for the component and extend it by an EXML component, bundling the application logic in the base class. The base class should therefore also take care of building the model bean.

Note however that when creating the Ext JS component configuration in the EXML component, the constructor of the base class has not yet been invoked. Because the component configuration must reference the model to bind the component's states, the model bean must be created before the base class constructor is used. This can be achieved by an accessor method that creates the bean using the call `com.coremedia.cms.editor.sdk.editorContext.getBeanFactory().createLocalBean()` upon first access. This is shown in the `getModel()` method below.

Example 5.8. Model bean factory method

```
import com.coremedia.ui.data.Bean;
import com.coremedia.ui.data.beanFactory;
import mypackage.config.myComponent;
import ext.Panel;

public class MyComponentBase extends Panel {
  ...
  private var model:Bean;

  /**
   * ...
   * @param config the config object
   */
  public function MyComponentBase(config:myComponent = undefined)
{
    super(config);
    initModel(config);
    ...
  }

  public function getModel():Bean {
    if (!model) {
      model = beanFactory.createLocalBean();
    }
    return model;
  }

  private function initModel(config:myComponent):void {
    getModel().set('myProperty', ...);
    ...
  }
  ...
}
```

Given this base class, you can access the model in the EXML class as follows:

Example 5.9. Model bean access

```
<exml:component xmlns:exml="http://www.jangaroo.new/exml/0.8"
                xmlns="exml:ext.config"
                xmlns:ui="exml:com.coremedia.ui.config"
                xmlns:mm="exml:mypackage.config"
                baseClass="MyComponentBase">
  <panel>
    <items>
      ...
      <textfield>
        <plugins>
          <ui:immediateChangeEventsPlugin/>
          <ui:bindPropertyPlugin bidirectional="true">
            <ui:bindTo>
              <ui:valueExpression expression="myProperty"
                                  context="{getModel()}"/>
            </ui:bindTo>
          </ui:bindPropertyPlugin>
        </plugins>
      </textfield>
      ...
    </items>
  </panel>
</exml:component>
```

Here a text field is configured to display the value of a property, but of course arbitrary widgets can be used.

In fact, the property is not directly accessed by the plugin, but indirectly through a value expression that, in this case, simply evaluates to a property value. Value expressions will be discussed in the next section.

5.3.6 Value Expressions

The interface `com.coremedia.ui.data.ValueExpression` describes objects that provide access to a possibly mutable value and that notify listeners when the value changes. They may also allow you to receive a value that can then become the next value of the expression. Value expressions may be as simple as defining a one-to-one wiring of a widget property to a model property, but they may encapsulate complex logic that accesses many objects to determine a result value. As an application developer, you can think of value expressions as an abstraction layer that hides that potential complexity from you, and use a common, simple interface when wiring up UI state to complex model state.

The Studio SDK offers the following primary implementations of the `ValueExpression` interface. You can use the factory methods from `com.coremedia.ui.data.ValueExpressionFactory` to create a `ValueExpression` programmatically from ActionScript.

- `PropertyPathExpression`. This is meant to be used in simple scenarios, where you want to attach a simple bean property to a corresponding widget property. It starts from a bean and navigates through a path of property

names to a value. Long paths can be separated with a dot. You can obtain this value expression flavor using `ValueExpressionFactory#create(expression, bean)`.

- `FunctionValueExpression`. Use this in scenarios where your UI state requires potentially complex calculations on the model, using multiple beans (remote or local). This value expression object wraps an ActionScript function computing the expression's value. When a listener is attached to the returned value expression, the current value of the expression is cached, and dependencies of the computation are tracked. As soon as a dependency is invalidated, the cached value is invalidated and eventually a change event is sent to all listeners (if the computed value has actually changed). You can use `ValueExpressionFactory#createFromFunction(function, ...args)` to create this flavor. See below for details on how to use `FunctionValueExpression`s.

In many cases, you can use the facilities provided by plugins (and thus use EXML to specify your value expression), without ever constructing a value expression programmatically. Nevertheless, value expressions are a vital part of the Studio SDK's data binding framework, so it is helpful to understand how they work.

Values

The method `getValue()` returns the current value of the expression. How this value is computed depends on the type of value expression used. Like bean properties, value expressions may evaluate to any ActionScript value.

When a value expression accesses remote beans that have not yet been loaded, its value is `undefined`. Getting the value or attaching a change listener (see below) subsequently triggers loading all remote bean necessary to evaluate the expression. If you need a defined value, you can use the `loadValue(Function)` method instead. The `loadValue` method ensures that all remote beans have been loaded and only then calls back the given function (and, in contrast to change listeners, only once, see below) with the concrete value, which is never `undefined`.

Be sure that the value is not undefined

Like remote beans, value expressions may turn out to be unreadable due to missing read rights. In this case, `getValue()` returns `undefined`, too, and the special condition is signaled by the method `isReadable()` returning `false`.

Events

A listener may be attached to a value expression using the method `addChangeListener(listener)` and removed using the method `removeChangeListener(listener)`. The listener must be a function that takes the value expression as its single argument. The listener may then query the value expression for the current value.

Contrary to bean events, value expression events are sent asynchronously after the calls modifying the value have already completed. The framework does however not guarantee that listeners are notified on *all* changes of the value. When the value is updated many times in quick succession, some intermediate values might not be visible to the listener.

The listener is also notified when the readability of the value changes.

As long as you have a listener attached to a value expression, the value expression may in turn be registered as a listener at other objects. To make sure that the value expression can be garbage collected, you must eventually remove all listeners added to it.

A common pattern when adding a listener to a value expression involves an upfront initialization and subsequent updates on events:

Example 5.10. Adding a listener and initializing

```
import com.coremedia.ui.data.ValueExpression;

public class MyComponentBase extends AnExtJSComponent {
  public function MyComponentBase(config:Object = undefined) {
    ...
    var valueExpr:ValueExpression = ...;
    valueExpr.addChangeListener(valueExprChanged);
    valueExprChanged(valueExpr);
    ...
  }

  private function valueExprChanged(valueExpr:
    ValueExpression):void
  {
    var value:* = valueExpr.getValue();
    ...
  }
  ...
}
```

By calling the private function once immediately after adding the listener, it is possible to reuse the functionality of the listener for initializing the component.

Property Path Expressions

The most commonly used value expression is the *property path expression*. It allows you to navigate from an object to a value by successively reading property values on which the next read operation takes place. For example, a property path expression may operate on the object `obj` and be configured to read the properties `a`, `b`, and then `c`. If the property `a` of `obj` is `obj1`, the property `b` of `obj1` is `obj2`, and the property `c` of `obj2` is 4, then the expression will evaluate to 4. A path of property names is denoted by a string that joins the property names with dots, in this case `"a.b.c"`. If you want to address array elements you have to add the index of the element with another dot, such as `a.b.c.3`, and not use the more obvious but false `a.b.c[3]` notation.

You can create a property path expression manually in the following way:

Example 5.11. Creating a property path expression

```
import com.coremedia.ui.data.ValueExpression;
import com.coremedia.ui.data.ValueExpressionFactory;
...
var ppe:ValueExpression =
  ValueExpressionFactory.create("a.b.c", obj);
```

The dot notation above might suggest that property path expressions operate exactly like ActionScript expressions, but that is not quite correct. Property path expressions support the following access methods for properties:

- read the property of a bean using the `get(property)` method;
- call a publicly defined getter method whose name consists of the string "get" followed the name of the property, first letter capitalized;
- call a publicly defined getter method whose name consists of the string "is" followed the name of the property, first letter capitalized;
- read from a publicly defined field of an object. This is the classic ActionScript case.

At different steps in the property path, different access methods may be used.

Even if there are many properties in the path, changes to any of the objects traversed while computing the value will trigger a recomputation of the expression value and potentially, if the value has changed, an event. This is only possible, however, for objects that can send property change events.

- For beans, a listener is registered using `addPropertyChangeListener()`.
- For instances of `ext.util.Observable`, a listener is registered using `addListener()`.

Property path expressions may be updated. When invoking `setValue(value)`, a new value for the value expression is established. This will only work if the last property in the property path is writable for the object computed by the prefix of the path. More precisely, a value may be

- written into a property of a bean using the `set(property,value)` method;
- passed to a publicly defined setter method that takes the new value as its single argument and whose name consists of the string "set" followed by the name of the property, first letter capitalized;
- written into a publicly defined field of an ActionScript class.

At various points of the API, a value expression is provided to allow a component to bind to varying data. Using the method `extendBy(extensionPath)` adds further property dereferencing steps to the existing expression. For example,

`ValueExpressionFactory.create("a.b.c", obj)` is equivalent to `ValueExpressionFactory.create("a", obj).extendBy("b.c")`.

To create a property path expression from within an EXML file, you can use the `valueExpression` element from the `exml:com.coremedia.ui.config` namespace.

```
...
<ui:valueExpression expression="myProperty"
  context="{getModel()}"/>
...
```

Example 5.12. The valueExpression EXML element

Function Value Expressions

Function value expressions differ from property path expressions in that they allow arbitrary ActionScript code to be executed while computing their values. This flexibility comes at a cost, however: such an expression cannot be used to update variables, only to compute values. They are therefore most useful to compute complex GUI state that is displayed later on.

To create a function value expression, you use the method `createFromFunction` of the class `ValueExpressionFactory`.

```
ValueExpressionFactory.createFromFunction(function():Object {
  return ...;
});
```

Example 5.13. Creating a function value expression

The function in the previous example did not take arguments. In this case, it can still use all variables in its scope as starting point for its computation or it might access global variables. To make the code more readable, you might want to define a named function in your ActionScript class and use that function when building the expression.

```
private function doSomething():void {
  ...
  expr = ValueExpressionFactory.
    createFromFunction(calculateSomething);
  ...
}

private function calculateSomething():Object {
  return ...;
}
```

Example 5.14. Creating a value expression from a private function

If you want to pass arguments to the function, you can provide them as additional argument of the factory method. The following code fragment uses this feature to pass a model bean to a static function.

Example 5.15. Creating a value expression from a static function

```
private function doSomething():void {
  ...
  expr = ValueExpressionFactory.
    createFromFunction(calculateSomething, getModel());
  ...
}

private static function calculateSomething(model:Bean):Object
{
  return ...;
}
```

value change events

Function value expressions fire value change events when their value changes. To this end, they track their dependencies on various objects when their value is computed. For accessed beans and value expressions, the dependency is taken into account automatically: whenever the bean or the value expression changes relevantly, the value of the function value expression changes automatically, and an event for the function value expression is fired.

If you access other mutable objects, you should make sure that these objects inherit from `Observable`, so that you can register the dependencies yourself. To this end, you can use the static methods of the class `DependencyTracker`. In particular, the method `dependOnObservable(Observable,String)` provides a way to specify the observable and the event name that indicates a relevant change. As a shortcut, the method `dependOnFieldValue(Field)` allows you to depend on the value of an input field.

Example 5.16. Manual dependency tracking

```
var observable:Obserable;
var field:Field;

private function calculateSomething():Object {
  DependencyTracker.dependOnObservable(observable, "fooEvent");
  DependencyTracker.dependOnFieldValue(field);
  ... observable.fooMethod() ...;
  ... field.getValue() ...;
  return ...;
}
```

If you register a dependency while no function value is being computed, the call to `DependencyTracker` is ignored. This means that you can register dependencies in your own functions, and the methods will work whether they are called in the context of a function value expression or not.

To create a function value expression in EXML, you have to insert an ActionScript block into the EXML:

Example 5.17. The bindPropertyPlugin EXML element

```
...
<exml:import
 class="com.coremedia.ui.data.ValueExpressionFactory"/>
...
<ui:bindPropertyPlugin bindTo="{ValueExpressionFactory.
```

```
  createFromFunction(calculateSomething)}"/>
...
```

This assumes that you have defined a function calculateSomething in the base class of your EXML component with visibility `protected`. Of course, you may also use static functions or anonymous functions specified inline in the EXML file, but the latter might be more difficult to read.

5.4 Remote CoreMedia Objects

Accessing content on the server

For accessing content, users and groups from *CoreMedia Studio*, a rich API is provided on top of the `Bean`/`RemoteBean` API. In particular, the interfaces `Content`, `User`, and `Group` all inherit from `RemoteBean`. The API aims at being similar to the *Unified API*, which provides access to the *CoreMedia* servers from Java. However, some adjustments were necessary to support the different flavor of concurrency found in JavaScript/ActionScript.

Please refer to the ActionScript documentation (ASDoc) for details about the individual interfaces and methods listed in the following overview.

5.4.1 Connection and Services

Usually, the *Studio* framework will already have taken care of the login when your code is invoked.

Creating a connection when not logged in

In special cases, for example if you are not in *CoreMedia Studio*, you can use the static method `com.coremedia.cap.Cap.prepare(Function)` to create a connection to the remote server. The URL of the CMS remote service to use is read from the global variable `coremediaRemoteServiceUri`. The `prepare` method calls the callback function when the connection has been established, passing a `com.coremedia.cap.common.CapConnection` as the single argument. This connection is not yet bound to a user, but it provides the method `getLoginService()`. On the returned `com.coremedia.cap.common.LoginService` you can call the `login(String, String, String, Function)` method to authenticate the current user, which enables access to other services of the connection.

Once a connection is established, the current session is stored under the key `session` in the application scope bean (obtainable from the current `editorContext` instance). The session provides access to the current user and back to the connection.

The methods `getContentRepository()`, `getUserRepository()`, and `getWorkflowRepository()` of the connection return objects of type `com.coremedia.cap.content.ContentRepository`, `com.coremedia.cap.user.UserRepository`, and `com.coremedia.cap.workflow.WorkflowRepository`, respectively. These repositories serve the same purpose as the identically named objects of the *Unified API*. However, the supported functionality is limited to the use cases required for content editing.

Content repository and services

The `ContentRepository` provides access to the `PublicationService` and the content `AccessControl` through the method `getPublicationService()` and `getAccessControl()`, respectively.

Unlike the *Unified API*, approval operations using the publication service also approve all folders on the path to a content. Publication is very similar to the *Unified*

API counterpart, but withdrawals are performed in a single step without the need to successively set a mark, approve it, and publish the withdrawal.

The `AccessControl` class in the package `com.coremedia.cap.content.authorization` allows you to check whether certain operations on contents are permitted for the current user. Some methods like `mayMove()` and `mayCreate()` are provided for special cases, but most checks are made using the method `mayPerform`, which takes a `Right` enumeration value to indicate the intended operation.

All these methods track the dependencies and can be used from within a `FunctionValueExpression`, even though you cannot register change listeners directly.

Workflow repository and services

The `WorkflowRepository` provides access to the `WorklistService` and the workflow `AccessControl` through the method `getWorklistService()` and `getAccessControl()`, respectively.

The `WorklistService` corresponds closely to the `WorklistService` of the *Unified API*. It provides access to all user-specific lists, but not the administration lists. In particular, you can retrieve the list of process definition that the current user may instantiate, the processes the user has created, but not started, the processes the user has created and started, the offered task and the accepted tasks. You can also obtain lists of tasks that encountered problems during their execution.

All these methods track the dependencies and can be used from within a `FunctionValueExpression`, even though you cannot register change listeners directly.

The `AccessControl` class in the package `com.coremedia.cap.workflow.authorization` allows you to check whether certain operations on workflow objects are permitted for the current user. The methods match the methods defined in the *Unified API*. While the rights are being retrieved, the methods will return `undefined`. Afterwards a Boolean value is answered. Note, however, that no changes of rights are propagated to the client. This is not normally a problem, because the built-in rights policies depend on the current user, only, and not on the workflow state.

5.4.2 Content

Content on the server

The package `com.coremedia.cap.content` of *CoreMedia Studio* provides classes for accessing content. A `Content` object represents a document or folder in the CoreMedia system. It can be obtained through the methods `getChild(...)` or `getContent(String)` of the content repository. Note that unlike in *Unified API*, the `String` parameter to the latter method is not an ID, but a URI path. You can get the URI path of a `Content` with the `Content#getUriPath()` method (inherited from `com.coremedia.ui.data.RemoteBean`).

You can also initiate a search request using the search service returned by `getSearchService()` or by navigating to a content from the root folder returned by `getRoot()`.

Accessing properties of content

Using `getProperties()`, it is possible to navigate to a secondary bean of type `com.coremedia.cap.content.ContentProperties` that contains all schema-defined properties of a content item. When updating properties, use the inherited, generic `set(property, value)` method of `com.coremedia.ui.data.Bean` with `Calendar`, `String`, or `Number` objects or arrays of `Content` objects as appropriate for the individual properties. Refrain from setting blob-valued and XML-valued properties at this time. As for all remote beans, the method `flush(callback)` can be called to force properties to be written to the server immediately.

The `Content` object itself is only responsible for the meta properties that are the same for all contents, for example the name property. The class `ContentPropertyNames` lists all these property names for your reference. As usual, these are also the property names for the events that are sent when a content changes.

The property `lifecycleStatus` is a special property that does not correspond to any *Unified API* feature. It indicates the simplified way in which *Studio* represents the approval, deletion, and publication flags to the user. The class `LifecycleStatus` contains constants for the supported states.

Following the *Unified API*, every content object is associated to a `ContentType` object by means of the `getType()` method. You can also retrieve types by name from the content repository. Given a type, you can create new instances of the type by means of the `create(Content, String, Function)` method.

The `move()` and `rename()` methods are shortcuts for setting the `parent` and `name` properties. As such, a callback provided with these calls receives a `FlushResult` as its single argument. The methods `copy()`, `checkIn()`, `checkOut()`, `revert()`, and `doDelete()` correspond to the equivalent *Unified API* methods. (The unusual name of the `doDelete()` method is caused by `delete` being a reserved word in ActionScript.)

Getting result objects

All operations receive result objects indicating whether the operation was successful. The result of a delete operation is recorded in a `DeleteResult`, with result codes being documented in `DeleteResultCodes`. Similarly, there are `CopyResult` and `CheckInResult` objects. Please see the ASDoc for details.

Through the method `getIssues()`, a `Content` object provides access to issues detected by the server-side validators. See Section 5.3.3, “Issues” [41] for details about the issue API.

5.4.3 Workflow

Workflows on the server

The package `com.coremedia.cap.workflow` of *CoreMedia Studio* provides classes for accessing worklists and workflow objects. A `WorkflowObject` represents a

`Task` or `Process` in the *Workflow Server*. Tasks provide the method `getContainingProcess()` to navigate to its process. Each task and process links to a definition object by means of its `getDefinition()` method. Definition objects are either instances of `TaskDefinition` or `ProcessDefinition`. Each task definition indicates a `TaskDefinitionType` through the method `getType()`, for example `USER` or `AUTOMATED`.

Using the methods `getTaskState()` and `getProcessState()` the current state of a task or process can be obtained as an enumeration value.

Accessing properties of workflow objects

The methods available for workflow objects and definitions correspond to the equivalent *Unified API* methods.

Using `getProperties()` on a task or process, it is possible to navigate to a secondary bean of type `WorkflowObjectProperties` that contains all schema-defined properties of a workflow object. When updating properties, use the inherited, generic `set(property, value)` method of `Bean` with `Boolean`, `String`, `Number`, `User`, `Group`, `Content`, or `Version` objects or arrays of such objects as appropriate for the individual properties. At the moment, timer are not supported. As for all remote beans, the method `flush(callback)` can be called to force properties to be written to the server immediately.

5.4.4 Structs

Structs are part of the *Unified API* and are thus a core product feature.

Storing dynamically structured content with Structs

Implemented by the interfaces `Struct` and `StructBuilder` in the Java API, structs provide a way to store dynamically typed, potentially nested objects in the content repository, and thus add the possibility of storing dynamically structured content to the *Content Server*'s static content type system. To this end, the document type schema may define XML properties with the grammar name `coremedia-struct-2008`. This grammar should use the XML schema `http://www.coremedia.com/2008/struct` as defined in `coremedia-struct-2008.xsd`.

In the ActionScript API, structs are modeled as `Bean` objects. They are directly modifiable. They implement the additional interface `com.coremedia.cap.struct.Struct` to provide access to their dynamic type.

Like every content property value, struct beans are provided as properties of the `ContentProperties` beans. If a struct bean contains a substruct at some property, that substruct is again represented as a struct bean.

Atomic properties of structs may be accessed just like regular bean properties. Structs can store strings, integers, Boolean, and links to documents as well as lists of these values. All struct properties can be read and written using the ordinary `Bean` interface. As usual, lists are represented as ActionScript `Array` objects. Do *not* modify the array returned for a list-valued property. To modify an array, clone the array, modify the clone, and set the new value at the bean.

In the special case of lists of structs, use the methods `addAt()` and `removeAt()` (of the struct containing the struct list) to insert or delete individual entries in the struct list. Note that `Struct` objects in struct lists represent a substruct at a fixed position of the list. For example, the `Struct` objects at position 2 will contain the values of the struct previously at position 1 after you insert a new struct at position 0 or 1.

Structs and substructs support property change events. Substructs do *not* support value change events. You can only listen to a single property of a substruct.

Top-level structs in the ActionScript API are never `null`. If a content property is bound to an empty XML text, a struct without properties is still accessible on the client. This makes it easier to fill initially empty struct properties.

The most convenient way to access a struct property is by means of a value expression. For example, for navigating from a content property bean to the property `bar` of the struct stored in the content property `foo`, you would use the property path `foo.bar`. You can use these property paths in the standard property fields provided by *CoreMedia Studio*. This case is shown in the following code fragment:

Example 5.18. Property paths into struct

```
<?xml version="1.0" encoding="UTF-8"?>
<exml:component xmlns:exml="http://www.jangaroo.net/exml/0.8"
    xmlns="exml:ext.config"
    xmlns:editor="exml:com.coremedia.cms.editor.sdk.config">
  <editor:documentForm itemId="MyDocumentType">
    <items>
      ...
      <editor:stringPropertyField propertyName="foo.bar"/>
      ...
    </items>
  </editor:documentForm>
</exml:component>
```

Dynamic addition of new property values

Structs support the dynamic addition of new property values. To this end, the interface `Struct` provides access to a type object implementing `com.coremedia.cap.struct.StructType` through the method `getType()`. You can call the `addXXXProperty()` methods for various property types during the initialization code that runs after the creation of a document.

Example 5.19. Adding struct properties

```
public function init(editorContext:IEditorContext):void {
  ...
  editorContext.registerContentInitializer("MyDocumentType",
    initStruct);
  ...
}

private function initStruct(content:Content):void {
  var properties:ContentProperties = content.getProperties();
  var struct:Struct = properties.get('foo') as Struct;
  struct.getType().addStringProperty('bar', 200);
}
```

While it is possible to add a property automatically during the first write, this is not recommended. Some property fields cannot handle an initial value of `undefined`. You should therefore only bind property fields to initialized properties.

5.4.5 Types and Property Descriptors

Both `Content` and `Struct` are derived from a common parent interface `CapStruct`, which takes the same responsibilities as its *Unified API* equivalent. It augments `Bean` objects by providing a type in the form of a `CapType`, the common parent of, for example, `ContentType` and `StructType`. Types can be arranged in a type hierarchy and they can be given a name.

A `CapType` provides access to `CapPropertyDescriptor` objects, which describe the individual properties allowed for a `CapObject`. In the `type` property a property descriptor indicates which value the property can take according to the constants defined in `CapPropertyDescriptorType`: string, integer, markup, and so on. Each property descriptor also declares whether the property is `atomic` and accepts plain values or is a `collection` and accepts arrays of appropriate values.

For certain descriptor types, more specific interfaces provide access for additional limitations on the property. A `StringPropertyDescriptor` declares a `length` attribute indicating the maximum length of a string stored in the property. A `BlobPropertyDescriptor` can limit the `contentType` (a MIME type string) of the property values. A `LinkPropertyDescriptor` specifies the type of linked objects and a `MarkupPropertyDescriptor` the grammar of stored XML data.

5.4.6 Concurrency

Being remote beans, the `Content` objects inherit the concurrent behavior of the bean layer. A request to load content data is issued upon first querying any property except for `isDocument()` and `isFolder()`. However, since the response arrives asynchronously and is handled in a subsequent execution, the getter methods will initially return `undefined`. You must therefore make your code robust to handle this situation - which commonly is done by attaching a value change listener that is invoked once the content properties become available, or create a property path expression and use its `loadValue(Function)` method (see Section 5.3.6, “Value Expressions” [44]). Depending on the execution sequence, content may be loaded due to some other, potentially unrelated request before you access it - but your code must not rely on it.

All singletons (`Cap`, `CapConnection`, `CapLoginService`, `session`/`CapSession`, `ContentRepository`, `UserRepository`) and all `ContentType` objects, however, *are* fully loaded before the *Studio* application's initialization process is finished (which is why these interfaces do not extend `RemoteBean`).

When you want to make sure that values have actually hit the server after an update, you can use `RemoteBean#flush(Function)`, and register a callback function.

5.5 Studio Component IoC

This section describes the component IoC concept for *CoreMedia Studio* and its use cases.

5.5.1 Motivation

CoreMedia Studio - based on Ext JS - consists of UI components. Each component is responsible for the user interaction of a local part of the whole Studio UI. The components are organized in the component hierarchy and the state of a component can depend on its place in the hierarchy. For example the publish button in the actions toolbar publishes the current content of the work area. The publish button in the library view however publishes the selected items in the library. So the *content* state of the publish button must be transferred from the outer container (the library or a premular) to the button. Even more: the state of the outer container must be synchronized with the state of child components when it changes.

A straight approach for the requirement combines the Ext JS configuration and the event mechanism: The container defines a value expression of the state and hands it down the component hierarchy - until the target component (the publish button in the example) is finally configured to use the value expression. This approach has two major drawbacks:

1. Many components along the way from the container to the component just pass the state configuration from one hierarchy level to the next without being interested in the state by themselves. This leads to a bunch of boilerplate, error-prone code. For example an intermediate container might forget to pass the configuration.
2. Second there are cases where the state configuration of the container is not available for the child components - For example in a plugin rule you want to add a new button to the toolbar of the library which will do some actions on the selected items. Hence, the value expression of the selected items must be passed to the button but you cannot access the value expression in the plugin rule in the standard, well-defined way. Alternatively all containers along the hierarchy could use the `defaults` configuration to apply default settings to all added items. Again it leads to boilerplate, error-prone code.

5.5.2 Inversion of Control

The approach above passes the state configuration between components that are statically assigned to one another. Instead, the state of a container could be passed dynamically in the runtime to its child components. This is exactly what the Studio component IoC does: It uses annotations in the ActionScript classes to declare which components provide or consume a component state (in the following *context*)

and leverages the Ext JS component hierarchy in the runtime to establish the dependency between the *context provider* and *context consumer*.

5.5.3 Annotations of Context Consumer and Context Provider

Instead of passing the state configuration from the container down to the child component Studio IoC requires that context providers and consumers declare themselves as provider or consumer and in which context they are interested in.

Assume that you have a property `exampleProperty` of a context consumer `exampleConsumer` and want to inject a provided property to the example property. The setter method of the property in the class `exampleConsumer` must be annotated:

```
[InjectFromExtParent]
public function setExampleProperty(value:String):void {
  exampleProperty = value;
}
```

The Studio Component IoC generates the name of the provided property out of the annotated method name. This annotation is called *implicit*. In the example the assumed name of the provided property is `exampleProperty`. But in most cases the method name will not reflect the name of the provided property. Hence, the annotation supports the optional parameter `variable`:

```
[InjectFromExtParent(variable='providedProperty')]
public function setExampleProperty(value:String):void {
  exampleProperty = value;
}
```

This annotation is called *explicit*. The name of the provided property in the example is then `providedProperty`. Still, the annotation is not flexible enough if you want to reuse `exampleConsumer` and to configure the name of the provided property. The optional parameter `variableNameConfig` does the job:

```
[InjectFromExtParent(variableNameConfig='examplePropertyVariableName')]
public function setExampleProperty(value:String):void {
  exampleProperty = value;
}
```

This annotation is called *configurable*. Then in order to inject the provided property, the name of the provided property has to be configured explicitly for each consumer in the EXML:

```
<editor:contextConsumer
examplePropertyVariableName="providedProperty"/>
```

The annotation of a context provider is done in similar way but will not be described in details here. To customize *Studio* you only need to know how to inject a provided

property and which (and where) provided properties exist. Section 7.8, “Customizing Studio using Component IoC” [143] describes how to customize *Studio* using the Studio component IoC.

5.6 Web Application Structure

CoreMedia Studio uses a web application for delivering both static content (like the JavaScript code defining the application) and dynamic content stored in the CMS.

Dynamic content is provided by means of a REST service embedded in a Spring web application context. See http://www.springsource.org/ for details about the Spring framework. In the following section, it is assumed that you know about the essential concepts of the Spring inversion of control (IoC) container.

You can extend and modify the application context by providing additional configuration files in the classpath. Such files must be named according to the pattern `component-*.xml` and they must be put into the directory `META-INF/coremedia`. It is recommended that the variable part of the file name is equivalent to the name of the Maven module in which you define the XML file and optionally Java classes required for your extension.

You must modify the application context to configure your content validation setup. See Section 7.15.1, "Validators" [176] for the details.

5.7 Localization

Text properties in *CoreMedia Studio* can be localized. English and German are supported out of the box; you can add your own localization bundles if required. To do so, proceed as follows:

1. Add the new locale to the `studio.locales` property in your Studio application's `application.properties` file.

   This property contains a comma-separated list of locales. The first element in the list is `en` and specifies the locale of values in the default properties files (that is, the files without a locale suffix). Therefore, you must not change this first entry; it must always remain `en` (see below).
2. Add properties files that follow the naming scheme for your added locale, as explained below.

Localized texts are stored in property files according to the Java property file syntax. The naming scheme of these files is:

```
<FileName>_<IsoLanguageCode>.properties
```

A property file with no language code contains properties in the default language English. Note that English is only a technical default. The default locale used for users opening *CoreMedia Studio* for the first time is determined by the best match between their browser language settings and all supported locales.

When one or several properties are missing in locale-specific properties files, their values are inherited from the default language (that is, they will appear in English rather than in the locale the user has set). However, there must be a `.properties` file for every supported locale as per the `studio.locales` property in the `application.properties`. The locale-specific properties file may be empty, but a missing file will result in an error on Studio startup.

Property files are placed beside ActionScript and EXML files in the proper package below the `src/main/joo` directory. They are compiled into ActionScript classes with the base file name (without the ISO language code), followed by `_properties`.

Each such class contains a static field `INSTANCE` which at runtime references an object declaring one attribute for each property declared in the `*.properties` file, using the property key as the attribute name and the localized value depending on the selected locale. For each property key that is a valid ActionScript identifier, the generated class also declares a getter function to simplify typed access.

Overriding existing properties

If you want to change predefined labels, tooltips or similar, you can override properties from existing properties classes. To this end, you should first define a new property class and then call the static method `ResourceBundle#override`

`Properties(destination, source)` to overwrite an existing property class with the values stored in your new property class. This method will never remove a property key, it will only update existing values.

Note that the call to `ResourceBundle.overrideProperties` references the ActionScript classes, not the property maps reachable through the static `INSTANCE` fields.

Generally, each Studio plugin module will contain at least one set of property files for localizing its own components or for adapting existing property files.

5.8 Multi-Site and Localization Management

CoreMedia provides a concept to handle multi-site and multi-language in a standardized way.

Configuration

The CoreMedia Site Model is defined via the bean `siteModel` of the *CoreMedia Studio* web application. Please refer the to the [[CoreMedia Digital Experience Platform 8 Developer Manual]] to know, how CoreMedia has designed multi-site and multi-language support.

SitesService

To access all the features of multi-site and multi-language, you can use the `SiteService`. The `SitesService` is available via the `EditorContext` with its `getSiteService()` Method.

With this, you have access to all available Sites and their properties - the root folder, the site indicator, etc. Furthermore, you have access to the Site Model specifications like the properties for master relations or of which document type the Site Indicator is. For a detailed understanding, you are asked to read the Studio API documentation as well.

5.9 Further Reading

At http://www.senchaexperts.com/api/extjs3.3/ you can find the API documentation of Ext JS 3.3.

http://www.jangaroo.net/ and https://github.com/CoreMedia/jangaroo-tools/wiki describe the Jangaroo language and tool chain.

http://cksource.com/ provides information about the rich text editor CKEditor.

The documentation of the ActionScript API is linked from the documentation page of *CoreMedia DXP 8*. The overview page can be found at `https://documentation.coremedia.com/cm7/overview/` . Note that classes or interfaces not mentioned in the API documentation pages are not public API. They are subject to change without notice.

The remote API for content is closely related to the *Unified API* provided for Java projects, although there are changes to accommodate for the different semantics of the base languages. Still, the *Unified API Developers Guide* gives a good overview of the involved concepts when dealing with content. Documents, folder, versions, properties, types, and the like are explained in detail as well as the structuring of the API into repositories, identifiable objects and immutable values.

6. Using the Development Environment

This section describes how to connect the *Content Server* and the *Preview CAE*. It provides pointers to information on Jangaroo tools supporting the build process and the IntelliJ IDEA IDE. Furthermore, some basic information on debugging Studio customizations is given.

6.1 Configuring Connections

CoreMedia Studio needs to be connected with the *Content Management Server* to access the repository and with the preview *CAE* to show the preview of the opened form. If you use *CoreMedia Blueprint*, everything is already configured properly for your local workspace. If you use a distributed environment you have to adapt the following properties:

Connecting with the Content Server

When you start the *Studio* web application with `mvn tomcat7:run` during development, you can configure the connection by supplying the arguments `-Dcontentserver.host=MYHOST` and (optionally) `-Dcontentserver.port=MYPORT` at the command line. Alternatively, you can configure the connection in the `application.properties` file in the `src/main/webapp/WEB-INF` directory. Use the `contentserver.host` and `contentserver.port` properties for the host and port of your *Content Server* , respectively.

Refer to the [CoreMedia DXP 8 Manual] to learn about building deployable artifacts.

Connecting with the Preview CAE

When you start the *Studio* web application with `mvn tomcat7:run` during development, you can configure the connection to the preview *CAE* in the `override-web.xml` file in the *CoreMedia Studio* web application directory. Simply change the value of the parameter `ProxyTo` to the URL of your *CAE*. When you deploy *Studio* in an application server like Tomcat, you should change the `application.properties` in the `src/main/webapp/WEB-INF` directory. The property `studio.previewUrlPrefix` contains the path to the preview controller up to, but not including the `preview` suffix.

If a different prefix is required for the final deployment, you have to add the `studio.properties` file to the deploy workspace in the `WEB-INF` directory and make the appropriate changes.

The property `studio.previewControllerPattern` contains the configurable preview controller pattern. If it is empty or not defined, then the *Studio* web application will use the default preview controller pattern `preview?id={0}`. If you want to use simple numeric IDs instead, then you can configure in the `studio.properties` as the following: `studio.previewControllerPattern=preview?id={1}`. The placeholder `0` and `1` are representing the CoreMedia ID and the numeric ID, respectively.

Note that *Elastic Social* users and user comments do not have numeric IDs. Hence, you should configure `preview?id={0}`. However, when using `preview?id={1}`, the placeholder `1` is replaced with the non-numeric ID as well and the preview application has to handle this special case or will fail to deliver.

6.2 Build Process

CoreMedia Studio provides artifacts for use with Maven. Since *CoreMedia Studio* builds upon Jangaroo, its build process is basically identical to Jangaroo's. The Jangaroo compiler documentation explains how to use the Jangaroo tools from the command line and how to use them with Ant or Maven. This covers the conversion from EXML to ActionScript and further down to Ext JS.

A detailed description of the Jangaroo build process with Maven is given in the Jangaroo tools wiki.

In the following section, you will find a description of some of the typical use cases that appear during *CoreMedia Studio* development using the *CoreMedia Project* workspace.

Compiling the Studio Project

Open a command line at the *CoreMedia Project* root directory. To compile all Studio modules, change to the `modules/studio/` directory and run

```
mvn clean install
```

This will remove all generated files before starting the compilation. To only recompile updated files, run

```
mvn install
```

Running the Studio Web Application

In the `modules/studio/studio-webapp/` directory of *CoreMedia Blueprint*, you can start the Studio web application in a Tomcat servlet container via Maven, like so:

```
mvn tomcat7:run
```

The recommended way of dynamically reassigning the server URLs that you want Studio to connect to is to add one or several Maven profiles to your local `settings.xml`, that redefine connection properties as follows:

```
<profile>
  <id>myStudio</id>
  <properties>
    <installation.host>myserver.mycompany.com</installation.host>
    <database.host>mydatabase.mycompany.com</database.host>
    <solr.host>mysolr.mycompany.com</solr.host>
    <mongo.db.host>mymongoserver.mycompany.com</mongo.db.host>
  </properties>
</profile>
```

You can then start your local Studio development web application by running

```
mvn -PmyStudio tomcat7:run
```

When only EXML and ActionScript files are recompiled from within IntelliJ IDEA, the Tomcat servlet container automatically serves the updated compiled class files. There is no need to stop and restart Tomcat.

In contrast, before recompiling any Java files, make sure to kill the Tomcat process. The Java Virtual Machine might not be able to load additional classes when JAR files are modified concurrently.

Configuring the Build Process

The Jangaroo compiler can be configured to check whether compiled code uses non-public API. To this end, the parameter `publicApiViolations` of the Jangaroo Maven plugin controls how the compiler handles usages of non-public API classes in your project code. The parameter can take the values `warn` to log a warning whenever such a class is used, `allow` to suppress such warnings, and `error` to stop the build with an error. The default value is `warn`, but you can set it to `error` as follows:

Example 6.1. Detecting public API violations

```
<plugin>
  <groupId>net.jangaroo</groupId>
  <artifactId>jangaroo-maven-plugin</artifactId>
  <configuration>
    <publicApiViolations>error</publicApiViolations>
  </configuration>
</plugin>
```

6.3 IDE Support

One of the rationales behind Jangaroo is to make the good parts of static typing, such as getting reliable and useful IDE support, available for the dynamic language JavaScript. This is described in more detail in the Jangaroo tools wiki.

Recent versions of the IDE *IntelliJ IDEA Ultimate Edition* have built-in support for ActionScript and JavaScript development. Jangaroo provides an IDEA plugin called *Jangaroo 0.9*, which bundles the functionality of two older plugins (*Jangaroo Language* for compiling ActionScript as described at https://github.com/CoreMedia/jangaroo-tools/wiki/Developing-Jangaroo-Applications-with-IntelliJ-IDEA and *Jangaroo EXML* for compiling `.exml` files as described at https://github.com/CoreMedia/jangaroo-tools/wiki/Developing-Ext-AS-Applications-with-IntelliJ-IDEA). The older plugins should not be activated jointly with the current unified *Jangaroo 0.9* plugin. They are still available for projects working with older Jangaroo versions, but will eventually be deleted.

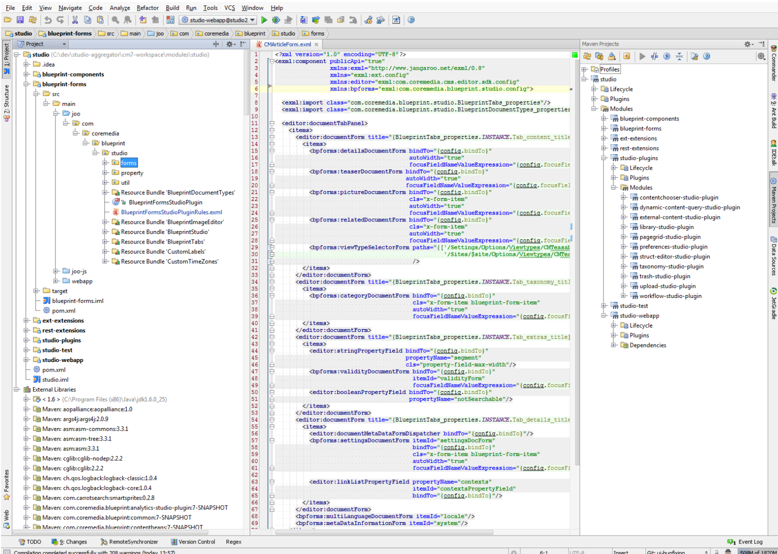


Figure 6.1. Studio project within the Project workspace in IntelliJ Idea

6.4 Debugging

CoreMedia Studio components and plugins consist of static resources (images, style sheets, JavaScript files) and JavaScript objects. Debugging a custom *CoreMedia Studio* component or plugin involves the following tasks:

- check whether the static resources have been loaded
- explore the runtime behavior of the customization, that is, the relevant JavaScript code or DOM nodes

In this section, tools and best practices for debugging your *CoreMedia Studio* customizations are described.

6.4.1 Browser Developer Tools

All modern browsers provide tools for web application debugging. These are usually simply called "Developer Tools" and can be invoked via a menu entry, a toolbar button, the F12 key or the key combination Ctrl+Shift+I.

As of today, using Google Chrome for debugging is recommended, since it currently offers the most mature developer tools and is the fastest, especially while debugging. Internet Explorer 11 is quite good in both disciplines, too, while Firefox trails the field especially in execution performance during debugging.

All modern browsers' developer tools provide tabs for different tools:

- DOM Explorer / Element / Inspector — Inspect the page's actual DOM elements as a DOM tree, with the option to select an element on the rendered page to reveal it in the tree. Selected DOM tree nodes are highlighted on the rendered page. The DOM can be watched for changes and modified interactively.
- Console — All JavaScript messages and errors are logged to this console, and it provides a read-eval-loop for JavaScript expressions.
- Network — Inspect all HTTP network traffic between the client-side application and the server, static resources as well as Ajax (XHR) requests. Most developer tools offer to disable the cache while they are active, to make sure that you always load the most recent version of code and other resources you just changed.
- Debugger / Sources — Inspect all loaded JavaScript and CSS sources, set breakpoints to debug in step-by-step mode. Most modern developer tools allow you to change sources interactively with immediate effect.
- Profiles / Profiler / Audits / Memory / Analysis — Diverse tools to measure your web application's client-side and network performance and memory

usage. Helpful to find memory leaks (see below) and track performance issues.

Using the debug versions of the JavaScript files

Since *CoreMedia Studio* is a Jangaroo application, please refer to the tutorial about Jangaroo debugging with Firebug at https://github.com/CoreMedia/jangaroo-tools/wiki/Tutorial-~-Debugging. Essentially, you have to load *CoreMedia Studio* with the `#joo.debug` parameter appended to the URL to debug the JavaScript code of your component. This parameter loads the debug versions of the JavaScript files. In particular, it loads every class in a separate file, which greatly simplifies debugging. In debugging mode, both the *Network* tab and the list of loaded scripts in the *Sources* / *Debugger* tab show the script files of your components. The line numbers in the script files match the line numbers from your ActionScript source files, which simplifies setting breakpoints at the appropriate spots in your code. Also, third-party-libraries like Ext JS and CKEditor are loaded in their human-readable (as opposed to "minified") versions when in Jangaroo debug mode. Last but not least, for developer convenience, *CoreMedia Studio*, skips the confirmation dialog that normally appears before reloading (F5).

All browser developer tools offer a convenient way to navigate to a certain script file or Jangaroo class (which, in debug mode, is a one-to-one mapping): With the Sources / Debugger tab active, press Ctrl-P (note that this invokes the print dialog when the focus is not on the developer tools!) and just start typing the name of the class (file) you want to debug, and the list is filtered incrementally. Some tools even support typing camel case prefixes of the class name, for example to find the class PreviewPanelToolbarBase in Google Chrome, press Ctrl-P and type "PrevPaToBa" to quickly reduce the number of suggestions.

To navigate to the desired line in the file, you can either add a colon (:) and the line number directly after the file search term, or press Ctrl-L or Ctrl-G (Goto Line) and enter the line number.

A very efficient way to locate a certain line of a Jangaroo class in Google Chrome's Developer Tools (to set a breakpoint, for instance) when working with IntelliJ IDEA is as follows. In IDEA, jump to the very start of the line (press Pos 1 repeatedly until there). Then, press Ctrl-Alt-Shift-C ("Copy Reference"). IDEA's status line shows a message that the file/line reference has been copied to the clipboard. Switch to Chrome Developer Tool's Sources tab (Alt-Tab suffices when changing back and forth) and press Ctrl-P. Now paste the file/line reference and replace the "a" of ".as" by "j" (for ".js"). The fastest way to do so is to use Ctrl-Left-Arrow twice, then Shift-Right, then type "j". Hitting Return, Chrome accepts the syntax file-path:line and takes you to the exact file and line.

The debugger allows you to set breakpoints, to automatically pause on errors, to step through the script at runtime and to evaluate expressions in the current scope of the script. In this context, the **Console** tab is also very helpful, because it offers a JavaScript shell for direct interaction with the current script. The console displays

the results of the expressions evaluated in the shell and also messages generated by the current script runtime.

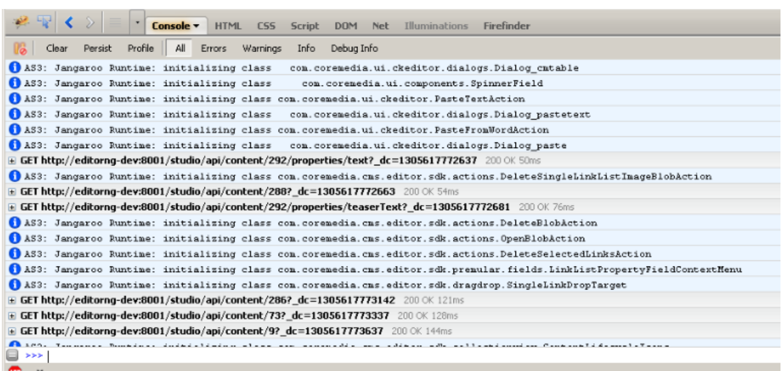


Figure 6.2. Firebug: console

6.4.2 Ext JS `debug.js`

Ext JS comes with a built-in debug console. Before you can use the console, you have to run *CoreMedia Studio* in debug mode by appending the `#joo.debug` URL parameter as described in the previous section. Then, you can activate the console by executing

```
Ext.log();
```

in the JavaScript console. The Ext debug console offers capabilities tailored for debugging the Ext component tree.

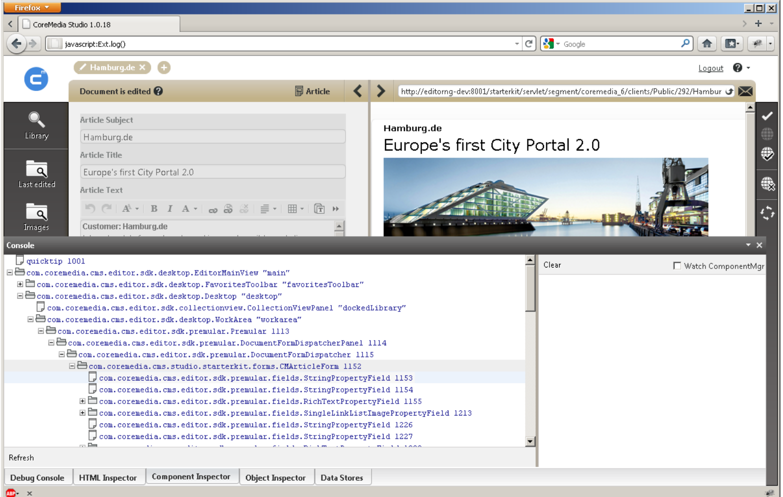


Figure 6.3. Ext component tree

The **Component Inspector** tab shows the Ext component tree, displaying the components' `xtypes` and `ids`. The element associated with the selected node is highlighted. The figure below shows that the properties form of the active article is highlighted when the component of xtype `CMArticleForm` with id *1152* is selected. Double-clicking this component opens a new view showing the properties of this component.

The Ext debug console also offers capabilities to explore the HTML structure of the current document and to execute JavaScript. However, compared to the browser's developer tools, these capabilities are rather limited.

6.4.3 Illuminations

Illuminations for Developers is a commercial third-party Firebug add-on that makes developing more intuitive when using Ext JS. It can be purchased at http://www.illuminations-for-developers.com.

Illuminations changes the concept of inspecting from HTML elements to Ext JS components in an extra overview panel in Firebug for Ext JS called Illuminations. The Illuminations panel lets you inspect widgets (usually derived from `Ext.Component`, but not always), data (Ext stores, records/models, fields), and elements (Ext. Element). These views show the hierarchical structure that results from your code.

Illuminations makes it easier to understand the Ext JS framework, makes objects more transparent and helps to debug the code.

Object Naming

Illuminations recognizes objects as named objects instead of "Object" in the console. Additionally, it gives you the information about the ID of the current component and the corresponding value.

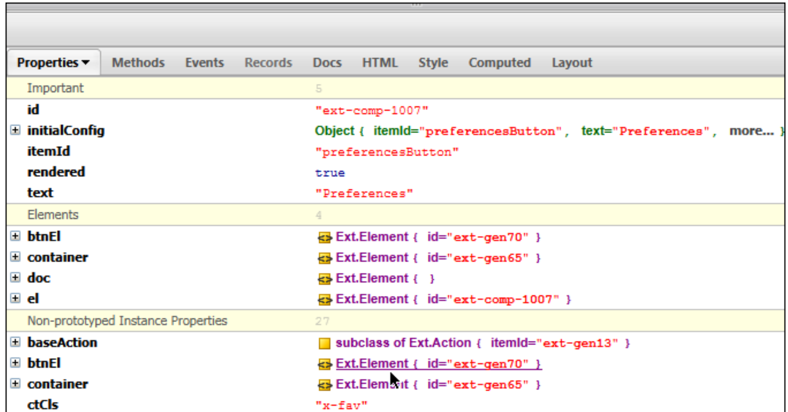


Figure 6.4. Illuminations: objects

Method Naming

Utilizing the option "Name Methods" as found in the Illuminations panel options menu, you get more telling names.

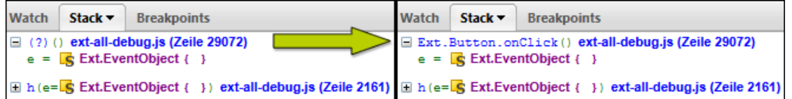


Figure 6.5. Illuminations: methods

Element Highlighting

When you hover the mouse over the items in the Illuminations panel, Illuminations highlights the components on the page, as hovering over an HTML element in firebug would do. It works for Ext components, Ext elements and composite elements.

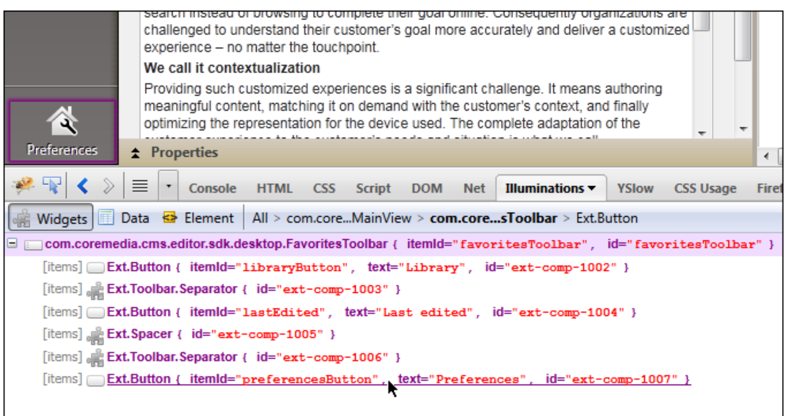


Figure 6.6. Illuminations: highlighting

Contextual Menu

By right-clicking on an element of the page, you can open a context menu with a new inspect item to open the selected Ext component in the properties panel. Ideally, Illuminations inspects some sort of UI widget, else an Ext element.

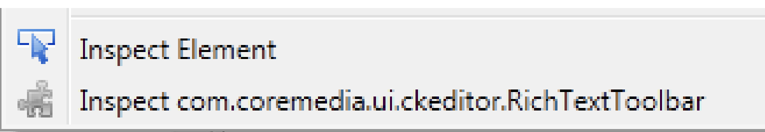


Figure 6.7. Illuminations: inspect

6.4.4 Tracing Memory Leaks

Ext JS applications can consume high amounts of memory in the browser. As long as memory is de-allocated when UI elements are disposed, the user has the choice to limit memory usage. But it becomes a problem when there are *memory leaks*. Fortunately, reloading the application's page (F5), with a few exceptions, frees memory again, but still, frequent reloading is undesirable for the user.

Memory leaks occur when an object is supposed to be no longer used, but undesired references to that object remain that keep it "alive", that is, from being garbage-collected. Such references are called *retainers*. In an Ext JS applications, such retainers are typically

- Ext's component manager. It maintains a global list of all active components. See below how to tackle memory leaks cause by the component manager (component leaks).
- Event listeners. When attaching your event listener function to some object, that object retains the event listener function and every object in the scope of that function, typically at least `this`.

- Drop zones. Like for components, Ext keeps a global list of all active drop zones. So when your custom component creates a drop zones, remember to explicitly destroy it together with your component.

Component Leaks

If a component is destroyed, it and, if it is a container, all its items, are removed from Ext's component manager registry. But there are cases when components fail to be destroyed:

- If two items of the same container use the same itemId, Ext does not complain, but one of them is kept even if the container is destroyed.

  Components that are created manually via ComponentMgr.create() have to be destroyed manually unless they are added to the items of a container.

Memory Leaks Caused by Non-Detached Listeners

Always remove any listeners that you attach to an `ext.util.Observable`, `com.coremedia.ui.data.Bean`, `com.coremedia.ui.data.ValueExpression`, or any other object that emits events. Even when using the option `{single: true}`, the event might not have been fired at all when your component is destroyed.

A typical error pattern is to attach some method `handleFoo` as event listener, but by mistake hand in another method with a similar name `handleFuu` when intending to remove the listener. No error whatsoever is reported, because trying to remove a function as listener that is not in the current set of listeners is silently ignored by `Observable#removeListener()` and all other event emitters.

A useful utility to automate removing listeners is to use `Observable#mon()` instead of `Observable#on()` (alias: `Observable#addListener()`). `mon` does not attach the listener to the caller, but to the first parameter, but binds it to the lifetime of the caller. For example, when your custom component creates a DOM element `elem` and registers a `click` listener like so: `this.mon(elem, "click", handleClick)`, the listener is automatically detached when your component (the caller, `this`) is destroyed.

It never makes sense to call `comp.mon(comp, ...)`, because when a component is destroyed, it removes its own listeners, anyway. Using `comp.mon(comp, "destroy", handleDestroy)` even leads to the handler *never* being called, because a component removes all `mon` listeners already in its `beforedestroy` phase. In contrast, `comp.on("destroy", handleDestroy)` works as expected.

Not only components, but any objects that register event handlers, most prominently actions, have to detach all event handlers again.

As actions do not have a `destroy` event and `onDestroy` method like components, you have to override `addComponent()` and `removeComponent()` to detect when an action starts and ends being used by any component. Introducing a simple counter field starting with zero, you should acquire resources (for example, register event listeners, populate fields) when `addComponent()` is called while the counter is zero before increasing, and release resources (remove event listeners, set fields to `null`) when `removeComponent()` is called while the counter is zero after decreasing.

To minimize the impact in case event listeners are not detached, and to avoid cyclic dependencies, keep the scope of any event handler function or method as small as possible. In the optimal case, the event handler function is a private static method, for example if it just toggles a style class of the DOM element given in the event object:

```
private function attachListeners():void {
  var el:Element = getEl();
  // bad style: using an anonymous function that
  // does not need its outer scope at all:
  el.addListener("mouseover", function(e:IEventObject) {
    e.getTarget().addClass("my-hover");
  });
  // good style: for such cases, use a static method:
  el.addListener("mouseout", removeHoverCls);
}

private static function removeHoverCls(e:IEventObject):void {
  e.getTarget().removeClass("my-hover");
});
```

If your event handler only needs access to `this`, declare it as a method as opposed to an anonymous function:

```
private var hoverCounter:int = 0;

private function attachListeners():void {
  var el:Element = getEl();
  // bad style: using an anonymous function that
  // only needs to access "this":
  el.addListener("mouseover", function(e:IEventObject) {
    ++hoverCounter;
  });
  // good style: for such cases, use a (non-static) method:
  el.addListener("mouseout", countHoverEvent);
}

private function countHoverEvent(e:IEventObject):void {
  ++hoverCounter;
});
```

In ActionScript, like in JavaScript, anonymous or inline functions have lexical scope, that is they can access any variable declared in the surrounding function or method. Since this scope usually contains a reference to the object that emits events (here: `el`), and that object stores your event handler function in its listener set, you create a cyclic reference between the two. Cyclic references are not bad per se, because garbage collection can handle them if all objects contained in the cycle are not

referenced from "outside". But firstly, as long as any of the objects is kept alive, all others are retained, too, and secondly, as discussed below, this makes finding the real culprit for memory leaks harder.

Memory Leaks Caused by Other References

Any reference to an object can cause it to stay alive. Thus, to find unwanted retainers, it makes sense to null-out all references a component keeps in its `onDestroy()` method, like in this code sketch:

```
public class MyComponent extends Component {

  private var foo:SomethingExpensive;

  public function MyComponent(config:myComponent) {
    super(config);
    foo = new SomethingExpensive();
  }

  protected function onDestroy():void {
    foo = null;
    super.onDestroy();
  }
}
```

You have to be careful that even after your component has been destroyed, certain asynchronous event callbacks may occur. Your event handlers have to be robust against fields already being `null`. Consider this example using a fictitious `timeout` event:

```
public class MyComponent extends Component {

  private var foo:SomethingExpensive;

  public function MyComponent(config:myComponent) {
    super(config);
    foo = new SomethingExpensive();
    addListener("timeout", handleTimeout);
  }

  private function handleTimeout():void {
    // Although we remove the listener in onDestroy,
    // an event may already be underway, so foo may
    // already be null in time it arrives:
    if (foo) {
      foo.doSomething();
    }
  }

  protected function onDestroy():void {
    removeListener("timeout", handleTimeout);
    foo = null;
    super.onDestroy();
  }
}
```

Detecting Memory Leaks

To check whether your customized Studio contains any component leaks, proceed as follows.

1. Open the suspicious UI, for example, a document tab containing your new property field. Wait until everything is rendered correctly and close the UI again. This is to ensure that helper components (a context menu, for instance) that are shared between instances and created with the first instance do not blur the view on real component leaks.
2. Store a snapshot of the current Ext component manager registry by executing the following command in the JavaScript console:

```
before = Ext.ComponentMgr.all.items.concat()
```

3. Open and close the UI again like before. Take a second snapshot:

```
after = Ext.ComponentMgr.all.items.concat()
```

4. In theory, the second snapshot should be exactly equal to the first. But some components are recreated occasionally, which is not bad if their old version is correctly destroyed. Thus, the first check is to simply compare the component count:

```
after.length - before.length
```

5. If there are more components in the second snapshot (positive difference), next goal is to determine their component type (xtype). This is achieved by the following code:

```
newComponents = after.filter(function(c) {
  return before.indexOf(c) === -1;
})
```

6. To get an overview of the new components, count how many components are of which type (xtype), using the following code:

```
byXtype = {};
newComponents.forEach(function(c) {
  var xtype = c.constructor.xtype;
  byXtype[xtype] = (byXtype[xtype] || 0) + 1;
});
byXtype
```

7. For custom EXML components, the xtypes in the resulting map indicate the config package, from which you can derive the Maven module, and the config class name, which corresponds to the EXML file name (using an upper case first letter).

To check whether your customized Studio contains any other memory leaks, proceed as follows.

1. Always append `#joo.debug` to the Studio web-app URL (see above). The representation of heap snapshots is a lot more detailed (at least in Chrome) and should even display your ActionScript class names as (guessed) object types.
2. Open the suspicious UI, for example, a document tab containing your new property field. Wait until everything is rendered correctly and close the UI again. In addition to what has been said regarding component leaks, this is to ensure that all needed data objects (remote beans) have been fetched from the server. In Studio, remote beans are cached, so they are not garbage-collected on purpose.
3. Take a heap snapshot. In Google Chrome, this is achieved as follows. In Developer Tools, select "Profiles". Under "Select profiling type", the option "Take Heap Snapshot" is preselected. The third option, "Record Heap Allocations", claims to be suitable for isolating memory leaks, but CoreMedia founds comparing heap snapshots simpler. Press the button "Take Snapshot". In the left column, Chrome adds an icon for the snapshot and shows a progress indicator while it is recorded. When recording is finished, the heap snapshot is shown as an expandable list of all JavaScript objects is shown, grouped by their (internal) type.
4. Repeat opening and closing the suspicious UI like in step 2.
5. Take a second heap snapshot. To do so, either you have to select "Profiles" on the left and proceed like in step 3, or simply click the "record" button (a gray filled circle).
6. Where the label "Summary" is shown, you can switch to "Comparison". The first snapshot is automatically selected for comparison. Now, you no longer see all objects, but only those that either have been removed ("Deleted") or have been created ("New") between snapshot one and two ("Delta").

Since the application is in the same state after opening and closing the suspicious UI, ideally, the comparison would be empty. In practice, however, this can never be achieved. What you have to look for are "expensive" objects, consuming lots of memory ("Alloc. Size", "Freed Size", "Size Delta"). The focus is "Size Delta", which tells you how much memory has leaked between snapshot one and two.

Since you cannot do much about memory leaks in Ext JS or in Studio Core, concentrate on your own extensions. Fortunately, you have loaded Studio with `#joo.debug`, and Chrome's Profiler manages to find the Jangaroo class names of objects. Thus, you can filter the comparison by the name of your ActionScript class, and it will only show objects of that class whose set of instances has changed.

Each entry in the upper part represents the set of all object. To inspect a concrete instance and its retainers, you have to expand the entry using the triangle / arrow,

and select an instance from the expanded list. For the selected instance, all retainers are now shown in the lower part of the heap analyzer.

Each root node in the "Retainers" tree represents the property of the instance directly referencing (retaining) the instance selected in the upper part. By expanding any node, you can drill down into its retainers, until you reach an instance that is globally retained, usually by the global JavaScript object `window`.

By default, the heap analyzer sorts child nodes by "Distance" (first column), so that you inspect the longest path when always expanding the first child node. This most likely, but not necessarily leads you to the "culprit" retainer, that is the instance that should no longer refer to the inspected instance. Many other retainers result from cyclic references, that is, they would have been garbage-collected together with the inspected object, if the "culprit" did not reference the inspected object. This is why it is recommended to reduce the number of references by cleaning up fields and listeners, even if this would not have been necessary without the memory leak (see above).

Hopefully, by inspecting retainers, you'll find a listener that has not been detached or a global reference that should be removed on destroy. If not, you can still clean up your component or action so that it at least leaks less memory.

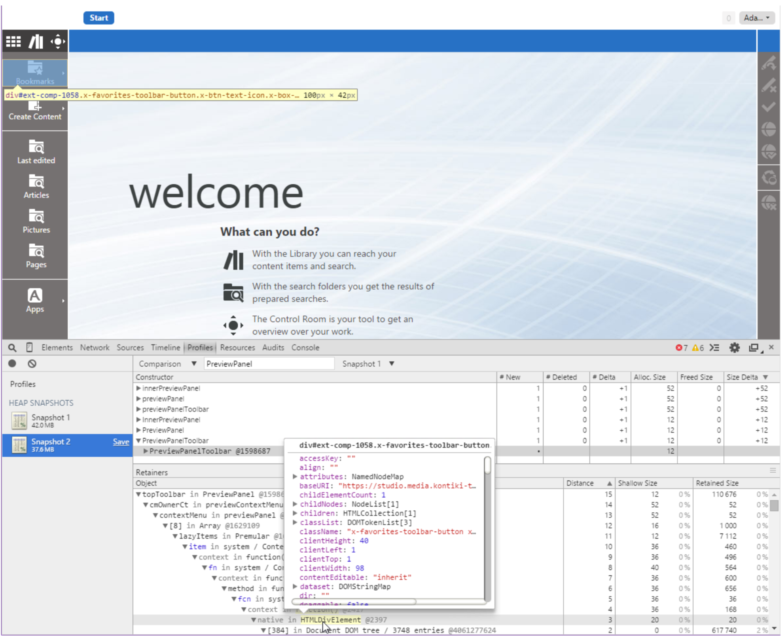


Figure 6.8. Google Chrome's Developer Tools Support Comparing Heap Snapshots

The screenshot shows Google Chrome's developer tools in action. Blueprint Studio has been loaded in debug mode. A document tab has been opened and closed

again, "Snapshot 1" has been taken, and after repeating this, "Snapshot 2" has been added. Then, both snapshots have been compared as described above and the developer has filtered for "PreviewPanel". The only retained instance of `PreviewPanelToolbar` has been selected, so that its retainers are shown in the lower part. In the expanded path, the mouse hovers over the almost-leaf `HTMLDivElement`, which is also automatically highlighted in the Studio UI. This reveals the culprit of the memory leak: The highlighted "Bookmarks" button in the favorites toolbar is the one who keeps an indirect reference to the `PreviewPanel` through its context menu.

7. Customizing CoreMedia Studio

This chapter describes different customization tasks for *CoreMedia Studio*.

- Section 7.1, “Studio Plugins” [84] describes the structure of *CoreMedia Studio* plugins.
- Section 7.2, “Localizing Labels” [94] describes how you can localize labels of *CoreMedia Studio*.
- Section 7.3, “Document Type Model” [97] describes how you can adapt *CoreMedia Studio* to your document type model, for example by localizing types and properties, defining document forms, and so on.
- Section 7.4, “Customizing Property Fields” [115] describes how you can create custom property fields and how you can customize the existing rich text property field.
- Section 7.6, “Coupling Studio and Embedded Preview” [138] describes how you can couple the Preview and Form of a document in the JSP templates of the *CAE* preview.
- Section 7.8, “Customizing Studio using Component IoC” [143] describes how to customize *CoreMedia Studio* using the Studio component IoC.
- Section 7.9, “Customizing Central Toolbars” [145] describes how to customize the CoreMedia toolbar with additional search folders or custom actions.
- Section 7.11, “Customizing the Library Window” [152] describes how you can customize the Library Window.
- Section 7.12, “Work Area Tabs” [160] describes how to integrate your own tab to *CoreMedia Studio*. how to determine which tabs are opened at start time and how to add actions to the work area tab context menu.
- Section 7.13, “Dashboard” [166] describes how to configure the dashboard of *CoreMedia Studio*.
- Section 7.14, “Configuring MIME Types” [175] describes how to configure MIME types for additional file types for *CoreMedia Studio*.
- Section 7.15, “Server-Side Content Processing” [176] describes how the processing of content can be influenced by custom strategies and how inconsistencies in the content structure can be detected or avoided.
- Section 7.16, “Available Locales” [186] describes how *CoreMedia Studio* assists the user in choosing a locale and how to configure the available locales.
- Section 7.17, “Notifications” [187] describes how to enrich *CoreMedia Studio* with custom notifications.

7.1 Studio Plugins

The way to easily customize and extend *CoreMedia Studio* is by using plugins. The Studio module in the *CoreMedia Blueprint workspace* demonstrates the usage of the plugin mechanism, and defines several plugins for Studio.

Note that a Studio plugin is not to be confused with an *Ext JS* component plugin. The former is an application-level construct; Studio plugins are designed to aggregate various extensions (custom UI elements and their functional code, together with the required UI elements to trigger the respective functionality). The latter means a per-component plugin and is purely an *Ext JS* mechanism. This section deals with Studio plugins; *Ext JS* plugins are described in Section 5.1.3, "Component Plugins" [30]. In this manual, the terms *Studio plugin* and *component plugin* are used, respectively, to avoid ambiguity.

Examples for *CoreMedia Studio* extension points that plugins may hook into are:

- Localization of document types and properties
- Custom forms for document types
- Custom collection *thumbnail view*, and custom columns in collection *list view*
- Custom tab types (example in Blueprint: Taxonomy Manager tab)
- Custom library search filters
- Allowed image types and respective blob properties for drag and drop into rich text fields
- Additional extensions to extension menu
- Document types without a valid preview

A plugin for *CoreMedia Studio* usually has the following structure:

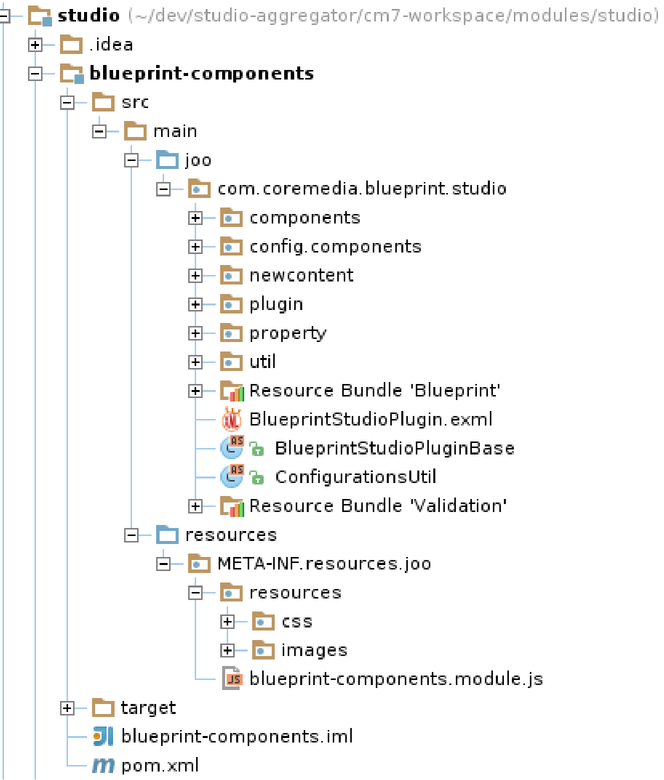


Figure 7.1. plugin structure

Structure of example

The example above depicts the layout of a typical Studio module in the *CoreMedia Blueprint workspace*. All plugins contain a `pom.xml` file that defines the dependencies of the plugin. The actual source code goes into a subdirectory named `joo`. The `resources` subfolder contains some bootstrapping code to register the plugin with *CoreMedia Studio*, and it may also contain additional static resources such as images or CSS files.

The module `blueprint-components`, for example, has a main package `com.coremedia.blueprint.studio` and holds two resource bundles, and an EXML file declaring the plugin and its applicable rules and configuration.

It is recommended to put the source of your plugin into a custom package. This package is reflected in the folder structure below `joo`. The package name for the example above is `com.coremedia.blueprint.studio` as it is *CoreMedia Blueprint*'s main Studio plugin.

Each plugin is described in an EXML file (in this example, this is `BlueprintStudioPlugin.exml`). This file declares the plugin's rule definitions (that is the various

Studio extension points that this plugin hook into) and configuration options. Typically, that EXML file is sufficient for a plugin declaration.

However, if you want to run arbitrary ActionScript code as part of your plugin's initialization phase, you can also introduce an ActionScript base class. In this case, you need to declare that base class in your main EXML file, make your base class extend `StudioPlugin`, and then override the `init()` method in your base class.

The Main Class

The main class of a plugin can either be defined as ActionScript code or as EXML. In the example in Figure 7.1, “plugin structure” [85] the main class is `BlueprintStudioPlugin` as EXML. For your own plugins, it is recommended to use a name schema like `<your plugin name>StudioPlugin`.

The main class for a plugin must implement the interface `com.coremedia.cms.editor.sdk.EditorPlugin`. The interface defines only one `init()` method that receives a context object implementing `IEditorContext` as its only parameter, which is supposed to be used to configure *CoreMedia Studio*.

In ActionScript, you can simply implement the interface in your source code. In EXML, on the other hand, you cannot implement interfaces. Therefore, Studio provides a base EXML element to inherit from, namely `<editor:studioPlugin>`. The corresponding ActionScript class `com.coremedia.cms.editor.configuration.StudioPlugin` not only implements the `IEditorContext` interface, it also delegates the `init()` call to all Studio plugins specified in its `configurations` config option.

The `IEditorContext` instance handed in to the `init()` method can be used for the following purposes:

- Configure which document types can be instantiated by the *CoreMedia Studio* user. This basically restricts the list of content types offered after clicking on the Create Document Icon in the Collection View (see Section 7.3.8, “Excluding Document Types from the Library” [113] for details). Note that only those documents are offered in the create content menu that the current user has the appropriate rights for in the selected folder - excluded document types will be placed on top of that rule (that is, you can exclude document type X from the menu even when the user has technically the rights to create documents of type X).
- Configure image properties for display in the thumbnail view and for drag and drop;
- Register hooks that fill certain properties after initial content creation (see Section 7.3.9, “Client-side initialization of new Documents” [114] for details);
- Add properties to the localization property bundles, or override existing properties (see Section 7.2, “Localizing Labels” [94] for details),
- Get access to the central bean factory and the application context bean,

- Get access to the REST session and indirectly to the associated repositories.
- Register content types for which *Studio* should not attempt to render an embedded preview
- Register a transformer function to post-process the preview URL generated for an existing content item for use in the embedded preview
- Get access to persistent per-user application settings, such as the tabs opened by the user or custom search folders
- Register symbol mappings for pasting external text from the system clipboard into a RichText property field, which can be useful when you have to paste documents from Microsoft Word with special non-standard characters

Note that a Studio plugin's `init()` method is allowed to perform asynchronous calls, which is essential if it needs server-side information (access user, groups, Content, and so on) during initialization. *CoreMedia Studio* waits for the plugin to handle all callbacks, only then the next plugin (if any) is initialized and eventually, *CoreMedia Studio* is started. However, you cannot use `window.setTimeout()` or `window.setInterval()` in Studio plugin initialization code!

Plugin Rules

The other essential part of a *CoreMedia Studio* plugin is the plugin rules it declares in its `<ui:rules>` element. Plugin rules are applied to components whenever they are created, which allows you to modify behavior of standard *CoreMedia Studio* components with component plugins. The `BlueprintStudio` plugin, for example, declares rules that add buttons to the favorites toolbar and to the preview panel's toolbar.

The rules element

The studio plugin file consists of one "rules" element that contains component elements. The components can be either identified by their global id or by namespace and xtype. For the latter case, you need to declare the required namespace(s) in the `<exml>` tag of the `Plugin` file. You can read a Studio plugin rule like this: "Whenever a component of the given xtype is built, add the following component plugin(s)."

You can use predefined Ext JS component plugins to modify framework components. The `BlueprintStudioPlugin` plugin, for example, uses the `addItemsPlugin` to add buttons to the favorites toolbar.

In the `BlueprintFormsStudioPlugin`, custom forms for the various Blueprint-defined document types are added by using the `addTabbedDocumentFormsPlugin` (which is a component plugin).

While in simple cases, the items to add can be specified directly inline in the Studio plugin EXML file, this is generally discouraged.

> The reason is that the Studio plugin class is instantiated as a singleton, and all EXML elements that represent objects that are not components or plugins, most prominently Actions, are instantiated immediately, too. This means that Actions are instantiated (too) early, and that a plugin rule may be applied several times with the same Action instance, leading to unexpected results.
>
> The best practice is to move the whole component plugin to a separate EXML file and reference this new plugin subclass from the Studio plugin rule. Since the new plugin is referenced by its ptype, a new plugin instance and thus a new Action instance is created for each application of the plugin rule as expected.

Execution order

The Ext JS plugins of any component are executed in a defined order:

1. Plugins provided directly in the component definition are initialized
2. Plugins defined in Studio plugin rules, starting with the plugins for the most generic applicable `xtype`, then those with successively more specific `xtypes`
3. Plugins configured for the component's ID

If that specification does not unambiguously decide the order of two plugins, plugins registered earlier are executed earlier. To make sure that a certain module's Studio plugins are registered after another module's Studio plugin, the former module must declare a Maven dependency on the latter module. This way, the Studio plugins run and register in a defined order.

For your own Studio plugin, you might want to use the file from the *CoreMedia Project* workspace as a starting point. The name of the Studio plugin file should reflect the functionality of the plugin, for example `<My-plugin-Name>StudioPlugin.exml` for better readability.

The following example shows how a button can be added to the actions toolbar on the right side of the work area:

Example 7.1. Adding a plugin rule to customize the actions toolbar

```
<editor:studioPlugin>
  <ui:rules>
    ...
    <editor:actionsToolbar>
      <plugins>
        <my:addActionsToolbarItemsPlugin/>
      </plugins>
    </editor:actionsToolbar>
    ...
  </ui:rules>
</editor:studioPlugin>
```

Because it is embedded in the element `<editor:actionsToolbar>` in the above declaration, your custom plugin `<my:addActionsToolbarItemsPlugin>` will be added to all instances of the `ActionsToolbar` class (which uses the `actionsToolbar` configuration class).

Your custom plugin is defined in a separate EXML file `AddActionsToolbarItemsPlugin.exml` that configures an `<addItemsPlugin>` to add a separator and a button with a custom action to the `ActionsToolbar` at index 5:

Example 7.2. Adding a separator and a button with a custom action to a toolbar

```
<exml:plugin xmlns...>
  <ui:addItemsPlugin index="5">
    <ui:items>
      <tbseparator/>
      <button>
        <baseAction>
          <my:myAction .../>
        </baseAction>
      </button>
    </ui:items>
  </ui:addItemsPlugin>
</exml:plugin>
```

Relative position of new component

While you can insert a component at a fixed position as shown above, it might also make sense to add the component after or before another component with a certain (global) ID, `itemId`, or `xtype`. To that end, the `addItemsPlugin` allows you to specify pattern objects so that new items are added before or after the represented objects. If the component you want to use as an "anchor component" is not a direct child of the component you plug into, you can set the `recursive` attribute in your rules declaration to `true`.

Nested extension points

When the component you want to modify is located inside a container that is also a public API extension point, you might have to access that container's API to provide context for your customizations. A typical use case for this is that you want to add a button to a toolbar that is nested below a container, but you need to apply your plugin rule to the container (and not the toolbar), because you need to access some API of that Container to configure the items to add (for example, access to the current selection managed by that container), or because the toolbar is reused by other containers, and you want your button to only appear in one specific context. Some *Studio* components define public API interfaces for accessing the runtime component instance, for example `<editor:collectionView>` creates a component that is documented to implement the public API interface `ICollectionView` (package `com.coremedia.cms.editor.sdk.collectionview`).

To express such nested extension point plugin rules, there is the plugin `<ui:nestedRulesPlugin>`. Its usage is similar to *CoreMedia Studio* plugin rules, namely is must contain an element `<ui:rules>` that again contains nested plugin rules. A nested plugin rule consists of the element of the sub-component to locate with an optional `itemId`, which in turn contains a `<plugins>` element with the plugins to add to that component. Typical plugins to use here are `addItemsPlugin`, `removeItemsPlugin`, and `replaceItemsPlugin`, all located in namespace `exml:com.coremedia.ui.config`.

For example, assume that to every LinkList property field, you want to add a custom action that needs access to the current selection of content items in the LinkList given as a config option `contentValueExpression` of type `ValueExpression`.

Like in the example above, you have to add a custom plugin to a *CoreMedia Studio* extension point in your *CoreMedia Studio* plugin EXML file:

Example 7.3. Adding a plugin rule to customize all LinkList property field toolbars

```
<editor:studioPlugin>
  <ui:rules>
    ...
    <editor:linkListPropertyField>
      <plugins>
        <my:customizeLinkListPropertyFieldPlugin/>
      </plugins>
    </editor:linkListPropertyField>
    ...
  </ui:rules>
</editor:studioPlugin>
```

Now, in your plugin `CustomizeLinkListPropertyFieldPlugin.exml`, instead of using `<ui:addItemsPlugin>` directly, you apply `<ui:nestedRulesPlugin>` to locate the toolbar you want to customize. Still, the component you plug into is a LinkList property field, and when your custom plugin is instantiated, that component is instantiated, too, and handed in as the config option `component`. It is good practice to assign the LinkList property field component as well as its initial configuration (when needed) to typed local EXML variables to avoid repeating longish expressions and type casts in inline code.

Example 7.4. Using `<ui:nestedRulesPlugin>` to customize a sub-component using its container's API

```
<exml:plugin
        xmlns:exml="http://www.jangaroo.net/exml/0.8"
        xmlns="exml:ext.config"
        xmlns:ui="exml:com.coremedia.ui.config"
        xmlns:editor="exml:com.coremedia.cms.editor.sdk.config"
        xmlns:my="exml:...">

  <exml:import
class="com.coremedia.cms.editor.sdk.premular.fields.LinkListPropertyField"/>

  <exml:import
class="com.coremedia.cms.editor.sdk.config.linkListPropertyField"/>


  <exml:var name="myLinkListPropertyField"
            type="LinkListPropertyField"
            value="{LinkListPropertyField(config.component)}"/>

  <exml:var name="linkListPropertyFieldConfig"
            type="linkListPropertyField"
            value="{linkListPropertyField(
                      config.component.initialConfig)}"/>

  <ui:nestedRulesPlugin>
    <ui:rules>
      <editor:linkListPropertyFieldToolbar>
        <plugins>
          <ui:addItemsPlugin>
            <ui:items>
              <tbseparator/>
              <ui:iconButton>
                <baseAction>
                  <my:myAction
                    contentValueExpression=
                      "{myLinkListPropertyField
```

```
                        .getSelectedValuesExpression()}"
                    forceReadOnlyValueExpression=
                      "{linkListPropertyFieldConfig
                        .forceReadOnlyValueExpression}"/>
                </baseAction>
              </ui:iconButton>
            </ui:items>
            <ui:before>
              <component
                itemId="{linkListPropertyFieldToolbar
                         .LINK_LIST_SEP_FIRST_ITEM_ID}"/>
            </ui:before>
          </ui:addItemsPlugin>
        </plugins>
      </editor:linkListPropertyFieldToolbar>
    </ui:rules>
  </ui:nestedRulesPlugin>
</exml:plugin>
```

Note how the above code makes use of the `xtype` / EXML element `linkListPropertyFieldToolbar` to locate the toolbar inside the `linkListPropertyField`, as well as to use an `..._ITEM_ID` constant from that config class to specify the new items' location.

Customizing nested components

As another example, assume you want to create your own component inheriting from `<editor:linkListPropertyField>`. You want to reuse the default toolbar that the standard link list component defines, but you want to add one additional button to that toolbar. In a very similar fashion to the example above concerning *CoreMedia Studio* plugins, you can then write your custom component's EXML file like this:

Example 7.5. Using `<ui:nestedRulesPlugin>` to customize a sub-component

```
<exml:component xmlns...>
  <exml:cfg name="additionalToolbarItems" type="Array"/>
  <editor:linkListPropertyField>
    <plugins mode="append">
      <ui:nestedRulesPlugin>
        <ui:rules>
          <editor:linkListPropertyFieldToolbar>
            <plugins>
              <ui:addItemsPlugin
                items="{config.additionalToolbarItems}"/>
            </plugins>
          </editor:linkListPropertyFieldToolbar>
        </ui:rules>
      </ui:nestedRulesPlugin>
    </plugins>
  </editor:linkListPropertyField>
</exml:component>
```

Note that when you inherit from a component and use the `<plugins>` element to declare the plugins you want to apply to this component, you overwrite the `plugins` definition of the component you inherit from. That means that all the plugins that the super component defines would not be used in your custom component. To avoid that, you have to set the `mode` attribute of the `plugins` element to either `append` or `prepend`, which will then add your custom plugin

definitions to the end of the super component's declarations, or insert them at the beginning, respectively.

Removing components

You might also want to remove certain components from their containers. In that case, you can add the `removeItemsPlugin` to the container component and remove items, again identifying them by pattern objects that can specify id, item id, or xtype.

In order the replace an existing component, you can use the `replaceItemsPlugin`. For this plugin, you specify one or more replacement components in the `items` property. Each item must specify an id or an item id and replaces the existing component with exactly that id or item id.

Register the plugin

Finally, a custom *CoreMedia Studio* plugin needs to be registered with the *Studio* application. This is done in a JavaScript file in the `resources` folder. In the example, this file is called `blueprint-components.module.js`. It is recommended that you choose a name following the schema `<put your plugin name here>.module.js`. The purpose of this file is to add the fully qualified main plugin class to the list of Studio plugins. For your own plugin, you need to change the third and fourth lines of the following example accordingly:

Example 7.6. Registering a plugin

```
joo.loadModule('${project.groupId}', '${project.artifactId}');
coremediaEditorPlugins.push({
  name:"My Plugin",
  mainClass:"com.my.company.MyStudioPlugin"
});
```

Group-specific plugin

If your plugin should only be active for a certain group of users, you can add a `requiredGroup` property to the plugin descriptor. The plugin will only be loaded if the user is a member of the given group.

The object pushed onto the array `coremediaEditorPlugins` 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`, restricting access to the plugin to members of that group.

OnlyIf plugin

You can also implement group specific and own conditions using the `onlyIf` plugin. Find further information in the ASDoc of com/coremedia/cms/editor/sdk/config/onlyIf.

To recapitulate, this is a brief overview of the configuration chain:

1. Maven dependencies introduce Studio plugin modules to *CoreMedia Studio*.
2. Studio plugin modules register Studio plugins in the `*-module.js` file.
3. Studio plugin rules definitions denote components by ID or xtype and add Ext JS plugins to those components.

4. The Ext JS plugins shown here change the list of items of the components. Any other Ext JS plugins can be used in the same way.

Load external resources

If you want to load external resources like style sheets or JavaScript files into *Studio*, you can load them with the module JS files mentioned above. Loading a JavaScript file works as follows:

Example 7.7. Loading an external script

```
joo.loadScript('<path to JavaScript file
relative to the web application root>');
```

Adding the following line loads a style sheet into *Studio*:

Example 7.8. Loading an external style sheet

```
joo.loadStyleSheet('<path to CSS-file
relative to the web application root>');
```

See the *CoreMedia Blueprint*'s main Studio plugin bootstrap code in `blueprint-components.module.js` for an example on how to load custom style sheets.

7.2 Localizing Labels

Many labels besides document types and property names can also be localized. Typical cases are labels or button texts, error messages or window titles. The localized texts are stored in property files. To use these property values, classes are generated by the EXML compiler following the singleton pattern. Property classes can be adapted as described in Section 5.7, “Localization” [61], typically overriding the existing value with values from a new customizing property class.

Predefined property classes of CoreMedia Studio

The following classes are predefined property classes defining labels and messages used throughout *CoreMedia Studio*.

- `Actions_properties`
- `Editor_properties`
- `EditorErrors_properties`
- `Publisher_properties`
- `Validators_properties`

See the ActionScript documentation for a list of defined properties.

Predefined property files of Blueprint Studio

The *CoreMedia Studio Blueprint* plugin contains two property files with localization entries in the `studio/blueprint-components/src/main/joo/com/coremedia/cms/studio/blueprint` directory: `BlueprintStudio.properties` and `BlueprintStudio_de.properties`. These files are used for custom search buttons in the favorites toolbar and for other labels that are not content type specific.

You can simply change the value of any of the properties as needed. While you can also add new properties to these files when building extensions of *CoreMedia Studio*, it is preferable to put new localization keys into new property files.

Adding a new resource bundle

If you want to add a new property file to contain your own localization key, proceed as follows:

1. Create a directory corresponding to the desired package of your resource bundle, for example, `<ModuleName>/src/main/joo/<PackagePath>`.

2. Create new properties files following the naming schema: `<PropertyFileName>.properties` and `<PropertyFileName>_de.properties`.
3. Add one or more keys and values, like so: `<KeyName>=<PropertyValue>`
4. Optionally, add the same key to each locale-specific properties file, using an appropriate translation. By default, there is only one translation (German), but you can add your own.
5. In an EXML file describing your custom component, import the resource bundle, using its fully qualified class name: `<exml:import class="<FullyQualifiedName>_properties"/>`
6. Address the resource bundle and key in the text attribute of the component where you want to use the label: `{<FileName>_properties.INSTANCE.<KeyName>}`. You will get code completion in a properly configured IDE once the properties bundle was compiled.
7. Alternatively, reference the `INSTANCE` object from an ActionScript class.

Example: Adding a search button

In order to introduce a new localized button to the favorites toolbar you could add the following component to the file `BlueprintFavoritesToolbarButtons.exml`:

Example 7.9. Adding a search button

```
<button itemId="exampleButton">
  <baseAction>
    <editor:showCollectionViewAction
text="{BlueprintStudio_properties.INSTANCE.doc_example_txt}"
published="false" editedByMe="true" contentType="CMArticle"/>
  </baseAction>
</button>
```

The attribute `text` of the `editor:ShowCollectionView` Element defines the text to be displayed in the Studio web application. On the top of the file `BlueprintFavoritesToolbarButtons.exml` you will see the following line:

```
<exml:import class="com.coremedia.cms.studio. \
blueprint.BlueprintStudio_properties"/>
```

This line imports the `BlueprintStudio.properties` file into the scope. Of course, you could also import your own file.

In order to have the label you want, you need to add it to the properties file. The `BlueprintStudio.properties` file starts like this after adding a string for the label:

Example 7.10. Example property file

```
doc_example_txt=My Example Button

SpacerTitle_navigation=Navigation
SpacerTitle_versions=Versions
```

```
SpacerTitle_layout=Layout
...
```

Override Standard Studio Labels

It is also possible to override the standard *Studio* labels, like so:

1. Create a property file with all labels you want to override, for example `CustomLabels.properties` and `CustomLabels_de.properties`.
2. Search for the key of the property that should be changed. All the keys are documented in the ActionScript API, such as `Action_withdraw_tooltip` in the resource bundle class `Actions_properties`.
3. In your `CustomLabels` bundle, set the new value for the key.
4. In the `init()` method of the `EditorPlugin`, override the `Actions_properties` bundle with the following code:

Example 7.11. Overriding properties

```
//override the standard studio labels with custom properties
ResourceBundle.overrideProperties(Actions_properties,
  CustomLabels_properties);
```

This can be done with every property of *Studio*. An example can also be found in the `BlueprintStudioPlugin`.

7.3 Document Type Model

Each CoreMedia CMS content application is based on an object-oriented document type model. Documents of different types often require different treatment. By tailoring CoreMedia Studio to the document type model, the support for dealing with documents is greatly improved.

- Section 7.3.1, “Localizing Types and Fields” [97] describes how to localize the names of document types and document properties.
- Section 7.3.2, “Defining Content Type Icons” [98] describes how to define icons for your document types in CoreMedia Studio.
- Section 7.3.3, “Customizing Document Forms” [101] describes how you can add or remove property fields to or from a document form.
- Section 7.3.4, “Image Cropping and Image Transformation” [108] describes how to enable the image cropping feature.
- Section 7.3.6, “Disabling Preview for Specific Document Types” [112] describes how you can disable the preview for a specific document type.
- Section 7.3.7, “Configuring Translation Support” [112] describes how you can configure the translation support.
- Section 7.3.8, “Excluding Document Types from the Library” [113] describes how you can exclude document types from the dropdown lists for document creation and document type search filtering.
- Section 7.3.9, “Client-side initialization of new Documents” [114] describes how you can initialize newly created documents.

7.3.1 Localizing Types and Fields

You can localize the names of document types and document properties by means of property files as described in Section 5.7, “Localization” [61]. To this end, you provide property files and use them to override the properties defined in `com.coremedia.cms.editor.ContentTypes_properties`. Typically, this is done while initializing a Studio plugin.

Example 7.12. Localizing document types

```
public function init(editorContext:IEditorContext):void {
  ResourceBundle.overrideProperties(ContentTypes_properties,
MyDocumentTypes_properties);
}
```

There are several kinds of property keys to overwrite when localizing document types:

- `<ContentTypeName>_text`: the name of the content type `<ContentTypeName>` in the given language;
- `<ContentTypeName>_toolTip`: the tooltip shown for the content type `<ContentTypeName>`;

- `<ContentTypeName>_icon`: the CSS class to attach to the HTML <div> elements that show the type icons for the content type `<ContentTypeName>` (see Section 7.3.2, “Defining Content Type Icons” [98] for details about these style classes);
- `<ContentTypeName>_<PropertyName>_text`: the name of the property `<PropertyName>` of a document of type `<ContentTypeName>` or a subtype thereof;
- `<ContentTypeName>_<PropertyName>_toolTip`: the tooltip shown for the property `<PropertyName>` of a document of type `<ContentTypeName>` or a subtype thereof.
- `<ContentTypeName>_<PropertyName>_emptyText`: the text to shown in the field when the property `<PropertyName>` of a document of type `<ContentTypeName>` is empty. This message typically prompts the user to enter a value.

When multiple localizations are defined for a single property, but different content types, the most specific type is used.

Content Types in Blueprint Studio

The *CoreMedia Studio Blueprint* plugin contains two property files `BlueprintDocumentTypes.properties` and `BlueprintDocumentTypes_de.properties` for localizing document type names and property names in the `studio/blueprint-forms/src/main/joo/com/coremedia/blueprint/studio` directory.

You can simply change the value of any of the properties as needed.

7.3.2 Defining Content Type Icons

A significant number of content type icons are already defined. See Table 7.1, “Content Type Icons” [98] for an overview. Special cases, though, might not be covered by these icons.

Table 7.1. Content Type Icons

Icon	CSS class
	`content-type-CMArticle-icon`
	`content-type-CMAudio-icon`
	`content-type-CMCSS-icon`
	`content-type-CMChannel-icon`

Icon	CSS class
	content-type-CMCollection-icon
	content-type-CMDownload-icon
	content-type-CMExternalLink-icon
	content-type-CMFavDirectory-icon
	content-type-CMFolder-icon
	content-type-CMGallery-icon
	content-type-CMHTML-icon
	content-type-CMImageMap-icon
	content-type-CMInteractive-icon
	content-type-CMJavaScript-icon
	content-type-CMMedia-icon
	content-type-CMNamedDynamicList-icon
	content-type-CMObject-icon
	content-type-CMPicture-icon
	content-type-CMSettings-icon
	content-type-CMSearchDirectory-icon
	content-type-CMSite-icon
	content-type-CMSitemap-icon

Icon	CSS class
	`content-type-CMTaxonomy-icon`
	`content-type-CMTeaser-icon`
	`content-type-CMVideo-icon`
	`content-type-CMViewtype-icon`
	`content-type-Dictionary-icon`
	`content-type-Preferences-icon`
	`content-type-Query-icon`

If you want to provide custom icons, you should use black outlines (`#3d4242`) and white fill (`#ffffff`). Instead of white, you may also use a gray gradient form `#ffffff` to `#b3b3b3`. A gradient is actually preferred for large icons. In order to maintain a style that is consistent with the default icons, use color sparingly, if at all. The icons have to be placed on a transparent background. All standard icons are strictly 2-dimensional.

You have to provide four different images. You can then add CSS rules to use your own icons as background images of type icon HTML elements. The four images are used in the following cases:

- 16x16-pixel icons for use on a white background. Your CSS styles should use this image when an element is tagged with your style class and the style class `content-type-xs`.
- 16x16-pixel icons for use on a light gray or colored background. Unlike the other icons, these icons should not use a white fill. Instead, use a transparent fill or a black to transparent gradient. Your CSS styles should use this image when an element is tagged with your style class and the style classes `content-type-xs` and `content-type-transparent`.
- 64x64-pixel icons for use on a white background. As a rule of thumb, use 2-pixel outlines instead of single pixels for 16x16 icons. Your CSS styles should use this image when an element is tagged with your style class and the style class `content-type-l`.

- 128x128-pixel icons for use on a white background. Your CSS styles should use this image when an element is tagged with your style class and the style class `content-type-xl`.

Assuming your style class is called `myIconClass`, you might want to define the following rules:

```
.content-type-xs.myIconClass{
  background-image:url('...')!important;
}
.content-type-transparent.content-type-xs.myIconClass{
  background-image:url('...')!important;
}
.content-type-l.myIconClass{
  background-image:url('...')!important;
}
.content-type-xl.myIconClass{
  background-image:url('...')!important;
}
```

If you omit the rule for the `content-type-transparent` class, the browser will fall back to the first rule, showing icons with a solid fill.

If you define many content type icons, consider grouping the icons in a single sprite image, using the `background-position` attribute in your CSS to select the correct icon.

If you want to show the content type icons in your Studio document tab, then you need to include the following rules for each content type style class:

```
.silicium-tab .x-tab-strip-text.content-type-myIconClass{
  background-image:url('.../16x16/myIcon-pos')!important;
}
.silicium-tab.x-tab-strip-active
.x-tab-strip-text.content-type-myIconClass{
  background-image:url('.../16x16/myIcon-neg')!important;
}
```

By using the rules above, the status icon (checked-out state, editing state etc.) will replace the content type icon, if applicable.

7.3.3 Customizing Document Forms

The following section describes how to customize the document forms, which constitute the main working component that your users will use. In earlier *Studio* versions, property fields were all contained in the main work area of the document form, and document metadata such as the filing information and version history were grouped in a collapsible section at the bottom of the form.

Using tabs for grouping

Current *Studio* versions offer a more flexible way of organizing your - potentially quite big - set of property fields into horizontal tabs. You can either use the default of two tabs, one for the main content properties, one for the metadata, respectively, or you can arrange properties freely on an arbitrary number of tabs.

Default two-tabbed document forms

In the simple (default) case, a CoreMedia document form has two tabs: the primary form fields (1) that enable you to edit the content of the object and another tab (2) that shows metadata such as the path to the document, and the versioning information for that document.

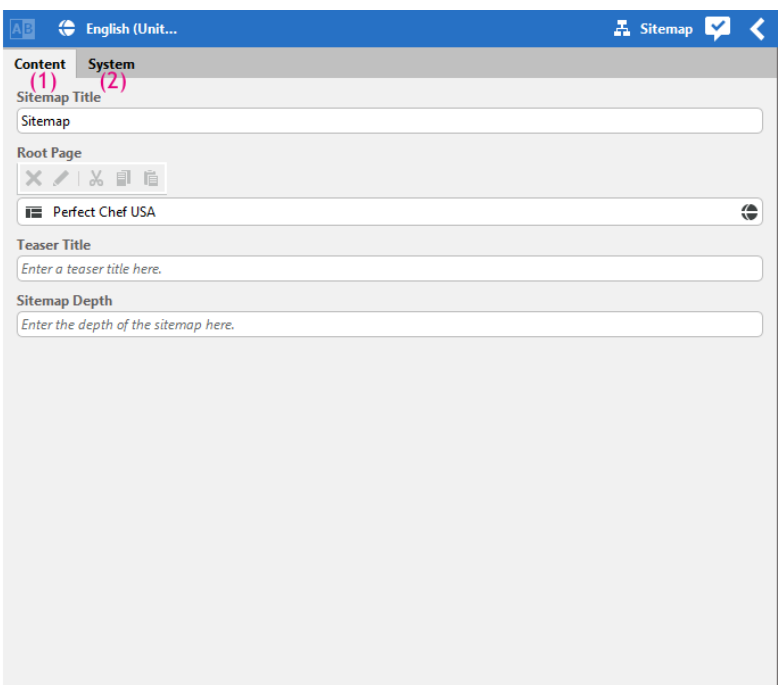


Figure 7.2. Document form with content and metadata properties

Both tabs of the form are defined in separate definition files. Forms should be defined by subclassing the predefined `DocumentForm` component.

CoreMedia Studio offers at least one predefined property field for each property type available for CoreMedia documents. See Table 7.2, “Property Fields” [104] for a list of all provided field types.

To customize a form, you need to adapt the respective form definition file (an EXML component) in `studio/blueprint-forms/src/main/joo/com/coremedia/blueprint/studio/forms/`. Containers used in the forms are defined in separate EXML files in the `/containers` sub directory. The following code shows a simple example for a standard `CMArticle` form definition:

Example 7.13. Article form

```
<?xml version="1.0" encoding="UTF-8"?>
<exml:component xmlns:exml="http://www.jangaroo.net/exml/0.8"
```

```
                  xmlns="exml:ext.config"
                  xmlns:ui="exml:com.coremedia.ui.config"

xmlns:editor="exml:com.coremedia.cms.editor.sdk.config">
  <editor:documentForm itemId="CMArticle">
     <items>
       <editor:stringPropertyField propertyName="subject"
        cls="cm-textfield-header"/>
       <editor:stringPropertyField propertyName="title"
        cls="cm-textfield-header"/>
       <editor:richTextPropertyField propertyName="text"/>
       <editor:imageLinkPropertyField
                propertyName="media"linkListContentType="CMPicture"

                 dataPropertyName="data" />
       <editor:stringPropertyField propertyName="teaserSubject"
                 cls="cm-textfield-header"/>
       <editor:stringPropertyField propertyName="teaserTitle"
        cls="cm-textfield-header"/>
       <editor:richTextPropertyField propertyName="teaserText"/>
       <editor:blobPropertyField propertyName="thumbnail"
        contentType="image/*"/>
       <editor:linkListPropertyField propertyName="related" />
       <editor:stringPropertyField propertyName="linktext"/>
     </items>
  </editor:documentForm>
</exml:component>
```

The property fields are defined in the `<items>` element of the `<editor:documentForm>` element. Each property field has at least an attribute `propertyName` which corresponds to the property name of the document type. The property name must be specified for each field. The document form also provides three additional properties to all fields without specifying them explicitly: `bindTo`, `hideIssues`, and `forceReadOnlyValueExpression`. The standard property fields recognize these options and custom property fields are encouraged to so, too. See Section 7.4, "Customizing Property Fields" [115] for details about developing new property fields.

- `bindTo`: A value expression that evaluates to the content object to show in the form. The content may change when the form content changes.
- `hideIssues`: This attribute is used to disable the highlighting of property fields with issues originating from validators. Validators will be described in Section 7.15.1, "Validators" [176]. If set on the document form, it applies to all property fields.
- `forceReadOnlyValueExpression`: A value expression that evaluates to true when the document form and all of its property fields should be shown in read-only mode, for example when showing the document form on the left side in master comparison mode.

Other attributes might vary depending on the property type. The `BlobPropertyField` editor, for example, has a property `contentType` that defines the MIME type. If you want to hide a property, you can simply remove the related `<edit`

or:<PropertyType>propertyField> element. The order of the editor elements defines the order in the form.

Table 7.2. Property Fields

Property Field	Used for	Description
stringPropertyField	String property	Shows string data.
integerPropertyField	Integer property	Shows integer number.
spinnerPropertyField	Integer property	Shows integer number, with arrow buttons to increase/decrease the current value, and mouse wheel support
booleanPropertyField	Integer property with 0/1 Boolean values	Shows a checkbox indicating checked=1, unchecked=0.
dateTimePropertyField	Date property	Shows date, time and time zone and provides appropriate picker elements.
linkListPropertyField	Link List property	Allows drag and drop.
contentListChooserPropertyField	Link List property	Shows a list of linkable contents and the current selection.
richTextPropertyField	CoreMedia RichText (XML) property	Shows the text in a WYSIWYG style and provides a fully featured toolbar.
xmlPropertyField	Generic XML property	Shows the raw XML text.
blobPropertyField	blob property for all MIME types	Shows the image and provides an upload dialog.
textAreaPropertyField	CoreMedia RichText (XML) property	Shows the text as plain text in a text area.
textAreaStringPropertyField	String property	Shows the text in a text area.
textBlobPropertyField	blob property of MIME type text/plain	Shows the blob as plain text in a text area.
structPropertyField	CoreMedia Struct property	Shows a generic editor for structs.

Showing derived contents

In a multi-site setting contents may be localized variants of each other. By including the component DerivedContentsList into your form you can show the list of

derived contents of any given document. Typically, this component is placed near the link list property that associates a master document to a derived document.

```
<editor:documentForm>
  <items>
    ...
    <editor:derivedContentsList/>
    ...
  </items>
</editor:documentForm>
```

Customizing columns in link list properties

Showing more columns

By default, the `linkListPropertyField` shows a document type icon, the name and the lifecycle status for each linked document. You can configure an array of columns to be shown using the `columns` property of the field component. Each array element must be an Ext JS grid column object. The available fields of the store backing the grid panel are `name`, `status`, `type`, and `typeCls`, These fields represent the name, the lifecycle status, the document type name and a style class for a document type icon, respectively.

If you need additional fields for your custom columns, you can add them using the `fields` property. Each field should be a `com.coremedia.ui.config.dataField`. The following example shows how a new column uses a custom field to display the `locale` property of linked documents.

```
<editor:linkListPropertyField propertyName="variants">
  <editor:fields>
    <ui:dataField name="locale"
                  mapping="properties.locale"
                  ifUnreadable=""/>
  </editor:fields>
  <editor:columns>
    <editor:typeIconColumn/>
    <editor:nameColumn/>
    <editor:statusColumn/>
    <gridcolumn id="locale"
                width="30"
                dataIndex="locale"/>
  </editor:columns>
</editor:linkListPropertyField>
```

Whereas the configured fields are added to the default fields, the configured columns completely replace the default columns. That is, if you want to keep the predefined fields, you have to repeat their definitions as shown in the example.

Multi-tab document forms

In situations where the default split into a content and a properties tab is not sufficient for your users, you can also define any number of arbitrary tabs, and freely assign property fields to them. Doing so requires a slightly more complex definition of your document forms, where individual tabs are nested within the root element of your EXML definition. The following pseudo code snippet outlines the basic structure of a tabbed document form.

```
<?xml version="1.0" encoding="UTF-8"?>
<exml:component xmlns:exml="http://www.jangaroo.net/exml/0.8"
                xmlns="exml:ext.config"

xmlns:editor="exml:com.coremedia.cms.editor.sdk.config">


  <editor:documentTabPanel>
    <items>
      <editor:documentForm title="First tab">
        <items>
          <editor:stringPropertyField name="property1"/>
          <editor:stringPropertyField name="property2"/>
        </items>
      </editor:documentForm>
      <editor:documentForm title="Second tab">
        <items>
          <editor:stringPropertyField name="property3"/>
          <editor:stringPropertyField name="property4"/>
        </items>
      </editor:documentForm>
      <editor:documentForm title="Third tab">
        <items>
          <editor:stringPropertyField name="property5"/>
          <editor:stringPropertyField name="property6"/>
        </items>
      </editor:documentForm>
       ...
    </items>
  </editor:documentTabPanel>
```

To register your custom document form, you need to add your EXML component to the `TabbedDocumentFormDispatcher`, like so:

```
<editor:tabbedDocumentFormDispatcher>
  <plugins>
    <editor:addTabbedDocumentFormsPlugin>
      <editor:documentTabPanels>
        <my:myFormDefinition1 itemId="CMArticle"/>
        <my:myFormDefinition2 itemId="CMTeaser"/>
        ...
      </editor:documentTabPanels>
    </editor:addTabbedDocumentFormsPlugin>
  </plugins>
</editor:tabbedDocumentFormDispatcher>
```

The above code plugs into the `TabbedDocumentFormDispatcher`, and registers two custom document forms from your own namespace titled `my`. Note that the `itemId` still corresponds to the name of the document type you want to apply your form for.

The document forms registered with the dispatcher are automatically used for both the regular document form and for left-hand form of the version comparison view and the master side-by-side view. When used on the left side, the `forceReadOnlyValueExpression` passed to the form is set to `true`, allowing your form to switch into a read-only mode.

When you choose to use multi-tab document forms, also note that you need to specify the built-in metadata components such as `<editor:versionHistory>`

or `<editor:documentInfo>`, because they are not automatically added. It is common practice to place these on a separate tab titled *System* or similar (which is also what *CoreMedia Blueprint* does), but of course you can add them to any place in the form that you want.

Collapsible Property Fields

To add several property fields to a group with an additional title, the component `<editor:collapsibleFormPanel>` can be used. All documents forms of *CoreMedia Blueprint* do use it to provide a better overview about related fields.

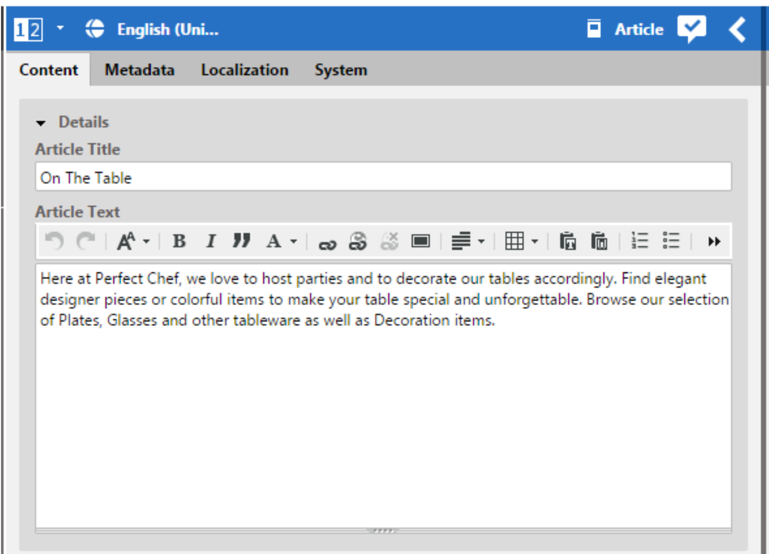


Figure 7.3. Document form with a collapsible form panel

Additionally, the collapsible form panel persists the collapsed status. For example, when the collapsible form panel is collapsed for the teaser title and teaser text of an article, the group is collapsed for all newly opened article documents too (except it contains an invalid field). This status information is stored in the user preferences of the user, so if the user logs into *Studio* on another computer, the same state will be restored.

```
<editor:collapsibleFormPanel
title="{CustomLabels_properties.INSTANCE.PropertyGroup_Details_label}"
 itemId="detailsDocumentForm">
  <items>
    <editor:stringPropertyField propertyName="title"/>
    <editor:richTextPropertyField propertyName="detailText"
                                  initialHeight="200"/>
  </items>
</editor:collapsibleFormPanel>
```

Each declaration of an `<editor:collapsibleFormPanel>` element should contain the attributes `title` and `itemId`. The `title` attribute applies a title to the panel (and also provides a meaning to the group). It is also used as click area for collapsing the panel. The `itemId` should be applied to persist the state of the group. If no `itemId` is provided, the collapsible state is not stored in the user preferences and therefore not applied when new documents of the same type are opened.

7.3.4 Image Cropping and Image Transformation

The Image Editor provides various image transformations which are stored in a separate struct property of the document. It also holds the original image data which is never modified - the transformations are applied only when previewing or delivering the image.

The Image Editor uses the same Image Transformation Framework to display the image within the image form as the CAE uses for delivering images to web sites, e.g. within the preview panel. See the Content Application Developer Manual for further details on image transformations.

The `ImageEditorPropertyField` is defined in the `CMPictureForm.exml` of the *Blueprint* and can be defined by using the config properties listed below. Properties marked with * are mandatory.

Table 7.3. `ImageEditorPropertyField` Configuration Settings

Config Property	Type	Description
`bindTo*`	`ValueExpression`	A property path expression leading to the content Bean whose properties are edited.
`propertyName*`	`String`	The name of the BLOB property containing image data.
`imageSettingsPropertyName*`	`String`	The name of the Struct property containing image transformation data.
`hideIssues`	`Boolean`	If true, no validation issues on this property field are shown. Defaults to false.
`forceReadOnlyValueExpression`	`String`	An optional ValueExpression which makes the component read-only if it is evaluated to true.

The `ImageEditorPropertyField` can be configured as follows:

Example 7.14. Configuring the Image Editor

```
<ie:imageEditorPropertyField bindTo="{config.bindTo}"
                             propertyName="data"
imageSettingsPropertyName="localSettings"/>
```

A crop is a subset of the image with a fixed aspect ratio and minimum size. Crops in the Image Editor are represented by variants. There are two different ways to configure variants: via Spring or as site specific variants directly in the content.

Spring Configuration for Variants

To configure global variants for all `CMPicture` documents, the `mediatransform.xml` has to be adjusted. Each variant is defined by one `Transformation` which holds all the information for that variant.

Example 7.15. Configuring the variants

```
<bean class="com.coremedia.cap.transform.Transformation">
    <property name="name" value="large4x3"/>
    <property name="widthRatio" value="4"/>
    <property name="heightRatio" value="3"/>
    <property name="minWidth" value="640"/>
    <property name="minHeight" value="480"/>
    <property name="previewWidth" value="400" />
</bean>
```

The configuration of variants via Spring is the default used by the`TransformImageService`.

Site Specific Image Variants

If not all sites should have the same fixed set of image variants, site specific image variants can be configured via content instead. Thereto a `CMSettings` document named `responsiveImageSettings` with the struct property `linkedSettings` has to be defined for every site (see also section "Content Configuration" below).

The feature for site specific variants is disabled by default. To enable it, the property `dynamicVariants` has to be set to `true` in the file`transform-image-service.properties`.

If loading the image variants fails for some reason, e.g. the image is not located within a site, the default variants configured in the `mediatransform.xml` will be applied instead. It is therefore recommended to apply all site specific variant configurations to the `mediatransform.xml` as well.

Rendering Site Specific Image Variants

When rendering images, the `TransformImageService` is used to access the variants of an image. An example for this can be found in the`CMPicture.asPreview.jsp`. In this template the `previewWidth` and `previewHeight` attributes of the `Transformation` class are used to calculate the image size in the preview. If these attributes are not set, `minWidth` and `minHeight` are used instead.

CAE Configuration

For the CAE, the class `TransformImageService` is responsible for loading site specific cropping information. The feature can be enabled by changing/adding the attribute `dynamicVariants` to true in the file `mediatransform.xml`. The class part of the *Blueprint* so it can be customized if necessary.

The `TransformImageService` will automatically look up the linked settings of the root channel and search for the "Responsive Image Settings" struct which contains the variant information.

Content Configuration

The "Responsive Image Settings" document not only contains image variants, but also various resolutions which may be used on different devices. The breakpoint values defined in the css for the corresponding theme are used to determine which resolution should be used. With the introduction of site specific image crops, additional struct properties can be configured for variants.

Variant Properties, the following are mandatory:

- `widthRatio`: minimum integer which defines the width component of the aspect ratio
- `heightRatio`: minimum integer which defines the height component of the aspect ratio
- `minWidth`: the value is used by the Studio to validate the minimum variant width (integer property)
- `minHeight`: the value is used by the Studio to validate the minimum variant height (integer property)

Pre-defined image sizes (resolutions), at least one pair should be defined per variant and must match the aspect ratio:

- `width`: defines the width of the image (integer property)
- `height`: defines the height of the image (integer property)

Properties for variant and pre-defined image sizes (properties listed within the predefined image size properties will always override the more general variant properties):

- `gamma`: the default gamma value of the picture (string property with numeric value from 0 to 1)
- `jpegQuality`: the default jpeg quality of the picture (string value with numeric value from 0 to 1)
- `sharpen`: boolean value to enabled/disable sharpening of the picture

- `removeMetadata`: boolean value to enabled/disable metadata removal of the transformed image

7.3.5 Enabling Image Map Editing

The image map editor comes as a panel component embedding an image view. The editor allows users to create hot zones (image map areas) and to attach documents to hot zones via drag and drop. The image map editor uses a configurable struct property name to store the image map configurations to a struct property of an image map document. It also offers a configuration option for the image to display. This allows you to store image map configurations in documents that do not have an image blob property themselves.

To enable image map editing in your project, include an image map editor component in your document's EXML form (*Blueprint* shows this in its `CMImageMapForm.exml` definition).

Example 7.16. Configuring an Image Map Editor

```
<im:imageMapEditor
  imageBlobValueExpression=

"{config.bindTo.extendBy('properties.pictures.0.properties.data')}"

       structPropertyName="localSettings"/>
```

In the example above, the source document has a link list property name `pictures` of cardinality `1`. So the image editor component is bound to the image stored at the `data` property of the linked image document. The map configuration is stored at the source document's `localSettings` property.

Enabling validation

Configure the ImageMapAreasValidator in the Spring application context to enable validation of the image map document. The validator generates an error issue if there is no image blob or if at least one of the defined image map areas does not have a valid link target. See also Section 7.15.1, “Validators” [176] for validation in general.

Example 7.17. Configuring a validator for image maps

```
<bean id="cmImageMapAreasValidator"
class="com.coremedia.rest.cap.validators.ImageMapAreasValidator">
  <property name="connection" ref="connection"/>
  <property name="contentType" value="CMImageMap"/>
  <property name="validatingSubtypes" value="true"/>
  <property name="imagePropertyPath" value="pictures.data"/>
  <property name="structProperty" value="localSettings"/>
</bean>
```

In the example above, the validator is configured for the document type `CMImageMap` and its subtypes. The image is stored in the blob property `data` of the

first document of link list property `pictures` of the image map document. The image map configuration is stored in the struct property `localSettings`.

7.3.6 Disabling Preview for Specific Document Types

For some document types a suitable preview representation is not easily generated. This applies to some built-in document types like `Dictionary` and `EditorPreferences`, but also to very technical document types storing CSS or script code.

The method `getDocumentTypesWithoutPreview()` from the editor context object grants access to an array of document type names for which no preview should be shown. Like in the case of document types excluded from creation as shown in the previous section, you can simply push additional document types into the mutable array returned from the method.

You can also use the `configureDocumentTypes` plugin to specify document types without preview, like in the following excerpt from `BlueprintFormsStudioPlugin`.

Example 7.18. Defining document types without preview

```
<editor:configureDocumentTypes
        names="CMAction,CMCSS,..."
        preview="false"/>
```

7.3.7 Configuring Translation Support

Source language document resolver

If you work with content in multiple languages and want to derive translated documents from source documents in a primary language, you can support the editors by providing a side-by-side view of both documents. To this end, a resolution strategy for matching translated documents can be configured.

A source language document resolver is simply a function that takes a `Content` object as its single argument and returns the `Content` from which the given content was derived. If no source document is available, the resolver returns null. So that the document forms for the translated document and the source document can be properly aligned in the side-by-side view, the returned content must belong to the same content type as the argument content. In the document model of CoreMedia Blueprint, the resolver function can simply follow the link list `master` to determine a source document.

Example 7.19. Blueprint source language document resolver

```
public function resolveMasterDocument(content:Content):Content {
  var contentProperties:ContentProperties = content.getProperties();

  if (contentProperties) {
    var readOnlyContents:Array =
      contentProperties.get('master') as Array;
    if (readOnlyContents) {
      return readOnlyContents[0] as Content;
    }
  }
```

```
  return null;
}
```

To configure a source language document resolver, the `configureDocumentTypes` Studio plugin can be used. A resolver is used for the given content type and all subtypes. If multiple resolvers are available, the resolver for the more specific content type takes precedence.

Example 7.20. Configuring a source language document resolver

```
<editor:configuration>
  <editor:configureDocumentTypes
    names="CMLocalized"
    sourceLanguageDocumentResolver="{resolveMasterDocument}"/>
</editor:configuration>
```

7.3.8 Excluding Document Types from the Library

The CoreMedia document type model is a very powerful concept to tailor *CoreMedia CMS* to your needs. However, in any typical project, there are at least a couple of document types mainly designed to manage technical metadata, such as site settings. In many cases you want to hide these document types from casual users of *CoreMedia Studio*, thereby keeping the interface simple and avoiding clutter. To do so, you can remove choices from the dropdown document type selector in the Library's create content menu, and from the dropdown used to restrict search results to certain document types.

You can add the document types that should not be shown to the list of excluded document types using the `IEditorContext`. The methods `getExcludedDocumentTypes()` and `getDocumentTypesExcludedFromSearch()` return an array holding the names of all document types excluded from the create document dropdown and search filter dropdown, respectively. Using the array's `push` method, you can add additional document types you wish to hide: `editorContext.getExcludedDocumentTypes().push('<DocType1>', ...)`

Example

Example 7.21. Defining excluded document types

```
editorContext.getExcludedDocumentTypes().push('Dictionary',
  'Preferences', 'Query', 'Folder_',
  'CMDynamicList', 'CMVisual',
  'EditorPreferences');
```

This call gets the array of excluded document types and adds Strings containing the names of the document types to exclude.

If you are using EXML for your plugin, you can also write the above exclusion instructions declaratively in your main Plugin EXML file:

```
<editor:configuration>
  <editor:configureDocumentTypes

names="Dictionary,Preferences,Query,Folder_,CMDynamicList,CMVisual,EditorPreferences"

        exclude="true" excludeFromSearch="true"/>
</editor:configuration>
```

Example 7.22. Defining excluded document types in EXML

7.3.9 Client-side initialization of new Documents

With a content initializer you can initialize the properties of a newly created document. A content initializer will be called while a new content object is being created by the `NewContentAction`. Only one initializer can be defined for each document type. You must register custom initializers with the `IEditorContext` class. Simply call the `registerContentInitializer(contentTypeName, initializer)` method.

Example

The following code defines a simple initializer that sets the content's language property to German by default:

```
editorContext.registerContentInitializer("CMTeaser", initLanguage);
...
private function initLanguage(content:Content):void {
  var properties:ContentProperties = content.getProperties();
  properties.set('lang', 'de');
}
```

Example 7.23. Defining a content initializer

Client-side initialization might be sufficient for simple initialization scenarios. If you have complex requirements, consider using server-side initialization: Refer to Section 7.15.2, “Intercepting Write Requests” [180] for details.

7.4 Customizing Property Fields

While *CoreMedia Studio* provides predefined property fields for strings, dates, link lists (including those handling images), and many others, you might want to use an own widget to display and edit a property according to your specific requirements.

Ext JS offers many components that can be used for this purpose. Often, some configuration will get you a long way to an appropriate widget. The main task that is always necessary is the binding of the new component to your data ("the model"). *Studio*'s client-side models are explained in more detail in Section 5.3, "Client-side Model" [37] and Section 5.4, "Remote CoreMedia Objects" [51]. While you could theoretically implement property fields in any way, adhering to certain conventions as described in the following section helps to make the property fields reusable.

Also, there are a number of standard plugins that simplify the task of writing a property field. These are introduced by way of an example in Section 7.4.2, "Standard Component StringPropertyField" [116]. Here you will find a simple recipe for creating property fields that use a predefined plugin to handle the data binding.

The rich text property field allows several customizations as shown in Section 7.4.5, "Customizing RichText Property Fields" [122].

7.4.1 Conventions for Property Fields

Property field are intended for use in document forms as described in Section 7.3.3, "Customizing Document Forms" [101]. To ensure the most convenient usage, custom property fields should adhere to the standard name for config options.

The option `propertyName` should define the name of the property to show and edit in the property field. While you can use a different name for this option, your document form definition become more readable when you use the `propertyName` option uniformly.

Further conventions arise, because a document form forwards a number of configuration option to all included components, that is, to all included property fields. By using the standard option names, you avoid repetitions and accidental omissions.

The option `bindTo` is a value expression that evaluates to the object that defines the property. If possible, the field should not assume that this object implements the `Content` interface, but rather that it is a bean with a property `properties` that stores another bean that contains the property given as `propertyName`. That will eventually make it possible to reuse the field for workflow forms.

For the same reason, a property field should not access built-in properties like `creationDate` and others. It should also refrain from performing other operations like `checkIn` on the returned bean. This is no significant limitation, because

property fields are typically reading and writing schema-defined properties, only. When property fields are used in the left half of the version comparison view, they are bound to an object that does implement the `Content` interface, but that is actually wrapping a version. In this case, the built-in properties of `Content` are present, but might not always return the value you expect. It always claims to be checked in and it returns the properties of the historic version, even though it reports the id of the versioned content. When accessing only the schema-defined properties, property field will behave as expected.

If the value expression provided through the option `forceReadOnlyValueExpression` evaluates to true, the property field should switch to a read-only mode. In this mode it should be possible to view property values and preferably to copy them, but it should be impossible to make updates. The value expression is set to true when a document form is used on the left side of a master side-by-side view or a version comparison view. The property field itself must take other reasons into account that might make the field read-only. To this end, the utility methods `isReadOnly` and `createReadOnlyValueExpression` in the class `PropertyEditorUtil` support you in making a property field read-only.

The class `PropertyEditorUtil` also contains methods for localizing property names, types, and so on.

7.4.2 Standard Component StringPropertyField

The task attempted in this section is to replicate the behavior of the standard `StringPropertyField`.

Create the new property field as an EXML component, since it is a visual component and needs no application logic. You inherit directly from the Ext JS component `TextField` that is used for displaying the property. Before you can start, you must set the stage for the XML file.

Example 7.24. Custom property field

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<exml:component xmlns:exml="http://www.jangaroo.net/exml/0.8"
                xmlns="exml:ext.config"
                xmlns:editor=
                "exml:com.coremedia.cms.editor.sdk.config"
                xmlns:ui="exml:com.coremedia.ui.config">
```

You need the `exml` namespace for the basic structure of the EXML file, the default name space for predefined Ext JS components, the `editor` namespace for CMS-specific components and plugins (the "Editor SDK"), and the `ui` namespace for generic plugins at the model layer (the "UI Toolkit").

The element `<exml:component/>` indicates that a component is defined. It does *not* specify that the new component should inherit from `Component` directly. If you need a base class, you can specify it using the `baseClass` attribute - this is shown further down in this example.

Afterwards, the configuration options supported by the class are described, using the `<exml:cfg>` elements. You can think of the set of these elements as the configuration API description of your component. Any component inherits the configuration options from its superclass(es).

```
<exml:cfg name="propertyName" type="String">
  <exml:description>
    The property to bind.
  </exml:description>
</exml:cfg>
<exml:cfg name="bindTo" type="com.coremedia.ui.data.ValueExpression">

  <exml:description>
    A value expression evaluating to the content whose
    property is being edited.
  </exml:description>
</exml:cfg>
```

The two properties `propertyName` and `bindTo` are mandatory for all property fields. The former declares the name of the property to be edited, which is used both for accessing the model and for localizing the property field. The latter declares a value expression evaluating to the `Content` object.

```
<exml:cfg name="hideIssues" type="Boolean">
  <exml:description>Don't show any validation issues on this property
 field.</exml:description>
</exml:cfg>
```

As a third configuration option, you can disable the visual indication of content errors or warnings via configuration. This option will later on passed to the appropriate plugin.

An optional description of the entire class follows.

```
<exml:description>...</exml:description>
```

You are now ready to define the base class and add some styling.

```
<textfield name="{'properties.' + config.propertyName}"
     anchor="100%"
     cls="string-property-field">
```

Several plugins are available to customize the behavior of the editor.

```
<plugins>
```

To register the property field properly with *Studio* for the purposes of preview-base editing and navigating directly to property field, you need to declare the following plugin:

```
<editor:propertyFieldPlugin propertyName="{config.propertyName}"/>
```

Using this plugin lets *Studio* know that your component is authoring a content property. Among other things, this will set up your component to cooperate properly

with the content errors and warnings navigation window, and with content shortcuts from the embedded preview.

Update backing model

By default, a component will flush its state to the server when it loses its focus, which typically happens when a users clicks into another property field. If you want to update the backing model more frequently, you can use the immediate change events plugin. The plugin sends a change event when the user has not typed anything for longer than a configurable "buffer" time (in milliseconds). This plugin works for `Field` components only, but when you look at the Ext JS class tree, you will find that many components are fields in disguise, even number fields and combo boxes.

```
<ui:immediateChangeEventsPlugin/>
```

In order to support content validation, a field should also be highlighted in red (when content errors are present), or orange (when content warnings are present). See Section 7.15.1, “Validators” [176] for information on how to set up server-side content validators. On the client side, the `showIssuesPlugin` as shown below handles all the work. It reads the issues generated on the server and attaches one of the style classes `issue-error` and `issue-warn` if an issue is present. Pass all relevant configuration options from the property field to the plugin, especially the options `bindTo` and `propertyName`.

Additionally, this plugin highlights the property field in differencing mode when the property value has changed. To this end, it attaches a style class `issue-change` to its component if the property is reported as changed by the server.

For struct properties, a dot-separated property path can be used as the property name to visualize issues and differences of a property nested in a struct value.

Because the string property field shown here is based on a plain `textfield`, all formatting rules are already provided in the standard style sheets. For custom components, it might be necessary to add CSS rules for the style classes `issue-error`, `issue-warn`, and `issue-change` in order to visualize issues and changes correctly.

The `propertyFieldPlugin` and the `showIssuesPlugin` are often, but not always attached to the same component. In some cases it may appropriate to designate an outer component as the component to scroll into view when navigating to a property, but to select an inner component to be tagged with issue style classes.

```
<editor:showIssuesPlugin bindTo="{config.bindTo}"
        propertyName="{config.propertyName}"
        ifUndefined=""
        hideIssues="{config.hideIssues}"/>
```

The property label is used when displaying the component in a form. Using the following plugin, you can make sure that the label is localized according to the standard localization pattern.

```
<editor:setPropertyLabelPlugin
        bindTo="{config.bindTo}"
        propertyName="{config.propertyName}"/>
```

Show default text

When the string field is empty, you want to display a message instructing the user to enter a text. This, too, can be localized uniformly, and the `setPropertyLabelPlugin` sets your property field up to play along nicely.

```
<editor:setPropertyEmptyTextPlugin
        bindTo="{config.bindTo}"
        propertyName="{config.propertyName}"/>
```

Read-only

Also, the component should be made read only (meaning that the user cannot enter any text but still can mark and copy the content) when the edited content is checked out by another user or is forced to be read only by the document panel:

```
<editor:bindReadOnlyPlugin
forceReadOnlyValueExpression="{config.forceReadOnlyValueExpression}"
bindTo="{config.bindTo}"/>
```

Data binding

Lastly, the edited value must be passed to the server, and the component should display the server-side value. This ("data binding") is typically done using the versatile `bindPropertyPlugin`, like shown below. Note that the immediate changes plugin just triggers the change event often enough, whereas the `bindPropertyPlugin` handles the wiring to the server side, and in turn triggers when a change event is fired.

```
<ui:bindPropertyPlugin
        bindTo="{config.bindTo.
          extendBy('properties', config.propertyName)}"
        bidirectional="true"/>
```

Finally, end the component definition.

```
</plugins></textfield></exml:component>
```

While the list of plugins may appear quite long at first, it is very helpful to be able to separate the different aspects of a property field in different plugins. If you want to provide a custom algorithm of reacting to an empty value, for example, you can easily do so by just omitting the respective plugin declaration, and providing custom handling code - either in the base class or possibly extracted into your own reusable plugin.

7.4.3 Compound Field

The following code example shows a more complex scenario, where a field for a URL is created that lets the user open a browser window or tab for the linked page with a single click.

The EXML declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
<exml:component xmlns:exml="http://www.jangaroo.net/exml/0.8"
                xmlns="exml:ext.config"

xmlns:editor="exml:com.coremedia.cms.editor.sdk.config"
                xmlns:ui="exml:com.coremedia.ui.config"

baseClass="com.coremedia.ui.examples.propertyField.UrlPropertyFieldBase">


  <exml:cfg name="bindTo"
type="com.coremedia.ui.data.ValueExpression">
    <exml:description>
      A property path expression leading to the Bean whose property
 is edited.
    </exml:description>
  </exml:cfg>

  <exml:cfg name="propertyName" type="String">
    <exml:description>
      The property of the Bean to bind in this field.
    </exml:description>
  </exml:cfg>

    <exml:description>
   A text field that binds to a string property being edited inside

    of a document form. It allows to open the link target in a new
 window.
  </exml:description>

  <container cls="url-property-wrapper" layout="form">
    <items>
      <!-- The URL is edited in a text field. -->
      <textfield itemId="urlTextField"
                 name="{'properties.' + config.propertyName}"
                 labelSeparator=""
                 cls="string-property-field">
        <plugins>
          <!-- register the new property editor -->
          <editor:propertyFieldPlugin
propertyName="{config.propertyName}"/>
          <!-- Generate an appropriate label. -->
          <editor:setPropertyLabelPlugin bindTo="{config.bindTo}"

propertyName="{config.propertyName}"/>
          <!-- Write back changes even before the user leaves the
field. -->
          <ui:immediateChangeEventsPlugin/>
          <!-- When the field is empty, an informational message
should appear. -->
         <editor:setPropertyEmptyTextPlugin bindTo="{config.bindTo}"


propertyName="{config.propertyName}"/>
          <!-- Disable the field as appropriate for a content form.
 -->
          <editor:bindDisablePlugin bindTo="{config.bindTo}"/>

          <!-- Bind the content of the field to the given content
property. -->
          <ui:bindPropertyPlugin
bindTo="{config.bindTo.extendBy('properties', config.propertyName)}"

                                 ifUndefined=""
                                 bidirectional="true"/>
        </plugins>
```

```
      </textfield>
      <!-- Add a link to the URL displayed in the field. The actual
 handling is done by the super class. -->
      <ui:textLink itemId="textLink"

text="{PropertyFieldExample_properties.INSTANCE.UrlPropertyField_open_text}"

                   handler="{openFrame}"/>
    </items>
  </container>
</exml:component>
```

The base class:

```
package com.coremedia.ui.examples.propertyField {

import com.coremedia.ui.data.ValueExpression;
import
com.coremedia.ui.examples.propertyField.config.urlPropertyField;

import ext.Container;

public class UrlPropertyFieldBase extends Container {
  public function UrlPropertyFieldBase(config:urlPropertyField) {
    super(config);
  }

  /**
   * A property path expression leading to the Bean whose property
 is edited.
   */
  public native function get bindTo():ValueExpression;

  /**
   * The property of the Bean to bind in this field.
   */
  public native function get propertyName():String;

  /**
   * Try to open a new window with the string currently stored in
the property used as the URL.
   */
  public function openFrame():void {
    var url:String = bindTo.extendBy('properties',
propertyName).getValue() as String;
    if (url) {
      window.open(url, 'externalLinkTarget')
    }
  }
}
}
```

The above is an example of a compound field, where you need to wrap multiple Ext JS components in a container. This is possible, but you must take care to declare and pass around all configuration properties that need to be set on subcomponents.

There is also some application logic, which is what the base class is for. While you could technically embed any code into the EXML file itself, it is good practice to separate out application code in an ActionScript base class. Note how the EXML component references the method `openFrame` from the base class using curly brackets:

```
<ui:textLink itemId="textLink"
                 text="{PropertyFieldExample_properties.
                 INSTANCE.UrlPropertyField_open_text}"
                 handler="{openFrame}"/>
```

Example 7.25. Using a base class method

7.4.4 Complex Setups

Keep in mind that somewhat counter-intuitively, the base class constructor has not run while the component tree is built in the constructor of the EXML class. In particular, this means that methods calls in the EXML file (not mere usages of methods as event handlers) will find the fields of the base class uninitialized. For example, calling `<textfield name="{computeName()}" .../>` would enter the method `computeName` before the base class constructor has run, so that some initialization would have to be done early on demand. On the other hand, in `<button handler="{handleButton}"/>` the method `handleButton` is only invoked after the component is initialized. If a method that is called early needs access to the configuration, you must pass the `config` object as a parameter: `<textfield name="{computeName(config)}" .../>`.

7.4.5 Customizing RichText Property Fields

A richtext property field consists of the richtext toolbar and a WYSIWYG editing area, the `richTextArea`, which is a wrapper for an instance of the CKEditor. The CKEditor provides richtext editing features via plugins. It is important to note that Ext JS and CKEditor are independent and offer their own JavaScript API.

The richtext toolbar is a standard ExtJS toolbar and contains buttons and menu items that perform richtext-related actions. There are a pre-defined set of buttons which are activated on this toolbar, which may be configured. This is described in Section "Customizing Richtext Toolbar" [127]. It is possible to add or remove buttons or menus from the toolbar. This may be done for pre-defined and custom actions.

The richtext property field comes with a set of pre-defined actions which can be activated, deactivated or configured. At the end of this section is a list of configuration options for these actions.

Most of these actions are wired closely to the CKEditor in the sense that the actions invoke CKEditor commands, which in turn are defined by CKEditor plugins. Some of these plugins like `pastefromword` and `pastetext` use CKEditor dialogs (with a custom CoreMedia skin to better integrate into the *Studio* UI).

Other actions are plain ExtJS actions (maybe using an ExtJS dialog) that interact with the CKEditor directly via its API.

It is also possible to define custom actions by writing plugins for the richtext property field or by using CKEditor plugins directly. This is described in Section "Customizing CKEditor" [129].

As with the pre-defined actions, you may write custom actions which invoke CKEditor commands or write custom ExtJS actions which use the CKEditor API. This is described in Section “Interacting with the CKEditor via API” [131]

You can remove entire CKEditor plugins if required. When you do so, you should also remove the corresponding buttons or menu items that are wired to commands defined in that plugin.

The following is a list of configuration options for pre-defined richtext actions:

- Section “Inline Images in RichText” [123]: Configure the creation and display of inline images, which are stored in image documents.
- Section “Adding table cell merge and split commands” [123]: Add merge and split table cell functionality (per default deactivated).
- Section “Adding Custom RichText Style Classes” [124]: Add custom richtext styles.
- Section “Customizing the Symbol Mapping” [126]: Configure the symbol mapping.

Inline Images in RichText

By dragging image documents from the library into a richtext field you can create inline images. The document types that are supported for this operation and the image blob properties that are accessed to display the images can be configured using the `registerRichTextDragDropImageType` method of the global `editorContext` object. You can also use the `configureDocumentTypes` plugin as shown in the `BlueprintFormsStudioPlugin` of *CoreMedia Blueprint*:

Example 7.26. Inline images in richtext

```
<editor:configureDocumentTypes
        names="CMPicture,CMImage"
        richTextDropImageProperty="data"/>
<editor:configureDefaultRichTextImageDocumentType
        defaultRichTextImageType="CMPicture"/>
```

The previous example also shows how the `configureDefaultRichTextImageDocumentType` plugin can be used to configure the document type that limits the search when the library is opened using the embedded image button of the richtext toolbar.

Adding table cell merge and split commands

There are predefined commands for merging and splitting of table cells that can easily be made available in the richtext toolbar. To do so use the `addItemsPlugin` as described in the previous chapter.

The code would be like this:

```
<editor:richTextPropertyField>
  <plugins>
    <ui:addItemsPlugin recursive="true">
      <ui:items>
        <menuseparator/>
        <menuitem
itemId="{CELL_MERGE_ITEM_ID}"><baseAction><ui:richTextAction
commandName="{richTextAction.COMMAND_CELL_MERGE}"/></baseAction></menuitem>

        <menuitem
itemId="{CELL_MERGE_RIGHT_ITEM_ID}"><baseAction><ui:richTextAction

commandName="{richTextAction.COMMAND_CELL_MERGE_RIGHT}"/></baseAction></menuitem>

        <menuitem
itemId="{CELL_MERGE_DOWN_ITEM_ID}"><baseAction><ui:richTextAction
commandName="{richTextAction.COMMAND_CELL_MERGE_DOWN}"/></baseAction></menuitem>

        <menuitem
itemId="{CELL_VERTICAL_SPLIT_ITEM_ID}"><baseAction><ui:richTextAction

commandName="{richTextAction.COMMAND_CELL_VERTICAL_SPLIT}"/></baseAction></menuitem>

        <menuitem
itemId="{CELL_HORIZONTAL_SPLIT_ITEM_ID}"><baseAction><ui:richTextAction

commandName="{richTextAction.COMMAND_CELL_HORIZONTAL_SPLIT}"/></baseAction></menuitem>

      </ui:items>
      <ui:after>
        <component
itemId="{richTextPropertyField.TABLE_REMOVE_ITEM_ID}"/>
      </ui:after>
    </ui:addItemsPlugin>
  </plugins>
</editor:richTextPropertyField>
```

Adding Custom RichText Style Classes

You can add custom richtext style classes to the CKEditor. Style classes can be applied to block elements (for example, `p`) or inline elements (for example, `span`). Moreover, you can define groups of style classes allowing only one style class of that group to be set at a time. To define own style class groups, you have to add them via the `customizeCKEditorPlugin`, using its `classGroups` attribute of the `config` object as shown in the following code listing.

> The group name must not contain hyphens.

Note, that when you apply any configuration as described in the listing, this will overwrite the default configuration in the product, rather than appending to it. Thus, you will typically want to re-add the defaults in your custom configuration - this is shown in the listing below, too.

```
<editor:studioPlugin>
  <ui:rules>
    <ui:richTextArea>
```

```
      <plugins>
        <ui:customizeCKEditorPlugin>
          <ui:config>
            <exml:object classGroups="{{
            'box' : {  /* name of the style class group */
              blockElements:'p',  /* block element(s) to which this
 */
              styleClasses: [     /* group should be applied */
                 'box--test-1',
                 'box--test-2'
              ]
            },
             /* re-add default style class group definitions  */
            'p' : {
              blockElements:'p',
              styleClasses: [
                 'p--heading-1',
                 'p--heading-2',
                 'p--heading-3'
              ]
            },
            'align' : {
              blockElements:'p',
              styleClasses: [
                 'align--left',
                 'align--right',
                 'align--center',
                 'align--justify'
              ]
            }
          }}"/>
          </ui:config>
        </ui:customizeCKEditorPlugin>
      </plugins>
    </ui:richTextArea>
  </ui:rules>
</editor:studioPlugin>
```

How to determine to which block element the style will be applied

The `blockElements` attribute is used to define which block elements the style should be applied to. Given the current cursor position when the respective command is invoked, the system will walk the DOM hierarchy upwards until it finds a block element whose name matches the one given in the `blockElements` attribute. The attribute may also contain an array of element names if the style class can be applied to different elements - in this case, the style will be applied to the first element found that matches any of the element names given. If you omit the attribute, the style group definition is treated as an inline style.

The `styleClasses` attribute is used to set an array of style class names. The naming format is up to you, but the "--" syntax given in the example is the best practice.

Adding CSS rules

To visualize a custom style in CKEditor, you need to add the respective CSS rules. As the CKEditor in the *Studio* is using a `div` container instead of an `iframe` you cannot use the `contentCss` configuration of the CKEditor, but have to load the CSS rules directly into the *Studio* (see section “Load external resources” [93]). Use `coremedia-richtext-1.0.css` as a reference on how to write the CSS rules so that they only apply to the CKEditor.

The command names necessary to apply the style classes to selected text will be `style_<classGroupName>_<styleClassName>`. The command name to remove the style class will be `style_<classGroupName>__remove`. Those commands can be added to the `richTextPropertyField` via the `addItemsPlugin` as shown in the next code listing.

```
<editor:richTextPropertyField>
  <plugins>
    <ui:addItemsPlugin recursive="true">
      <ui:items>
        <button text="box">
          <menu>
            <menu>
              <items>
                <acme:boxButton text="test 1"
richtextcommand="style_box_box--test-1"/>
                <acme:boxButton text="test 2"
richtextcommand="style_box_box--test-2"/>
                <menuseparator/>
                <acme:boxButton text="remove box style"
richtextcommand="style_box__remove"/>
              </items>
            </menu>
          </menu>
        </button>
      </ui:items>
      <ui:after>
        <component itemId="{...}"/>
      </ui:after>
    </ui:addItemsPlugin>
  </plugins>
</editor:richTextPropertyField>
```

In this example, the `BoxButton` is used as a wrapper around the richtext action using the mentioned commands. It is defined in a `BoxButton.exml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<exml:component xmlns:exml="http://www.jangaroo.net/exml/0.8"
                xmlns="exml:ext.config"
                xmlns:ui="exml:com.coremedia.ui.config">
  <exml:cfg name="richtextcommand" type="String"></exml:cfg>
  <menucheckitem group="box">
    <baseAction>
      <ui:richTextAction commandName="{config.richtextcommand}"/>
    </baseAction>
  </menucheckitem>
</exml:component>
```

Customizing the Symbol Mapping

Mapping Word symbol font items

When pasting rich text from Microsoft Word into *CoreMedia Studio*, some characters of the pasted text might originate from the Word symbol font. *CoreMedia Studio* translates such characters using a mapping table containing all commonly used characters. However, if your editorial staff uses more obscure symbols, wrong characters may appear after pasting.

To ensure that the characters shown in the rich text area correspond to the symbol character from a Word document, you can define an extension to the symbol

mapping. You map each additional character to the HTML entity or the Unicode character in a JavaScript object. Afterwards you can pass that object to the method `registerRichTextSymbolMapping` of the editor context during the initialization phase of a Studio plugin.

The following code shows how new symbol mappings are registered at startup time.

Example 7.27. Configuring the rich text symbol mapping

```
public function init(editorContext:IEditorContext):void {
  ...
  editorContext.registerRichTextSymbolMapping({
    'Ä': '&otimes;',
    'Å': '&oplus;'
  });
  ...
}
```

Customizing Richtext Toolbar

The buttons and menu items of the toolbar can be customized by applying the `addItemsPlugin` and `removeItemsPlugin` to `richTextPropertyField`. The item ids of the buttons and menu items provided are listed as constants in the ASDoc of `richTextPropertyField`.

It is also possible to add a toolbar button for a custom plugin or a CKEditor plugin.

Add action to new button

When adding a new button to the toolbar and you want it to perform a CKEditor command, you can use the `richTextAction` with the configured command name. Currently used commands are listed as constants in the ASDoc.

When adding or extending a menu in the toolbar and the menu items should perform `richTextActions` for context-sensitive CKEditor commands, you should use the `richTextMenuCheckItem` for a correct representation of the enabled and active states of the command. See the ASDoc for more information.

It is recommended to add the functionality into a Studio plugin that can be used in the `richTextPropertyField` configuration (see Section 7.1, “Studio Plugins” [84] for Studio plugins). The following code, included in a file `CustomizeRichTextPlugin.exml`, moves the italic button between the internal link and external link button and removes the heading 3 paragraph format menu from the rich text toolbar.

Example 7.28. Customizing the rich text editor toolbar

```
<?xml version="1.0" encoding="UTF-8"?>
<exml:plugin xmlns:exml="http://www.jangaroo.net/exml/0.8"
             xmlns="exml:ext.config"
             xmlns:ui="exml:com.coremedia.ui.config">
  <exml:import
class="com.coremedia.cms.editor.sdk.config.richTextPropertyField"/>


  <ui:nestedRulesPlugin>
```

```
    <ui:rules>
      <toolbar>
        <plugins>
          <ui:removeItemsPlugin>
            <ui:items>
              <component
itemId="{richTextPropertyField.ITALIC_BUTTON_ITEM_ID}"/>
              <component
itemId="{richTextPropertyField.PARAGRAPH_HEADING3_ITEM_ID}"/>
            </ui:items>
          </ui:removeItemsPlugin>
          <ui:addItemsPlugin>
            <ui:items>
              <ui:iconButton
itemId="{richTextPropertyField.ITALIC_BUTTON_ITEM_ID}">
                <baseAction>
                  <ui:richTextAction
commandName="{richTextAction.COMMAND_ITALIC}"/>
                </baseAction>
              </ui:iconButton>
            </ui:items>
            <ui:after>
              <component
itemId="{richTextPropertyField.INTERNAL_LINK_BUTTON_ITEM_ID}"/>
            </ui:after>
          </ui:addItemsPlugin>
        </plugins>
      </toolbar>
    </ui:rules>
  </ui:nestedRulesPlugin>
</exml:plugin>
```

The `baseAction`, as in the above example, can also reference a custom action defined in a custom or CKEditor plugin. In this case, the `commandName` of the `richtextAction` is the name given in the plugin definition.

You can either apply the plugin to all rich text fields or only to a specific content type. When you add it to your `*StudioPlugin.exml` file (the `<bp:customizeRichTextPlugin/>` line), then the plugin is applied to all rich text fields:

```
<editor:studioPlugin>
  <ui:rules>
    …

    <editor:richTextPropertyField>
      <plugins>
        <bp:customizeRichTextPlugin/>
      </plugins>
    </editor:richTextPropertyField>
  </ui:rules>
</editor:studioPlugin>
```

When the plugin should only be applied to a specific rich text field, you have to add it to a specific `*DocumentForm.exml` file:

```
<editor:documentTabPanel>
  <items>
   …
  <editor:richTextPropertyField propertyName="detailText">
    <plugins mode="append">
      <bp:customizeRichTextPlugin/>
```

```
    </plugins>
  </editor:richTextPropertyField>
</items>
</editor:documentTabPanel>
```

Here, it is important, that you use the `mode="append"` attribute. Otherwise, you would remove all plugins that are already defined for this field.

Add custom icon

You may also add a custom icon to the toolbar or use one bundled with an existing CKEditor plugin. To do this, apply the `addItemsPlugin` as above to the `richTextPropertyField`. The `iconButton` can take the arguments `iconCls`, `text` and `tooltip` in order to apply and localize the custom icon. The `iconCls` property defines the css class of the icon. The icon image location and style may then be added to the css using the css class name defined by the `iconCls`.

Example 7.29. Adding a custom icon to the rich text editor toolbar

```
<ui:iconButton
iconCls="{MyPluginLabels_properties.INSTANCE.MyPlugin_icon}"

tooltip="{MyPluginLabels_properties.INSTANCE.MyPlugin_tooltip}"

text="{MyPluginLabels_properties.INSTANCE.MyPlugin_text}">
```

As in the example above, these three properties may be defined in a separate properties bundle which can be localized.

Customizing CKEditor

CKEditor plugins

The CKEditor provides richtext editing capabilities in a browser independent way. It has a plugin-driven architecture. Plugins are JavaScript files that are loaded at the end of the CKEditor loading process, before the initialization and activation of CKEditor instances. Plugins are named and defined in a file named `plugin.js` which resides under a path matching the plugin's name. Plugins may add UI features, change the behavior of existing UI components or add data manipulation features. The CKEditor provides automatic runtime plugin dependency management.

The custom plugin `my-plugin` can be added to a *CoreMedia Studio* project by editing the following file

```
my-project/resources/META-INF/resources/ckeditor/_source/plugins/my-plugin/plugin.js
```

If not already done in the parent project, the path of the `plugin.js` has to be configured as a additional resource in the project `pom.xml`:

Example 7.30. Adding resource path to `pom.xml`

```
<resources>
  ...
  <resource>

<directory>${basedir}/src/main/resources/META-INF/resources/ckeditor/_source</directory>

    <targetPath>META-INF/resources/ckeditor</targetPath>
```

```
  </resource>
</resources>
```

Note that when explicitly configuring custom resources in a maven pom, you will need to include the maven default resource rules, such as copying of src/main/resources and src/main/generated-resources to the target (represented as ... in the example above). Please consult the blueprint sources for example usages.

The content of the `plugin.js` may be similar to

Example 7.31. Customizing the CKEditor

```
CKEDITOR.plugins.add('my-plugin',
{
  beforeInit(editor){
    ...
  },
  init : function(editor) {
    ...
  },
  lang : [...],
  requires: [...]
});
```

The argument passed to the `add` method is a so-called plugin definition whose `beforeInit` and `init` functions are called upon creation of every CKEditor instance in that package. The definition may also provide the `lang` and `requires` attributes which respectively define valid languages for the plugin and a list of required plugins.

The official CKEditor API documentation is available at http://docs.ckeditor.com/#!/api.

The custom plugin can now be registered by the CKEditor. This is done by using the `addCKEditorPluginsPlugin` with your `richTextArea`:

```
<ui:richTextArea>
  <plugins>
    <ui:addCKEditorPluginsPlugin plugins="my-plugin"/>
  </plugins>
</ui:richTextArea>
```

You can remove predefined plugins so that they are not loaded by CKEditor. This is done by using the `removeCKEditorPluginsPlugin` in your `richTextArea`. To remove the CKEditor plugin `about`, for example, add the following to your declaration:

```
<ui:richTextArea>
  <plugins>
    <ui:removeCKEditorPluginsPlugin plugins="about"/>
  </plugins>
</ui:richTextArea>
```

The list of additional CKEditor plugins loaded by *CoreMedia Studio* by default is documented in the ASDoc of `richTextArea` as the constant `defaultCKEditorExtraPlugins`. The list of standard CKEditor plugins, that are excluded by default are listed in the ASDoc of `richTextArea` as the constant `defaultCKEditorRemovePlugins`.

To change other configuration options of CKEditor, you can use the `customizeCKEditorPlugin` with your `richTextArea`. A list of CKEditor configuration options can be found here: CKEditor.config For example, to instruct the CKEditor to add 2 spaces to the text when hitting the TAB key, use the following code:

```
<ui:richTextArea>
  <plugins>
    <ui:customizeCKEditorPlugin>
      <ui:config>
        <exml:object tabSpaces="2"/>
      </ui:config>
    </ui:customizeCKEditorPlugin>
  </plugins>
</ui:richTextArea>
```

Items or Buttons which execute custom CKEditor commands have to be added to the richtext toolbar using the `AddItemsPlugin` as described in Section “Customizing Richtext Toolbar” [127]. This cannot be done in the CKEditor directly.

Interacting with the CKEditor via API

Interacting with the CKEditor via API

If you want to interact with the CKEditor without writing a CKEditor plugin, you can add a standard ExtJS action to the toolbar of the `richTextPropertyField`. To gain access to the CKEditor you have to create a `baseClass` for your action and add the following method:

```
[InjectFromExtParent]
public function setCKEditor(editor:*):void {
  ...
}
```

The injected `editor` object is of type `CKEDITOR.editor` (see http://docs.ckeditor.com/#!/api/CKEDITOR.editor) and can be used according to your needs.

However, there are two things to consider when writing your own custom actions:

- *Undo / Redo:* In order to be able to undo / redo the changes your action has made, you have to send one `saveSnapshot` event before and one after making the changes, like so:

  ```
  editor.fire('saveSnapshot');
  // perform changes ...
  editor.fire('saveSnapshot');
  ```

- *Saving Changes:* In order to update the bound content with the changes your action has made, it may be necessary to send an additional `save` event. This

event is recognized by the property field which will then trigger the update. The CKEditor already tracks changes and the property field will react to it, but in some cases this is not possible. You should check if the content gets checked-out when your action is performed, and if not add the following code:

```
// perform changes ...
editor.fire('save');
```

7.5 Upgrading the CKEditor

The upgrade from CKEditor 3 to 4.5.7 provides many advantages in stability, bugfixes and the opportunity to expand functionality with minimal migration costs. As the bugfixes and new functionality have been integrated into the standard CKEditor API and plugins, there is now the possibility to use CKEditor functionality out of the box.

This section will outline the steps needed to take in order to upgrade standard and custom plugins to CKEditor 4. At the end is a list of CoreMedia bug reports which have been solved with this upgrade.

7.5.1 Upgrading RichTextArea Plugins from CKEditor 3 to 4

- Update the version of `jangaroo-libs` to the latest version. This contains the source code for CKEditor 4.
- Choose the CKEditor and CoreMedia plugins that should be loaded by applying the `addCKEditorPluginsPlugin` or `removeCKEditorPluginsPlugin` to a Studio plugin. Choose the buttons which will appear in the richtext toolbar by applying the `additemsplugin` or `removeitemsplugin` to the `richtextpropertyfield`. See Section “Customizing Richtext Toolbar” [127]. Below are tables of CKEditor and CoreMedia Richtext plugins loaded by default.
- Event listening has changed in CKEditor 4, which may affect custom plugins. As the events to select menu items from a custom dropdown menu (e.g. custom styles) have changed, the `richTextMenuCheckItem` has been introduced to replace the `menucheckitem` in dropdown menus. If the menu has a base class, make sure to make this change there too.
- `CKEDITOR.dom.selection.getRanges()` returns undefined if the selection has length 0. Code which uses this range should check if it exists before calling properties or methods on it.
- CSS styles can no longer be added on a editor instance basis (e.g. `editor.addCss()`). As the CKEditor in the *Studio* is using a `div` container now instead of an `iframe` you cannot use the `contentCss` configuration of the CKEditor anymore, but have to load the CSS rules directly into the Studio (see section “Load external resources” [93]).
- Global themes are no longer supported and thus the global coremedia theme has been removed.
- Skins can be copied from the standard CKEditor collection and customized for styling (dialogs). The coremedia skin is the default. It provides some adjustments for dialogs, so that they better integrate into the *Studio* UI.

For more CKEditor API changes, see the CKEditor upgrade guide:

- CKEditor 3 Upgrade Guide
- API Changes in CKEditor 4

If you are also upgrading RichTextArea Plugins from older versions of CoreMedia, please note that

- InjectFromExtParent annotation must be added to `setCkEditor()` methods
- The `RichtextToolbar` does not exist anymore

7.5.2 Migrating Richtext Editor Dialogs

The architecture for richtext editor dialogs has changed. It is now possible to use standard CKEditor dialogs and style them with skins. The coremedia skin is available as a template which may be customized.

If you have custom CKEditor plugins that use dialogs then your old solution will not work with CKEditor 4 and you have to migrate it. The reason for this, is that coremedia shipped custom coremedia plugin for CKEditor, called `'extdialog'`, which patched the standard CKEditor dialogs and forwarded the calls that they would normally receive over to Ext-Dialogs and other way around from Ext-Dialogs to CKEditor. Such Ext-Dialogs could be defined in EXML and styled with CSS in the same way as other studio components. Unfortunately this blocked the posibilty for coremedia customers to use the CKEditor plugins containing dialogs out of the box.

With the upgrade to CKEditor 4 *Coremedia* removed the 'extdialog'-Plugin gaining the ability to use OTB CKEditor Plugins with CKEditor 4. The dialogs included in such plugins would be displayed with standard CKEditor CSS Styles, unless they are styled with coremedia skins as mentioned above.

Migration Steps

To migrate your CKEditor plugins, which include custom dialogs, you need to take following steps:

1. Define an action (e.g. `MyPluginAction.exml` and `MyPluginActionBase.as`) and a dialog (e.g. `MyPluginDialog.exml` and `MyPluginDialog.as`).
2. Inject CKEditor into your action by adding the following method to `MyPluginActionBase.as`:

```
[InjectFromExtParent]
public function setCKEditor(editor:*):void {
  ...
}
```

For more on injection and inversion of control in studio see Section 7.8, “Customizing Studio using Component IoC” [143].

3. Your dialog has to receive CKEditor as a config parameter

```
<exml:cfg name="editor" type="*"/>
```

in MyPluginAction.exml.

```
protected native function get editor():*;
```

in MyPluginActionBase.as

4. The handler method of your action must instantiate the dialog and pass the *ckEditor* (that it got injected with the help of `InjectFromExtParent` annotation) to the newly instantiated dialog.

5. In the EXML of the dialog you can define the components you need as well as the OK- and CANCEL-Handler for your dialog. In the base class of the dialog you can then get the reference to the CKEditor instance. The logic, that was earlier programmed in JavaScript in the CKEditor-dialog of your plugin, now should be transformed into Action Script of the base class of Ext-Dialog (`MyPluginDialog.as`).

6. Now you can delete the folder with the CKEditor-dialog from your plugin as well as the following code registering this dialog from its `plugin.js`:

```
editor.addCommand(commandName, new
CKEDITOR.dialogCommand(dialogName));
CKEDITOR.dialog.add(dialogName, this.path +
'dialogs/my-plugin-dialog.js');
```

In many cases you should consider deleting your plugin, completely, if the whole logic can be ported to Action Script base classes of your Ext-Action (`MyPluginActionBase.as`) and Ext-Dialog (`MyPluginDialog.as`) that would communicate with the injected CKEditor per API calls.

7.5.3 CKEditor plugins available

Table 7.4. CKEditor plugins loaded by default

CKEditor Plugin	Functionality	Requires, required by	Dialog
basicstyles	Bold, italics, etc.		
blockquote	Blockquote		
contextmenu (disabled by default in RichTextArea)	Context Menu		

CKEditor Plugin	Functionality	Requires, required by	Dialog
find (disabled by default in RichTextArea)	Find and replace		Styled CKEditor dialog
list	Numbered and bullet lists	Requires indentlist	
indentlist	Indent and outdent of list items	Requires indent, list	
pastefromword	Maps Word formatting to richtext	Requires clipboard	Styled CKEditor dialog
pastetext	Removes formatting from text	Requires clipboard	Styled CKEditor dialog
table	Add and edit tables	Requires tabletools, menu, floatpanel, panel, showborders	CM dialog
undo	Undo and redo		
link	Add and remove external links	Requires fakeobjects	CM dialog
entities		Required by Moderation Panel Comment View richtext area (for Elastic Social)	
divarea, wysiwygarea	CKEditor editing area	Required by CM richtext area. CKEditor is now enclosed in a div tag rather than an iframe.	
dialog, dialogui	Dialog elements	Required by plugins which use CKEditor dialogs	

Table 7.5. CM richtext plugins loaded by default

CM Richtext Plugin	Functionality	Requires, required by	Dialog
cmrichtextwriter, cmrichtextdataprocessor	Writing and processing of CM richtext	Requires htmlwriter	
classstyles	Headings, alignment (max. 1 style per group)		CM context menu, uses `richTextMenuCheckItem`

CM Richtext Plugin	Functionality	Requires, required by	Dialog
cmstyles	Inline styles such as underline and strikethrough		CM context menu, uses `richTextMenuCheckItem`

7.6 Coupling Studio and Embedded Preview

In the [CoreMedia Content Application Developer Manual/Adding Document Metadata] it is described in detail how to use the *Content Application Engine* to include metadata in Web documents.

This section explains how to access metadata of documents that are shown in the Studio's embedded preview.

7.6.1 Built-in Processing of Content and Property Metadata

CoreMedia Studio automatically accesses and interprets content and property metadata in order to connect preview and document form. When the user edits a property that is mapped to a preview DOM element via metadata, all changes are reflected in the embedded preview, either instantly (for simple properties like strings) or through automatically reloading the preview.

Moving the mouse cursor over the preview will highlight elements with attached content and/or property metadata. Right-clicking one of these elements in the preview focuses the corresponding form field, if possible. If the clicked element belongs to a content object different from the content object currently displayed in the document form, a context menu is opened that shows a breadcrumb to navigate through the metadata hierarchy down to the clicked content object, and it offers the options to open the content in a new tab or in the library.

7.6.2 Using the Preview Metadata Service

As described in [CoreMedia Content Application Developer Manual/Adding Document Metadata], it is possible to include arbitrary metadata in Web documents by means of the FreeMarker macro `<@cm.metadata>` or the custom JSP tags `<cm:metadata>`, `<cm:property>` and `<cm:object>`. In the rendered Web document, the different metadata chunks are included as JSON-serialized values of the custom HTML attribute `data-cm-metadata` of different DOM nodes. While metadata can be added using FreeMarker or JSP, this section uses the JSP tags in its examples.

The Metadata Service Interface

Communication between Studio and CAE web application

In Chapter 3, *Deployment* [18] it is described that the preview *CAE* web application and *Studio* communicate via an internal messaging system. This messaging system is also used to transfer metadata from the preview side to the *Studio* side. To hide this low-level layer from the Studio developer, CoreMedia offers a *metadata service* for each instance of a preview panel that runs in *CoreMedia Studio*. Given a preview panel, its metadata service can be obtained as follows (please see the API docu-

mentation of `PreviewPanel` for further information on how to obtain a preview panel component).

```
var previewPanel:PreviewPanel = ... ;
var metadataService:IMetadataService =
  previewPanel.getMetadataService();
```

The metadata service interface currently offers just one method, namely:

```
IMetadataService.getMetadataTree(selectionProperties:Array = null)
```

Via this method, the metadata of the associated preview panel's document can be retrieved. Metadata embedded in the preview document is represented in terms of a tree. This *metadata tree* originates from the DOM tree of the preview document: Hierarchical relationships between the metadata tree nodes correspond to hierarchical relationships between the DOM tree nodes that the respective metadata chunks are attached to. Consequently, the metadata tree is basically a projection of the DOM tree to its metadata information.

It is possible to further filter the metadata tree by means of the method's optional parameter, namely an array of properties. If such properties are supplied, the metadata tree contains only nodes that have at least one of these properties. In addition, other properties than the given properties are filtered out. Such a filtered metadata tree is a projection of the metadata tree that contains all metadata. The above statement about the correspondence of hierarchical relationships in the metadata tree and the DOM tree still holds.

Working with the Metadata Tree

When working with the metadata tree, you have two data structures to your convenience:

- `com.coremedia.cms.editor.sdk.preview.metadata.MetadataTree`: This data structure represents the whole tree and, for example, offers methods for accessing specific nodes (by their ID) or getting a list of all tree nodes (in breadth-first order).
- `com.coremedia.cms.editor.sdk.preview.metadata.MetadataTreeNode`: This data structure represents a single metadata tree node. It offers a range of methods like retrieving the parent or the children of a node, finding specific parent nodes upwards in the hierarchy or specific child nodes downwards in the hierarchy or accessing properties of a metadata tree node.

In the following you will find two examples of how to use the metadata tree. Suppose that the JSP templates on the *CAE* side have been prepared to include metadata about content. At different points throughout the JSP templates the code might look as follows:

```
...
<cm:metadata var="contentMetadata">
  <cm:property name="contentInfo">
    <cm:property name="title"
                 value="${self.content.title}"/>
    <cm:property name="keywords"
                 value="${self.content.keywords}"/>
  </cm:property>
</cm:metadata>

<div ${contentMetadata}>
  ...
</div>
...
```

In a preview document there might be multiple of such content-related metadata chunks attached to different DOM nodes. Suppose you want to gather the titles of all the contents that are included in such metadata chunks. One way to gather these titles in an array is the following:

```
var metadataService:IMetadataService = ... ;
var metadataTree:MetadataTree = metadataService.getMetadataTree();
var result:Array = [];
if (metadataTree.getRoot()) {
  var nodesToProcess:Array = [metadataTree.getRoot()];
}
var arrayIndex:int = 0;
while (arrayIndex < nodesToProcess.length) {
  var currentNode:MetadataTreeNode = nodesToProcess[arrayIndex];
  if (currentNode.getProperty("contentInfo")) {
    var title:String =
      currentNode.getProperty("contentInfo").title;
    result.push(title);
  }
  if (currentNode.getChildren()) {
    nodesToProcess =
      nodesToProcess.concat(currentNode.getChildren());
  }
  arrayIndex++;
}
```

In this example, the whole metadata tree is traversed in a breadth-first manner. For each node it has to be checked whether it has the `contentInfo` property as there might be metadata nodes with completely other information.

The code can be simplified considerably if a filtered metadata tree is retrieved:

```
var metadataService:IMetadataService = ... ;
var metadataTree:MetadataTree =
  metadataService.getMetadataTree(["contentInfo"]);
var result:Array = [];
var metadataNodesList:Array = metadataTree.getAsList();
metadataNodesList.forEach(function (node:MetadataTreeNode) {
  result.push(node.getProperty("contentInfo").title);
}
```

In this case, the metadata tree is filtered on retrieval, namely for metadata nodes that contain the `contentInfo` property. Now it is sufficient to get all metadata tree nodes as an array, walk through it and gather the content titles.

Listening to Metadata Availability/Changes

A metadata service is always associated with a specific preview panel. When a document is opened in a preview panel, it takes some time until its metadata is loaded. This happens asynchronously via the above mentioned message service. Consequently, it is necessary to have a mechanism to listen to the availability of a document's metadata. In addition, changes to the metadata may occur when the displayed document of the preview panel changes. Thus, it is also necessary to listen to metadata changes.

To this end, the method `IMetadataService.getMetadataTree()` is dependency-tracked. This means that it is possible to listen to changes to the returned metadata tree by using a function value expression (see `com.coremedia.ui.data.dependencies.DependencyTracker` and `com.coremedia.ui.data.ValueExpressionFactory.createFromFunction`). The following example is provided to illustrate this process:

```
var previewPnl:PreviewPanel = ... ;
ValueExpressionFactory.createFromFunction(
  function():MetadataTreeNode {
    var metadataTree:MetadataTree =
      previewPnl.getMetadataService().getMetadataTree();
    return metadataTree.getRoot() ? metadataTree : undefined;
  }
).loadValue(
  function(metadataTree:MetadataTree):void {
    // metadata tree loaded!
    metadataTree.getAsList()...;
  }
);
```

In this example `MetadataTree.getRoot()` is used as an indicator of whether the metadata has already been loaded (if not, the method returns `null`). A function value expression is created around a function that simply determines the existence of a metadata root node, returning `undefined` as long as it does not exist. Afterwards the value expression is loaded, which automatically retries to invoke the function until it returns a non `undefined` value. As soon as it does, the metadata has been loaded and the callback function can now process the metadata tree.

7.7 Storing Preferences

A custom component may have to store user preferences persistently. To this end, the editor context object implements the method `getPreferences` of the interface `IEditorContext`. The method returns a `Struct` object that is stored in the `EditorPreferences` document of the current user. You can modify this struct using the standard struct API as described in Section 5.4.4, "Structs" [54].

The class `PreferencesUtil` provides two utility methods for reading and writing complex objects in the preferences struct: `getPreferencesJSONProperty` and `updatePreferencesJSONProperty`. These methods support strings, numbers, Boolean, contents, and complex objects and arrays containing such values. The Studio API uses these methods internally for persisting saved searches (including custom filters), open tabs, dashboard widget states, and bookmarks.

7.8 Customizing Studio using Component IoC

Section 5.5, “Studio Component IoC” [57] describes how a property of an Ext JS parent component can be injected to a child component. Here you learn how a button or a menu item can be added to toolbars or menus of *Studio* and its base action can be configured using the Component IoC.

7.8.1 Content Actions

The newly public API class `ContentAction` is the most prominent context consumer in *Studio* and uses the configurable annotation:

```
 [InjectFromExtParent(variableNameConfig='contentVariableName')]
 public function setContents(contents:Array):void {
   ...
 }
```

While the class is an *abstract* class, all its child classes accept the injection of the contents property as long as the annotated method is not overridden. For example `approveAction`, `publishAction` and more. So such actions' content can be injected using the `contentVariableName` configuration parameter.

7.8.2 Example: Add a disapprove button to the actions toolbar

Assume that you want to add a new button to the actions toolbar. Clicking the button should disapprove the current content of the work area. This is the plugin rule which will do the job:

```
...
<editor:studioPlugin>
  <ui:rules>
    <editor:actionsToolbar>
      <plugins>
        <ui:addItemsPlugin>
          <ui:items>
            <button>
              <baseAction>
                <editor:disapproveAction
contentVariableName="{actionsToolbar.CONTENT_VARIABLE_NAME}"/>
              </baseAction>
            </button>
          </ui:items>
        </ui:addItemsPlugin>
      </plugins>
    </editor:actionsToolbar>
  </ui:rules>
</editor:studioPlugin>
...
```

The configuration class `actionsToolbar` is a public API context provider and describes in its AS doc the constant `CONTENT_VARIABLE_NAME`: [The context

property name for the content on which the actions will operate. It is the current content in the work area.]

7.8.3 Studio Component Map

To add a new button or menu item to *Studio* using the IoC you need a kind of geographical knowledge of *Studio*. The following maps present the context providers (gray box) and their provided properties (blue box) in CoreMedia Studio. The gray boxes without the provided properties are extension points where you can add a new button or menu item. The green boxes represent the default actions of the extension points.

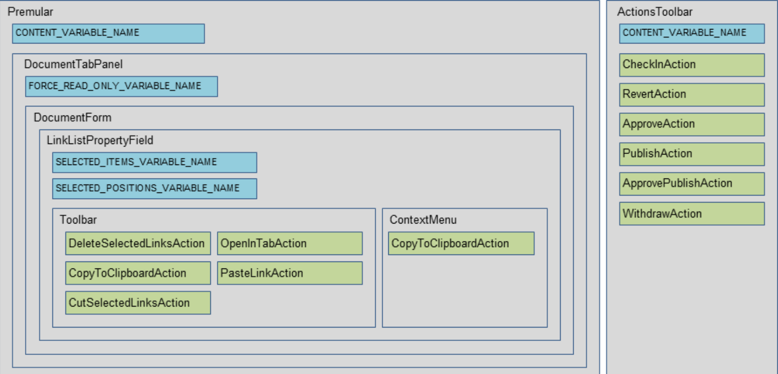


Figure 7.4. Premular and Actions Toolbar

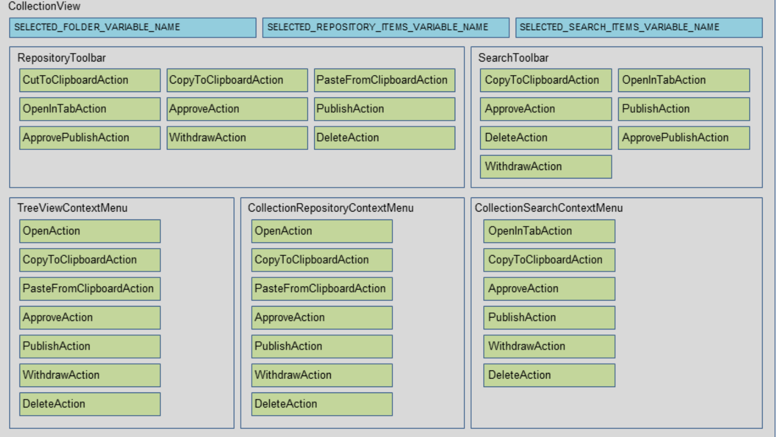


Figure 7.5. Collection View

7.9 Customizing Central Toolbars

Toolbars contain buttons for making functionality quickly accessible. There are two central toolbars that you might want to customize:

- The Favorites toolbar on the left, which contains shortcuts to often-used functions
- The Actions toolbar on the right, which contains buttons for completing the work on the current content

The following section describes how you can use the `addItemsPlugin` to add your custom button to an existing toolbar.

It is good practice to wrap the custom UI component's actual functionality (that is, what your button will do when clicked) in `Action` objects, so that these actions can be reused even if the visual representation of the button is changed to another component. The background of actions is described in Section 5.1.4, "Actions" [31].

7.9.1 Adding buttons to the Favorites Toolbar

The Favorites Toolbar on the left side is vertically divided into three subcontainers:

- The top section, which in Blueprint's default configuration has buttons to open the Library, to open Bookmarks, and to create content
- The middle section (`favoritesToolbarUserItemsArea`), which contains user-managed search folders. Users can add, delete, and modify their own, often used search folders in this section. Therefore, your ability to preconfigure this section is inherently different from the top and bottom section of the Favorites Toolbar. For details, see Section 7.9.2, "Providing default Search Folders" [146] below
- The bottom section, which in Blueprint's default configuration has a menu button for developer-defined extensions. You can add buttons within the extensions menu button (but you can also add top-level buttons above or below

If you want to add fixed buttons to the Favorites Toolbar (that is, buttons that can not be modified or removed by the user), you need to add them to either the top or the bottom section of the Favorites Toolbar.

The main `BlueprintStudioPlugin.exml` file shows how you can easily use the `addItemsPlugin` to add your own buttons to the top and bottom sections of the toolbar, respectively:

```
<editor:favoritesToolbar>
  <plugins>
    <ui:addItemsPlugin>
      <ui:items>
        <bp:newContentMenuButton/>
```

```
      </ui:items>
      <ui:after>
        <component
itemId="{favoritesToolbar.LIBRARY_BUTTON_ITEM_ID}"/>
      </ui:after>
    </ui:addItemsPlugin>
  </plugins>
</editor:favoritesToolbar>
```

Ensuring a proper order of the items of the favorites toolbar helps significantly in making the application usable. Note how an `after` constraint is used to put the new button to a specific place. It uses the framework-predefined `itemId` of the Library button to describe the desired location of the added button, and references the new button via an element with Blueprint namespace - the button itself is declared in a separate EXML file in the Blueprint Studio plugin.

To add a simple test button with an action to the *Blueprint* definition, enter the following code inside the `<items>` element (see Section 7.2, "Localizing Labels" [94] to learn how to localize the label of the button):

Example 7.32. Adding a search for documents to be published

```
<button itemId="exampleButton">
  <baseAction>
    <editor:showCollectionViewAction text="toBePublished"
     published="false" editedByMe="true" contentType="CMArticle"/>
  </baseAction>
</button>
```

This code snippet will add a "search folder" button to the favorites toolbar that uses a `showCollectionViewAction` to open the Library window in a mode that searches for a restricted set of content items (please see the API documentation for `showCollectionViewAction` for more details).

7.9.2 Providing default Search Folders

The middle section of the *CoreMedia Studio*'s favorites toolbar is made up of the component `favoritesToolbarUserItemsArea` which contains user-defined search folders. When you click a search folder, the collection view opens up in search mode showing the results of a predefined query. The user can create custom search folders via the `Save Search` button of the collection view's toolbar in search mode. Users can also modify existing search folders, change their order, rename them, or delete them altogether.

As a developer, you can provide a default set of search folders to your first-time users, so that the middle section won't appear empty on a user's first login to Studio.

Note that the configuration option shown below explains solely the default set of search folders that users will see on their first login. When Studio detects that there are no custom search folders defined yet for the user logging in, this default

set will be copied to this user's settings - from then on, management of the search folder section is completely up to the user, and your configuration will be ignored. If you want to permanently add buttons (including buttons representing search folders) to the Favorites Toolbar, please refer to Section 7.9.1, “Adding buttons to the Favorites Toolbar” [145] above.

You can add default search folders by using the `addArrayItemsPlugin` on the `favoritesToolbarUserItemsArea`. Each array item has to include the relevant search parameters that you want to pass to the collection view on opening. These parameters are modularized in terms of the different parts of the collection view in search mode. Thus, each array item is a nested JavaScript object literal that itself contains possibly multiple objects for the various parameter parts. These embedded objects can be accessed via unique keys (see below). In addition, each array item is given a unique name that will also be used as the display text for the resulting search folder in the favorites toolbar.

By default, the different search parameters of the collection view are divided into the following parts:

- The *main part* (key `_main`), featuring the search parameters `searchText`, `contentType`, `mode`, `view`, `folder`, `orderBy`, and `limit`.

  Note that for the `folder` property, it is possible to use both of the following notations:

  1. `folder: {$Ref: "content/9"}` (Rest URI path)
  2. `folder: {path: "/Sites/Media"}` (content repository path)
- The *status filter* (key `status`), featuring the search parameters `inProduction`, `editedByMe`, `editedByOthers`, `notEdited`, `approved`, `published` and `deleted`.
- The *last edited filter* (key `lastEdited`), featuring the search parameter `lastEditedBy`.

Further possible parameters may arise due to plugged in additional filters (c.f. Section 7.11.5, “Adding Search Filters” [155]) where each of them makes up its own part of search parameters. In the source code example below, a default search folder is plugged in that shows all documents under the content repository path folder `/Sites/Media` that were last edited by the user. You can see that the array item is composed of two of the three parts listed above and has been given a name.

Example 7.33. Adding a custom search folder

```
<editor:studioPlugin>
  <ui:rules>
    <editor:favoritesToolbarUserItemsArea>
      <plugins>
        <ui:addArrayItemsPlugin arrayProperty=
              "{favoritesToolbarUserItemsArea.DEFAULT_ITEMS}"
          items='{[
```

```
            {_main:{contentType: "Document_",
                    folder: {path: "/Sites/Media"},
                    mode: "search",
                    view: "list",
                    limit: 50},
             lastEdited: {lastEditedBy: "me"},
             _name: "Last edited"}
          ]}'/>
      </plugins>
    </editor:favoritesToolbarUserItemsArea>
  </ui:rules>
</editor:studioPlugin>
```

If in doubt about the actual format for a default search folder entry, you can always customize a search manually in *CoreMedia Studio*, save it and have a look at the user's preferences where they get saved.

7.9.3 Adding a Button with a Custom Action

The previous sections described how the predefined actions are wrapped in buttons and added to a toolbar. However, sometimes it is necessary to develop a custom action, for example to open a special window or to start a wizard. In Section 5.1.4, “Actions” [31] you will find a more detailed explanation of actions, but the recipe shown here should be enough in many cases.

All actions inherit from `ext.Action`. For example, an action `mypackage.MyAction` might look like this:

Example 7.34. Creating a custom action

```
package mypackage {
import ext.Action;
import mypackage.config.myAction;

public class MyAction extends Action {
  private var foo:String;

  /**
   * @param config
   */
  public function MyAction(config:myAction = null) {
    super(Ext.apply({handler: doAction}, config));
    foo = config.foo;
    ...
  }

  private function doAction():void {
    ...
  }
  ...
}
}
```

It uses the complementing `myAction` configuration class:

Example 7.35. Creating a custom action configuration class

```
package mypackage.config {
import ext.config.action;

[ExtConfig(target="mypackage.MyAction")]
public class myAction extends action {
  private var foo:String;

  public function myAction(config:Object = null) {
    super(config || {});
  }

  /**
   * the foo
   */
  public native function get foo():String;
  /**
   * @private
   */
  public native function set foo(value:String):void;
  ...
}
}
```

You can access this class from ActionScript and (more commonly) from EXML. Like always in EXML, the name of the configuration class determines the element name and its package is used as the namespace URI suffix:

Example 7.36. Using a custom action

```
...
<button>
  <baseAction>
    <mp:myAction text="do something"
                 foo="bar"/>
  </baseAction>
</button>
...
```

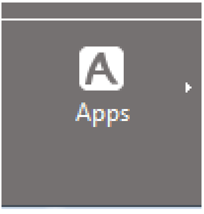
For example, such a button with a base action might by added to the Favorites toolbar or the Actions toolbar as shown in the previous sections. The previous fragment assumes that you have defined the `mp` namespace so that it references the module containing `mypackage` (`exml:mypackage.config`).

Note that you can use all configuration parameters inherited from `ext.action`, here `text`.

Contrary to pure Ext JS, EXML supports configuring a component with properties and an action at the same time by merging the component configuration and the action configuration. If both the component and the action declare a configuration property, the component configuration property value is used.

7.9.4 Adding a Button to the Apps Menu

The Apps Menu as shown in the figure below is used to group buttons for custom actions in one place. It is located in the bottom of the Favorite toolbar of *Studio*.



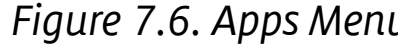


Figure 7.6. Apps Menu

You can add your custom button to the Apps Menu by using the `addItemsPlugin` as shown below:

```
<editor:extensionsMenu>
  <plugins>
    <ui:addItemsPlugin>
      <ui:items>
        ...
      </ui:items>
    </ui:addItemsPlugin>
  </plugins>
</editor:extensionsMenu>
```

7.9.5 Adding Disapprove Buttons

You can revoke the status of the approved document using the disapprove action. The disapprove action can be enabled in *CoreMedia Studio* so that the disapprove action is part of the actions toolbar, the collection repository context menu and the collection search context menu.

You enable the disapprove action by using the plugin `enableDisapprovePlugin`. For example by inserting the following code snippet inside `editor:configuration` in your Studio plugin EXML.

Example 7.37. Adding disapprove action using enableDisapprovePlugin

```
<editor:configuration>
  <editor:enableDisapprovePlugin/>
</editor:configuration>
```

7.10 Inheritance of Property Values

The *CAE* sometimes renders fallbacks if a content property is not set, for example, by using values of other properties instead. This is similar inheritance of a default value for an empty property. To visualize this in *Studio*, you may use content of a property editor from another property editor as the default empty text.

This is currently possible for a few of property fields. One is the `StringPropertyField` and the other one is the `TextAreaPropertyField`. While the `StringPropertyField` may inherit its content from another `StringPropertyField`, the `TextAreaPropertyField` may inherit its content from a `StringPropertyField` or a `RichTextPropertyField`.

In order to use this visualization, you may use the `StringPropertyFieldDelegationPlugin` or the `TextAreaPropertyFieldDelegationPlugin` attached to the property field that should inherit the value.

Example 7.38. Configuring Property Inheritance

```
<documentForm>
  ...
  <!-- inherit its content from the title property-->
  <editor:stringPropertyField propertyName="teaserTitle">
    <plugins mode="append">
      <editor:stringPropertyFieldDelegatePlugin
delegatePropertyName="title"/>
    </plugins>
  </editor:stringPropertyField>
  <!-- inherit its content from the detailText property-->
  <editor:textAreaPropertyField propertyName="teaserText">
    <plugins mode="append">
      <editor:textAreaPropertyFieldDelegatePlugin
delegatePropertyName="detailText"/>
    </plugins>
  </editor:textAreaPropertyField>
  ...
</documentForm
```

7.11 Customizing the Library Window

You can configure the library window in the following ways:

- by defining the columns that are displayed in the list view in the repository mode;
- by defining additional fields for the columns that should be displayed in the list views;
- by defining the columns that are displayed in the list view in the search mode and configuring the columns so that the results in the search mode can be sorted;
- by defining the blob properties that are displayed in the thumbnail view for different document types;
- by adding custom filters for the search mode of the library window.
- by making columns sortable and provide a detailed configuration how to sort.

If you are interested in opening the library from a toolbar button, see Section 7.9, “Customizing Central Toolbars” [145].

7.11.1 Defining List View Columns in Repository Mode

The list view of the library window is implemented using an Ext JS grid panel. A grid panel aggregates columns that refer to fields of an underlying store. For adding a new column, you usually have to add both a column definition and a field definition.

Although the editor context in the form of the interface `IEditorContext` allows a direct configuration in the form of the methods `addListViewDataField` and `setRepositoryListViewColumns`, the recommended way of defining columns uses the `configureListViewPlugin` in an EXML file containing Studio plugin. *CoreMedia Blueprint* defines custom columns of the repository mode in the file `LibraryStudioPlugin.exml`:

Example 7.39. Defining list view columns in the repository mode

```
<ui:rules>
...
  <editor:startup>
    <plugins>
      <editor:configureListViewPlugin>
        ...
        <editor:repositoryListViewColumns>
          <editor:listViewTypeIconColumn width="75"
showTypeName="true"/>
          <editor:listViewNameColumn sortable="true"/>
          <editor:listViewStatusColumn width="46"/>
          <editor:listViewCreationDateColumn width="120"
sortable="true"/>
          <editor:freshnessColumn sortable="true" hidden="true"/>
        </editor:repositoryListViewColumns>
```

```
      ...
      </editor:configureListViewPlugin>
    </plugins>
  </editor:startup>
...
</ui:rules>
```

The property `repositoryListViewColumns` lists **all** columns that should be displayed (not just the ones you want to add to the default) in the repository mode. Some columns in this example use predefined components from the Editor SDK, whereas some special columns use just a configured Ext JS standard grid column.

The `listViewTypeColumn`, `listViewNameColumn`, `listViewCreationDateColumn`, and `freshnessColumn` columns represent the standard columns that would be present without additional configuration (id and width of the column has to be defined if necessary), displaying a document's type, name, date of creation, and modification date, respectively. The `listViewStatusColumn` component represents an additional column that displays a document's lifecycle status (in production, approved, ...) and checked-out state. These columns can be made sortable by setting the attribute `sortable` to `true`. To enable sorting for other columns have a look at Section 7.11.6, “Make Columns Sortable in Search and Repository View” [158].

7.11.2 Defining Additional Data Fields for List Views

If you need additional fields in the underlying store, you can add fields using the `listViewDataFields` property of the `configureListViewPlugin`. The standard columns do not need an explicit field configuration. But if, for example, you may want to display the name of the user who created a content, the implementation would look like this:

Example 7.40. Defining list view fields

```
<editor:configureListViewPlugin>
  <editor:listViewDataFields>
    ...
    <datafield name="creator"
      mapping="creator.name"/>
  </editor:listViewDataFields>
  <editor:repositoryListViewColumns>
    ...
    <gridcolumn id="creator"
      header="Creator"
      sortable="false"
      dataIndex="creator"/>
  </editor:repositoryListViewColumns>
</editor:configureListViewPlugin>
```

In this case, an Ext JS `gridcolumn` is used for display, setting the column's attributes as needed. The definition of the field is slightly complex, because the property `name` of the property `creator` of each content in the search result should be accessed. To this end, a non-trivial `mapping` property will be added, but the `name`

attribute of the data field and the `dataIndex` attribute of the column will be kept simple and in sync. If the mapping property were identical to the name property of the field, it could have been omitted.

7.11.3 Defining List View Columns in Search Mode

The columns in the search mode are similarly configured but instead the property `searchListViewColumns` is used to list all columns of the search list. *CoreMedia Blueprint* defines custom columns of the search mode again in the file `LibraryStudioPlugin.exml`:

Example 7.41. Defining list view columns in the search mode

```
<ui:rules>
...
  <editor:startup>
    <plugins>
      <editor:configureListViewPlugin>
        <editor:listViewDataFields>
          <datafield name="site" mapping="parent.path"
convert="{getImportantPathInfo}"/>
          ...
        </editor:listViewDataFields>
        ...
        <editor:searchListViewColumns>
          <editor:listViewTypeIconColumn width="75"
showTypeName="true"/>
          <editor:listViewNameColumn sortable="true"/>
          <gridcolumn id="site"

header="{LibraryStudioPlugin_properties.INSTANCE.ListView_column_site_header}"

                      sortable="false"
                      menuDisabled="true"
                      dataIndex="site"/>
          <editor:listViewStatusColumn width="46"/>
          <editor:listViewCreationDateColumn sortable="true"
width="120"/>
          <editor:freshnessColumn  sortable="true" hidden="true"/>
        </editor:searchListViewColumns>
      </editor:configureListViewPlugin>
    </plugins>
  </editor:startup>
...
</ui:rules>
```

First you can see in the example above that an additional field `site` is defined and used for the `site` column.

Second the name, creation date and freshness columns are configured to be sortable so that the editor can now sort the search results by the name, creation date and freshness.

> If you define columns by your own, make sure that the `freshnessColumn` is configured because this column will be used as the default sort column. Otherwise the Studio user will get this error message on the console:

```
Invalid Saved Search Folder: Can not sort by sortfield
freshness. It will be sorted by 'Last Modified' instead.
```

Third the freshness column is sortable but hidden. It means that the column will not be shown in the search list by default but the freshness as a sort criterion (which is the default sort criterion) will be available and shown in a drop down box for sort criteria in the search toolbar. The column can also be unhidden by the user via the grid header menu.

The `listViewNameColumn`, `listViewCreationDateColumn` and `freshnessColumn` columns are standard columns that can be configured to be sortable without additional configuration. To enable sorting for other columns have a look at Section 7.11.6, “Make Columns Sortable in Search and Repository View” [158].

7.11.4 Configuring the Thumbnail View

The thumbnail view of the library window can show a preview image of documents with a blob property holding the image data. If you want to do so, you need to register your document type and configure the name of the blob property you want the thumbnail preview to be generated from. From ActionScript, use the `registerImageDocumentType` method of the `IEditorContext`. You can also use the standard plug `configureDocumentTypes`, setting the `imageProperty` for a given set of document types.

This is how the mapping is registered in the editor plugin of *CoreMedia Blueprint*:

Example 7.42. Configuring the thumbnail view

```
<editor:configureDocumentTypes names="CMPicture,CMImage"
                               imageProperty="data"/>
```

The configured property applies to exactly the given document types, only. It is not inherited by subtypes.

7.11.5 Adding Search Filters

The search mode of the library offers a filter panel at the left side of the window in which you can, by default, select the editing state of documents to be included in the search result. Depending on your editorial needs, you can add custom search filters that further restrict the search result. For example, you might want to search only for recently edited documents or for documents in a particular language.

Inheriting from FilterFieldset

For defining a custom filter, you can inherit from the class `FilterFieldset`. This class implements the interface `SearchFilter` and provides the framework for implementing a custom filter easily. Your implementation uses an ActionScript base class and an EXML user interface definition inheriting from that class. Both

classes communicate by means of a model bean provided by the framework's base class through the method `getStateBean()`.

See section Section 7.7, “Storing Preferences” [142] for details of how the value stored in the state bean is persisted and for the limits on the allowed property values.

In your base class, you need to override two methods. The method `buildQuery()` can use the current state stored in the model bean to assemble a Solr query string. Query strings from individual filters will be combined using the `AND` operator. By returning an empty string or `null`, you can indicate that the filter should not currently impose any restrictions on the search result. The following example shows how a property `foo` is retrieved and how a query is built from it.

```
public class FooFilterFieldsetBase extends FilterFieldset {
  override public function buildQuery():String {
    var foo:Number = getStateBean().get('foo');
    if (foo === 0) {
      return null;
    } else {
      return "foo:" + foo + " OR foo:-1";
    }
  }
  ...
```

The method `getDefaultState()` returns an object mapping all properties of the state bean to their defaults. It is used for initialization, for determining whether the current state of your UI represents the filter's default state, and for manually resetting the filter. In the above example, the respective filter's default state is represented by the special value "0", and consequently, you must use "0" as the filter's default value:

```
  ...
  override public function getDefaultState():Object {
    return { foo:0 };
  }
}
```

Now you can create the EXML definition of the actual UI for the new filter. Because the item ID of the filter component is used when identifying the filter later on, it often makes sense to specify the item ID directly in the EXML file. The basic structure is shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<exml:component xmlns:exml="http://www.jangaroo.net/exml/0.8"
                xmlns="exml:ext.config"
                xmlns:editor=
                  "exml:com.coremedia.cms.editor.sdk.config"
                baseClass="FooFilterFieldsetBase">
  <editor:filterFieldset
          itemId="fooFilter"
          title="Foo">
    <items>
      ...
    </items>
```

```
  </editor:filterFieldset>
</exml:component>
```

Synchronizing UI with model state

To synchronize your UI component(s) with the model state stored in the bean returned by `getStateBean()`, you might want to use the various existing bind plugins. The example below shows a typical configuration of the `bindPropertyPlugin` that would work in a filter fieldset for a text field or a combo box.

```
  ...
  <plugins>
    <ui:immediateChangeEventsPlugin/>
    <ui:bindPropertyPlugin bidirectional="true">
      <ui:bindTo>
        <ui:valueExpression expression="foo"
                            context="{getStateBean()}"/>
      </ui:bindTo>
    </ui:bindPropertyPlugin>
  </plugins>
  ...
```

The use of the `immediateChangeEventsPlugin` will ensure that changes of the UI component are propagated to the model quickly and not just after the field looses focus. If you have more complex requirements, a bind plugin might be insufficient, so that you have to synchronize the model and the view using custom ActionScript code.

If you attach the style class `collection-status-filters` to the outermost container in your field set, you might find it easier to achieve a visual style that matches that of the predefined filters.

Use the `addItemsPlugin` to add your custom filter to the Studio Library filter section. The component to configure is the `SearchFilters` class.

```
<editor:studioPlugin>
  <ui:rules>
    <editor:searchFilters>
      <plugins>
        <ui:addItemsPlugin>
          <ui:items>
            <foo:FooFilterFieldset/>
          </ui:items>
        </ui:addItemsPlugin>
      </plugins>
    </editor:searchFilters>
  </ui:rules>
</editor:studioPlugin>
```

Opening the Library in certain filter state

You can also open the library in a certain filter state, for example from a button in the favorites toolbar. To that end, the `showCollectionViewAction` provides a property `filters` that can take `SearchFilterState` objects, which are configured using the `<searchFilterState>` element in EXML. So that the action can configure the correct filter, the `filterId` attribute must be given, matching the item id of the configured filter fieldset. Additionally, any number of additional attributes may be configured for the `<searchFilterState>` element using the

exml:untyped XML namespace. The names and values of the attributes are exactly the property names and values of the state bean used by the filter set.

```
<editor:studioPlugin>
  <ui:rules>
    <editor:favoritesToolbar>
      <plugins>
        <ui:addItemsPlugin>
          <ui:items>
            <button ...>
              <baseAction>
                <editor:showCollectionViewAction
                        contentType="CMArticle">
                  <editor:filters>
                    <editor:searchFilterState
                            xmlns:untyped="exml:untyped"
                            filterId="fooFilter"
                            untyped:foo="{1}"/>
                  </editor:filters>
                </editor:showCollectionViewAction>
              </baseAction>
            </button>
          </ui:items>
        </ui:addItemsPlugin>
      </plugins>
    </editor:favoritesToolbar>
  </ui:rules>
</editor:studioPlugin>
```

If you prefer a type-safe configuration, you can also define an EXML subclass of SearchFilterState that declares the parameters explicitly.

7.11.6 Make Columns Sortable in Search and Repository View

Sorting can be enabled for custom columns by setting two mandatory attributes in the gridcolumn definition. The attribute sortable has to be set to true to enable sorting. The attribute sortField has to specify the Solr index column that should be used for sorting.

Example 7.43. Two additional attributes for sorting.

```
<gridcolumn id="creator"
            header="Creator"
            sortable="true"
            dataIndex="creator"
            u:sortField="creator"/>
```

For extended configuration purposes there are two optional attributes. The attribute extendOrderBy enables sorting by more than one column. The value of the attribute is a function which returns an array with additional sort criteria. The function will get two parameters. The first parameter is the primary sort field, the second parameter is the primary sort direction. The following example does not only sort by creator but also by name and creation date. The value for the function parameter field is "creator", the value of the parameter direction depends on the user's choice and can be "asc" or "desc".

```
<gridcolumn id="creator"
            header="Creator"
            sortable="true"
            dataIndex="creator"
            u:sortField="creator"
            u:extendOrderBy="{
              function(field:String, direction:String):Array {
                var orderBys:Array = [];
                orderBys.push('name ' + direction);
                orderBys.push('creationdate ' + direction);
                return orderBys;
              }
            }"/>
```

Example 7.44. Optional `extendOrderBy` Attribute for sort by more than one column.

The optional attribute `sortDirection` enables you to restrict the sort direction to only one direction. This is useful if sorting does only make sense in one direction. For example a user is usually not interested in the less relevant search result. So you want to disable sorting for relevance ascending. Possible attribute values are "asc" or "desc" where the value is the enabled sort direction.

```
<gridcolumn header="Relevance"
              id="score"
              dataIndex="score"
              sortable="true"
              u:sortField="score"
              u:sortDirection="desc"/>
```

Example 7.45. Optional `sortDirection` Attribute to enable only one sort direction.

You can make even hidden grid columns sortable. Hidden columns are not shown in the grid but users can select them from the sort drop down field. This is useful if columns do not have meaningful values (again relevance for example) or if you just do not want to blow up the grid too much. Hidden columns that do not have their `hideable` config option set to `false` can also be unhidden by the user using the grid header menu.

At last you can define one default sort column for each list in the collection view. The default sort column will be used when the user has not specified a sort criteria. To configure add the attribute `defaultSortColumn` with value `true`. For more fine grained configuration the attribute `defaultSortDirection` can be set to `asc` or `desc` to sort ascending or descending by default.

```
<gridcolumn id="creator"
            header="Creator"
            sortable="true"
            dataIndex="creator"
            u:defaultSortColumn="true"
            u:defaultSortDirection="desc"
            />
```

Example 7.46. `defaultSortColumn` Attribute to configure one column as the default for sorting.

7.12 Work Area Tabs

CoreMedia Studio organizes working items in a so called work area. The work area is a tab panel with the tabs containing currently opened working items. *CoreMedia Studio* restores open tabs (and their content) after successful relogin or reload of the website. The tabs usually contain CoreMedia-specific content but you can integrate your own customized tab into the work area. This section shows how it can be done using an example code. The example introduces a browse tab which consists of a URL trigger field and an iFrame in which the content of the URL is displayed.

7.12.1 Configuring a Work Area Tab

First you have to configure the tab which should be displayed in the work area. This must be an `ext.Panel` or any extended one. CoreMedia recommends that you configure your tab as a separate component in EXML. The rationale for this will be described below. In the example there are two such components: `BrowseTab.exml` and `CoreMediaTab.exml` (where the latter one uses the first one). Both have a configuration parameter `url` which is the key to persisting tab state across sessions and website reloads as explained below in Section 7.12.4, “Storing the State of a Work Area Tab” [161].

7.12.2 Configure an Action to Open a Work Area Tab

In most cases you will use an action to open your own tab. In the example, a button is plugged into the Favorites toolbar. Clicking the button triggers an `openTabAction` to open the browse tab.

Example 7.47. Adding a button to open a tab

```
...
<editor:favoritesToolbar>
  <plugins>
    <ui:addItemsPlugin>
      <ui:items>
      ...
         <!-- Add a button that opens a browse tab -->
         <button itemId="browseTab">
           <baseAction>
             <editor:openTabAction text="...">
               <editor:tab>
                 <example:browseTab/>
               </editor:tab>
             </editor:openTabAction>
           </baseAction>
         </button>
      </ui:items>
      ...
    </ui:addItemsPlugin>
  </plugins>
</editor:favoritesToolbar>
...
```

The browseTab from above is configured as the tab configuration parameter of openTabAction. A new browse tab is then opened every time when clicking the button. In addition, all open browse tabs will be reopened in the work area after the reload of *CoreMedia Studio*. For that *CoreMedia Studio* stores the xtypes of the open tabs as user preference when opening, closing or selecting tabs. When loading the work area instances of the xtypes are generated and added to the work area. This is basically why you should configure each tab in a separate EXML. Nevertheless, you will see below in Section 7.12.4, "Storing the State of a Work Area Tab" [161] how you can save other state of the tab than the xtype in the user preference.

7.12.3 Configure a Singleton Work Area Tab

The previously shown openTabAction has an additional Boolean configuration parameter singleton. In the example a button that opens a coreMediaTab is added, which is a browse tab with the fix URL of the CoreMedia homepage:

Example 7.48. Adding a button to open a browser tab

```
...
<editor:favoritesToolbar>
  <plugins>
    <ui:addItemsPlugin>
      <ui:items>
        <!-- Add a button that opens the CoreMedia homepage browse
 tab as singleton -->
        <button itemId="coremediaTab">
          <baseAction>
            <editor:openTabAction singleton="true" text="...">
               <editor:tab>
                 <example:coreMediaTab/>
               </editor:tab>
             </editor:openTabAction>
          </baseAction>
        </button>
      </ui:items>
      ...
    </ui:addItemsPlugin>
  </plugins>
</editor:favoritesToolbar>
...
```

In the work area there will be no more than one opened coreMediaTab: When clicking the button the already opened coreMediaTab will be active instead of opening a new one.

7.12.4 Storing the State of a Work Area Tab

You probably want to persist the state of your tabs across sessions and website reloads. As described above, the xtype of all open tabs is stored automatically which allows you to create the correct tab instances when reloading. However, this does not help to persist the content of the tabs. You have to take care of persisting tab state yourself. For example, when the user sets the URL of the browse tab in the example the URL will be restored after reload. Such internal state of the

tab can be stored implementing the interface `StateHolder` as `BrowseTabBase` of the example does:

Example 7.49. Base class for browser tab

```
...
public class BrowseTabBase extends Panel implements StateHolder{
...
  public function getStateValueExpression():ValueExpression {
    if (!stateValueExpression) {
      stateValueExpression =
      ValueExpressionFactory.createFromValue({url: url});
    }
    return stateValueExpression;
  }
...
}
}
```

To store the states of the open tabs *CoreMedia Studio* uses `getStateValueExpression` of each tab which implements the interface. See section Section 7.7, “Storing Preferences” [142] for details of how the state is persisted and for the limits on the allowed state structures. You must make sure that proper state is delivered via the state value expression. In `BrowseTabBase` this is achieved in the following way:

```
...
  internal function reloadHandler():void {
    var url:String = getTrigger().getValue();
    getBrowseFrame().setUrl(url);
    if (url) {
      setTitle(url);
    }
    //store the url as state in the user preference
    getStateValueExpression().setValue({url: url});
  }
}
}
```

The `reloadHandler` is invoked when the user clicks on the trigger button. The value of the trigger becomes the URL of the iFrame of the tab. Finally, the state value is set to `{url: url}`: As described above, `url` is a configuration parameter of `browseTab` and consequently, `{url:url}` is a configuration object with the parameter `url` with the trigger value. This configuration object will be copied to the configuration object of `browseTab` when restoring it. So `browseTab`'s configuration parameter `url` is then set to the stored value.

7.12.5 Customizing the Start up Behavior

After successful login, *Studio* restores the tabs of the last session. This default behavior can be disabled by calling the `setDefaultTabStateManagerEnabled(enable)` method of `IEditorContext` class.

When you set this value to `false`, *Studio* will start with a blank working area (that is, no documents or other tabs are open). This might be handy if you want to customize the startup behavior. When, for example, you want to open all documents

that a given search query finds on startup, you can do that with code like the following:

```
package com.coremedia.ui.examples.openCheckedOutDocuments {
import com.coremedia.cap.common.session;
import com.coremedia.cap.content.search.SearchParameters;
import com.coremedia.cap.user.User;
import com.coremedia.cms.editor.EditorErrors_properties;
import com.coremedia.cms.editor.sdk.editorContext;
import com.coremedia.cms.editor.sdk.util.MessageBoxUtil;
import com.coremedia.ui.data.Bean;
import com.coremedia.ui.data.PropertyChangeEvent;
import
com.coremedia.ui.examples.openCheckedOutDocuments.config.openCheckedOutDocumentsPlugin;

import ext.Component;
import ext.Container;
import ext.Plugin;
import ext.util.StringUtil;

public class OpenCheckedOutDocumentsPlugin implements Plugin{

  private const MAX_OPEN_TABS:int = 10;

  public function
OpenCheckedOutDocumentsPlugin(config:openCheckedOutDocumentsPlugin
 = null) {
  }

  public function init(component:Component):void {
    //get the top level container
    var mainView:Container =
component.findParentBy(function(container:Container){
      return !container.ownerCt;
    });

    mainView.addListener('afterrender', openDocuments, null, {
      single: true
    });

  }

  private function openDocuments():void {
    // Perform query to determine documents checked out by me.
   var searchParameters:SearchParameters = createSearchParameters();

    var searchResult:Bean =
session.getConnection().getContentRepository().getSearchService().search(searchParameters);

    // When the query result is loaded ...
    searchResult.addPropertyChangeListener(SearchParameters.HITS,
function openInTabs(event:PropertyChangeEvent):void {
      // ... open all documents in tabs.
      var searchResult:Array = event.newValue;
      if (searchResult && searchResult.length > 0) {

editorContext.getContentTabManager().openDocuments(searchResult.slice(0,
 MAX_OPEN_TABS));
        if (searchResult.length > MAX_OPEN_TABS) {

MessageBoxUtil.showInfo(EditorErrors_properties.INSTANCE.editorStart_tooManyDocuments_title,


EditorErrors_properties.INSTANCE.editorStart_tooManyDocuments_message);

        }
      }
```

```
    });
    searchResult.get(SearchParameters.HITS);
  }

  private function createSearchParameters():SearchParameters {
    var searchParameters:SearchParameters = new SearchParameters();

    searchParameters.filterQuery = [ getQueryFilterString()];

    //searchParameters.contentType = ['Document_'];
    searchParameters.orderBy = ['freshness asc'];

    return searchParameters;
  }

  private function getQueryFilterString():String {
    var filterQueries:Array = [];

    // retrieve user URI for parametrized filter expressions:
    var user:User = session.getUser();
    var userUri:String = "<" + user.getUriPath() + ">";

    // filter documents checked out by me
    filterQueries.push("ischeckedout:true");
    filterQueries.push(StringUtil.format("editor:{0}", userUri));

    return filterQueries.join(" AND ");
  }
}
}
```

7.12.6 Customizing the Work Area Tab Context Menu

The context menu for work area tabs comes with several predefined actions like close operations and options for checking in or reverting contents. In addition, the `WorkAreaTabContextMenu` is an extension point for plugging in your own actions.

The `WorkAreaTabContextMenu` is a Studio IOC context provider (see Section 7.8, "Customizing Studio using Component IoC" [143]). However, instead of accessing the provided context variables directly via *Studio* IOC it is recommended to implement your custom actions as subclasses of `AbstractTabContextMenuAction` or `AbstractTabContextMenuContentAction`. In both cases, the context-clicked tab and tab panel can be accessed via the methods `getContextClickedTab():Panel` and `getContextClickedTabPanel():TabPanel` respectively. In addition, `AbstractTabContextMenuContentAction` provides the methods `getContextClickedContent():Content` and `getContextClickedContents():Array` for obtaining the content of the context-clicked tab and all contents of work area tabs respectively. Note that only `Premular` tabs have content other than `undefined`.

Using these methods, subclasses should override the method `checkDisabled():Boolean` to decide whether the action should be disabled. In addition, these methods should suffice to provide enough information to implement the action's behavior.

For example, the following two code samples show how to add an action for checking in all contents of opened work area tabs.

```
<ui:rules>
  ...
  <editor:workAreaTabContextMenu>
    <plugins>
      <ui:addItemsPlugin>
        <ui:items>
          <menuseparator/>
          <menuitem>
            <baseAction>
              <custom:checkInAllContentTabsAction
                text="Check in all contents"/>
            </baseAction>
          </menuitem>
        </ui:items>
      </ui:addItemsPlugin>
    </plugins>
  </editor:workAreaTabContextMenu>
...
</ui:rules>
```

```
public class CheckInAllContentTabsAction extends
  AbstractTabContextMenuContentAction {
  ...

  private function handler():void {
    getContextClickedContents().
      forEach(function (content:Content):void {
        if (content.isCheckedOutByCurrentSession()) {
          content.checkIn();
        }
      }
    }
  }

  override protected function checkDisable():Boolean {
    var atLeastOneContentTabInEditMode:Boolean = false;
    getContextClickedContents().
      forEach(function (content:Content):void {
        if (content.isCheckedOutByCurrentSession()) {
          atLeastOneContentTabInEditMode = true;
        }
    });
    return !atLeastOneContentTabInEditMode;
  }
}
```

7.13 Dashboard

CoreMedia Studio provides a dashboard as a special tab type. On the dashboard, users may freely arrange so-called *widgets*, which display data that the user should be aware of. While your users may configure the dashboard according to their particular needs, it is your task as a developer to determine which widget types are available to them and to configure a suitable default dashboard for the first login.

If no default dashboard is configured, the dashboard will not be available at all. When you configure at least one type of widget, the dashboard button appears in the upper right corner of the screen, and users may start working with their own dashboard.

7.13.1 Concepts

Three rows

Studio dashboard widgets are organized in three columns of equal width that span the entire work area. Each widget may fill one or more fixed-height rows, depending on its `rowspan` attribute. Widgets cannot span multiple columns. Users can adjust the height of each individual widget when they adjust their widget configuration.

There may be many fundamentally different widget types for various purposes. Generally, widgets are used to display current information that a user is likely to be interested in, without requiring immediate action. However, there may also be widgets that allow the user to make simple updates or interact with other users. Due to the limited size of a widget, complex interactions are likely moved to a tab or a separate dialog.

Each widget type must provide a user interface that displays the actual information for this widget. Additionally, each widget type *may* opt to provide a user interface to configure a particular instance of the widget type on the user's dashboard. Users can choose a "configuration mode" for each widget, and in this mode, the configuration UI is displayed, which can be used to modify the appearance and functionality of the widget. Multiple widgets of the same type may be shown on the dashboard and each such widget can be in a different configuration state. Note the "configurability" of a widget is optional. For non-configurable widget types, the widget may just show an explanatory text describing its functionality.

State is stored persistently

For each user, the set of widgets, their positions, sizes, and states are stored persistently, allowing you to restore the widgets when the dashboard is closed and reopened. Many widget types provide a corresponding state class that allows you to define the state of the widget when configuring an initial dashboard. Widget state object and widget types are matched with each other by means of a widget type id.

Besides creating the user interfaces, the widget type in the form of an object implementing the `WidgetType` interface is also responsible for providing a type

name, description, icon, default `rowspan`, and for computing a title, possibly depending on the current widget state. Optionally, the widget type may also provide tools to be included in the header bar of the widget. Tools can allow the user to start operations based on the current widget state.

7.13.2 Defining the Dashboard

You can configure the dashboard by selecting which widgets the user may add to the dashboard and by describing the initial widget configuration of the dashboard.

To this end, the dashboard configuration is available through the method `getDashboardConfiguration()` of the `editorContext` object. It provides a list of `WidgetType` objects in the `types` property and a list of `WidgetState` objects in the `widgets` property.

Usually, you will not access the configuration object directly, but rather through the `configureDashboardPlugin`, which also offers a `types` and a `widgets` property and takes care of merging these values into the global configuration at the correct time.

The widget state objects in the property `widgets` determine the widgets to be shown when the user first opens the dashboard. You should therefore select widgets that a typical novice user would find interesting.

Each widget state object must be an instance of the class `WidgetState`, or a subclass thereof. The class `WidgetState` itself defines only the properties `widgetTypeId`, `rowspan`, and `column`, indicating the widget type, the relative height of the widget and the placement of the widget, respectively.

Widget types for all initial widgets have to be provided, but you will typically add more widget types for advanced users. Widget types and widget state objects are matched by their id, which can be specified using the `widgetTypeId` property of the state object. Predefined state objects will typically provide the correct ID automatically.

The following example shows how the `configureDashboardPlugin` is used inside an EXML Studio plugin specification.

Example 7.50. Dashboard Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<exml:class xmlns:exml="http://www.jangaroo.net/exml/0.8"
  xmlns="exml:ext.config"
  xmlns:editor="exml:com.coremedia.cms.editor.sdk.config">

  <editor:studioPlugin>
    <editor:configuration>
      <editor:configureDashboardPlugin>
        <editor:widgets>
          <editor:simpleSearchWidgetState
             contentType="CMArticle"/>
          <editor:simpleSearchWidgetState
             contentType="CMPicture"
```

```
            column="1"/>
        </editor:widgets>

        <editor:types>
          <editor:simpleSearchWidgetType/>
        </editor:types>
      </editor:configureDashboardPlugin>
    </editor:configuration>
  </editor:studioPlugin>
</exml:class>
```

You can see a single widget type being configured, `simpleSearchWidgetType`. In this example, the widget type provides no configuration option itself, but some widget type classes can be customized by configuration.

In the example, there are two widgets using the defined type. By specifying a `simpleSearchWidgetState`, the widget type id is set to match the `simpleSearchWidgetType`. The two widgets start off with a specific state. As a rule, any configuration options that can be provided using a state object should also be configurable when the widget is in edit mode.

For the second widget, a column is specified. Unless a column property is given, each widget is placed in the same column as the previous widget and the first widget is placed in the leftmost column. For the column property use either a numeric column id from 0 to 2 or one of the constants `SAME` or `NEXT` from the class `widgetState`, indicating to stay in the same column or to progress one column to the right. The leftmost column is used as the next column of the rightmost column.

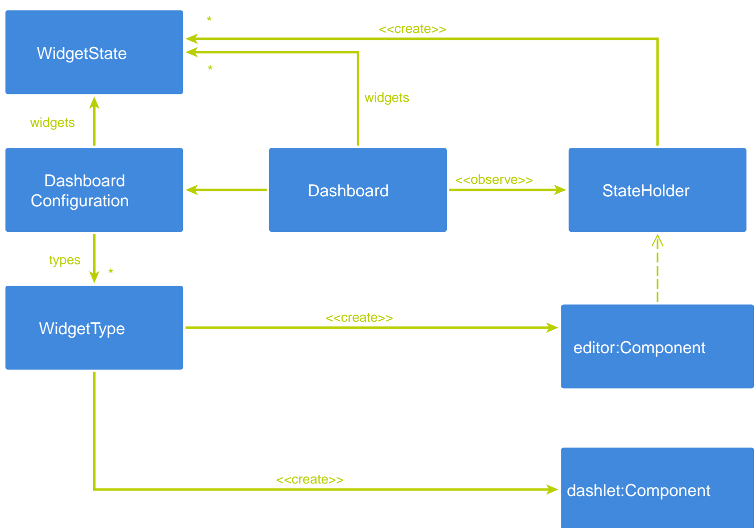


Figure 7.7. Dashboard UML overview

7.13.3 Predefined Widget Types

There are a number of predefined widgets that are immediately usable through simple configuration. All configuration classes of these widgets are located in the package `com.coremedia.cms.editor.sdk.config`. The following table summarizes the existing widgets.

Table 7.6. Predefined widget types

Name	Description
`fixedSearchWidgetType`	Displays the result of exactly one preconfigured search.
`simpleSearchWidgetType`	Displays the result of a search for contents of a configurable type containing a configurable text.

The individual types and their configuration options are subsequently explained in more detail.

Fixed Search Widget

Widget types based on the class `fixedSearchWidgetType` display the result of exactly one preconfigured search. Because this widget type does not offer any editable state, you should provide the search to execute when you define the widget type. In this way, you can define fixed search widget types showing checked-out documents or the most recently edited pages or arbitrary other searches.

For each type, you should at least specify the `name` under which the type can be selected in the dropdown box when adding a new widget. At your option, you may also set a `title` or a `description` to be shown for your type.

Because you can define multiple types, you must also provide different widget type IDs. You can then use a plain `widgetState` element with the chosen type ID and placement attributes to instantiate the widget.

An example configuration of this widget might look like this:

Example 7.51. Fixed Search widget Configuration

```
<editor:configureDashboardPlugin>
  <editor:widgets>
    <editor:widgetState widgetTypeId="editedByOthers"/>
  </editor:widgets>

  <editor:types>
    <editor:fixedSearchWidgetType
            id="editedByOthers"
            name="Edited by others">
      <editor:search>
        <editor:searchState editedByOthers="true"
                            editedByMe="false"
                            notEdited="false"
```

```
                             approved="false"
                             published="false"/>
      </editor:search>
    </editor:fixedSearchWidgetType>
  </editor:types>
</editor:configureDashboardPlugin>
```

Simple Search Widget

A widget of type `simpleSearchWidgetType` displays the result of a search for contents of a configurable type containing a configurable text. By default, the search is limited to the preferred site, if such a site is set. Through the state class `simpleSearchWidgetState`, the dashlet provides the associated configuration options `contentType`, `searchText`, and `preferredSite`.

An example configuration of this widget might look like this:

Example 7.52. Simple Search Widget Configuration

```
<editor:configureDashboardPlugin>
  <editor:widgets>
    <editor:simpleSearchWidgetState contentType="CMPicture"/>
  </editor:widgets>

  <editor:types>
    <editor:simpleSearchWidgetType/>
  </editor:types>
</editor:configureDashboardPlugin>
```

7.13.4 Adding Custom Widget Types

You can define your own widget types and add widgets of this type to the dashboard. This section will guide you through all the necessary steps, covering rather simple widgets as well as more sophisticated ones.

Widget Type and Widget Component

When creating own widgets, you typically start off by creating a custom *widget type*. As described in the previous sections, the dashboard is configured in terms of columns and widget states. Each widget state carries a *widget type id* which associates it with its widget type. In order to get from widget states to the actual widget instances shown on the dashboard, the different widget types are consulted. A widget type is responsible for creating the widget components from their associated widget states.

You could define your own widget type by creating a class from scratch that implements the interface `WidgetType`. However, a convenient default implementation `ComponentBasedwidgetType`, is provided out of the box. For many cases it is sufficient to let your own widget type extend it. In order to do so, you have to define a *widget component* that defines the UI for widgets of your new widget type.

For instance, the predefined `SimpleSearchWidgetType` is simply defined in EXML as follows:

Example 7.53. Simple Search Widget Type

```
<exml:class ... >
  ...
  <editor:componentBasedWidgetType
          name="..."
          description="..."
          iconCls="...">
    <editor:widgetComponent>
      <editor:simpleSearchWidget/>
    </editor:widgetComponent>
  </editor:componentBasedWidgetType>
</exml:class>
```

Besides setting the parameters `name`, `description` and `iconCls`, the widget component `SimpleSearchWidget` is set. The following listing shows a fragment of the `SimpleSearchWidget`:

Example 7.54. Simple Search Widget Component

```
<exml:component ...
                baseClass="...SimpleSearchWidgetBase">
  ...
  <exml:cfg name="searchText" type="String">
    <exml:description>
      ...
    </exml:description>
  </exml:cfg>

  <exml:cfg name="contentType" type="String">
    <exml:description>
      ...
    </exml:description>
  </exml:cfg>

  <exml:constant name="CONTENT_LIST_ITEM_ID" value="contentList"/>

  <container height="100%">
    <items>
      <editor:widgetContentList itemId="{CONTENT_LIST_ITEM_ID}"
                     contentList="{getContentValueExpression()}"/>
      ...
    </items>
  </container>
</exml:component>
```

The component can be configured with the parameters `searchText` and `contentType` in order to show a corresponding search result. Executing the search and obtaining the search results is carried out in the base class `SimpleSearchWidgetBase`. When extending that class, a value expression that references the search result can be obtained via `getContentValueExpression()` and is used by a `WidgetContentList` to display the result.

There is one further important aspect concerning the base class `SimpleSearchWidgetBase`. It implements the `Reloadable` interface. This indicates that a reload button should be placed in the widget header, calling the widget's `reload()`

method for refreshing the widget's contents. In this case, the base class simply triggers a new search.

Configurable and Stateful Widgets

The `WidgetType` interface also features the creation of an editor component for a widget at runtime. Again, if you opt to implement the interface yourself, you have to provide this functionality from scratch. If you choose your type to extend `ComponentBasedWidgetType`, you simply have to add an editor component, just as you did for the widget component. Consequently, the EXML definition for the `SimpleSearchWidgetType` for simple search widgets that are configurable at runtime looks as follows:

Example 7.55. Simple Search widget Type with Editor Component

```
<exml:class ... >
  ...
  <editor:componentBasedWidgetType
          name="..."
          description="..."
          iconCls="...">
    <editor:widgetComponent>
      <editor:simpleSearchWidget/>
    </editor:widgetComponent>
    <editor:editorComponent>
      <editor:simpleSearchWidgetEditor/>
    </editor:editorComponent>
  </editor:componentBasedWidgetType>
</exml:class>
```

Now widgets of this type have their own editor component when a widget on the dashboard is in edit mode.

However, without further wiring, the changes a user makes in edit mode do not carry over to the widget component. For the simple search widget it is expected that the user can choose a search text and content type in edit mode and that the widget shows a corresponding search result in widget mode. To make this happen, `SimpleSearchWidgetEditor` has to implement the `StateHolder` interface. The method `getStateValueExpression()` has to be implemented in a way that the value expression refers to a simple JavaScript object containing the configuration properties to be applied to the widget component. Thus, for the simple search widget, these properties are `searchText` and `contentType`.

See section Section 7.7, “Storing Preferences” [142] for details of how the state values are persisted and for the limits on the allowed objects.

You could just implement the `StateHolder` interface yourself. For convenience, CoreMedia recommends, that you let your editor component extend `StatefulContainer`. This component inherently implements `StateHolder`. It can be configured with a list of property names along with default values and automatically takes care of building a *state model bean* from them. This state model bean is the basis for the evaluation of the value expression that is returned via `getState`

ValueExpression(). Additionally, the bean can be consulted via getModel() from subclasses of StatefulContainer. This can be utilized for binding the model state to the user interface state. The following listing exemplifies this for the case of SimpleSearchWidgetEditor:

Example 7.56. Simple Search widget Editor Component

```
<exml:component ...>
  ...
  <ui:statefulContainer layout="form"
                        properties="searchText,contentType">
    <items>
      <editor:contentTypeSelector fieldLabel="..."
                                  width="auto"
                                  fieldClass="">
        <plugins mode="append">
          <ui:fullWidthPlugin/>
        </plugins>
        <editor:contentTypeValueExpression>
          <ui:valueExpression expression="contentType"
                              context="{getModel()}"/>
        </editor:contentTypeValueExpression>
      </editor:contentTypeSelector>
      <textfield fieldLabel="..."
                 width="auto">
        <plugins>
          <ui:bindPropertyPlugin bidirectional="true">
            <ui:bindTo>
              <ui:valueExpression expression="searchText"
                                  context="{getModel()}"/>
            </ui:bindTo>
          </ui:bindPropertyPlugin>
          <ui:immediateChangeEventsPlugin/>
          <ui:fullWidthPlugin/>
        </plugins>
      </textfield>
    </items>
    <ui:propertyDefaults>
      <exml:object contentType="{ContentTypeNames.DOCUMENT}"/>
    </ui:propertyDefaults>
  </ui:statefulContainer>
</exml:component>
```

This editor component for the simple search widget extends StatefulContainer and is configured to build a state model for the two properties searchText and contentType. For the content type property, a default is set. The editor component offers the user a combo box for selecting a content type and a text field for entering a search text. The user's input is tied to the state model via value expressions that use getModel() (inherited from StatefulContainer) as their context. This results in keeping the state model updated. Implementing the StateHolder interface yourself is not necessary. It is automatically taken care of by StatefulContainer on the basis of the always up-to-date state model.

All in all, this results in the simple search widget editor being stateful. When the user switches between widget mode and edit mode for this widget, the editor will keep its state (search text and content type). The state is only lost if the user selects a different widget type in edit mode.

In some cases, it might be useful to not only have the editor of a widget being stateful, but also the widget itself. This can be realized in the same way shown here for the editor: by implementing the `StateHolder` interface.

Custom Widget State Class

In many cases, it is not necessary to create you own widget state class for your custom widget type. As shown earlier in this chapter, the predefined class `WidgetState` allows you to set the dashboard column, the widget type and the widget's `rowspan`. This is sufficient unless you want to put widgets of your type into the default dashboard and at the same time use a configuration other than the default. However, if you want to do just that, CoreMedia recommends that you create your own widget state class as an extension to `WidgetState`. For the simple search widget, the custom state class `SimpleSearchWidgetState` looks as follows:

Example 7.57. widget State Class for Simple Search widget

```
<exml:class ...>
  ...

  <exml:cfg name="searchText" type="String">
    <exml:description>
      ...
    </exml:description>
  </exml:cfg>

  <exml:cfg name="contentType" type="String">
    <exml:description>
      ...
    </exml:description>
  </exml:cfg>

  <exml:cfg name="preferredSite" type="Boolean">
    <exml:description>
      ...
    </exml:description>
  </exml:cfg>

  <editor:widgetState widgetTypeId="{simpleSearchDashlet.xtype}"/>
</exml:class>
```

This class allows you to launch simple search widgets initially with the configuration properties `searchText` and `contentType` being set. They are set via the dashboard configuration prior to the dashboard's launch instead of being set by the user via the `SimpleSearchWidgetEditor` component at runtime (although this is of course possible afterwards).

The `widgetTypeId` for the `SimpleSearchWidgetState` is set to the `xtype` of `SimpleSearchWidget`. This is because widget types that extend `ComponentBasedWidgetType` by default take the `xtype` of their widget component as their `id`.

7.14 Configuring MIME Types

When a blob is uploaded into a property field, *CoreMedia Studio* selects an appropriate MIME type based on the name of the uploaded file. For the most common file name extensions, a MIME type is already preconfigured. You may add further extensions as needed. If the extension is completely unknown, the MIME type suggested by the uploading browser will be used.

To add custom file name extensions, add a mapping from the file extension to the desired MIME type in the file `WEB-INF/mime.properties` of the Studio web application.

Example 7.58. Configuring MIME types

```
mp2=audio/x-mpeg
```

The given example registers files with the extension `mp2` as MPEG files.

7.15 Server-Side Content Processing

Several operations on content can be implemented on the server side using the Unified API from Java. Especially, you may want to place restrictions on the content that is stored in your repository. This may be achieved by pointing the editors to invalid content, by normalizing content during writes or by inhibiting writes that violate your constraints.

- Section 7.15.1, “Validators” [176] describes how to add validation for values stored in the content repository.
- Section 7.15.2, “Intercepting Write Requests” [180] describes how to modify writes before they are executed.
- Section 7.15.3, “Immediate Validation” [183] describes how to inhibit undesirable writes.
- Section 7.15.4, “Post-processing Write Requests” [184] describes how to take additional action after a write has been completed.

7.15.1 Validators

CoreMedia Studio supports server-side validation based on a project-specific configuration. To this end, validators are configured in the REST service web application. Validators can analyze content and report issues which are available at the client side as described in Section 5.3.3, “Issues” [41]. Validators are implemented in Java and injected into the Spring application context of the web application. See Section 5.6, “Web Application Structure” [60] for an introduction of the server-side architecture.

Predefined validators

CoreMedia Studio offers several predefined validators and a convenient API to implement your own, based on project-specific content validation requirements. The table below gives an overview of the default validators, which reside in the package `com.coremedia.rest.validators` (for details, please consult the API documentation available at the CoreMedia download area).

Table 7.7. Selected predefined validators available in CoreMedia Studio

name	behavior
`EmailValidator`	checks for a valid email address according to RFC822
`ImageMapAreasValidator`	checks for non-empty image and correctly linked areas in an image map. See also Section 7.3.5, “Enabling Image Map Editing” [111]

name	behavior
ListMaxLengthValidator and ListMinLengthValidator	checks for maximum/minimum number of documents linked in a linklist
MaxIntegerValidator and MinIntegerValidator	checks for a maximum/minimum integer value
MaxLengthValidator and MinLengthValidator	checks for a maximum/minimum length of a String
NotEmptyValidator	checks whether a field is empty; works on strings, linklists, and blobs
RegExpValidator	checks whether a given (configurable) regular expression matches against the value given in the property
UniqueListEntriesValidator	checks against duplicate links in a linklist (that is, the same document is linked at least twice in the same linklist)
UriValidator and UrlValidator	checks for valid URIs or URLs, respectively

Implementing Validators

Single-property and multi-property validators

You can implement a validator for a single property or a validator that takes multiple properties into account when computing issues. Single-property validators are generally more reusable across document types and across projects and should cover the vast majority of use cases.

For a single-property validator, you can implement the interface PropertyValidator of the package com.coremedia.rest.validation. The easiest way of doing this is by inheriting from the class ObjectPropertyValidatorBase and implementing the method isValid(Object).

Example 7.59. Implementing a property validator

```
public class MyValidator extends ObjectPropertyValidatorBase {
  @Override
  protected boolean isValid(Object value) {
    return ...;
  }
}
```

If you know that all property values belong to a given Java class, you can inherit from PropertyValidatorBase instead, specifying the value type as the generic type argument of the base class and passing a class object of the value class to the base class's constructor. You can then implement a more specific isValid method that immediately receives an argument of the correct type.

To enable a property validator, you register it in a content type validator that is defined in the Spring application context. The following code snippet shows how the validator is applied to the property `myProperty` of the document type `MyDocumentType`. Here the validator is configured to apply to all subtypes of the given document type, too. By default, the validator would only apply to exactly the given document type.

Example 7.60. Configuring a property validator

```
<bean parent="contentTypeValidator">
  <property name="contentType" value="MyDocumentType"/>
  <property name="validatingSubtypes" value="true"/>
  <property name="validators">
    <list>
      <bean class="MyValidator">
        <property name="property" value="myProperty"/>
      </bean>
    </list>
  </property>
</bean>
```

See the Javadoc of the REST Service API and especially the packages `com.coremedia.rest.validators` and `com.coremedia.rest.cap.validators` for the predefined validators.

For all validators that inherit from `PropertyValidatorBase`, which includes all standard validators, you can set the field `code` in the Spring configuration to an issue code of your choice. If you choose not to do so, the class name of the validator implementation will be used as the issue code. For example, the validator `com.coremedia.rest.validators.RegExpValidator` creates issue with code `RegExpValidator` by default.

To provide multiple validators for a single document type you can either provide multiple beans inheriting from `contentTypeValidator` or, more commonly, multiple validators in the `validators` property of a single content type validator.

If you want to handle multiple properties of a content at once, your validator should inherit from the base class `ContentTypeValidatorBase` The single method to implement is `validate(Content, Issues)`, which receives the content to analyze as its first argument and an `Issues` object as its second argument. Whenever a problem is detected, you can call the method `addIssue(severity, property, code, ...)` of the issues object to register a new issue.

Example 7.61. Implementing a content validator

```
public class MyContentValidator extends ContentTypeValidatorBase {
  @Override
  public void validate(Content content, Issues issues) {
    if (...) {
      issues.addIssue(Severity.ERROR, "myProperty", "myCode");
    }
  }
}
```

By inheriting from `ContentTypeValidatorBase` you can easily specify the name of the content type to which is validator is applied when configuring the validator into the Spring application context.

Example 7.62. Configuring a content validator

```
<bean class="MyContentValidator">
  <property name="connection" ref="connection"/>
  <property name="contentType" value="MyDocumentType"/>
</bean>
```

You can also implement the interface `CapTypeValidator` directly, if you do not want to make use of the convenience methods of `ContentTypeValidatorBase`. Finally, by implementing `com.coremedia.rest.validation.Validator<Content>` you could create validators that are not even bound to a document type. This should only be necessary in very rare cases.

Defining and Localizing Validator Messages

CoreMedia Studio ships with predefined validator messages for the built-in validators. The messages are defined in property files, following the idiom described in Section 5.7, "Localization" [61]. However, you might still want to add your own localized messages if you add custom validators or if you want to provide more specific message for individual properties.

To this end, you should start by adding a new set of property files containing your localized messages. Make sure to add the base property file and an additional property file for each non-default language.

Augment the central validator property file with your own properties. The central property file is `com.coremedia.cms.editor.sdk.validation.Validators`, so that it can be updated as follows:

Example 7.63. Configuring validator messages

```
  ResourceBundle.overrideProperties(Validators_properties,
    MyValidators_properties);
```

Now you can add localized message to the base property file and optionally to every language variant, using an appropriate translation.

There are three kinds of keys using the following schemes:

1. `Validator_<IssueCode>_text` is used as the generic message for the respective issue code.
2. `PropertyValidator_<PropertyName>_<IssueCode>_text` is used when the issue code appears for a property of a specific name.
3. `ContentValidator_<ContentType>_<PropertyName>_<IssueCode>_text` is used when the issue code appears for a property of a specific

name for a document with the given content type or any subtypes thereof. A localized message for a more specific content type takes precedence.

Generally, more specific settings take precedence over more general settings. For example `ContentValidator_*` keys take precedence over `Validator_*` keys, if applicable.

Each localized message may contain the substitution tokens `{0}`, `{1}`, and so on. Before being displayed, these tokens are replaced by the corresponding issue argument (counting from 0).

Tying Document Validation to Editor Actions

It is possible to tie the validation of a document to editor actions via the `validateBefore` property defined in `studio.properties`. This property is to configure *Studio* to prevent certain activity on content items when they still contain errors. More specifically, you can specify that either checking in content or approving (and thus publishing) content will be not allowed in the presence of content errors. Setting the value of the `validateBefore` property to "CHECKIN" entails the check of both `Checkin` and Approve actions. Currently, the only supported options are "CHECKIN" or "APPROVE". Leaving the property value empty means that no such checks are imposed, and editors are allowed to check in, approve and publish even when content errors are detected.

7.15.2 Intercepting Write Requests

Write requests that have been issued by the client can be intercepted by custom procedures in the server. To this end, write interceptor objects can be configured in the Spring application context of the Studio REST service. Typical use cases include:

- Setting initial property values right during content creation, ensuring that a completely empty content cannot be encountered even temporarily.
- Replacing the value to be written, for example, to automatically scale down an image to predefined maximum dimensions.
- Computing derived values, for example, to extract the dimensions (or other metadata) of an uploaded image and storing them in separate properties.

> Replacing values is not normally useful for text properties, because text values are saved continuously as the user enters data, and a write interceptor might not be able to operate appropriately during the first saves. For blobs or link lists, the impact on the user experience is typically less of a problem. In any case, when using interceptors, you need to make sure that the user experience is not impacted negatively.

Developing Write Interceptors

In order to process write requests as described above, create a class implementing the interface `ContentWriteInterceptor`. Alternatively, your class can also inherit from `ContentWriteInterceptorBase`, which already defines methods to configure the content type to which the write interceptor applies, and the priority at which the interceptor runs compared to other applicable interceptors.

This leaves only the method `intercept(ContentWriteRequest)` to be implemented in custom code. The argument of the `intercept` method provides access to all information needed for processing the current request, which is either an update request or a create request.

Get values from write request

The method `getProperties()` of the `WriteRequest` object returns a mutable map from property names to values that represents the intended write request. Write interceptors can read this map to determine the desired changes. They may also modify the map (which includes the ability to add additional name/value pairs if required), thereby requesting modification of the original write request, and/or additional write operations. If multiple write interceptors run in succession, they see the effects of the previous interceptors' modifications in this map.

If a blob has been created in the write request by uploading a file via Studio, it is available as `UploadedBlob` in the properties of the `WriteRequest`, providing access to the original filename.

Get content for request

The method `getEntity()` returns the content on which an update request is being executed. A write interceptor may use this method to determine the context of a write request, for example to determine the site in which the content is placed in a multi-site setting or to determine the exact type of the content. Do not write to the content object. To modify the content, update the properties map as explained above.

The method `getEntity()` returns `null` for a create request, because a write interceptor is called before a content is created. So that the interceptor is able to respond to the context of a create request, the `ContentWriteRequest` object provides the methods `getParent()`, `getName()`, and `getType()`, which provide access to the folder, the name of the document to be created, and the content type to be instantiated.

Reporting issues

Finally, an issues object can be retrieved by calling `getIssues()`. This object functions as shown in Section 7.15.1, “Validators” [176]. In this context, it allows an interceptor to report problems observed in the write request. If a write interceptor reports any issues with error severity using the method `addIssue(...)` of the issues object, the write request will automatically be canceled and an error description will be shown at the client side. If issues of severity warn are detected, the write is executed, but a message box is still shown. In any case, the issues are not persisted, so that the only issues shown for a content permanently are the issues computed by the regular validators.

The following example shows the basic structure of a custom interceptor for images. A field for the name of the affected blob property is provided. The `intercept()` method checks whether the indicated property is updated, retrieves the new value and provides a replacement value using the properties map.

Example 7.64. Defining a Write Interceptor

```
public class MyInterceptor extends ContentWriteInterceptorBase {
  private String imageProperty;

  public void setImageProperty(String imageProperty) {
    this.imageProperty = imageProperty;
  }

  public void intercept(ContentWriteRequest request) {
    Map<String,Object> properties = request.getProperties();
    if (properties.containsKey(imageProperty)) {
      Object value = properties.get(imageProperty);
      if (value instanceof Blob) {
        ...
        properties.put(imageProperty, updatedValue);
      }
    }
  }
}
```

Configuring Write Interceptors

Enabling the interceptor

A write interceptor is enabled by simply defining a bean in the Spring application context of the Studio web application. The interception framework automatically collects all interceptor beans and applies them in order whenever an update is requested. Interceptors with numerically lower priorities are executed first.

Priority of interceptor

For a write interceptor implemented using the class `ContentWriteInterceptorBase`, the priority is configured through the `priority` property. Such interceptors also provide the property `type`, indicating that an interceptor should only run for instances of specific content types. While the setter `setType()` receives a `ContentType` parameter, it is possible to simply provide the content type name as a string in the Spring bean definition file. The type name will be automatically converted to a `ContentType` object.

Furthermore, you need to configure whether the interceptor also applies to instances of subtypes of the given type through the property `isInterceptingSubtypes`. Like for validators, this property defaults to `false`, meaning that interception applies only to documents of the exact type.

Each write interceptor may also introduce additional configuration options of its own.

A typical definition might look like this:

Example 7.65. Configuring a Write Interceptor

```
<bean id="myInterceptor" class="MyInterceptor">
  <property name="type" value="CMPicture"/>
```

```
  <property name="imageProperty" value="data"/>
</bean>
```

7.15.3 Immediate Validation

Write requests that violate hard constraints of your document type model can be aborted when a validator fails. Typical use cases include:

- Preventing a client from uploading an image that is too large.
- Making sure that a document does not link to itself directly.

Blocking writes is not normally useful for text properties, because text values are saved continuously as the user enters data, and a write interceptor might not be able to operate appropriately during the first saves. For blobs or link lists, the impact on the user experience is typically less of a problem. In any case, you need to make sure that the user experience is not impacted negatively.

For implementing immediate validation, you can create an instance of the class `ValidatingContentWriteInterceptor` as a Spring bean and populate its `validators` property with a list of `PropertyValidator` objects. When the validators are configured to report an error issue, an offending write will not be executed (that is, the requested value will not be saved).

A configuration that limits the size of images in the `data` property of `CMPicture` documents to 1 Mbyte might look like this (class names are wrapped for layout reasons):

Example 7.66. Configuring Immediate Validation

```
<bean id="myValidatingInterceptor"
      class="com.coremedia.rest.cap.intercept.
        ValidatingContentWriteInterceptor">
  <property name="type" value="CMPicture"/>
  <property name="validators">
    <list>
      <bean class="com.coremedia.rest.cap.validators.
              MaxBlobSizeValidator">
        <property name="property" value="data"/>
        <property name="maxSize" value="1000000"/>
      </bean>
    </list>
  </property>
</bean>
```

Remember that the validators become active during creation, too, so that an immediate validator might validate initial values set by an earlier write interceptor.

7.15.4 Post-processing Write Requests

Write requests that have been executed by the server can be post processed by custom procedures. To this end, write post-processor objects can be configured in the Spring application context of the Studio REST service.

In most cases, a write interceptor is better suited for reacting to update requests, because an interceptor can still block an update completely and because it is more efficient to make sure that the right value are written immediately. But especially during content creation it might be necessary to create links to the generated content, which would be impossible before the content has actually been created.

Note that post-processors are not executed atomically with the actual write, so that the write is persisted even if a post-processor exits with an exception.

Developing Write Post-processors

In order to post process write requests as described above, create a class implementing the interface `ContentWritePostprocessor`. Alternatively, your class can also inherit from `ContentWritePostprocessorBase`, which already defines methods to configure the content type to which the write interceptor applies, and the priority at which the interceptor runs compared to other applicable interceptors.

This leaves only the method `postProcess(WriteReport<Content>)` to be implemented in custom code. The argument of the `postProcess` method provides access to all information needed for post processing the current request, which is either an update request or a create request.

The method `getEntity()` returns the content on which an update request has been executed. A write interceptor may use this method to determine the context of a write request.

The method `getOverwrittenProperties()` of the `WriteReport` object returns a map from property names to the values that have been overwritten during the write request. The new values can be retrieved as the current property value of the content returned from the method `getEntity()`.

Configuring Write Post-processors

A write post-processor is enabled by simply defining a bean in the Spring application context of the Studio web application. The interceptor framework automatically collects all post-processor beans and applies them in order whenever an update is requested. Post-processors with numerically lower priorities are executed first.

Priority of post-processor

For a write post-processor implemented using the class `ContentWritePostprocessorBase`, the priority is configured through the `priority` property. Such

post-processors also provide the property `type`, indicating that a post-processor should only run for instances of specific content types.

Furthermore, you need to configure whether the post-processor also applies to instances of subtypes of the given type through the property `isPostprocessingSubtypes`. Like for validators, this property defaults to `false`, meaning that post-processing applies only to documents of the exact type.

Each write post-processor may also introduce additional configuration options of its own.

7.16 Available Locales

As the `locale` property of a content item is just a plain string property, *CoreMedia Studio* provides assistance with setting the locales and keeping them consistent.

For this purpose a special content item is maintained that stores a list of language tags. These tags are used to restrict the selectable locales when cloning a site or setting a content item's `locale` property. To this end a new property field called `AvailableLocalesPropertyField` is used in the Blueprint content forms, which displays the available locales as a combo box.

The locales are rendered to the user in a readable representation that is localized for the current *Studio* language. The property field can also be configured to show an empty entry that sets the field value to the empty string.

When editing the list of available locales a validator will warn you if a language tag does not match the *BCP 47* standard (http://www.rfc-editor.org/rfc/bcp/bcp47.txt) and it will show an error if a language tag is defined multiple times.

The content item and property storing the locales can be configured with the following two *Studio* properties:

```
studio.availableLocalesContentPath=/Settings/Options/Settings/LocaleSettings
studio.availableLocalesPropertyPath=settings.availableLocales
```

7.17 Notifications

7.17.1 Configure Notifications

By default the amount of notifications requested by the *Studio* is limited to 20. This value is customizable via the Spring property `notifications.limit`. The property can be overwritten in the `application.properties` of the *Studio* webapplication or any other Spring `properties` file that is loaded for the *Studio* context.

7.17.2 Adding Custom Notifications

On several occasions, *CoreMedia Studio* shows notifications (see also Section 2.7 of the *CoreMedia Studio* User Manual). It is easily possible to add your own custom notifications to *CoreMedia Studio*. In the following the necessary steps are described.

For your server-side module where you want to create a notification, make sure you add a Maven dependency on `notification-api`. This module contains the `NotificationService` API.

Also, make sure that your Web-App as a whole has a Maven dependency on `notification-elastic`. This module contains an *Elastic Core*-based implementation of the `NotificationService`. For the Blueprint Studio Web-App this is already taken care of by the extension module `bpbase-notification-studio-lib`. By default, the provided `NotificationService` uses mongoDb. If for some reason you want to use a memory-based `NotificationService`, combine the Maven dependency `notification-elastic` with `core-memory`.

Finally, take care of declaring a `NotificationService` Spring bean, either via component-scan or explicit declaration.

For the Studio client side, you have to add the Maven dependency `notification-studio-client` to the module where you want to develop new notification UIs. In addition, you have to activate the notifications framework via plugin (for the Blueprint Studio, this is already taken care of by the extension module `bpbase-notification-studio-plugin`):

```
<editor:studioPlugin>
  <editor:configuration>
    <notifications:notificationsStudioPlugin/>
  </editor:configuration>
</editor:studioPlugin>
```

7.17.3 Creating Notifications (Server Side)

To create notifications on the server side, simply inject the `NotificationService` and use it at the appropriate position (event/request handler, REST method, task etc.) to create a new notification with the method `createNotification`:

```
Notification createNotification(@Nonnull String type,
                                @Nonnull Object recipient,
                                @Nonnull String key,
                               @Nullable List<Object> parameters);
```

A notification always has a combination of `type` and `key`. The key is basically a sub-type and will be used to determine the correct localization text key on the client side. An example of a type / key combination is "publicationWorkflow" / "offered".

A notification has a `recipient`. This parameter is typed as `Object`. For Studio notifications, it has to be a `User` object.

Additional `parameters` will be used on the client side to parametrize the notification's text. In advanced cases they are additionally used to configure actions and customize the notification's UI. Details are explained below.

7.17.4 Displaying Notifications (Client Side)

For displaying notifications in *CoreMedia Studio*, three levels are distinguished:

1. Simply displaying the notification in terms of a text message and an icon. For example, the notification might inform the user that a new publication workflow has arrived in its inbox.
2. The same as in 1. but with an additional click action handler. For example, clicking the publication workflow notification might open the publication workflow inbox in the Studio Control Room.
3. Completely customizing the display and controls of the notification.

Levels 1 and 2 are considered as the typical cases for displaying notifications. For these, CoreMedia offers default components. However, in certain cases it might be necessary/desired to develop a more refined notification UI.

Level 1: Simple Notification Display

For just displaying a notification in terms of an icon and a text message, you simply have to provide an icon class property and a text key property. These properties must match the patterns `Notification_{notificationType}_iconCls` and `Notification_{notificationType}_{notificationKey}_msg` respectively. For the example of a publication workflow notification from above, the properties look as follows:

```
Notification_publicationWorkflow_iconCls =
                publication-workflow-notification-icon
Notification_publicationWorkflow_offered_msg =
                 The workflow {0} is new in your inbox.
```

In this example, the message property has a placeholder. By default, the parameters of the notification (see notification creation above) are inserted in the placeholders one after the other. Consequently, the parameters have to be Strings. However, it is also possible to compute the placeholder insertions from the notification's parameters (for example, if you have a complex bean as a parameter that should be the basis for all placeholder insertions). In this case your notification's Studio component (see below) has to implement the interface com.coremedia.cms.editor.notification.components.TextParametersPreProcessor.

You define your properties in your own resource bundle (WorkflowNotifications_properties, for instance) and have to make sure to copy it onto the resource bundle com.coremedia.cms.editor.notification.Notifications_properties which is provided by us:

```
<editor:studioPlugin>
  <editor:configuration>
    <editor:copyResourceBundleProperties
            destination="{Notifications_properties}"
            source="{WorkflowNotifications_properties}"/>
  </editor:configuration>
</editor:studioPlugin>
```

Level 2: Simple Notification Display with Click Action

In many cases it is not enough to just display a notification. Normally, a notification is a request to the user to do something. So it should be possible to click the notification and be directed to the part of Studio where the user can do something about it.

In order to add an action click handler to your notification, you have to register your own notification component. You always register a notification component for a specific notification type:

```
<editor:studioPlugin>
  <editor:configuration>
    <notifications:registerNotificationDetailsPlugin
          notificationType="publicationWorkflow">
      <notifications:notificationDetailsComponentConfig>
        <wfnotifications:workflowNotificationDetailsComponent/>
      </notifications:notificationDetailsComponentConfig>
    </notifications:registerNotificationDetailsPlugin>
  </editor:configuration>
</editor:studioPlugin>
```

You do not have to do any component developing for level 2. You can simply let your notification component extend defaultNotificationDetails and add your notification action as its baseAction. You need to let your action extend

`NotificationAction`. This yields numerous benefits like accessing the notification via the method `NotificationAction.getNotification()`. Consequently, you have also access to all the notification's parameters.

```
<exml:component ... >

  <notifications:defaultNotificationDetails>
    <baseAction>
      <wfnotifications:showInboxForWorkflowNotificationAction/>
    </baseAction>
  </notifications:defaultNotificationDetails>
</exml:component>
```

Level 3: Custom Notification Display

You are free to develop your own notification component that does not inherit from `defaultNotificationDetails`. CoreMedia gives no further guidelines here but point out that your component at least has to inherit from `notificationDetails`. You register your custom component just as it was described above.

8. Security

In this chapter you will get to know about security mechanisms in *CoreMedia Studio*. This chapter does not cover general deployment aspects but focuses on application level security topics.

8.1 Preview Integration

It is recommended to serve the preview application and *CoreMedia Studio* application from different origins (the origin includes protocol, host, port), as described in Section 3.3, "Basic Preview Configuration" [21]. By separating the application origins, the browser ensures that both applications run independently in their own environment without direct access to each other (see Same-origin policy). Potential vulnerabilities in the preview application can not automatically propagate into the Studio application and vice versa.

It is highly recommended serving both, *CoreMedia Studio* and the embedded preview over HTTPS. The unencrypted HTTP protocol should only be used in a well separated development environment. Due to several browser constraints regarding mixed content it is highly discouraged to serve *CoreMedia Studio* and the embedded preview over different protocols.

8.2 Content Security Policy

Cross-site scripting (XSS) vulnerabilities are a severe threat for all high profile web applications like *CoreMedia Studio*. While conscientious output escaping always has to be the first choice in order to avoid cross-site scripting attacks, most modern web browsers offer a new standard called Content Security Policy (CSP) as a second line of defense (see http://www.w3.org/TR/CSP/).

Default Policy

The standard Blueprint *CoreMedia Studio* enables Content Security Policy by default. It sends at least the following default CSP header to the browser.

```
default-src 'none';
style-src 'self' 'unsafe-inline';
script-src 'self' 'unsafe-eval';
img-src 'self';
connect-src 'self';
object-src 'self';
font-src 'self';
media-src 'self';
frame-src <YOUR_PREVIEW_ORIGIN>
```

The header value represents the minimum set of directives to comply with the Studio's and its third-party library requirements. Both, the `unsafe-inline` value of the `style-src` directive and the `unsafe-eval` value of the `script-src` directive are required by Ext JS.

Customize Policy

Each of the CSP directives that are included in the default header plus the `report-uri` directive can be easily customized.

Note that weakening the policy settings can have severe effects on the application's security. Especially re-enabling inline script execution is considered harmful as it thwarts all efforts to prevent XSS.

Customization is done via a set of `studio.security.csp.*` properties in the `WEB-INF/application.properties` property file of the *Studio* web application. Each property is responsible for one Content Security Policy directive.

- `studio.security.csp.scriptSrc`: Takes a list of values for the `script-src` policy directive. Default values are `'self'`, `'unsafe-eval'`.
- `studio.security.csp.styleSrc`: Takes a list of values for the `style-src` policy directive. Default values are `'self'`, `'unsafe-inline'`.
- `studio.security.csp.frameSrc`: Takes a list of values for the `frame-src` policy directive. The hierarchy of default values for this directive is as follows.

  - `studio.previewUrlWhitelist` values if specified.
  - Schema and authority of `studio.previewUrlPrefix` if specified.
  - `'self'`
- `studio.security.csp.connectSrc`: Takes a list of values for the `connect-src` policy directive. Default value is `'self'`.
- `studio.security.csp.fontSrc`: Takes a list of values for the `font-src` policy directive. Default value is `'self'`.
- `studio.security.csp.imgSrc`: Takes a list of values for the `img-src` policy directive. Default value is `'self'`.
- `studio.security.csp.mediaSrc`: Takes a list of values for the `media-src` policy directive. Default value is `'self'`.
- `studio.security.csp.objectSrc`: Takes a list of values for the `object-src` policy directive. Default value is `'self'`.
- `studio.security.csp.reportUri`: Takes a list of values for the `report-uri` policy directive. If no custom list is provided, the directive is not included in the CSP header.
- `studio.security.csp.frameAncestors`: Takes a list of values for the `frame-ancestors` policy directive. Default value is `'none'`. This directive is used to defend clickjacking attacks.

  Please note that the `frame-ancestors` directive is part of the Content Security Policy Level 2 standard which is not yet supported by all the browsers that support Content Security Policy Level 1. If required, similar functionality can be achieved for 'legacy' browsers by setting an appropriate `X-Frame-Options` header.

Here is an example how an adapted property would look like.

```
studio.security.csp.objectSrc='self',www.exampleDomain.com
```

Write CSP Compliant Code

According to the default policy, inline JavaScript will not be executed. This restriction bans both inline `script` blocks and inline event handlers (for example `onclick="..."`). The first restriction wipes out a huge class of cross-site scripting attacks by making it impossible to accidentally execute scripts provided by a malicious third-party. It does, however, require a clean separation between content and behavior (which is good practice anyway). The required code changes for inline JavaScript code can be summarized as follows:

- Inline `script` blocks needs to move into external JavaScript files.
- Inline event handler definitions must be rewritten in terms of `addEventListener` and extracted into component code.

CSP violations can be easily discovered by monitoring the browser console. All violations are logged as errors including further details about the violation type and culprit.

Customize CSP Mode

CoreMedia Studio can run in one of four supported CSP modes.

- `ENFORCE`: Full CSP protection is enabled. All directives are enforced and reported.
- `ENFORCE_ALLOW_DISABLE`: Enable full CSP protection unless the `disableCsp` query parameter is set to 'true'. This mode is not recommended for a production environment.
- `REPORT`: CSP protection is enabled in report only mode. All violations are reported using the `report-uri` directives configured in `studio.security.csp.reportUri` but the directives are not enforced. This mode is not recommended for a production environment.
- `DISABLE`: CSP protection is disabled. This setting is not recommended.

The configuration is done via the `studio.security.csp.mode` key of the `WEB-INF/application.properties` property file of the *Studio* web application.

8.3 Single Sign On Integration

The default *CoreMedia Studio* authentication process is implemented based on Spring Security. Due to this open standard it is easy to replace the standard authentication mechanism with a common redirect based SSO system like *Atlassian Crowd* or *CAS*. While the authentication process can be replaced, the *CoreMedia Content Server* still needs to have a matching user provider configured in order to perform a fine grained authorization. Please refer to the [CoreMedia Content Server Manual] for further details about user providers.

This documentation does not replace the SSO manufacturers manual about how to integrate with Spring Security. This section only covers *CoreMedia Studio* specific adjustments that need to be made to a generic integration.

Do not modify the authentication process and the Spring Security filter chain unless you know what you are doing. An improperly configured security context can cause severe security issues.

Custom Component

The first step to integrate with a single sign on system is to create a custom component as replacement for the `editing-rest-security-component`. The `editing-rest-security-component` contains the configuration for the default built-in authentication process. It is not required anymore once there is a SSO integration in place. To replace the component simply replace the `editing-rest-security-component` dependency in the `pom.xml` of the `studio-webapp` with a dependency on the new component.

For further details about component artifacts and how to create them, please refer to the section *Application Architecture* in the [CoreMedia Digital Experience Platform 8 Developer Manual].

Generic Spring Security Context

The new component has to provide a Spring Security context that holds all the required configuration to authenticate users against your SSO system. Simply create a file `/META-INF/coremedia/component-XYZ.xml` in the new component and include the following import statement.

Example 8.1. Import base context

```
<import resource="classpath:
/com/coremedia/rest/cap/authentication/editing-rest-security-base.xml"/>
```

Next, create a generic Spring Security context based on the SSO manufacturer's documentation.

Studio Spring Security Context

The core elements of a Spring Security context are the `http` and the `authentication-manager` element. The `http` element is the parent of all functionality related to the web, the `authentication-manager` holds the configured `authentication-provider` elements.

Your generic Spring security context for a redirect based SSO solution could look something like:

Example 8.2. Spring Security context

```
<security:http entry-point-ref="YOUR_ENTRY_POINT" auto-config="false">
  <security:custom-filter position="FORM_LOGIN_FILTER" ref='YOUR_LOGIN_FILTER'/>
  <security:custom-filter position="LOGOUT_FILTER" ref='YOUR_LOGOUT_FILTER'/>

  <security:intercept-url pattern="/api/**" access="YOUR_AUTHORITY"/>
  <security:intercept-url pattern="/index.html" access="YOUR_AUTHORITY"/>
  <security:custom-filter .../>

  <security:session-management session-fixation-protection="newSession"/>
  <security:csrf request-matcher-ref="YOUR_CSRF_REQUEST_MATCHER"/>
</security:http>

<security:authentication-manager alias="authenticationManager">
  <security:authentication-provider ref='YOUR_AUTHENTICATION_PROVIDER'/>
</security:authentication-manager>
```

Login

CoreMedia Studio only imposes very minimal constraints to the login process.

Depending on the chosen SSO system the login itself is either performed by a Spring Security filter (internal login page) or an external system (external login page). The only requirement for this part of the login is that at least one recognizable authority is granted to the authenticated user (typically `ROLE_XYZ`). This authority needs to match the one in the `intercept-url` elements of the Spring Security `http` configuration.

The second requirement for the login procedure involves the authentication entry point referenced in the `http` configuration element. The entry point implementation for a redirect based SSO system usually does some sort of redirect to a login page. While this is sensible behavior for a 'normal' request, it is not expected for *Studio* REST calls which are XmlHttpRequests to a dedicated `/api` path. The *Studio* REST client can not handle redirects to pages reasonably. Unauthenticated REST calls should trigger a 403 response instead which is then handled by Studio with a popup message. A separate handling of REST calls and non REST calls can be achieved with a delegating entry point like the following.

Example 8.3. Delegating entry point

```
<bean id="delegatingEntryPoint"

class="org.springframework.security.web.authentication.DelegatingAuthenticationEntryPoint">

  <constructor-arg>
    <map>
      <entry key-ref="apiMatcher" value-ref="forbiddenAEP"/>
    </map>
  </constructor-arg>
  <property name="defaultEntryPoint" ref="SSO_SPECIFIC_ENTRY_POINT"/>
</bean>

<bean id="forbiddenAEP"

class="org.springframework.security.web.authentication.Http403ForbiddenEntryPoint"/>

<bean id="apiMatcher"
      class="org.springframework.security.web.util.matcher.AntPathRequestMatcher">
  <constructor-arg value="/api/**"/>
</bean>
```

Authorization

In addition to the authorization happening in the *Content Server* Spring Security is used to perform a pre-authorization at HTTP level. For a redirect based SSO system, it is best practice to pre-authenticate all requests to the REST API (`/api` path) and to the `index.html`. A set of `intercept-url` elements in the `http` configuration checks for the granted authority that the SSO system assigns to authenticated users.

Logout

CoreMedia Studio expects a logout listener to listen to POST requests the context relative path `/logout`. It has to trigger at least the default Spring Security `SecurityContextLogoutHandler` and the predefined `capLogoutHandler` bean. While the `SecurityContextLogoutHandler` resets the security context, the `capLogoutHandler` ensures that all `CapConnection`s for the current user are closed and released.

The logout listener must not listen to GET requests as this might result in a CSRF vulnerability. For simplicity reasons you can use the `logoutRequestMatcher` bean from the base security context.

A simple logout filter might look similar to this:

Example 8.4. Logout filter

```
<bean id="logoutFilter"
class="org.springframework.security.web.authentication.logout.LogoutFilter">
  <constructor-arg value="LOGOUT_SUCCESS_TARGET"/>
  <constructor-arg>
    <list>
      <ref bean="capLogoutHandler"/>
      <ref bean="SSO_SPECIFIC_LOGOUT_HANDLER_IF_NEEDED"/>
```

```
      <bean
class="org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler"/>

    </list>
  </constructor-arg>
  <property name="logoutRequestMatcher" ref="logoutRequestMatcher"/>
</bean>
```

Depending on the chosen SSO system it might be required to add another SSO specific logout handler or define additional single sign out filters in the Spring Security filter chain.

Other Configuration

While not required for the core functionality it is still highly recommended including a `csrf` and `session-management` configuration in your `http` settings.

The `csrf` configuration is used to enable the Spring Security CSRF protection. It must be enabled for all vulnerable HTTP verbs like `POST`, `PUT`, `DELETE`, the *Studio* client ensures that a valid token is included in the affected requests.

The `session-management` configuration together with the `session-fixation-protection` attribute is used to explicitly enable the Spring Security session fixation protection. The attribute value can safely be set to `newSession`.

User Finder

After finishing the configuration of the Spring Security context, there is one last *Studio* specific step to do.

So far you have set up a Spring Security context that is using the default Spring Security authentication providers and user detail services for your SSO system to authenticate users and load user details. These user details are usually represented by a SSO specific details object linked to the Spring Security `Authentication` object.

While keeping the default implementations in the authentication process hugely simplifies the SSO configuration, *CoreMedia Studio* still needs to know the matching `com.coremedia.cap.user.User` for the current SSO specific user details. Each individual Unified API operation has to be performed in the name of the currently authenticated `User` in order to be able to perform a fine grained authorization in the *CoreMedia Content Server*. To do this mapping between SSO specific user details and a `User` for the chosen SSO system, you have to implement a `SpringSecurityUserFinder`.

The `SpringSecurityCapUserFinder` interface consists of only one method that finds a `User` for a given `Authentication` object. In order to write a finder for the chosen SSO system you can simply extend the `AbstractSpringSecurityCapUserFinder` that already has a `CapConnection` available.

Example 8.5. User finder

```
public class XYZSpringSecurityCapUserFinder
        extends AbstractSpringSecurityCapUserFinder
        implements SpringSecurityCapUserFinder {

  @Override
  public User findCapUser(Authentication authentication) {
    Object principal = authentication.getPrincipal();
    if (principal instanceof XYZ) {
      String username = GET_USER_NAME_FROM_USER_DETAILS;
      return getCapConnection().getUserRepository()
              .getUserByName(username, DOMAIN);
    }
    return null;
  }
}
```

The custom user finder is enabled by replacing the Spring bean `springSecurityCapUserFinder` in the Spring context.

Example 8.6. Enable user finder

```
<customize:replace id="customSpringSecurityCapUserFinder"
                   bean="springSecurityCapUserFinder">
  <bean class="XYZSpringSecurityCapUserFinder"
        parent="abstractSpringSecurityCapUserFinder"/>
</customize:replace>
```

Session Tracking Mode

In order to prevent the `JSESSIONID` from appearing as an URL parameter it is recommended to add the following configuration to your `web.xml`:

```
<session-config>
  <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

8.4 Auto Logout

CoreMedia Studio provides two complementing mechanisms for automatically logging out inactive users: server-side session management and client-side activity tracking.

Jointly, these two algorithms keep the number of active sessions to a minimum, reducing the opportunity for an attacker to hijack a *Studio* session. The session timeouts for these algorithms can be configured separately. You should strive for a balance between security and user convenience.

Server-Side Session Management

A login to *CoreMedia Studio* is supported by a servlet session that is established with the web application container. If the client application in the browser does not contact the web application for a certain time, the servlet session will be closed by the container.

When the servlet session dies and the *Studio* client contact the server again, the condition will be detected and an appropriate error message is shown. The user will need to log in again.

Note that this timeout appears typically when the browser is closed or when the client machine is suspended or shut down. As long as *Studio* is open in a running browser, it continually fetches events from the server using HTTP requests. These requests keep the session alive.

You can configure the timeout in the `web.xml` file of the *Studio* web application. Most containers set a default value of 30 minutes. Because the *Studio* client contacts the server at least every 20 seconds, you may opt to reduce the timeout significantly. You should not reduce it to less than a couple of minutes, though, so that temporary network problems do not cause *Studio* to disconnect.

Client-Side Activity Tracking

In order to detect that the user is not interacting with a running *CoreMedia Studio*, a client-side process continually detects mouse movements and write requests, which provide a good indication of use activity.

When the user is inactive for too long, the *CoreMedia Studio* session is closed and the login screen is shown again. This timeout can be configured using the application property `studio.security.autoLogout.delay`. By default, the timeout is set to 30 minutes.

8.5 Logging

In order to support the detection of attacks and analysis of incidents, authentication failures as well as successful authentication events are logged by *CoreMedia Studio*. Example 8.7, “Example Output” [202] shows some typical log entries.

Example 8.7. Example Output

```
2015-07-07 13:43:30 [WARN]
  Http401AuthenticationFailureHandler [] -
  Failed login - User: Rick,
  IP: 127.0.0.1 (http-bio-8080-exec-8)
2015-07-07 13:51:11 [INFO]
  Http200AuthenticationSuccessHandler [] -
  Successful login - User: Rick (coremedia:///cap/user/8),
  IP: 127.0.0.1 (http-bio-8080-exec-6)
```

Marker Hierarchy

To get a better overview of security events you might want to duplicate or even redirect such events to extra access logs. To do so *CoreMedia Studio* uses a SLF4j Marker hierarchy

Example 8.8. Marker Hierarchy

- **coremedia** - root marker
  - **security** - security related entries
    - **authentication** - for example login or logout events
    - **authorization** - events such as missing rights for certain actions

Filtering

Filtering log entries is described in Logback's Online Documentation, Chapter 7: Filters. To redirect or duplicate security related log events you will define a filter for an appender using the JaninoEventEvaluator. Mind that you will require a runtime dependency on `org.codehaus.janino:janino`.

Example 8.9. Configure Access Log

```
<appender name="access"
          class="ch.qos.logback.core.FileAppender">
  <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator>
      <expression><![CDATA[
        marker != null && marker.contains("authentication");
        ]]></expression>
    </evaluator>
    <OnMismatch>DENY</OnMismatch>
    <OnMatch>ACCEPT</OnMatch>
  </filter>
```

```
  <encoder><pattern>${log.pattern}</pattern></encoder>
  <file>access.log</file>
</appender>
```

Example 8.9, “Configure Access Log” [202] shows an example how to log authentication events to a file named `access.log`. `marker` refers to a variable exported by `JaninoEventEvaluator` before parsing. Only authentication events will be logged here.

Example 8.10. Configure Security Log

```
<appender name="security"
          class="ch.qos.logback.core.FileAppender">
  <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator>
      <expression><![CDATA[
        marker != null && marker.contains("security");
        ]]></expression>
    </evaluator>
    <OnMismatch>DENY</OnMismatch>
    <OnMatch>ACCEPT</OnMatch>
  </filter>
  <encoder><pattern>${log.pattern}</pattern></encoder>
  <file>security.log</file>
</appender>
```

Example 8.10, “Configure Security Log” [203] shows an example how to log any security related events to a file named `security.log`. As `security` contains `authentication` also authentication log entries will go here.

Example 8.11. Configure Default Log

```
<appender name="default"
          class="ch.qos.logback.core.FileAppender">
  <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator>
      <expression><![CDATA[
        marker != null && marker.contains("security");
        ]]></expression>
    </evaluator>
    <OnMismatch>NEUTRAL</OnMismatch>
    <OnMatch>DENY</OnMatch>
  </filter>
  <encoder><pattern>${log.pattern}</pattern></encoder>
  <file>default.log</file>
</appender>
```

Example 8.11, “Configure Default Log” [203] shows an example for an appender which ignores any security related log entries. You might want to use this approach to hide login/logout entries from unauthorized personal.

Example 8.12. Configure Logger

```
<logger name="com.coremedia"
        additivity="false"
        level="info">
  <appender-ref ref="security"/>
  <appender-ref ref="access"/>
  <appender-ref ref="default"/>
</logger>
```

Example 8.11, “Configure Default Log” [203] eventually binds all appenders to the given logger.

Example 8.13. Suppress Security Logging

```
<turboFilter class="ch.qos.logback.classic.turbo.MarkerFilter">
  <Marker>security</Marker>
  <OnMatch>DENY</OnMatch>
</turboFilter>
```

Example 8.13, “Suppress Security Logging” [204] is just another example in case you completely want to suppress security log entries using so called turbo filters.

Glossary

Blob	Binary Large Object or short blob, a property type for binary objects, such as graphics.
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 Adaptive 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 Importer*
- *CoreMedia Site Manager*
- *CoreMedia Studio*
- *CoreMedia Search Engine*
- *CoreMedia Adaptive Personalization*
- *CoreMedia CMS for SAP Netweaver ® Portal*
- *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 CoreMedia employees. CoreMedia contributions are hosted on Github at https://github.com/coremedia-contributions.

Controm Room
: *Controm 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 of 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 exists.

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 supporting the declarative development of complex Ext JS components. EXML is Jangaroo's equivalent to Adobe Flex MXML and compiles down to Actions Script.
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.
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.
Importer	Component of the CoreMedia system for importing external content of varying format.
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 ActionScript as an input language which is compiled down to JavaScript. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net.
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.
JSP	JSP (Java Server Pages) is a template technology based on Java for generating dynamic HTML pages. It consists of HTML code fragments in which Java code can be embedded.
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-Site 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.

Personalisation
: On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.

Projects
: A project is a collection of content items in CoreMedia CMS created by a specific user. A project can be managed as a unit, published or put in a workflow, for example.

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 items 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

Swing component of CoreMedia for editing content items, managing users and workflows.

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, JSPs 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, pre-defined 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 web application

The *User Changes* web 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.

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.

WebDAV

WebDAV stands for World Wide Web Distributed Authoring and Versioning Protocol. It is an extension of the Hypertext Transfer Protocol (HTTP), which offers a standardised method for the distributed work on different data via the internet. This adds the possibility to the CoreMedia system to easily access CoreMedia resources via external programs. A WebDAV enabled application like Microsoft Word is thus able to open Word documents stored in the CoreMedia system. For further information, see http://www.webdav.org.

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 and global-search-and-replace 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

Symbols

#joo.debug, 71
-Dcontentserver.host, 66

A

Access Control (content), 52
Access Control (workflow), 52
actions, 31
ActionScript
    documentation, 34
annotation
    configurable, 58
    explicit, 58
    implicit, 58
architecture, 14
ASDoc, 34

B

beans, 38
    properties, 38
    remote, 38, 40
    singleton, 40
    state, 39
browser developer tools, 70
    drill-down, 71
button
    add to Favorites Toolbar, 145
    apps menu, 149
    custom action, 148
    disapprove, 150

C

callback function, 42
    successful, 42
CKEditor, 17, 129
    add plugin, 129
    blockElements, 125
    custom style classes, 124
    default plugins, 130
    mapping characters while copying, 126
    plugin definition, 130
    plugins, 129
    visualize style, 125
ckeditor
    richtext, 133
com.coremedia.cap.struct.Struct, 54
com.coremedia.ui.data.bean, 38
compiling, 67
component
    extending, 30
    plugin mechanism, 30
component map, 144
components
    configuration, 35
concurrency, 56
connection
    command line parameter, 66
    create, 51
    reassigning server URL, 67
    using profiles, 67
    with Content Server, 66
    with Preview CAE, 66
Content, 52
content
    accessing properties, 53
content type icons, 98
ContentProperties, 53
ContentRepository, 51
ContentTypes_properties, 97
ContentWritePostprocessor, 184
context
    annotate, 58
Control Room
    configuration, 20

D

dashboard, 166
    configuration, 167
    configureDashboardPlugin, 167
    UML overview, 168

widgets, 166
debugging, 70
debugging applications, 25
derived contents list, 104
document form
article example, 102
hide property, 103
link list properties, 105
document forms, 101
adding tabs, 105
customize, 101
disabling preview, 112
document types
exclude from library, 113
documents
client-side initializers, 114

E

example
add disapprove button, 143
EXML, 16
typed language, 32
Ext AS, 32
file types, 32
Ext JS, 16, 27
activate console, 72
components, 29
debugging, 72
plugins, 37
xtype, 27
ext.Component, 29
ext.ComponentMgr, 29
Ext.getCmP, 29
ext.Viewport, 29

F

Firebug
debugging, 25
forms (see document forms)
function value expressions, 48
changed value, 49
passing arguments, 48

I

icons
content types, 98
four types, 100
predefined content type icons, 98
sprites, 101
IDE
setup, 24
IDEA plugin, 69
IEditorContext
usages, 86
Illuminations, 73
image cropping, 108
defining crops, 108
enabling, 108
image map, 111
enabling, 111
validation, 111
Inheritance
property, 151
interceptor
enabling, 182
example, 182
get content, 181
get file name, 181
get request values, 181
issues, 181
primary, 182
interceptors, 180
issues, 41
codes, 42
marking invalid, 41

J

Jangaroo, 16, 32
compiler, 33
debugging, 71
documentation, 33
IDEA plugin, 69
user group, 34

L

labels, 94
Blueprint properties, 94

example, 95
new resource bundle, 94
overriding standard labels, 96
predefined property classes, 94
library
customizing, 152
list view columns, 152
search filter, 155
thumbnail view, 155
list views
additional data fields, 153
search mode, 154
localization, 61
Blueprint properties, 98
default language, 61
document types and fields, 97
overriding default properties, 97
overwrite existing, 61

M

memory leaks, 75
retainers, 75
metadata
example, 139
listen to changes, 141
Metadata Service, 138
metadata tree
filter, 140
traverse breadth-first, 140
MetadataTree, 139
MetadataTreeNode, 139
MIME types, 175
adding, 175
model beans, 43
MongoDB
Collaboration, 20
multisite
sitesservice, 63
MVC pattern, 37

N

non-public API usage, 68

O

OperationResult, 42

P

pbe.studioUrlWhitelist, 22
plugin rule, 85
plugins, 84
Preferences, 142
preview
communicate with Studio, 138
configuration, 21
whitelist of URLs, 22
Process, 53
ProcessDefinition, 53
ProcessState, 54
properties, 38, 115
events, 39
example String property, 116
inherit from base class, 116
updating, 38
property
injecting, 143
property field
compound field, 119
data binding, 119
default text, 119
mandatory properties, 117
read-only, 119
register, 117
richtext, 122
update model, 118
validating, 118
property path expressions
access methods, 47
Property Value Inheritance, 151
PublicationService, 51

R

remote beans, 38, 40
get URL, 40
load content, 40
properties ready to use, 41
subclasses, 40
replaceItemsPlugin, 92

repository connection, 19
repository.url, 19
richtext property
  inline images, 123
  table cell merge and split, 123
  toolbar, 127
running Studio web application, 67

S

search filter
  add, 155
  default state, 156
  open library in filter state, 157
  Solr query string, 156
search folder
  addArrayItemsPlugin, 146
  search parameters, 147
search folders
  providing defaults, 146
search mode
  freshness, 155
  searchable lists, 154
server-side validation, 41
structs, 54
  adding new properties, 55
Studio
  compiling, 67
  plugins, 84
  running, 67
Studio plugin
  adding button, 88
  loading external resources, 93
  main class, 86
  register, 92
  relative position of new component, 89
  removing components, 92
  replacing components, 92
  structure, 84
Studio plugins
  execution order, 88
  rules, 87
studio.previewControllerPattern, 66

T

Task, 53
TaskDefinition, 53
TaskDefinitionType, 53
TaskState, 54
toolbar
  order items, 146
toolbars, 145
translation
  define source language document, 112

U

Uniform access layer, 37
UploadedBlob, 181
User Changes web application
  configuration, 20

V

validators, 176
  editor actions, 180
  enabling, 177
  immediate validation, 183
  implementing, 177
  localize messages, 179
  messages, 179
  multiple properties, 178
  predefined, 176
  server-side, 176
  single-property, 177
value expression
  events, 45
  listener, 45
  no undefined result, 45
  property path expression, 46
value expressions, 37, 44
  getValue, 45
  implementations, 44

W

widget
  configuration mode, 166
  getting search results, 169
  reload button, 171
widgets
  adding custom types, 170
  predefined, 169

work area
  action to open, 160
  customize context menu, 164
  restore, 162
  start with blank area, 162
  storing state of tab, 161
  tabs, 160
WorkflowObject, 53
WorkflowObjectProperties, 54
WorkflowRepository, 52
WorklistService, 52
workspace
  setup, 24
write post-processor
  priority, 184
write post-processors, 184
  configuring, 184
write requests
  interceptors, 180
  post process, 184