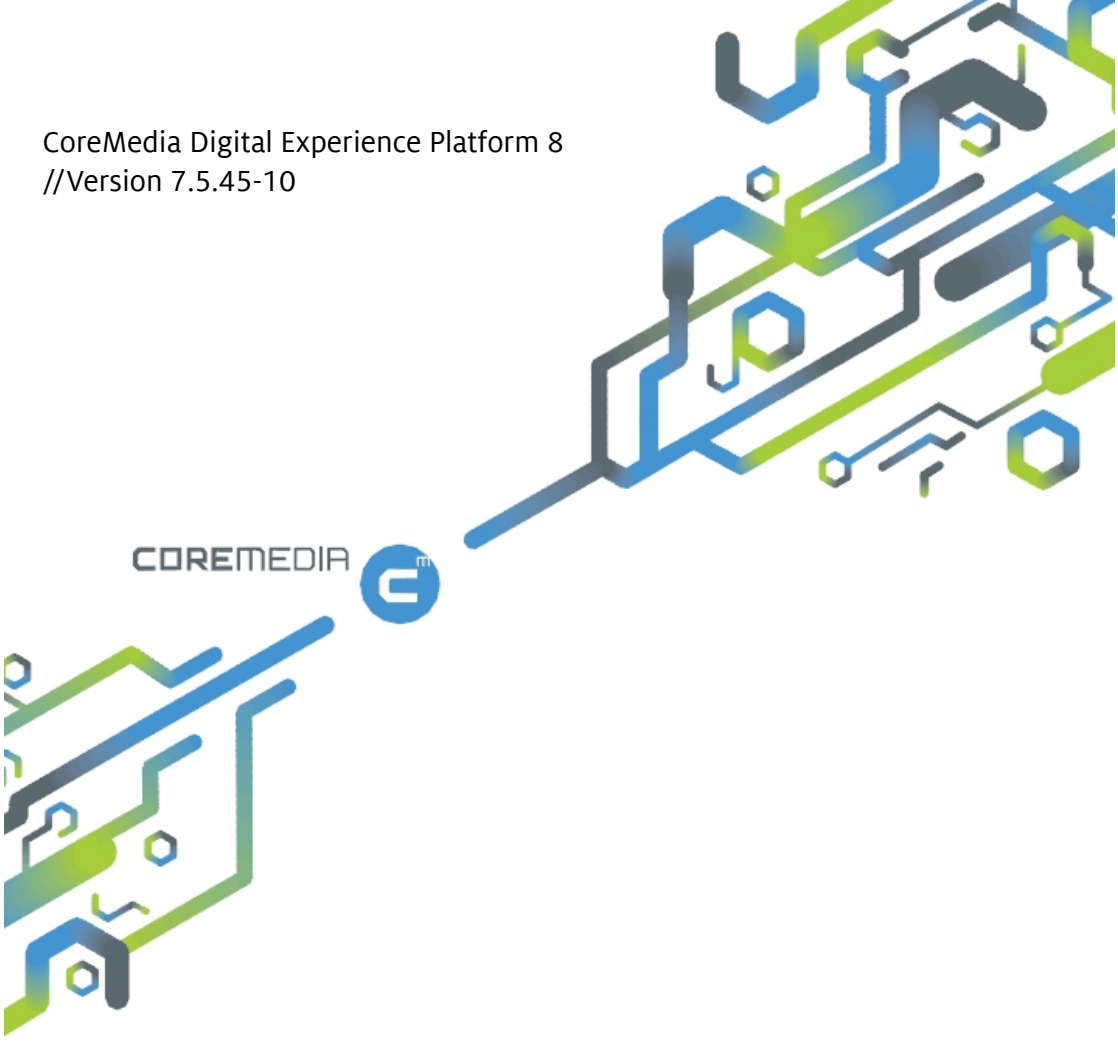
CoreMedia Digital Experience Platform 8
//Version 7.5.45-10



CoreMedia Site Manager Developer Manual

CoreMedia Site Manager Developer 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. Preface 1
1.1. Structure of the Manual 2
1.2. Audience 3
1.3. Typographic Conventions 4
1.4. CoreMedia Services 6
1.4.1. Registration 6
1.4.2. CoreMedia Releases 6
1.4.3. Documentation 7
1.4.4. CoreMedia Training 9
1.4.5. CoreMedia Support 10
1.5. Change Chapter 13
2. Site Manager Overview 14
2.1. The BeanParser 15
2.2. Description of the CoreMedia editor.dtd 17
3. Operation and Configuration 18
3.1. Defining The User Login 19
3.2. Define the Locale 21
3.3. Starting the Editor 22
3.4. Defining XML Files For Configuration 23
3.5. Defining Group Specific Configuration Files 29
3.6. Configuration Using coremedia-richtext-1.0.css 32
3.6.1. Supported CSS Attributes 32
3.6.2. Extend the coremedia-richtext-1.0.css file 33
3.6.3. Localize the New Styles and Style Groups 34
3.6.4. Add to Content Editor 36
3.7. Configuring the Struct Editor 39
3.8. Disable Workflow 40
3.9. Enable Direct Publication 41
3.10. Define the Browser for Web Extensions 42
3.11. Enable the Spell Checker 45
3.12. Troubleshooting 47
3.12.1. Taking a Thread Dump 49
4. Programming and Customization 50
4.1. How To 51
4.1.1. How To Access Arbitrary Resources 51
4.2. Program Own Initializers 52
4.3. Program Own Validators 54
4.4. Program Own Language Resolver Factories 56
4.5. Program Own PropertyEditors 58
4.6. Program Own Predicate Classes 60
4.7. Program Own Renderers 62

4.8. Program Own Commands 64
4.8.1. Register Commands 65
4.8.2. Localize Commands 66
4.8.3. Add Command to Document View 67
4.8.4. Add Command to Explorer View 68
4.8.5. Add Command to Context Menu 69
4.8.6. Add Action to RichTextPane 70
4.9. Program Own ResourceNamingFactory Classes 72
4.10. Localization 73
4.10.1. Localize the Editor 73
4.10.2. Localize for Use with WebStart 75
5. Appendix 76
5.1. Classes Delivered for Site Manager Configuration 77
5.1.1. Property Editors 77
5.1.2. View Classes 96
5.1.3. Predicate Classes 97
5.1.4. Column Classes 101
5.1.5. Renderer Classes 104
5.1.6. Initializer Classes 106
5.1.7. Validator Classes 107
5.1.8. Comparator Classes 109
5.2. Configuration Possibilities in the XML Files 112
5.2.1. General Configuration 113
5.2.2. Defining Group Specific Configuration Files 126
5.2.3. Configuring Document Types 129
5.2.4. Configuring Document Windows 136
5.2.5. Configuring Table Views 139
5.2.6. Configuring the Spell checker 146
5.2.7. Configuring the Workflow 149
5.2.8. Configuring Web Extensions 153
5.2.9. Example Configuration of the Document Overview 156
5.2.10. Example Configuration of the Document Window 159
5.3. Configuration Possibilities in editor.properties 162
5.4. Configuration Possibilities in proxy.properties 165
5.5. Configuration of The Site Manager in capclient.properties 167
5.6. Configuration Possibilities in workflowclient.properties 168

5.7. Configuration Possibilities in language-mapping.properties 169
Glossary 170
Index 177

List of Figures

2.1. The Explorer window of the Site Manager 14
3.1. The localized style group 36
3.2. The tabbed Struct editor 39
5.1. Configured file creation dialog 96
5.2. Example of a tabbed document view 97
5.3. Configured Query window 159
5.4. Dish document without special configuration 160
5.5. Dish document after the configuration 161

List of Tables

1.1. Typographic conventions 4
1.2. Pictographs 4
1.3. CoreMedia manuals 7
1.4. Log files check list 11
1.5. Changes 13
3.1. Some attributes of the element Editor 20
3.2. Attributes of element <Locale> 21
3.3. Attributes of the <ConfigGroup> element 31
3.4. Attribute of the <Configuration> element 31
3.5. Attributes of StyleGroup element 37
3.6. Editor classes 39
3.7. An attribute of the element Editor 40
3.8. An attribute of the element Editor 41
3.9. The attributes of the <WebBrowser> element 43
3.10. Attribute of the element SpellChecker 45
4.1. Parameters of the getInitialValue method 52
4.2. Return values of the getInitialValue method 52
4.3. Parameters of the validate method 54
4.4. Default types of the properties 54
4.5. Parameters of the getLanguageResolver method 56
4.6. Commands to subclass from 64
4.7. Register a new command 65
4.8. Steps to extend GenericDocumentView. 68
4.9. How to integrate your command into the explorer view 68
4.10. Add command to context menu 69
4.11. How to integrate actions into the RichTextPane 70
5.1. Property editors for the workflow 77
5.2. Property editors for strings 78
5.3. Property editors for integers 80
5.4. Property editors for dates 82
5.5. Some attributes of the RichTextPane 83
5.6. Attributes of NewDocumentDialogSettings 87
5.7. The attribute of the PasteTransformation element 89
5.8. The attributes of the TransformElement element 89
5.9. Attributes of the IgnoreElement element 89
5.10. Attributes of the Attribute element 90
5.11. Editors for blob fields 91

5.12. Attributes of NewDocumentDialogSettings and DocumentChooserSettings 94
5.13. More attributes of NewDocumentDialogSettings 95
5.14. View classes 96
5.15. Predicate classes for filtering documents types. 98
5.16. Predicate classes for filtering folders 98
5.17. Predicate classes for filtering workflows 99
5.18. Predicate classes for filtering documents 99
5.19. Programming own predicates 101
5.20. Column classes for workflows 102
5.21. Column classes for predefined columns 102
5.22. Column classes for user defined document properties 103
5.23. Implicit properties 103
5.24. Provided Renderer classes of CoreMedia CAP 105
5.25. Initializer classes 106
5.26. Validator classes 107
5.27. Comparators for document types 109
5.28. Server-side comparators for sorting rows 110
5.29. Client-side comparators for sorting rows 111
5.30. Comparators for sorting folders 111
5.31. Comparators for sorting workflows 111
5.32. The attributes of the element Editor 114
5.33. Attribute of element <AuthenticationFactory> 116
5.34. Attributes of the DocumentTableLayout element. 116
5.35. Attributes of element <Locale> 117
5.36. Attribute of the <Bundle> element 118
5.37. Attributes of the element Preview 118
5.38. Attributes of the element Browser 120
5.39. Attributes of the element RemoteControl 121
5.40. Parameters of the remote control URI 121
5.41. Attributes of element <FrameFactory> 123
5.42. Attribute of element <PropertyModelFactory> 124
5.43. The attributes of the <ResourceNamingFactory> element 124
5.44. The attributes of the <WebBrowser> element 126
5.45. Attributes of the <ConfigGroup> element 128
5.46. Attribute of the <Configuration> element 129
5.47. Attribute of the DocumentType element 130
5.48. Attributes of the <PropertyType> element 131
5.49. Regular patterns to use with the attribute validPattern 131
5.50. Attributes of the element <Validator> 132

5.51. Attributes of the element <Initializer> 133
5.52. Attribute of the element ModelClass 133
5.53. Attribute of element <Comparator> 134
5.54. Attributes of the DocumentTypePredicate element 135
5.55. Attribute of the Predicate element 136
5.56. Attributes of the Documents element 136
5.57. Attributes of element <Document> 137
5.58. Attributes of element <Property> 138
5.59. Attributes of element <Tab> 139
5.60. Attributes of the <Explorer> element 139
5.61. Attributes of element <Filter> 142
5.62. Attribute of element <TableDefinition> 143
5.63. Attributes of element <ColumnDefinition> 143
5.64. Attributes of element <NamedDocumentVersionComparator> 145
5.65. Attribute of the <Renderer> element 145
5.66. Attributes of the <DisplayMap> element 146
5.67. Attribute of the element SpellChecker 146
5.68. Attributes of the MainDictionary element 147
5.69. Attribute of the <CustomDictionary> element 148
5.70. Attribute of the PropertyLanguageResolverFactory element 149
5.71. Attribute of element <Process> 151
5.72. Attribute of the <Task> element 152
5.73. Attribute of <WorkflowStartup> element 152
5.74. Attributes of element <Variable> 153
5.75. Attributes of <AggregationVariable> element 153
5.76. Attributes of the WebContext element 154
5.77. Attributes of the <WebExtension> element 155
5.78. Attributes of the <Pattern> element 156
5.79. editor.properties 162
5.80. proxy.properties 165
5.81. capclient.properties 167
5.82. Parameters of the workflowclient.properties file 168

List of Examples

2.1. Example of a BeanParser XML file 15
3.1. Example for the Locale element 21
3.2. Add files to server.policy 28
3.3. Add the style group to the editor 36
3.4. Disabling the workflow in the editor.xml 40
3.5. Disabling the workflow 41
3.6. Example of a Spellchecker element 45
4.1. Integrate Initializer in editor.xml 52
4.2. Example of an Initializer 53
4.3. Integrate validator in editor.xml 55
4.4. Simple customized validator 55
4.5. Example of a language resolver 56
4.6. How to integrate a property editor 58
4.7. Example of a property editor 58
4.8. How to integrate the Predicate class 61
4.9. Example of a customized Predicate class 61
4.10. How to integrate a Renderer in the editor 62
4.11. Example of a customized Renderer class 62
4.12. Localize command 67
4.13. Integrate bundle 67
4.14. How to integrate into editor.xml 68
4.15. How to integrate a resource naming factory 72
4.16. Example of a resource naming factory 72
5.1. Example for the use of a property editor 77
5.2. Example of PasteTransformation 90
5.3. PlainXmlPropertyEditor configuration example 90
5.4. Example for the configuration of a document view 96
5.5. Example for the use of a filter 97
5.6. Example for the use of a column class 101
5.7. Example for the use of a renderer class 104
5.8. Example for the use of an initializer 106
5.9. Example for the use of a validator 107
5.10. Example for the use of a Comparator 109
5.11. Example for the Editor element in editor-startup.xml 113
5.12. Example of the DocumentTableLayout element 116
5.13. Example for the Locale element 117
5.14. Example for the Bundle element 117
5.15. Example for the Preview element 118

5.16. Example for the Browser element 119
5.17. Example for the RemoteControl element 120
5.18. Example of the FrameFactory element 123
5.19. Example for the PropertyModelFactory element 123
5.20. Example of the ResourceNamingFactory element 124
5.21. Example for the DocumentTypes element 129
5.22. Example of a DocumentType element 130
5.23. Example of a PropertyType element 131
5.24. Example of the Validator element. 132
5.25. Example of the Initializer element 132
5.26. Element ModelClass 133
5.27. Example for sorting the offered document types and the folders in the folder view. 133
5.28. Example for the DocumentTypePredicate element 134
5.29. Example for the Predicate element used in a Filter element 135
5.30. Example for the Documents element 136
5.31. Example for the Document element 136
5.32. Example for the Property element 138
5.33. Example for the Tab element 138
5.34. Example for the Explorer element 139
5.35. Example for the ResourceChooser element 140
5.36. Example for the Query element 141
5.37. Example of the Search element 141
5.38. Example for the Treesorter element 141
5.39. Example for the TreeFilter element 142
5.40. Example for the Filter element 142
5.41. Example for the TableDefinition element 143
5.42. Example for the ColumnDefinition element 143
5.43. Example for the NamedDocumentVersionComparator 144
5.44. Example for the Renderer element 145
5.45. Example for the DisplayMap element 145
5.46. Example of a Spellchecker element 146
5.47. Example of a MainDictionary element 147
5.48. Example of a CustomDictionary element 148
5.49. Example of a PropertyLanguageResolver Factory element 149
5.50. Example of the Workflow element 149
5.51. Example for the Processes element 150
5.52. Example for the Process element 150
5.53. Example for the Code element 151

5.54. Example for the Task element 151
5.55. Example for the WorkflowStartup element 152
5.56. Example for the Variable element 152
5.57. Example for the AggregationVariable element 153
5.58. The query tags 157
5.59. Example configuration for document type display 157
5.60. Example configuration for document name display 158
5.61. Example configuration for the structured text column 158
5.62. Creation of two filters 159
5.63. Code example for configuration of the editor 161

1. Preface

The *CoreMedia CMS* is the future-proof standard software solution for production, administration and distribution of multimedia content for digital services.

Content applications such as high-volume web sites and device-independent multi-channel services are implemented cost-effectively in minimal time using *CoreMedia CMS*.

This *Site Manager Developer Manual* is written for developers and administrators - people who set up and tune, who integrate and implement *CoreMedia CMS*. It describes how to make all the features of the *Site Manager* work well and create a unique workplace fitting the customers demands.

Out-of-the-box functions for complete editing processes in the easily used *Site Manager*, rapid prototyping features, an acclaimed architecture and a matching, proven process for content application projects guarantee excellent results and high, sustained customer satisfaction.

1.1 Structure of the Manual

This manual provides information on the customizing of the *Site Manager*.

- Chapter 2, *Site Manager Overview* [14] outlines a *Site Manager* overview,
- Chapter 3, *Operation and Configuration* [18] describes the customization and operation of the editor,
- Chapter 4, *Programming and Customization* [50] shows how to use the editor API for own extensions,
- Chapter 5, *Appendix* [76] contains a reference of all XML elements and delivered classes for customization.

1.2 Audience

This manual is addressed to developers of CoreMedia projects who want to configure and customize the *Site Manager*.

For an even more detailed documentation, this manual is augmented by the comprehensive *Site Manager* Javadoc pages.

1.3 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.

1.4 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 1.4.1, “Registration” [6] 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 1.4.1, “Registration” [6] describes how to register for the usage of the services.
- Section 1.4.2, “CoreMedia Releases” [6] describes where to find the download of the software.
- Section 1.4.3, “Documentation” [7] describes the CoreMedia documentation. This includes an overview of the manuals and the URL where to find the documentation.
- Section 1.4.4, “CoreMedia Training” [9] describes CoreMedia training. This includes the training calendar,the curriculum and certification information.
- Section 1.4.5, “CoreMedia Support” [10] describes the CoreMedia support.

1.4.1 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 1.4.5, “CoreMedia Support” [10]) by email to request further access depending on your customer, partner or freelancer status so that you can use the CoreMedia services.

1.4.2 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 1.4.1, “Registration” [6] 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 1.4.5, “CoreMedia Support” [10]) to get your licences.

1.4.3 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

1.4.4 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

1.4.5 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 1.4.1, “Registration” [6]. 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`

1.5 Change Chapter

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

Section	Version	Description

Table 1.5. Changes

2. Site Manager Overview

This manual describes the customization of the *Site Manager* by means of predefined or self-implemented classes. The *Site Manager* is a Java Swing application which offers a full-grown editor API to the developer. The editor consists of four main windows which are all subject to configuration except the Query window:

- The Explorer window in which you can inspect and edit the folder and document structure.
- The Document window in which you can edit the documents.
- The Query window in which you can retrieve information.
- The Workflow window in which you can create and edit workflows.

  Configuration of the first three windows is covered in this manual but the Workflow window is covered in the *Workflow Manual*.

The next figure shows the Explorer window of the *CoreMedia Site Manager*.

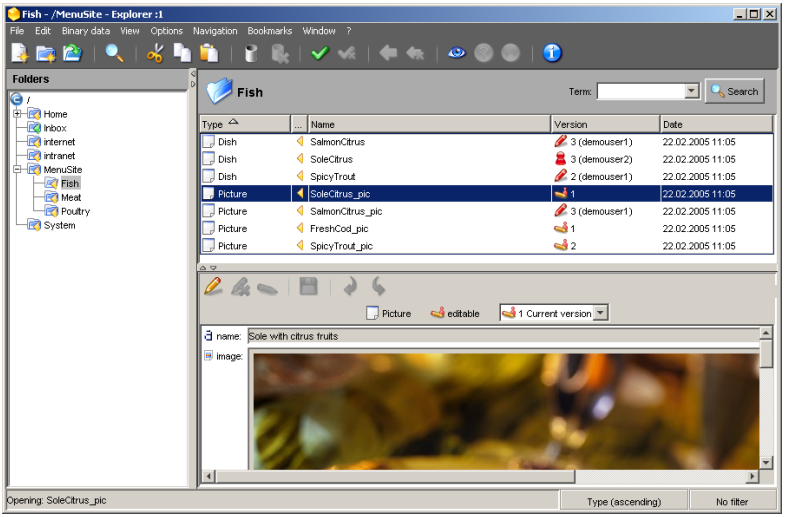


Figure 2.1. The Explorer window of the Site Manager

For more details on the editor GUI please refer to the [CoreMedia User Manual].

2.1 The BeanParser

The XML files used to configure *CoreMedia CMS* components are processed by the *BeanParser*, which is a basic part of the system. As such, it is used to

- configure document views,
- configure editor.

The *BeanParser* processes the XML files as follows:

- For each XML element it tries to instantiate an object of a class, which is determined by a factory or via the `class` attribute. The object is created via Java Reflection and a zero-argument constructor.
- If the XML element occurs inside another XML element, it tries to set the object created by the inner element on the object created by the outer element. For this, it calls a setter method and passes the object. The setter method may be named `set<Element Name>()`, `add<ElementName>()` or simply `set()` or `add()`.
- For each attribute of an element it calls a setter method on the object that was created when parsing the element start tag. The setter method may be named `set<AttributeName>()`, `add<AttributeName>()` or simply `set()` or `add()`.

Example:

Assume the following XML file:

Example 2.1. Example of a BeanParser XML file

```
        <FirstElement class="com.example.FirstElement"
attribute1="Ho">
    <SecondElement class="com.example.SecondElement"
      attribute="Hi"/>
</FirstElement>
```

The BeanParser will execute the following steps:

1. Create an instance of class `com.example.FirstElement`.
2. Call `setAttribute1("Ho")` on that instance.
3. Create an instance of class `com.example.SecondElement`.
4. Call `setAttribute("Hi")` on that second instance.
5. Call `firstElement.setSecondElement(secondElement)`, that is, set the object created in step 3 on the object created in step 1.

Advanced features:

The class attribute has a special meaning as it determines the name of the class to instantiate objects from. For this attribute, no setter methods has to be defined inside the class.

The *BeanParser* works without an XML Document Type Definition (DTD), but in connection with a DTD, it makes use of `ID` and `IDREF` feature of the XML Parsers. The object, that has been created by the element with the `IDREF` attribute, is substituted by the object that is defined the corresponding `ID` attribute. Again, no setter methods have to be defined inside the involved classes.

2.2 Description of the CoreMedia editor.dtd

The elements and corresponding attributes allowed in the XML configuration files are defined in the `coremedia-editor.dtd` file, in fact a (pseudo) DTD. The DTD is called "pseudo" because it contains positions at which it can be extended, depending on the classes used. These points are shown by the placeholder `%varies;` You find the DTD in the `lib/cap-schema-bundle-<version>.jar` file in the zipped `xml` folder of your *CoreMedia Content Server* installation.

In Section 5.2, “Configuration Possibilities in the XML Files” [112] the meaning of elements and of the corresponding attributes is described. In addition, a short example for the syntax of each element is given.

3. Operation and Configuration

The administrator can configure the *Site Manager* using two types of files. Property files, which are mainly used for technical concerns like the connection to the server or the logging of the editor and XML files which are mainly used to configure the appearance of the *Site Manager,* for example which information is shown in the document overview, which buttons are shown in the toolbar or which fields are hidden in the document window. Some of the tasks you can do with the property and XML configuration files are described in this chapter but you will find an exhaustive summary of all elements in Chapter 5, *Appendix* [76].

The following property files exist:

- `capclient.properties`
- `editor.properties`
- `proxy.properties`
- `Optional mime.properties`
- `language-mapping.properties`

3.1 Defining The User Login

You can shorten the login time if you predefine the user name, password and domain of the user. The login window of the *Site Manager* is automatically filled with these data. You can even skip this window completely, so that the user will login immediately.

Not mandatory: By default, the user set in the environment of the computer will be used.

Configure in one of the following files:

- `editor.properties`
- `editor-startup.xml`

The settings of `editor.properties` will be overwritten by the `editor-startup.xml` settings.

Predefine login data

editing.properties

In the `editor.properties` file use the following properties:

- Enter the name of the user into the `login.username` property.
- Enter the password of the user into the `login.password` property.
- Enter the domain of the user into the `login.domain` property.
- For immediate login enter "true" into the `login.immediate` property.

`editor-startup.xml`

In the `editor-startup.xml` file use the following properties:

- Enter the name of the user into the attribute `loginName` of the `<Editor>` element.
- Enter the password of the user into the attribute `loginPassword` of the `<Editor>` element.
- Enter the domain of the user into the attribute `loginDomain` of the `<Editor>` element.
- For immediate login enter "true" into the attribute `loginImmediate` of the `<Editor>` element.

The <Editor> element has more attributes that can be used. They are shown in the following table:

Table 3.1. Some attributes of the element Editor

Attribute	Description
class	This attribute is used to enter the editor class to use. Default is `hox.corem.editor.generic.GenericEditor`.
`loginName`	This attribute is used to enter the default name for login. If no name is entered, the name from the environment is used. You can always change the name during login, it is just a preset. If a login name should be predefined, it must be set in the `editor-startup.xml` file. If a global `editor.xml` file is used for all users it might be sensible to set the login name in the `editor.properties` file.
`loginPassword`	This attribute is used to enter the default password for login. If no password is entered, the login name is used. You can always change the password during login, it is just a preset. If a login password should be predefined, it must be set in the `editor-startup.xml` file. If a global `editor.xml` file is used for all users it might be sensible to set the login password in the `editor.properties` file.
`loginDomain`	This attribute is used to enter the default domain for login. You can always change the domain during login, it is just a preset. If a login domain should be predefined, it must be set in the `editor-startup.xml` file.
`loginImmediate`	If this attribute is set to "true", an attempt is made to connect directly to the server with the login data given above. The login window does not appear. The default value is "false".
`showCurrentUser`	If this attribute is set to "true", the name of the current user of the editor is shown at the top of the window. Default is "false", that is, the user name is not shown.
`startup`	This attribute defines the *Site Manager* window to start with. Possible values are "OpenExplorer" "OpenQuery", "OpenWorkflow", "OpenUserManager" which will open the respective window. As default, the *Site Manager* starts with the explorer window ("normal" user) or with the user manager window ("administrator" user).
`startupMode`	This attribute defines the start-up mode for administrators. If set to "4.2", the super user with ID "0" always starts with the User Manager window. All other users will start with the window defined using "startup". If set to "5.0" the super user with ID "0" and all members of the administration group start with the User Manager window. All other users will start with the window defined using "startup". Default setting is "5.0".

3.2 Define the Locale

<Locale>

Child elements:

Parent elements: `<Editor>`

You can select the language and country settings which should be used by the *Site Manager* with the element `<Locale>`. These settings determine the language used in the GUI of the *Site Manager*. The locale that you set in `editor-startup.xml` will be used for the Login screen you can overwrite this setting with a `<Locale>` element in the `editor.xml` file. So you can define group specific localizations for example.

Example 3.1. Example for the Locale element

```
        <Editor>
        <Locale language="de" country="DE"/>
        .
        .
        .
</Editor>
```

Table 3.2. Attributes of element <Locale>

Attribute	Description
language	The language used in the program. At present, there are locales for English ("en") and German ("de"). The locales follow the usage in `java.util.Locale`.
country	Country-specific settings. At present, there are locales for the United States ("US") and Germany ("DE"). The locales follow the usage in `java.util.Locale`.

3.3 Starting the Editor

As a developer you will start the *Site Manager* from the workspace using Maven as follows:

```
mvn install
cd  editor-components\editor\target\editor
bin\cm.exe editor
```

A *Site Manager* that is installed under Windows can be started directly via the link in the menu setup during installation.

If you don't use Java Web Start and multiple LAN connections are active under windows you can select the IP address of the LAN connection with which the *CM Editor* should communicate. For this, use the Java Virtual Machine Parameter `ooc.boa.host=<ip-address>` in the file `editor.jpif`.

If the editor should start under Unix for administration purposes, the following must be entered in the `bin` directory of the CoreMedia installation:

```
cm editor &
```

3.4 Defining XML Files For Configuration

The *Site Manager* can be configured using different XML files. The names of these files are defined in the `editor.properties` file. The files are a small bootstrap file (`editor.startup.configuration`) necessary for some settings before the actual start of the editor, and the XML files for the customization of the editor. The following use cases are supported:

- Everyone uses the same configuration file. This file is defined by the property `editor.configuration`.
- Everyone uses the same common configuration file with additional group (`group.configuration`) and/or user (`user.configuration`) specific configuration files.
- Each group has its own configuration files defined by `group.configuration` and the `<ConfigGroups>` element in the editor startup file. Additional user specific configuration files can be used. The groups might share configuration files but they do not have to.

When a user is member of more than one group, for which specific configuration files exist, then the system chooses an arbitrary group for which the configuration file is taken. So, it is good practice, to have only one group for each user for which a configuration file exists. If this is no option, you can define your own selection scheme as described in this manual.

The files are evaluated in a specific order:

No <ConfigGroups> element used

1. Bootstrap file defined by `editor.startup.configuration`,
2. Common configuration file defined by `editor.configuration`
3. Group specific files defined by `group.configuration`
4. User specific files defined by `user.configuration`.

<ConfigGroups> element used

1. Bootstrap file defined by `editor.startup.configuration`,
2. Group specific files defined by `group.configuration` and in `<ConfigGroups>` in the `editor-startup.xml` file (see Section 3.5, “Defining Group Specific Configuration Files” [29] for details).
3. User specific files defined by `user.configuration`.

Be aware, that you need to define a file in `editor.configuration` in both cases, even if it will not be used in the second case, otherwise an error occurs.

Example for a configuration without a <ConfigGroups> element

You have made the following settings in `editor.properties`:

```
      editor.startup.configuration=editor-startup.xml
editor.configuration=editor.xml
group.configuration=editor-group-{0}.xml
user.configuration=editor-user-{0}.xml
```

A user named "Axel" who is only member of the group "editors" logs in and the following configuration files are applied in the shown order:

1. `editor-startup.xml`
2. `editor.xml`
3. `editor-group-editors.xml`
4. `editor-user-Axel.xml`

Example for a configuration with a <ConfigGroups> element

You have made the following settings in `editor.properties`:

```
      editor.startup.configuration=editor-startup.xml
editor.configuration=editor.xml
group.configuration=editor-group-{0}.xml
user.configuration=editor-user-{0}.xml
```

And these settings in `editor-startup.xml`:

```
      <ConfigGroups>
   <ConfigGroup name="editors">
      <Configuration name="common"/>
      <Configuration name="editor"/>
   </ConfigGroup>
</ConfigGroups>
```

The same user as before logs in and the following configuration files are applied in the shown order:

1. `editor-startup.xml`
2. `editor-goup-common.xml`
3. `editor-group-editor.xml`
4. `editor-user-Axel.xml`

How files merge

All elements which can occur only once - due to the `coremedia-editor.dtd` - will be overwritten by the settings of the succeeding configuration file. The following elements use inheritance:

- `Bundle` and `Explorer` elements will be added.
- Existing `Document` and `Process` definitions of `Documents` and `Processes` elements, will be overwritten, new definitions will be added.
- Existing `DocumentType` definitions of `DocumentTypes` elements use inheritance on the `PropertyType` element level. That is, existing `PropertyType` definitions (for example a `Validator` set for the property Name) will be overwritten and new definitions will be added.

Example:

The interesting parts of the `editor.xml` look as follows:

```
      <Bundle name="first/bundle"/>
<SpellChecker enabled="true"/>
<Documents>
  <Document type="Article">
    <Property name="Headline" editorClass="FirstClass"/>
  </Document>
</Documents>
<DocumentTypes>
  <DocumentType name="Article">
    <PropertyType name="Editor">
      <Validator class="NotEmpty"/>
      <Initializer class="SetChiefEditor"/>
    <PropertyType/>
  </DocumentType>
</DocumentTypes>
```

The interesting parts of the group specific editor definition look as follows:

```
      <Bundle name="second/bundle"/>
<SpellChecker enabled="false"/>
<Documents>
  <Document type="Image">
    <Property name="Caption" editorClass="SecondClass"/>
  </Document>
</Documents>
<DocumentTypes>
  <DocumentType name="Article">
    <PropertyType name="Editor">
      <ModelClass class="MyModel"/>
      <Initializer class="SetEditor"/>
    <PropertyType/>
  </DocumentType>
</DocumentTypes>
```

Applying both editor definitions will result in the following behavior:

- Both bundles will be used.
- The spell checker will be disabled.
- `Article` documents will use the property editor `FirstClass` with the property `Headline` and `Image` documents the property editor `SecondClass` with the property `Caption`.
- `Article` documents will use the validator class `NotEmpty`, the initializer class `SetEditor` and the model class `MyModel` with the property `Editor`.

See Section 5.2, “Configuration Possibilities in the XML Files” [112] for a detailed description of the properties.

You can either use locally stored files for each client or administrate these files centrally on the *Content Server* and deliver them to the clients.

- For locally stored files adding the path relative to <CoreMediaHome>.
- For centrally stored files adding the URL to the Content Server as described below.

Not mandatory: You only need to do this configuration if you want to use group or user specific configuration files or if you want to administrate the files centrally on the *Content Server*. Otherwise, the following files in the folder `<CoreMediaInstall>/properties/corem` are used by default:

- `editor-startup.xml`
- `editor.xml`

Configure in the following file:

- `editor.properties`

Defining the XML configuration files

1. Enter the name of the XML file with the path relative to `<CoreMediaHome>` into the appropriate properties.

- `editor.startup.configuration`

→ `editor.configuration`

Defining user or group specific configuration files

1. Enter the name of the XML files with the path relative to `<CoreMediaHome>` into the appropriate properties.

   → `group.configuration`

   → `user.configuration`

   You must add the wildcard {0} to the name of the XML file. This wildcard will be replaced by the group name or the user name of the user. If the user belongs to multiple groups, the system will choose an arbitrary group.

2. If required configure the `<ConfigGroups>` element in the editor startup file as described in Section 3.5, “Defining Group Specific Configuration Files” [29].

Example:

→ `group.configuration=http://localhost:44441/coremedia/files/properties/corem/editor-{0}.xml`.
The user belongs to the groups Editors and CvD. Thus, the files with the URL `http://localhost:44441/coremedia/files/properties/corem/editor-Editors.xml` and `http://localhost:44441/coremedia/files/properties/corem/editor-CvD.xml` will be loaded in an arbitrary order.

Note, that you can define multiple configuration files for a specific group, which will be evaluated in a specific order. See the *Site Manager Developer Manual* for a description of the XML element `<ConfigGroups>` for details.

Administrating the XML files on the server

1. Store the XML files in the `properties/corem` directory of the *Content Server*.
2. Enter the following URL into the configuration property of the XML file in the `editor.properties` file:

   `http://<ServerHost>:<ServerPort>/coremedia/files/properties/corem/<editor-xmlfilename>`.

   Replace `<ServerHost>` with the host name of the *Content Server* and `<ServerPort>` with the port of the *Content Server*.

3. Enter the names of the XML files in the `properties/policy/server.policy file` of the server in order to allow the server to deliver the files. The entry must look as shown in the next code example:

Example 3.2. Add files to server.policy

```
        grant codeBase "http://localhost/servlets/fileservlet" {
    .
    .
    permission java.io.FilePermission
    "properties${/}corem${/}editor.xml", "read";
    permission java.io.FilePermission
    "properties${/}corem${/}editor-CVD.xml", "read";
};
```

3.5 Defining Group Specific Configuration Files

The *Site Manager* is configured with XML files. It is possible to define special configuration files for distinct groups or users of the CoreMedia system. To configure the usage of special configuration files you may adapt the following properties in the `editor.properties` file (see Section 3.4, "Defining XML Files For Configuration" [23] for details):

- `editor.startup.configuration`
- `editor.configuration`
- `group.configuration`
- `user.configuration`

If you only use `group.configuration`, you can define one specific configuration file for each group. To have multiple configuration files for one group, you may configure the set of files and in which order they are parsed in `editor-startup.xml` (default) or in the file configured by `editor.startup.configuration`. Mind that group configuration in `editor-startup.xml` overrides the mechanism one configuration file per group which especially means: If users are not member of any group configured in `<ConfigGroups>` no group configurations are applied to these users.

In both cases, that is either with one configuration file per group or with multiple configuration files per group you have to set the property `group.configuration` to point to configuration files with a path relative to `<CoreMediaHome>` or to the URL where to find the files. The path/URL defined has to contain a wildcard `{0}` which will be replaced either by the group name or by the names as defined in the `<Configuration>` element (see below).

Example:

```
group.configuration=properties/corem/editor-{0}.xml
```

The *Content Server* will look in the `properties/corem` directory for a file called `editor-<PlaceHolder>.xml` where `<PlaceHolder>` will be replaced by the values of the `name` attribute of the `<Configuration>` element described below or by the group name if no `<ConfigGroups>` element is used.

If a user is member of more than one group, the exact behavior reading group configuration files is undetermined. If multiple matching `<ConfigGroup>` exist, one of them is chosen by random. If `<ConfigGroups>` configuration is not used but direct mapping groups to configuration files all matching configuration files are read but in an undetermined order. To determine the exact behavior you have to implement your own selection scheme. Proceed as follows:

1. Extend `GenericEditor`
2. Override the `getConfigurationGroupNames(UserModel user)` method which is inherited from `AbstractEditor` with your own selection scheme. The default implementation of the method either returns the configuration file names as configured in the `<Configuration>` element (first case) and if no `<ConfigGroups>` element is used the unordered list of groups a user is member of. You might want to use the convenience method `getUserConfigGroups(UserModel user)` to create your own implementation. For further reference see the Javadoc.
3. Add your class to the `class` attribute of the `<Editor>` element in the `editor-startup.xml` file.

<ConfigGroups>

Child elements: `<ConfigGroup>`

Parent elements: `<Editor>`

```
<Editor>
  <ConfigGroups>
     .
     .
  </ConfigGroups>
</Editor>
```

This element combines the elements for the group configuration.

The element has no attributes. If `<ConfigGoups>` is not used but `group.configuration` is set, only the general editor configuration file (default: `editor.xml`) and the matching group specific configuration files will be applied. See the *Site Manager* chapter in the *Administration and Operations Manual* for details.

<ConfigGroup>

Child elements: `<Configuration>`

Parent elements: `<ConfigGroup>`

```
    <ConfigGroups>
  <ConfigGroup name="editor" domain="main">
     .
     .
  </ConfigGroup>
</ConfigGroups>
```

This element defines for which group and domain the configuration should be used. It groups the `<Configuration>` elements.

Attribute	Description
name	The name of an existing group in the CoreMedia user management for which the configuration will be used.
domain	The domain of the group.

Table 3.3. Attributes of the <ConfigGroup> element

<Configuration>

Child elements:

Parent elements: `<ConfigGroup>`

```
      <ConfigGroups>
   <ConfigGroup name="editor">
      <Configuration name="common"/>
      <Configuration name="special"/>
   </ConfigGroup>
</ConfigGroups>
```

This element defines the name with which the placeholder in `group.configuration` will be replaced and the order in which multiple configuration files are applied. In the example above the placeholder will first be replaced with "common" and then with "special", if the user is member of the "editor" group. This especially means that in case of conflicting settings the settings from the special file will override the settings in the common file.

Attribute	Description
name	Name which will replace the placeholder in the `group.configuration` property of `editor.properties`. In general, this is not the name of an existing group, but it can be.

Table 3.4. Attribute of the <Configuration> element

3.6 Configuration Using coremedia-richtext-1.0.css

The *Site Manager* offers a lot of formatting options in the *RichText pane*. Nevertheless, you might want to have your own formatting options ready to hand. For this purpose, it is possible to extend the existing options and to add new ones. To do so, three steps are sufficient that will be described in the subsequent paragraphs:

1. Extend the file `coremedia-richtext-1.0.css` to your needs as described in Section 3.6.2, “Extend the coremedia-richtext-1.0.css file” [33].
2. Localize the new styles and style groups in a bundle file (optional) as described in Section 3.6.3, “Localize the New Styles and Style Groups” [34].
3. Add the new style groups to your *Site Manager* as described in Section 3.6.4, “Add to Content Editor” [36].

The formatted inline text or block element will be marked as follows:

- **inline text**

  Inline text will be embedded in the `<span>` tag with the attribute `class` containing the format option. Example:

  ```
  <span class="font-size--36">My big text</span>
  ```

- **block element**

  Block elements will have an additional class attribute containing the format information. Example:

  ```
  <p class="background-color--black">My test paragraph</p>
  ```

3.6.1 Supported CSS Attributes

As opposed to the usual browsers the CSS support in Java/Swing is limited and does not cover the complete CSS attributes. Therefore, Java/Swing will display some CSS attributes flawed or not at all. See the Javadoc of the `javax.swing.text.html.CSS` class of your used JVM for the supported CSS attributes.

Because of these limitations you should always use the preview function of the *Site Manager* for a check of the used CSS styles.

Don't merge the attributes in the `coremedia-richtext-1.0.css` file. For example don't write "border: solid 1px red" but "border-style:solid; border-width: 1px; border-color: red;". If you edit `coremedia-richtext-1.0.css` with tools like Microsoft Frontpage, take care that the attributes are not merged on saving.

3.6.2 Extend the coremedia-richtext-1.0.css file

The aim of the `coremedia-richtext-1.0.css` file is twofold. First, it defines the look of the XML elements in the RichText pane according to the CSS definitions (see http://www.w3c.org), but secondly it is used as the definition of the possible attributes.

The RichText pane supports only a subset of CSS.

You can use well known CSS syntax to define your own style groups and styles. Nevertheless, the RichText pane of the *Site Manager* does not support the display of all possible CSS formats. Some formats will be displayed in a WYSIWYG style (such as bold, italic, understroke ...) others will be displayed symbolic (color, value of a Style group etc.). The actual layout of the text depends on the definitions and structure of the generated website and can only be seen in the HTML preview of the browser.

Getting the file and add it to the editor

The `coremedia-richtext-1.0.css` file is included in a JAR file. So in order to add your extensions you have to get the file and put the changed file to a new location. Proceed as follows:

1. Build the `editor-components` module in the development workspace of *CoreMedia Project*.
2. Extract the `coremedia-richtext-1.0.css` file from the `cap-editor-resources.jar` file in the `target/../lib` directory.
3. Customize the file to your needs.
4. Put the file into the `properties/css` directory of the `editor` module.
5. Configure the property `editor.richtext.css.location` in `editor.properties` to the `properties/css/coremedia-richtext-1.0.css` location in order to override the default CSS file.

Define new style groups

A style group is a list of CSS style classes, that share a common prefix ending with "--". For example, `[font-name--arial, font-name--times]` is a style group `font-name` consisting of the two styles classes `font-name--arial` and `font-name--times`. You can limit the usage of your style group to single elements by adding the name of the element in front of the style group separated by a dot.

Due to limitations in the Swing CSS class use only lower case letters for the names of style groups and style classes.

Simply add the new StyleGroup to the `coremedia-richtext-1.0.css` file following the naming pattern given above. The following example adds a new *style group* `inlineformat` with the two style classes `code` and `glossary` to the CSS file. Text marked as code will be displayed with red background, text marked as glossary with blue background.

```
.inlineformat--code { background-color: red; }

.inlineformat--glossary { background-color: blue; }
```

Define new style classes

The style classes are defined as described above. Simply add an entry following the scheme:

```
<LimitedElement>.<StyleGroupName>--<StyleClassName> { <layout
definition> }
```

Define free text style group

If you want to define a style group without predefined style classes where the user can enter own text proceed as follows:

1. Define a style group without style classes, for example:

   ```
   .freetext {background: blue;}
   ```

2. Add the style group to the *Content Editor* as explained in Section 3.6.4, “Add to Content Editor” [36].

This style sheet group will only appear in the attribute editors but not in the tool bar.

3.6.3 Localize the New Styles and Style Groups

After defining the new style group and style classes you can localize it for use in the *Content Editor*. Follow these steps:

1. Add a Bundle element to the `editor-startup.xml` file. The Bundle element defines the localization file to use.
2. Create the localization file in the resource section of a JAR module, for example in the `editor-customizations` module of the developer workspace.
3. Enter the localization entries.
4. Restart the *Content Editor*.

If you do not localize your entries, the names will be taken from the CSS file.

Please read Section 4.10, “Localization” [73] for more details on localization.

Enter the localization entries

The following keys can be used to localize your StyleGroup:

- `richtext-stylesheet-<StyleGroupName>Label=`<Label of the StyleGroup used in menus>
- `richtext-stylesheet-<StyleGroupName>ToolBarLabel=`<Label of the StyleGroup in the tool bar, usually empty, since icons are used mostly>
- `richtext-stylesheet-<StyleGroupName>Mnemonic=`<Key Mnemonic for the StyleGroup>
- `richtext-stylesheet-<StyleGroupName>ToolTip=`<Tooltip shown on mouse over>
- `richtext-stylesheet-<StyleGroupName>Image=`<URL of a small (16*16) icon used in menus>
- `richtext-stylesheet-<StyleGroupName>ToolBar3Image=`<URL of a small (16*16) icon used in the tool bar>

The following keys can be used to localize each style option:

- `richtext-stylesheet-<StyleGroupName>--<StyleOption>Label=`<Label of the style option>
- `richtext-stylesheet-<StyleGroupName>--<StyleOption>ToolTip=`<Tooltip of the style option>
- `richtext-stylesheet-<StyleGroupName>--<StyleOption>Image=`<URL of a small (16*16) icon used for the style option>
- `richtext-stylesheet-<StyleGroupName>--removeClassAttributeValueLabel=`<Label of the style option which removes the formatting, usually "---" is used>
- `richtext-stylesheet-<StyleGroupName>--removeClassAttributeValueToolTip=`<The tool tip of the remove option>

Example:

Here you see the localization of the `inline-format` style group with the `code` style option defined previously. The image options have been omitted.

```
richtext-stylesheet-inlineformatLabel=Text format
richtext-stylesheet-inlineformatToolBarLabel=
richtext-stylesheet-inlineformatMnemonic=i
richtext-stylesheet-inlineformatToolTip=Selects special options for inline text
richtext-stylesheet-inlineformat--codeLabel=Code
richtext-stylesheet-inlineformat--codeToolTip=Formats text as code
```

- `richtext-stylesheet-inlineformat--removeClassAttribute ValueLabel=---`
- `richtext-stylesheet-inlineformat--removeClassAttribute ValueToolTip=Removes inline formatting`

In Figure 3.1, “The localized style group” [36] you can see the result of the localized style group.

Figure 3.1. The localized style group

3.6.4 Add to Content Editor

If you have defined your style groups, you need to add them to the *Site Manager*. The right place is the `editor.xml` file using the attribute `addStyleSheetGroups` of the `Property` element. The following example shows how to add your newly defined style group `inlineformat` and the predefined style group `list-item` to the property `RichText` of the "Article" document:

Example 3.3. Add the style group to the editor

```
          <Documents>
  <Document type="Article">
    <Property name="RichText" editorClass="RichTextPane"
     addStyleSheetGroups="inlineformat(inline) list-style(ol,ul)
     freetext:string(inline)">
      <Toolbar>
        <StyleGroup name="inlineformat" show="button"/>
      </Toolbar>
    </Property>
  </Document>
.
.
</Documents>
```

After you have restarted the *Content Editor* the style group `inlineformat` will appear in the formatting button of the toolbar and the tab *Inline text* of the attribute editor and the style group `list-style` will appear in the *List* tab of the attribute editor.

The attribute addStyleSheetGroups

As you can see from the example above, multiple style groups can be added to the `addStyleSheetGroups` attribute, simply separated by blanks. The element names in the brackets are optional. They limit the offering of the style group to the elements in brackets. Therefore, the example style group `inline-format` will only be offered for inline text but not, for example, for paragraphs and the style group `list-style` will be offered to bullet and numbered lists. You can use the `p`, `table`,

`tr`, `td`, `li`, `ul`, `ol`, a and `img` elements of the `coremedia-richtext-1.0.dtd` file and the additional keywords `inline`, `block` or `flow`.

- `inline`: The style group will be offered for all inline elements.
- `block`: The style group will be offered for all block elements.
- `flow`: The style group will be offered for all elements.

If you use one of these additional keywords, you cannot add any other element or keyword to the style group. For example `inline-format(ul,block)` is forbidden.

If you want to add a style group for free text you have to add the attribute `string` or `identifier`:

- `string`: Enter free text.
- `identifier`: Enter free text with characters which are valid for very strict CSS identifiers.

`freetext:string(inline)`, for instance

The elements Toolbar and StyleGroup

You can use the elements `Toolbar` and `StyleGroup` to configure the appearance of the style groups in the toolbar of the RichTextPane. You can define for each style group,

- to be displayed as a combo box (default),
- to be added to the A button,
- not to be shown in the toolbar.

Add the element `Toolbar` as a child element of the element `Property` and the element `StyleGroup` as a child of `Toolbar`. `StyleGroup` has the following two attributes:

Table 3.5. Attributes of StyleGroup element

Attribute	Description
name	Name of the style group to configure.
show	How to display the style group. The following options exist: - `button`: Add style group elements to the A button. - `combobox`: Add as a combo box to the toolbar.

Attribute	Description
	→ `false`: Do not show style group in toolbar.

3.7 Configuring the Struct Editor

The Struct editor offers a convenient way to edit Struct properties. The Struct editor comes in two flavors, one with tabs to switch between an XML view and a formatted view and the default editor only offering the formatted view.

You have to configure the editors in the `editor.xml` file. In order to configure the editors, simply add the appropriate class to the `editorClass` attribute of the `<Property>` element

Editor	Class
Default Struct editor	`hox.corem.editor.toolkit.property.StructRichTextPane`
Tabbed Struct editor	`hox.corem.editor.toolkit.property.TabbedStructPropertyEditor`

Table 3.6. Editor classes

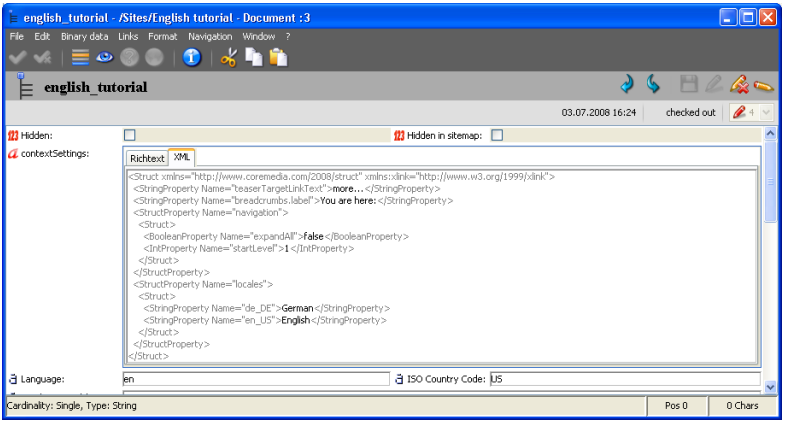


Figure 3.2. The tabbed Struct editor

3.8 Disable Workflow

In some cases, users are not required to use workflows. You can completely hide all references to the workflow component in the editor by setting the attribute `enableWorkflow` of the `<Editor>` element.

Example 3.4. Disabling the workflow in the editor.xml

```
        <Editor
  enableWorkflow="false">
        .
        .
        .
</Editor>
```

Table 3.7. An attribute of the element Editor

Attribute	Description
`enableWorkflow`	If "false", removes all references to the workflow component from the GUI and ensures that no connection to the workflow server is established. Default is "true".

3.9 Enable Direct Publication

By default, the approve and publication buttons are only enabled for users of the administrator(0) group. You can enable the feature for all users by setting some attributes of the `<Editor>` element, thereby supporting publications without workflows.

Example 3.5. Disabling the workflow

```
        <Editor
  enableDirectPublication="false">
        .
        .
        .
</Editor>
```

Table 3.8. An attribute of the element Editor

Attribute	Description
`enableDirectPublication`	If "true", ensures that the buttons for approval, disapproval, publication, and publication preview are displayed as buttons and included as menu items in all relevant windows for all users. Default is "false".

3.10 Define the Browser for Web Extensions

You can define the browser for the web extensions using the element `<WebBrowsers>`. Multiple browsers for different locales and operating systems can be defined. The browser for the preview can be chosen from the **File|Preview** menu of the overview window. If you do not define any browser, the web extensions cannot be executed.

‹WebBrowsers›

Child elements: `<WebBrowser>`

Parent elements: `<Editor>`

```
<Editor>
  ..
    <WebBrowsers>
      ..
    </WebBrowsers>
..
</Editor>
```

You can use the `<WebBrowsers>` element to configure web browser definitions for Web Extensions such as the preview with the `<WebBrowser>` child element. The `<WebBrowsers>` element has no attributes.

‹WebBrowser›

Parent elements: ‹*WebBrowsers*›

```
      <WebBrowsers>
  <!-- Standard Windows IE installation -->
  <WebBrowser id="Internet Explorer" os="win"
  command="c:\\Program Files\\Internet Explorer\\Iexplore.exe %s"/>


  <!-- IE installation in german locale on Windows -->
  <WebBrowser id="Internet Explorer" os="win" language="de"
  command="c:\\Programme\\Internet Explorer\\Iexplore.exe %s"/>
</WebBrowsers>
```

This element configures web browser installations for a given locale of the *Site Manager* and operating system. Web extensions (see `<WebExtension>`) may open several web browsers (Preview) or the first matching web browser. Therefore, the order of `<WebBrowser>` elements is important.

The example above configures two Windows web browsers, one with language attribute set to ‘de’. If a web extension running on German locale wants to select

a browser, it should open the German browser. A precedence list defines which browser is selected.

1. `os`
2. `language`
3. `country`
4. no attribute

In the example above, for both browsers the `os` attribute has been set but the German browser is selected because it has a `language` attribute that matches the language of the German locale. If you delete the `os` attribute in the German browser configuration, the other browser will be opened.

In rare conditions a matching browser can not be opened. Take, for example, the configuration above and call a preview web browser from a *Site Manager* with a German locale on a French Windows system. The command `c:\\Programme\\Internet Explorer\\Iexplore.exe %s` can not be executed on the French system because "Programme" will not be found. In this case, the first browser is taken that can be opened, independently of any `os` or `language` settings.

Table 3.9. The attributes of the <WebBrowser> element

Attribute	Description
`id`	The name of the browser, such as Internet Explorer. Use the same id for the same browser application, like FireFox for all Firefox configurations.
`os`	The name of the operating system or a prefix thereof. It must equal or be a prefix of the Java system property `os.name`. This attribute is optional. If not set, the command must be executable on all operating systems your *Site Manager* runs on.
`language`	The language of the locale. The value must conform to a valid language in a Java `java.util.Locale` instance. For the English language the valid value is 'en' for the German language the valid value is 'de'. This attribute is optional.
`country`	The country of the locale. The value must conform to a valid country in a Java `java.util.Locale` instance. For the USA the valid value is 'US' for Germany the valid value is 'DE'. This attribute is optional.
`command`	The command to start a browser with a given URL on the configured operating system. For the Internet Explorer on an English Windows installation the command looks as follows: `c:\\Program Files\\Internet Explorer\\Iexplore.exe %s`

Attribute	Description
	The suffix `%s` is the placeholder for the URL to load in to the browser.
`optional`	Specifies whether this browser is optional. This feature is used by the Preview web extension when doing a preview with all configured browsers (for example by clicking the Preview button in the toolbar or by selecting **File|Preview|All**). The *Site Manager* only shows errors for non-optional browsers or if no browser could be started at all. Allowed values are true and false. Default is false

3.11 Enable the Spell Checker

You can activate the *Spellchecker* using the `<Spellchecker>` element in the `editor.xml` file.

<Spellchecker>

Child elements:`<MainDictionary>`, `<CustomDictionary>`

Parent elements: `<Editor>`

Example 3.6. Example of a Spellchecker element

```
      <Editor>
  [...]
  <SpellChecker enabled="false" />

  <SpellChecker enabled="true" os="Windows">
    <MainDictionary class=
       "com.coremedia.spellchecker.Bridge2JavaWordDictionary"/>

    <CustomDictionary class=
       "hox.corem.editor.spellchecker.Dictionary"/>
  </SpellChecker>

  [...]
</Editor>
```

If you have activated the *CoreMedia Spellchecker* on your computer, and you want to use Word afterwards, you have to start Word with the "/w" option. Otherwise, all macros contained in a Word document that you want to edit later, would be executed without any warning.

This is because the *CoreMedia Spellchecker* starts a Word instance from an automation client for spell checking. This disables the macro security settings of Word. To circumvent this security problem, you have to start word with the "/w" option (see http://support.microsoft.com/kb/210565 for details).

Table 3.10. Attribute of the element SpellChecker

Attribute	Description
`enabled`	This attribute determines whether the spellchecker should be used ("true") or should be disabled ("false"). By default, "false" is used.
`os`	Restricts the configured spellchecker to a given operating system. The value is compared to Java's system property `os.name`. Common value is for example `Windows`. The configuration with the best (that is, longest) match wins. So for example if you have a spellchecker configured with `os="Windows"` and another with `os="Windows 7"` and you are running on Windows 7 the second one will be taken. Default is to match

Attribute	Description
	all operating systems. So the example above says: Disable the spellchecker on all operating systems but on Windows.

3.12 Troubleshooting

The Site Manager does not start under Windows.

Possible cause:

When redirecting the log outputs to a file (`OUTPUT_REDIRECT=log`), under Windows it can occur that the *Site Manager* does not start. The reason for this can be another process (*Site Manager*, Text Viewer or similar), which has opened the log file in the meantime. Under Windows a file can only be opened by one process at any one time. The *Site Manager* does not start, when it cannot open and write to its log file.

Possible solutions:

a) Close the program which is accessing the log file of the *Site Manager*. Restart the *Site Manager*.

b) If it cannot be determined which program is accessing the log file, the user must log off the Windows system and log in again. Afterwards, the *Site Manager* can be restarted.

After installation of the Site Manager on the client computer, only an empty root directory appears after logging in.

Possible causes:

a) No subfolders have been set up.

b) The server cannot reach the client computer. The server (as well as the client) must be able to resolve the name of the computer to contact in the network.

Possible solutions:

a) Set up subdirectories.

b) If you are using DNS, the correct client computer name must be entered. After this, check that the local client computer name (machine name) matches the client computer name entered in the DNS.

If you are not using DNS, the computer name of the client must be entered in the file `/etc/hosts` on the server. Correspondingly, the server name must be entered under `$WINNT/system32/drivers/etc/Hosts` on the client.

Publishing resources apart from workflow

In case of emergency it might become necessary to publish documents apart from a workflow. As administrator, you can publish as follows:

→ Select **Publication** from the **File** menu.

- Click on the publication symbol.
- In the Overview window or in the Publication window, select **Publish** from the context menu. The document has been published. The status is shown by the symbol for 'published'.

I cannot write to the Word dictionary/the spell checker does not use my user dictionary for suggestions

It's not possible to write from the *Site Manager* to the Word user dictionary. Whether Word uses the Word user dictionary or not depends on the configuration of Microsoft Word and can not be influenced by the *Site Manager*. Spelling suggestions are computed by Word only using Word dictionaries.

Cross-language installations

If OS and Microsoft Office have been installed in different language versions, this might cause errors when using the spell checker with the *CM Editor.*

Example: Your computer runs on a German Windows, your Office application is an English one - as are your Site Manager and the language of your CoreMedia project.

Explanation: The default language had been set to German - no matter if dictionary nor grammar available - automatically due to the user profile of the German operating system. This problem can be abstracted for any cross-language installations, so proceed in analogy for other languages. It is a genuine Microsoft problem, likely to occur with both Windows XP and Windows 2000.

Solution: It is not sufficient just to change the **Language Settings** within your Office application. You have to go **Programs | Microsoft Office Tools | Microsoft Office Language Settings**. Set the default language to English and remove German from the list of enabled languages. This does not change the installed Office components themselves.

The Site Manager does not start under Windows because `msvcr100.dll` is missing

Possible cause:

When you start the *Site Manager* by executing the `cm editor` command in the `bin` directory, and use Java 7 32-bit it can occur that the *Site Manager* does not start. The reason for this is that the file `msvcr100.dll` is missing from your computer.

Possible solutions:

a) Install the *Microsoft Visual C++ 2010 Redistributable Package (x86)* on your computer.

b) Replace the provided `msvcr100.dll` in the `bin` directory of the *Site Manager* with `msvcr100.dll` 32-bit.

3.12.1 Taking a Thread Dump

When the *Site Manager* does not respond, or is performing poorly, you can take a thread dump to identify the problem or attach it as a file to a support ticket.

There are three options to take a thread dump depending on how the *Site Manager* was started and what version of Java you are running.

- Point your web browser to the URL: `http://localhost:<port>/coremedia/threaddump` (the default port is 44444). This generates a thread dump in the browser window. You can copy and paste it into an email for example. Additionally, the thread dump is written to a file in your user home directory. The filename matches the pattern `threaddump.editor.<timestamp>`. This file can be attached to a support ticket to help determine the cause of the problem. For this option the minimum required Java version is JRE 5.0 and the remote control of the *Site Manager* must be enabled but it is independent of how the editor was started.

- When you started the editor via `bin/cm editor` in the `bin` directory of the CoreMedia installation, you can also take a thread dump by pressing Ctrl-Break (Windows) or Ctrl-\ (Unix, Linux, Mac OS X) in the console you used to start the editor. A thread dump will be generated in the console.

- When you started the *CoreMedia Editor* via the web browser using Java Web Start and the Java Console window is enabled, (on Windows, this can be done by going to the Windows Control Panel, and double-clicking the Java icon, then enabling the Java Console on the Advanced Tab) you can press 'v' in the Java Console window. A thread dump will be generated in the console. For this option the minimum required Java version is JRE 6.0.

4. Programming and Customization

Components to customize

This chapter deals with the customization of the *Site Manager* by programming own extensions. You will find chapters covering the following topics:

- Tasks useful for different customization
- Validators
- Initializer
- Language resolvers
- Property editors
- Predicates
- Implement commands
- Add buttons to the toolbar
- Add menu items to menus

Please refer to the *Site Manager API* for more details on the classes described in the following sections. For workflow issues you may find more information in the *CoreMedia Developer Manual*.

The [CoreMedia DXP 8 Manual] provides further information how to configure the editor as a part of *CoreMedia Project*. You will find an example integration of the *Site Manager* in the development workspace. Start with the `editor-components` module in the development workspace.

4.1 How To ...

This chapter describes tasks that can be used for different customizations of the editor.

4.1.1 How To Access Arbitrary Resources

Question:

How can I access arbitrary resources?

Answer:

1. Get the root folder of the repository:

→ `FolderModel rootFolder=Editor.getEditor().getResourceFactory.getRootFolder();`

→ This returns a `FolderModel` which can be used for further processing.

2. Get the resource you are looking for:

→ `ResourceModel myResource=rootFolder.pathLookup(new String[]({"myFolderName", "myDocumentName"}));`

or

`ResourceModel myResource=rootFolder.pathLookup(new String("myFolderName/myDocumentName"));`

→ This will return a `ResourceModel` of the searched resource or null if the resource does not exist.

4.2 Program Own Initializers

Initializers fill the fields of a newly created document with default values.

Interface to implement

For own initializers you need to implement the interface hox.corem.editor.initialization.Initializer with the method `getInitialValue`.

Parameters to use

The `getInitialValue` method gets the following parameters:

Parameter	Type	Description
document	DocumentModel	The document which contains the property to initialize.
`propertyType`	`PropertyTypeModel`	The property to initialize.

Table 4.1. Parameters of the getInitialValue method

Return Type

Depending on the property type for which the initializer is intended to use the method may return the following values (return type is `Object`):

Property	Default PropertyModel	Value
String	`StringModel`	String
Integer	`IntegerModel`	Integer
Date	`CalendarModel`	Calendar
LinkList	`LinkListModel`	`ResourceHolder[]`
SgmlText	`SgmlTextModel`	`org.w3c.Document`
Blob	`BlobModel`	byte[]

Table 4.2. Return values of the getInitialValue method

Integrate your Initializer into the Site Manager using `editor.xml`

You can integrate your Initializer into the *Site Manager* using the element `Initializer` of the `editor.xml` file as shown in example.

```
<DocumentTypes>
 <DocumentType name="SomeType">
  <PropertyType name="SomeProperty">
   <Initializer
   class="com.customer.example.editor.SimpleInitializer"/>
  </PropertyType>
```

Example 4.1. Integrate Initializer in editor.xml

```
 </DocumentType>
</DocumentTypes>
```

Example:

The next example shows a simple Initializer which checks whether the property to initialize is of type String or not. If it is, the property will get the name of the creator of the document as the initial setting.

Example 4.2. Example of an Initializer

```
import hox.corem.editor.initialization.Initializer;
import hox.corem.editor.proxy.*;
public class SimpleInitializer implements Initializer {
 public Object getInitialValue(DocumentModel doc,
  PropertyTypeModel p) {
  String result = "";
  try {
   int ver = doc.getLatestVersion();
   PropertyModel pm = doc.getPropertyModel(ver, p.getName());
   if (pm instanceof StringModel) {
    result = doc.getCreator().getName();
   }
  } catch (Exception e) { }
  return result;
 }
}
```

4.3 Program Own Validators

Validators are used to check the values of document properties at check-in time. If the validator throws a ValidationException, a window pops up which shows a message containing the exception message.

Interface to implement

Own validators must implement the interface hox.corem.editor.validation.Validator2 with the method `validate`. Validator2 replaces the deprecated Validator interface which was memory consuming when it comes to the validation of Blobs.

Parameters to use

The `validate` method gets the following parameters:

Table 4.3. Parameters of the validate method

Parameter	Type	Description
document	Document-Model	The document with the property to validate. The Document-Model is for read access only, do not try to modify any document here
`propertyType`	`PropertyTypeModel`	The property type of the property to validate
value	Object	The value of the property, not a PropertyModel but the value of the PropertyModel (see table below for the default types)
`allProperties`	Map	A map indexed by property name of all the documents properties

Return types

The method returns a value. The type depends on the property which has been validated.

The properties have the following types:

Table 4.4. Default types of the properties

Property	Default PropertyModel	Value
String	`StringModel`	String
Integer	`IntegerModel`	Integer
Date	`CalendarModel`	Calendar
LinkList	`LinkListModel`	Object[] containing ResourceHolder objects

Property	Default PropertyModel	Value
XmlText	`XmlPropertyModel`	`org.w3c.Document`
Blob	BlobModel	`hox.corem.editor.proxy.BlobValue`

General hints

- Use the SimpleValidationException instead of ValidationException since ValidationException is abstract. The constructor takes parameters for error messages and hints for problem resolution, these parameters are in fact property names for the property file bundles (see the [CAP Editor API] for details).
- Don't try to change any documents in the validator. Violating this rule may lead to deadlocks, inconsistent states, swallowed events etc. since the check-in locks the proxy, which prevents events from the server to be processed. Thus, you see inconsistent states in your validator.

Integrate your validator into the Site Manager using `editor.xml`

You can integrate your validator into the *Site Manager* using the element `Validator` of the `editor.xml` file as shown in Example 4.3, "Integrate validator in editor.xml" [55].

Example 4.3. Integrate validator in editor.xml

```
<DocumentTypes>
  <DocumentType name="Dish">
  <PropertyType name="Price" initialValue="30"/>
      <PropertyType name="Name">
         <Validator
class="com.customer.example.editor.SimpleValidator"/>
      </PropertyType>
 </DocumentType>
</DocumentTypes>
```

Example:

The next example shows a validator which simply returns the value to be validated.

Example 4.4. Simple customized validator

```
package com.customer.example.editor;
import java.util.Map;
import hox.corem.editor.proxy.*;
import hox.corem.editor.validation.*;
public class SimpleValidator implements Validator2 {
  public Object validate(DocumentModel doc, PropertyTypeModel pte,

  Object value, Map props) throws ValidationException {
    return value;
  }
}
```

4.4 Program Own Language Resolver Factories

A property language resolver factory is used to determine the language of a property which will be used for the spell checker.

Interface to implement

Own property language resolver factories must implement the interface hox.corem.editor.PropertyLanguageResolverFactory with the method `getLanguageResolver`. In addition, an inner class must be created which implements the interface LanguageResolver (see Example 4.5, "Example of a language resolver" [56]).

Parameters to use

The `getLanguageResolver` method is called with the following parameters:

Parameter	Type	Description
document	DocumentModel	The document for which the language resolver should be built.
`propertyType`	`PropertyTypeModel`	The property of the document for which the language resolver should be built.

Table 4.5. Parameters of the getLanguageResolver method

Return types

The `getLanguageResolver` method returns a hox.gui.LanguageResolver.

Integrate your PropertyLanguageResolverFactory into the Site Manager using `editor.xml`

You can integrate your PropertyLanguageResolverFactory into the *Site Manager* using the PropertyLanguageResolverFactory element.

Example

The following example shows a language Resolver which returns locale_US as the language.

Example 4.5. Example of a language resolver

```
package com.coremedia.customer.example.editor;

import java.util.Locale;

import hox.gui.LanguageResolver;

import hox.log.Log;
import hox.corem.editor.Editor;
import hox.corem.editor.proxy.DocumentModel;
import hox.corem.editor.proxy.PropertyTypeModel;
import hox.corem.editor.PropertyLanguageResolverFactory;
```

```
public class EnglishLanguageResolverFactory implements
PropertyLanguageResolverFactory {

 private static class EnglishPropertyLanguageResolver
 implements LanguageResolver {

  public Locale getLanguage() {
   Editor.getLog().write(Log.LEVEL_DEBUG,
   "EnglishLanguageResolverFactory: "+Locale.US);
  return Locale.US;
  }

 }

 private static LanguageResolver englishPropertyLanguageResolver =

 new EnglishPropertyLanguageResolver();


 public LanguageResolver getLanguageResolver(DocumentModel document,

 PropertyTypeModel propertyType) {
 return englishPropertyLanguageResolver;
 }
}
```

4.5 Program Own PropertyEditors

Property editors are used to edit properties in the document view of the *Site Manager*. There are a lot of property editors provided by CoreMedia (see Section 5.1.1, “Property Editors” [77]) but sometimes it might be useful to extend the editors due to own needs.

Interface to implement

All property editors implement the interface PropertyEditor or one of its subinterfaces (see Javadoc). Nevertheless, you will normally subclass one of the existing property editors instead of implementing PropertyEditor or its subinterfaces. If you want to support search and replace within the property, then the property editor must implement the additional interface SearchableTextComponent and its method `isSearchable()` must return `true`. Search and replace affects the search and replace dialogs and the Global-Search-and-Replace-Workflow. The file `cap-examples.jar` contains a simple example for a property editor with search and replace functionality: `BasicStringEditor`.

Classes to subclass

The most common way to write own property editors is to subclass one of the existing editors. See Section 5.1.1, “Property Editors” [77] for a list of supplied property editors.

Integrate your Property Editor into the Site Manager using `editor.xml`

You can integrate your property editor into the *Site Manager* using the attribute `editorClass` of the `Property` element as shown in Example 4.6, “How to integrate a property editor” [58].

Example 4.6. How to integrate a property editor

```
<Documents>
   ...
   <Document type="SomeType">
      <Property name="Locale"
        editorClass="com.customer.example.editor.LocaleEditor"/>
   </Document>
   ...
</Documents>
```

Example

The following example shows a property editor which extends the ComboBoxStringEditor. The property editor adds all available locales to the combo box from which the user can select one.

Example 4.7. Example of a property editor

```
package com.customer.example.editor;
import java.util.Locale;
```

```
import hox.corem.editor.toolkit.property.ComboBoxStringEditor;
public class LocaleEditor extends ComboBoxStringEditor {

  public LocaleEditor() {
    super.setFixedChoice(true);
    Locale[] l = Locale.getAvailableLocales();
    for (int i = 0; i<l.length; i++)
      { addHistoryItem(l[i].toString()); }
  } // LocaleEditor()

  public void setFixedChoice(boolean b) { /* ignore! */ }

}
```

4.6 Program Own Predicate Classes

Predicate classes enable selective view of objects. They can filter different types of objects.

Interface to implement

For own predicates you need to implement the interface com.coremedia.common.util.Predicate<Object> with the method `include`.

Parameters to use

The `include` method gets only one parameter of type `Object`. Depending on the element of the `editor.xml` file where the predicate is used, the method can be called with different object types.

- `<Filter>`

  If the `<Predicate>` element is used in a `<Filter>` element, the documents shown in the document overview of the *Site Manager* can be filtered, due to different conditions. Thus, the objects to be filtered are documents of the type hox.corem.editor.proxy.DocumentTypeModel.
- `<Treefilter>`

  If the `<Predicate>` element is used in a `<Treefilter>` element, the folders shown in the folder view of the *Site Manager* can be filtered. The objects to be filtered are folders of the type hox.corem.editor.proxy.ResourceHolder.
- **<DocumentTypes>**

  If a `<Predicate>` or `<DocumentTypePredicate>` element is used in a `<DocumentTypes>` element, the document types which can be created, moved, copied or selected in document choosers are filtered. Thus, the objects to be filtered are document types hox.corem.editor.proxy.DocumentTypeModel.
- **<Processes>**

  If the `<Predicate>` element is used in a `<Processes>` element, the workflows offered for initiating in the Menu **File|New Workflow...** are filtered. The objects to be filtered are workflows of the type `com.coremedia.workflow.WfProcess`.

Integrate your predicate into the Site Manager using `editor.xml`

You can integrate your predicate into the *Site Manager* using the element `Predicate` or `DocumentTypePredicate` of `editor.xml` as shown in example.

```
<Explorer name="configurable-explorer-factory">
 <Filter name="ownFilter">
  <Predicate class="mySimplePredicate"/>
   .
   .
 </Filter>
<Editor>
```

Example 4.8. How to integrate the Predicate class

Example

The next example shows a predicate which simply returns "false" for all input. As a result, all items are filtered!

```
package com.customer.example;

import com.coremedia.common.util.Predicate;
import hox.corem.editor.Editor;

public class NewPredicate implements Predicate<Object> {
  public boolean include(Object obj) {
    boolean result = false;
    Editor.getLog().debug("NewPredicate.include(): "+obj);
    return result;
  }
}
```

Example 4.9. Example of a customized Predicate class

4.7 Program Own Renderers

A renderer is used to render the table cells of the document table view in the Explorer, Workflow, Query and ResourceChooser window.

Class to subclass

In order to create your own renderer class you need to subclass the abstract hox.corem.editor.toolkit.table.columnrenderer.LayoutColumnRenderer class and implement the following two methods:

- `getComponent`

  This method returns the `JComponent` which will be used for the table cell. The method is called without parameters.
- `customizeComponent`

  This method is called with the `JComponent` returned from `getComponent` and an `Object` which contains the data to render. The actual object which will be passed to the method depends on the window for which the renderer is defined. In the Explorer window a ResourceHolder is passed and in the Workflow window a WfInstance is passed to the method.

Integrate your renderer into the Site Manager using `editor.xml`

You can integrate your renderer into the *Site Manager* using the element `Renderer` which can be used within the element `ColumnDefinition`. `ColumnDefinition` can be used within `TableDefinition` which can be used within the elements `Explorer, Query, Workflow` and `ResourceChooser`.

Example 4.10. How to integrate a Renderer in the editor

```
<Explorer name="configurable-explorer">
  ...
 <TableDefinition>
 ..
  <ColumnDefinition class="StringColumn"
  name="" width="40" weight="0.0">
   <Renderer class=
   "com.customer.example.editor.ConstantLayoutColumnRenderer"/>
  </ColumnDefinition>
  ...
 </TableDefinition>
</Explorer>
```

Example

The following example shows a very simple renderer which inserts a `JLabel` saying "Hello" in each cell of the table column independent of the column type.

Example 4.11. Example of a customized Renderer class

```
package com.customer.example.editor;
```

```
import javax.swing.JComponent;
import javax.swing.JLabel;
import
hox.corem.editor.toolkit.table.columnrenderer.LayoutColumnRenderer;

public class ConstantLayoutColumnRenderer extends
  LayoutColumnRenderer {

 public JComponent getComponent() {
  return new JLabel();
 }

 public void customizeComponent(JComponent component, Object data)

 {
  JLabel label = (JLabel)component;
  label.setOpaque(true);
  label.setText("Hello");
 }
}
```

Why returning the `JLabel` first and entering the text ("Hello") later? This is because `customizeComponent` will be called later, short before the rendering. It is good practice to execute "expensive" operations at this place.

4.8 Program Own Commands

In this chapter, you will learn how to extend the *Site Manager* with buttons and menu entries which call commands on the current selection. A command is called with a target as a parameter and it executes an operation on this target following the Command Pattern. Therefore, you need to register your command at a CommandManager.

Interface to implement

All commands implement the Command interface. You need to implement (or overwrite) the `execute` and the `isExecutable` methods. The execute method gets the Context and `target` as parameters (see *Editor API* for details).

Classes to subclass

There are a different predefined commands for different tasks. You can subclass each of them (or its subclasses) for your needs. Please refer to the *Editor API* for details.

Table 4.6. Commands to subclass from

Name	Description
CommandSequence	A command which consists of a sequence of simple commands.
CreateProcessInstance	A command which creates a new WfProcessInstance from a WfProcess.
EnumerationCommand	A command which operations on enumerations and executes the given Command.
GlobalCommand	A GlobalCommand works on global, that is, application wide targets. It provides convenience methods to allow small and simple Commands.
MapCommand	A generic command which works on a map.
ResourceCommand	A command which works on resources.
ResourceEnumerationCommand	A command which works on a set of resources.
ResourceHolderCommand	A command which works on resource holders.
SearchText	A command which searches for the text in text components.
StringSelectionCommand	A command which acts on string selections.
TextActionCommand	A command which bridges from commands to actions on text components.
WfInstanceCommand	A command which works on a WfInstance.

Integrate your command into the Site Manager using `editor.xml`

There are no elements in the `editor.xml` to integrate the commands directly. You need to subclass an editor or view class.

- Integrate the command into the document view

  In this case you need to extend the `hox.corem.editor.generic.GenericDocumentView` class. The class can be added to the `editor.xml` file using the attribute `viewClass` of the element `Document`. See Section 4.8.3, "Add Command to Document View" [67] for an example.
- Integrate the command into the explorer view

  You simply need to set your own explorer view using the attribute `explorerViewClass` of the element `FrameFactory` in the `editor.xml` file. See Section 4.8.4, "Add Command to Explorer View" [68] for an example.

4.8.1 Register Commands

After you have written your command, you need to register it at the CommandManager, at the ListenerManager and at the manager of the chosen GUI element. This will make the command usable and visible. Optionally, you may associate your command with an activation model. These models activate or deactivate the command depending on a specific condition. See the field summary in the Javadoc of the DocumentView for the available activation models. To register your command proceed as follows:

Table 4.7. Register a new command

Step	Description
1.	Extend the view class.
	class MyView extends hox.corem.editor.generic.GenericDocumentView
2.	Get a CommandManager.
	`CommandManager commandMgr = getCommandManager();`
3.	Register your command at the CommandManager.
	`commandMgr.registerCommand("my-first-command", new com.coremedia.extensions.MyFirstCommand());`
4.	If you want to associate your command with an activation model you have to call the `associateActivationModel()` method. See the field summary of the DocumentView class for the available activation models.
	`commandMgr.associateActivationModel("my-first-command", approveModel);`

Step	Description
5.	Get a manager for the GUI component in which you want to insert your command. Possible are: → ToolBarManager → MenuBarManager → PopupMenuManager

	`ToolBarManager toolBarMgr = getToolBarManager();`
	`MenuBarManager menuBarMgr = getMenuBarManager();`
	`PopupMenuManager popupMenuMgr = getPopupMenuManager();`

6.	Get a ListenerManager in order to make your command live.
	`ListenerManager listenerMgr = getListenerManager();`
7.	If you want to insert your command in the tool bar or in a pop-up menu you need to get one.
	`JToolBar toolBar = toolBarMgr.getToolBar(); or JPopup Menu popUpMenu = popupMenuMgr.getPopupMenu();`
8.	Add your command to the ListenerManager and to the GUI component at once.

	`toolBarMgr.addItemBefore(toolBar, null, listenerMgr.cre ateToolBarButton("my-first-command"));`
	`or`
	`menuBarMgr.addItemBefore("edit-menu", null, listenerM gr.createMenuItem("my-first-command"));`
	`or`
	`popupMenuMgr.addItemBefore("linklist-menu", moveBottom Command, listenerMgr.createPopupMenuItem("my-first- command"));`

4.8.2 Localize Commands

You can define some attributes for your command in a properties file. The attributes are shown in the GUI and the names must follow the scheme:

`<command-name><suffix>=<value>`

`<Suffix>` can take the following values:

- `Label`: The text which is shown in the menus.
- `MenuItemLabel`: The text of a menu item.
- `Image`: The icon which is shown in the toolbar.
- `ToolTip`: The text which is shown as a tooltip.
- `Mnemonic`: The shortcut which can be used to start the command.

Please notice that `<command-name>` is not the class name of your command but the name registered at the CommandManager. So it is possible to register several commands with the same name.

Example:

You have created a command called `MakeCheaper`:

Example 4.12. Localize command

```
MakeCheaperMenuItemLabel=Reduce Price
MakeCheaperToolTip=Reduces all prices by 10 percent
MakeCheaperImage=/com/customer/cap/reduce.gif
```

The name of this file is `<Myname>.properties`. You can integrate it using the `Bundle` element of the `editor-startup.xml` file. The file name must be entered without file extension. It must appear at the right position of the classpath.

Example:

Example 4.13. Integrate bundle

```
 <Editor>
 <Locale language="de" country="DE"/>
 <Bundle name="com/customer/cap/mybundle"/>
</Editor>
```

Localization of the attributes can be done as described in Section 4.10, “Localization” [73]. Simply add the language suffix after the `<mybundle>` part of the name.

Example:

The name of the french version would be:

`mybundle_fr.properties`

4.8.3 Add Command to Document View

The document view of the *Site Manager* is created by the class hox.corem.editor.generic.GenericDocumentView. This class builds a container for a menu bar, a tool bar and a property section which contains the data of the document. If you want to use own commands in the menu bar or tool bar of the document view you need

to extend GenericDocumentView and overwrite the method `getComponent` which returns a `JComponent`. Follow the steps described in the next table.

Table 4.8. Steps to extend GenericDocumentView.

Step	Description
1	Extend GenericDocumentView
	public class `MyGenericDocumentView extends GenericDocumentVie`
2	Get the standard components from the superclass
	`JComponent component= super.getComponent()`
3	Now you can get references on the manager components and register your command as described in Section 4.8.1, “Register Commands” [65]
	`CommandManager commandMgr = getCommandManager();` `.....`
4	Return the new `JComponent`
	`return component`

After you are finished with your view class, you can insert it into the attribute `viewClass` of the element `Document` of the `editor.xml` file.

Example 4.14. How to integrate into editor.xml

```
<Documents>
 <Document type="dish" viewClass="com.custom.cap.MakeCheapClass">
 .
 .
 </Document>
</Documents>
```

4.8.4 Add Command to Explorer View

If you want to use own commands in the menu bar or tool bar of the explorer view you simply need to extend the `ExplorerView` class and add it to the editor using the attribute `explorerViewClass` of the `FrameFactory` element. Follow the steps described in the next table.

Table 4.9. How to integrate your command into the explorer view

Step	Description
1.	Extend ExplorerView.
	`public class MyExplorerView extends ExplorerView`
2.	Overwrite the `getComponent` method.
	`public JComponent getComponent() {`

Step	Description
3.	Get references to the CommandManager the ListenerManager and/or the ToolBarManager and/or the MenuBarManager.

	`CommandManager commandMgr = getCommandManager();`
	`ListenerManager listenerMgr = getListenerManager();`
	`ToolBarManager toolBarMgr = getToolBarManager();`
	`MenuBarManager menuBarMgr = getMenuBarManager();`

4.	Now you can get references on the manager components and register your command as described in Section 4.8.1, “Register Commands” [65].
	`commandMgr.registerCommand("makecheaper", new com.coremedia.extensions.MyFirstCommand); ...`
5.	Call the super method `getComponent()`. It's important to call this method *after* you have associated the command with an activation model.
	`super.getComponent()`
6.	Return the `ExplorerView`.
	`return this;` `}`

4.8.5 Add Command to Context Menu

You can find context menus in each view of the *Site Manager*. In this chapter, you will learn how to add a command to a context menu which belongs to a property editor. The steps shown in the next table will add the command to a menu belonging to a link list editor. The name of the menu is "linklist-menu" and your command will be added at the end of the menu.

Table 4.10. Add command to context menu

Step	Description
1	Extend the property editor for which you want to add your command
	`public class MyLinkListEditor extends GenericLinkListEditor {`
2	Overwrite the `getComponent` method
	`public JComponent getComponent() {`
3	Call the `getComponent` method of the parent class to create the default behavior
	`super.getComponent();`

Step	Description
4	Get the references to the manager components and register your command as described in Section 4.8.1, “Register Commands” [65].

	`CommandManager commandMgr = Services.getCommandManager(context);`
	
	`PopupMenuManager popupMenuMgr = Services.getPopupMenuManager(context);`
	
	`ListenerManager listenerMgr = Services.getListenerManager(context);`
	
	`commandMgr.registerCommand("My-Command", new MyCommand());`

5	Add a separator and your command to the end of the pop-up menu

	`popupMenuMgr.addItemAfter("linklist-menu", moveBottomCommand, listenerMgr.createMenuSeparator("Custom"));`
	
	`popupMenuMgr.addItemAfter("linklist-menu", "Custom", listenerMgr.createPopupMenuItem("My-Command"));`

6	Return the `JComponent` to the `Site Manager`

	`return this;`
	
	`}`

4.8.6 Add Action to RichTextPane

You can use custom actions to perform some actions in the RichTextPane. To this end, you have to perform the following steps:

Table 4.11. How to integrate actions into the RichTextPane

Step	Description
1	Create the command.

	`public class InsertElementCommand extends TagActionCommand {`
	
	`public static String COMMAND_NAME = "insert-element";`
	
	`public InsertElementCommand() {`
	
	`super(COMMAND_NAME);`
	
	`}`
	
	`}`

Step	Description
2	Register the command (see Section 4.8.1, “Register Commands” [65] for details):
3	Create the action you want to perform. The name of the action must match the name of the command above. The following code shows an example action which inserts an image into the RichTextPane.

	`public class InsertElementAction extends AbstractLinkAction {`
	`public InsertElementAction() {`
	`super(InsertElementCommand.COMMAND_NAME);`
	`}`
	`public void actionPerformed(ActionEvent event) {`
	`JEditorPane editor = getEditor(event);`
	`StyledEditorKit kit = getStyledEditorKit(editor);`
	`XHTMLDocument document = getXHTMLDocument(editor);`
	`document.removeSelection(editor);`
	`String uri = getReferenceToInternalDocument(12) + "/" + "blobDataPropertyName";`
	`document.insertImage(kit, editor, uri);`
	`}`
	`}`

4	Subclass hox.corem.editor.toolkit.property.RichTextPane (for example as `MyRichTextPane`) and override the `createDefaultEditorKit()` method to return your own RichTextEditorKit.
5	Define your Action in the `EditorKit` class and override `getActions` to add your custom action to the action array.

	`private final Action insertElementAction = new InsertElementAction();`
	`public Action[] getActions() {`
	`return TextAction.augmentList(super.getActions(), insertElementAction);`
	`}`

6	Configure your `MyRichTextPane` in the `editor.xml` file as described in Section 4.5, “Program Own PropertyEditors” [58].

4.9 Program Own ResourceNamingFactory Classes

A resource naming factory creates and modifies names of resources and folders. This is intended to enable customization of how resources and folders are named or renamed in different projects.

Interface to implement

Own resource naming factories must implement hox.corem.editor.ResourceNamingFactory. Please read the Editor API for more details.

Classes to subclass

The easiest way to write own resource naming factories is to subclass BasicResourceNamingFactory and to overwrite the appropriate methods. Please read the Editor API for more details.

Integrate your resource naming factory into the Site Manager using editor.xml

You can integrate your resource naming factory into the *Site Manager* using the attribute `class` of the ResourceNamingFactory element as shown in Example 4.15, "How to integrate a resource naming factory" [72].

Example 4.15. How to integrate a resource naming factory

```
<Editor>
 <ResourceNamingFactory class="MyResourceNames"/>
 .
 .
</Editor>
```

Example:

The following example shows a simple resource naming factory which allows only documents with the name "image" within the Test directory.

Example 4.16. Example of a resource naming factory

```
public MyResourceNames extends BasicResourceNamingFactory {
public boolean isValidResourceName(FolderModel folder,
ResourceTypeModel resourceType,
String name) {
return !(!folder.getName().equals("Test") || "image".equals(name))

&& super.isValidResourceName(folder, resourceType, name);
}
}
```

4.10 Localization

The *Site Manager* is internationalized and is therefore prepared for configuration to different languages. Within a localization, program texts in menus, dialogs etc., as well as the names of the document types and properties which are shown, can be adjusted.

The language and the country-specific settings which the *Site Manager* displays on the user interface are configured with the attributes

```
language="en"
```

and

```
country="UK"
```

of the `Locale` element in the file `properties/corem/editor-startup.xml`. Details for configuring the language settings are given in the *Administration Manual*.

Internationalization of the *Site Manager* is based on the class `java.util.Locale`. On program start, the default locale of the Java environment is set to the value given in the configuration file `properties/corem/editor-startup.xml`. For identification of the locale, the ISO 639 language codes are used (see http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt). New locales should use these language codes in order to remain compatible.

4.10.1 Localize the Editor

For localization of texts and names and custom icons shown in the *Site Manager* defined in the *CoreMedia Server*, a file following the naming scheme name[_<locale>].properties can be used. It can be located in either class path or a JAR file located in the `lib/` directory.

Example for the naming:

German: `com/customer/cap/editor.properties`

English: `com/customer/cap/editor_en.properties`

French: `com/customer/cap/editor_fr.properties`

The Bundle name `com/customer/cap/editor` is configured with the Bundle element of the `editor-startup.xml`.

```
<Bundle name="com/customer/cap/editor"/>
```

Within the files, name/value pairs define the localized names. For each object a name and a tooltip text can be configured. The naming scheme is as follows:

`<Name of the object defined in the CoreMedia Server>[/<Name of the property>]Label` or

`<Name of the object>[/<Name of the property>]ToolTip.`

The value is the localized name.

If you want to change the embedded labels of the *Site Manager* (for example a menu item or the label of a button) you need to know the keys of these labels. You will find the keys in the file `hox/corem/editor/toolkit/property/property_de.properties` which is located in the `cap-editor-resources.jar` file of an editor installation.

Example:

To localize some German names for use in English language, create a file `$COREM_HOME/classes/com/customer/cap/editor_en.properties`. In order to rename a document type called `Bild` to `Image`, define:

```
BildLabel=Image
```

To rename the property `Piktogramm` to `Thumbnail` and to add a tooltip saying "Preview Image" define:

```
Bild/PiktogrammLabel=Thumbnail

Bild/PiktogrammToolTip=Preview Image
```

It can be seen that properties are designated via the preceding document type name and are separated by a slash. Therefore, the names of document types must not contain slashes for proper localization of properties. This file can be stored in a JAR file to make distribution easier.

Adding or Changing Document Type Icons

Out-of-the-box the *Site Manager* comes with only a few icons for predefined document types (Query, document...). In order to improve the usability of your customized editor, you should add own document type icons. The editor supports icon sizes of 16x16 and 32x32 pixels that are used in different views. The icons can be in GIF and PNG file formats. You have to add the images to your custom `properties` file using the keys `<doctypename>Image` for 16x16 icons and `<doctypename>TitleImage` for 32x32 images. In order to make the icons available, proceed as follows:

1. Store your icons in a directory, for example `com/customer/cap/icons`
2. Add the icon names to your property file using the keys described above.
3. Pack these directories in a JAR file and put this file into the `lib/` directory of your editor installation.

4. Add the name of the bundle to the `editor-startup.xml` file.

That's it. The editor will automatically use your new icons.

Example:

You already have an `editor.properties` file with some customizations of the editor. The file is stored in `com/custom/cap` and therefore the entry in `editor-startup.xml` is:

```
<Bundle name="com/custom/cap/editor"/>
```

Now you want to add new icons for the Query document and your custom Teaser document. Store the images `Query.png`, `QueryLarge.png`, `Teaser.png` and `TeaserLarge.png` in the `com/custom/cap/icons` directory and add the following lines to the `editor.properties` file:

```
QueryImage=/com/custom/cap/icons/Query.png
QueryTitleImage=/com/custom/cap/icons/QueryLarge.png
TeaserImage=/com/custom/cap/icons/Teaser.png
TeaserTitleImage=/com/custom/cap/icons/TeaserLarge.png
```

At last, you create a JAR file from the files and put it into the `lib/` directory of your editor.

4.10.2 Localize for Use with WebStart

Localization of the *Site Manager* for use with WebStart is similar to the standard localization described before. You only have to provide the property file as a signed JAR file to WebStart. Proceed as follows:

1. Create a property file, here `editor.properties`, with the localized properties in the `resources` directory of the `editor-customizations` module.
2. Build the editor with Maven in the workspace. Maven will automatically build and sign a JAR file and put it into the `editor.jnlp` file.

Now, the editor can use the localized properties.

5. Appendix

In this chapter you will find all predefined classes usable for the editor configuration and a reference view of all XML elements of the editor DTD.

5.1 Classes Delivered for Site Manager Configuration

To configure the *Site Manager* for your specific needs the *Editor API* is available. You can use it to program your own editor, filter or validator classes. To keep the effort for the user as small as possible, the *Site Manager* is delivered with a series of classes which already enable configuration. These delivered classes are described in the following sections.

5.1.1 Property Editors

Property editors are connected with fields in document types or with variables in workflows. In this way, for example, you can apply a combo box to a field.

```
<Property name="Author" editorClass="ComboBoxStringEditor">
        <HistoryItem value="TestEditor"/>
        <HistoryItem value="Editor"/>
</Property>
```

Example 5.1. Example for the use of a property editor

Workflow Editors

The following property editors are available for variables in workflows.

Class	Description
`JCheckboxBooleanEditor`	A check box. "True" is shown as checked, "False" as unchecked.
`ComboBoxDocumentTypeEditor`	A combo box with the allowed document types.
`GroupChooserEditor`	A window which shows the available group.
`ResourceChooserEditor`	A resource chooser window, like the one in the editor window.
`UserChooserEditor`	A window which shows the available users.

Table 5.1. Property editors for the workflow

String Editors

The following property editors are available for string fields:

Table 5.2. Property editors for strings

Class	Description
JTextFieldStringEditor	Simple text field. This is the default editor. The JTextFieldStringEditor can be configured with the following attributes: → `fontName="Name of the font"` This property is used for setting the font of the Text field in the *Site Manager*. It should be noted that the fonts depend on the particular configuration of the Java environment. If the font entered does not exist, all available fonts are shown in the log. → `fontSize="Size of the font"` This parameter adjusts the font size of Text field in the *Site Manager*. The size can be set between 10 and 24. If there is no exactly matching character set for this font size, the font is scaled accordingly. This can cause an awkward appearance of the font. → `spellCheckingEnabled="false"` This parameter disables spell checking for the Text field in the *Site Manager*. Spell checking is enabled by default.
ComboBoxStringEditor	A selection list. You can enable own entries in the combo box using the attribute `fixedChoice` with the `<Property>` element. "false" will allow you to type own entries in the combo box. "true" is default and will allow only the predefined items. You can limit the number of characters, which are allowed to be entered, by using the attribute `columns`. Example: `<Property name="CopiesTo" editorClass="ComboBoxStringEditor" columns="15">` If you are using the `columns` attribute you have to take care for two things: 1. Do not enter a `<HistoryItem>` or `<FunctionItem>` with a length longer than defined in `columns`. 2. Be sure that no strings longer than defined in `columns` have been stored in the repository before. The best practice would be to configure a validator to guarantee correct server side storage. You can disable spell checking using the attribute `spellCheckingEnabled="false"`. Spell checking is enabled by default. The selection options can be added with three different multiple child elements: → `<HistoryItem value="abc"/>`: Shows and uses exactly the values that are defined in the `value` attribute of this element.

Class	Description

	→ `<FunctionItem`: You can use the attribute `class` to define the class which should be used when you select the entry. The attribute label (optional) defines the name of the function item which is shown in the combo box. Own classes must extend the abstract class hox.gui.editor.combobox.FunctionItem.
	
	→ `<LabeledItem value="60" label="One minute"/>` Use the label attribute to define the text shown in the combo box. You will see the localized version of the value of the label attribute (localized via `classes/hox/corem/editor/toolkit/property/property.properties`). The value used is the one defined in the attribute `value`.
	
	Examples:
	
	`<Property name="Name" editorClass="ComboBoxStringEditor">`
	
	`<HistoryItem value="60"/>`
	
	`</Property>`
	
	Will show 60 in the combo box and will write 60 into the property when you select it.
	
	`<Property name="Name" editorClass="ComboBoxStringEditor">`
	
	`<FunctionItem class="myFunctionItems.MyStringClass" label="Select Me"/>`
	
	`</Property>`
	
	Will show "Select Me" in the combo box and will write the result of the class `MyStringClass` into the property when you select the entry.
	
	`<Property name="Name" editorClass="ComboBoxStringEditor">`
	
	`<LabeledItem value="60" label="One minute"/>`
	
	`</Property>`
	
	Will show "One minute" in the combo box and will write 60 into the property when you select the entry.

JCheckBoxStringEditor	A check box. If the box is chosen, "true" will be saved, otherwise "false".
JPasswordFieldStringEditor	Text field which allows you to enter a password. Thus, the input characters will be displayed hidden.

Class	Description
JTextPaneStringEditor	Text field with more than one line. That is, if the length of your string exceeds the width of your string text field a new line will be started. Internally, this will not be stored as a line break. You can insert line breaks hitting the <Return> key. The length of the text is limited to the length defined in the `document-type.xml` file. The `JTextPaneStringEditor` can be configured with the following attributes: → `fontName="Name of the font"` This property is used for setting the font of the Text field in the *Site Manager*. It should be noted that the fonts depend on the particular configuration of the Java environment. If the font entered does not exist, all available fonts are shown in the log. → `fontSize="Size of the font"` This parameter adjusts the font size of Text field in the *Site Manager*. The size can be set between 10 and 24. If there is no exactly matching character set for this font size, the font is scaled accordingly. This can cause an awkward appearance of the font. → `spellCheckingEnabled="false"` This parameter disables spell checking for the Text field in the *Site Manager*. Spell checking is enabled by default.

Integer Editors

The following property editors are available for integer fields:

Table 5.3. Property editors for integers

Class	Description
JTextFieldIntegerEditor	Simple integer field. This is the default editor.
JCheckBoxIntegerEditor	A check box. If the box is checked, "1" will be saved, otherwise "0".
ComboBoxIntegerEditor	Selection list. If this class is used, the attribute `fixedChoice` can be added to the `<Property>` element. "false" will allow you to type own entries in the combo box. "true" is default and will allow only the predefined items. You can limit the number of digits which are allowed to enter, by using the attribute `columns`. Example:

Class	Description
	`<Property name="Copies" editorClass="ComboBoxIntegerEditor" columns="3">` If you are using this attribute you have to take care for two things: 1. Do not enter a `<HistoryItem>` with a length longer than defined in `maximumSize`. 2. Be sure that no integers longer than defined in `columns` have been stored in the repository before. The best practice would be to configure a validator to guarantee correct server side storage.
	The selection options can be added with two different multiple child elements: → `<HistoryItem value="123"/>` The values defined in this element are shown as a combo box. → `<LabeledItem value="60" label="One minute"/>` The values defined in this element are shown as a combo box. In the combo box you will see the localized version of the value of the `label` attribute (localized via `classes/hox/corem/editor/toolkit/property/property.properties`). The value used is the one defined in `value`.

	→ `<FunctionItem>`
	
	The attribute `class` defines the class which should be used. The attribute `label` (optional) defines the name of the function item.
	
	Own classes must implement the interface hox.gui.editor.combobox.FunctionItem.
	
	Example:
	
	`<Property name="Value" editorClass="ComboBoxIntegerEditor">`
	
	`<FunctionItem class="myFunctionItem.MyIntegerClass"/>`
	
	`</Property>`

GroupChooserIntegerEditor	This editor shows all groups defined in the CoreMedia system. One group can be chosen and the ID of the group will be stored.
UserChooserIntegerEditor	This editor shows all users defined in the CoreMedia system. One user can be chosen and the ID of the user will be stored.

Date Editors

The following editors are available for entering dates:

Table 5.4. Property editors for dates

Class	Description
DatePickEditor	A graphical date selector.
JTextFieldDateEditor	Simple text field. This is the default editor. Using the attribute `format`, the date format can be defined. The default is '`dd.MM.yyyy HH:mm`'.

ComboBoxDateEditor	This editor is used to configure predefined dates which can be chosen from a combo box. If this class is used, the attribute `fixedChoice` can be added to the `<Property>` element. "false" will allow you to type own entries in the combo box. "true" is default and will allow only the predefined items. Two different sub elements of `<Property>` can be chosen:
	→ `<HistoryItem>`
	The attribute `value` can be used to define a specific date. The date format of these entries can be configured with the attribute `format` of the element `<Property>`, with default '`dd.MM.yyyy HH:mm`'. All letters defined in the class `java.text.SimpleDateFormat` can be used to define the format.
	Example:
	`<Property name="Date" editorClass="ComboBoxDateEditor" format="dd.MM.yyyy">`
	`<HistoryItem value="01.01.2001"/>`
	`</Property>`
	Thus, "01.01.2001" is shown in the selection list of the combo box.

	→ `<FunctionItem>` The attribute `class` can be used to define some variable dates which depends on the actual date. The attribute `label` (optional) defines the name of the function item. To do so, the following classes can be used: → NotApplicable: No date is shown. → Tomorrow: The date of tomorrow is chosen. → Today: The date of today is chosen. → NextWeek: The date of today in a week is chosen.

Class	Description

	→ NextMonth: The date of today in a month is chosen.
	→ NextYear: The date of today in a year is chosen.
	→ EndOfMonth: The date of the end of this month is chosen.
	→ EndOfYear: The date of the end of this year is chosen.
	
	The whole qualified name of the class must be used (see example below). Own classes can be written, which must implement the interface hox.gui.editor.combobox.FunctionItem.
	
	Example:
	
	`<Property name="Date" editorClass="ComboBoxDateEditor" format="dd.MM.yyyy">`
	
	`<FunctionItem class="hox.corem.editor.toolkit.property.items.Tomorrow"/>`
	
	`</Property>`
	
	Thus, "Tomorrow" is shown in the combo box. If you choose this entry, the date of tomorrow is shown in the format defined in `<Property>`.

XML Editors

The following editors are available for XML fields:

Editors for XML text

RichTextField

Text component for editing text in only one line. The text must correspond to the `coremedia-richtext-1.0.dtd`. All the attributes from RichTextPane are valid.

RichTextPane

Text component for editing text which corresponds to the CoreMedia Rich Text DTD (`lib/xml/coremedia-richtext-1.0.dtd`). You can configure the RichTextPane with the following attributes:

Table 5.5. Some attributes of the RichTextPane

Attribute	Value	Default	Description
`addStyleSheetGroups`	`<StyleSheetGroupName>(<UsedOnEle`		Using this attribute, you can assign a style sheet group defined in the `coremedia-richtext-1.0.css` file to a property. The added style groups ap-

Attribute	Value	Default	Description
	ment>,<UsedOnElement>,...)..		pear automatically in the corresponding attribute editor and in the toolbar. You need to replace <StyleSheetGroupName> with the name of the style sheet group as defined in the CSS file and <UsedOnElement> with the name of the element to which the style should be applied. The elements p, table, tr, td, li, ul, ol, a, img from the coremedia-richtext-1.0.dtd can be used. In addition, the special keywords inline, block and flow can be used. inline stands for any inline element, block for any block element and flow for any possible element (see Section 3.6.4, "Add to Content Editor" [36] for details). A special keyword cannot be combined with any other element or keyword. If you want to use a free text field to enter the value of a style sheet group, you have to add :string behind the name of the style sheet group. Be aware, that this free text will not be rendered specifically, because it is not defined in the coremedia-richtext-1.0.css file. It will appear as ordinary text. But you can define additional combo box entries, for example for some default values which will be rendered as defined in the CSS file. Example: addStyleSheetGroups="myGroup1:string(td) myGroup2:string"
showUnknownStyles	true, false	true	If this attribute is "true" unknown styles will be shown in the attribute editor along with a button to remove them from the element. If set to "false", the unknown styles will not be shown.
nestedTablesAllowed	true, false	true	If this attribute is set "true" you are allowed to use nested tables in the RichTextPane.
maximumTableCells	Integer	250	This attribute defines the maximum number of table cells allowed for a newly created table.

Attribute	Value	Default	Description
`spellCheckingEnabled`	true, false	true	This attribute enables or disables spell checking for the rich text pane.
`internalLinkDocumentType`	Document type		Using this attribute, you can set a default document type used for the internal link chooser. For example, if you set `internalLinkDocumentType="Article"` only Article documents will be shown in the chooser by default.
`internalLinkTarget`	new, replace, embed, other, none	replace	Using this attribute, you can configure the default targets for internal links in the RichTextPane
`externalLinkTarget`	new, replace, embed, other, none	new	Using this attribute, you can configure the default targets for external links in the RichTextPane
`imageDocumentType`	Document type		Using this attribute, you can set a default document type used for the image document chooser. For example, if you set `imageDocumentType="Image"` only Image documents will be shown in the chooser by default.

The following attributes disable the respective menu items of the RichText pane.

- `enableTableAttributes="false"`
- `enableTableModifying="false"`
- `enableTableCellModifying="false"`
- `enableInsertTables="false"`
- `enableTables="false"`
- `enableClassAttributes="false"`
- `enableTextAlignment="false"`
- `enableNumberedLists="false"`
- `enableBulletList="false"`
- `enableLists="false"`
- `enableLanguages="false"`

- `enableListIndention="false"`
- `enableListOutdention="false"`
- `enableInternalLinks="false"`
- `enableExternalLinks="false"`
- `enableLinks="false"`
- `enableInsertImages="false"`
- `enableSubScript="false"`
- `enableSuperScript="false"`
- `enableStrikeThrough="false"`
- `enableUnderline="false"`
- `enableBold="false"`
- `enableItalic="false"`
- `enableRemoveTextFormatting="false"`
- `enableFontSize="false"`
- `enableFontColor="false"`
- `enableFont="false"`
- `enableBackgroundColor="false"`
- `enableHeadings="false"`
- `enableBlockQuote="false"`

  If these attributes are set "false", you can disable the respective menu items and tools of the RichText pane. Default is "true". You must not use `enableTables` with other table settings (for example `enableTableModifying`), `enableLinks` with other link settings and `enableLists` with other list settings

In the following, you will find child elements of the `<property>` element with the editor class `RichTextPane`. You can use these elements to define the transformation of HTML elements of text in the clipboard into elements of the `coremedia-richtext-1.0.dtd` (see Example 5.2, "Example of PasteTransformation" [90]) and to configure the file creation dialog.

This configuration affects copying within a RichtText pane as well as between an external application and a RichText pane. HTML elements which are neither configured using `<PasteTransformation>` nor included in the standard configuration (see Javadoc com.coremedia.cap.gui.richtext.RichTextPasteConfig) will be ignored.

The file creation dialogs of the `RichText pane` is used when you move a blob from the file system into a RichText pane. Use the child element `< NewDocumentDialogSettings>` for your settings.

<NewDocumentDialogSettings>

Child elements:

Parent element: `<Property>`

Use the `<NewDocumentDialogSettings>` element if you want to customize the file creation dialog.

Table 5.6. Attributes of NewDocumentDialogSettings

Attribute	Value	Default	Description
`createPreselectedFolder`	true, false	false	If true, the preselected folder defined with `preselectedResource` will be created if it does not exist yet. Ignored if `preselectedResourceId` is set.
`preselectedType`	Document type		Name of the preselected document type. If no preselected type is defined or the preselected type is not able to store the blob data, the first matching type will be used in a new dialog.
`preselectedResource`	Resource path		Absolute path or path relative to the current document, which defines the preselected resource. Alternative to the `preselectedResourceId` attribute. When both attributes are given, the `preselectedResourceId` takes precedence unless no resource with the given id exists.
`preselectedResourceId`	Integer		ID of the preselected resource. Alternative to the `preselectedResource` attribute. Additional feature available only using the id: Instead of specifying a folder id you may also specify a document id. In this case the document with the given id serves a kind of token where new documents will be created because the document will be created in the very same directory where the referenced document is in.
`rootFolder`	Folder path		Name of the folder which defines the root of the file chooser dialog. Alternative use to the `rootFolderId` attribute. If both attributes

Attribute	Value	Default	Description
			are given, the `rootFolderId` takes precedence.
`rootFolderId`	Integer		ID of the folder which defines the root of the file chooser dialog. Alternative use to the `rootFolder` attribute.
`upperBound`	Document type		Configures the sub type of the shown document types. That is, you will only see document types which are super types of the defined document type (including the specified document type unless it is abstract). `typePredicate` overrides any bounds set.
`lowerBound`	Document type		Configures the super type of the shown document types. That is, you will only see document types which are sub types of the defined document type (including the specified document type unless it is abstract). `typePredicate` overrides any bounds set.
`typePredicate`	Class path		A class of type com.coremedia.common.util.Predicate<Object> with a no-argument constructor which filters the shown document types. If you set a `typePredicate` it overrides any `upperBound` or `lowerBound` set.
`resourceName`	Name		The name of the new document. If no resource name is defined, the name field will be empty. For all subsequent calls, the previously entered name will be used.
`openDocument`	true, false	true	Defines the state of the *Open document* check box in the dialog. If true, the newly created document will automatically be opened.
`fieldName`	Field name		Name of the preselected document field where the blob should be stored. If no name is defined or the document field is not able to store the blob data, the first matching field of the document type is shown.

<PasteTransformation>

Child elements: `<TransformElement>*, <IgnoreElement>*`

Parent element: `<Property>`

Use the `<PasteTransformation>` element if you want to customize the paste operation of the RichText pane.

Attribute	Description
`extendDefault`	If this attribute is set to "true", the standard paste configuration will be extended by the new configuration. If it's set to "false", the standard configuration will be replaced by the new configuration. Default is "true".

Table 5.7. The attribute of the PasteTransformation element

<TransformElement>

Child element: `<Attribute>*`

Parent element: `<PasteTransformation>`

Use this element to match the HTML element from the clipboard with the element of the `coremedia-richtext-1.0.dtd` to insert.

Attribute	Description
name	The name of the HTML element which should be transformed.
to	The name of the element into which the HTML element should be transformed/which will be inserted into the RichText pane. If `to` is not set, the name of the HTML element will be used. Please notice, that only elements according to the `coremedia-richtext-1.0.dtd` are allowed.

Table 5.8. The attributes of the TransformElement element

<IgnoreElement>

Child element:

Parent element: `<PasteTransformation>`

Use this element to define HTML elements which should not be inserted into the RichText pane.

Attribute	Description
name	Name of the HTML element to ignore.
recursive	"false" (Default): Only the element defined in `name` will be ignored. "true": The element defined in `name` and all contained elements will be recursively be ignored.

Table 5.9. Attributes of the IgnoreElement element

<Attribute>

Parent element: `<TransformElement>`

Use this element to define an attribute of an HTML element which should be inserted into the RichText pane. Attributes of an HTML element which are not matched by an `<Attribute>` element will be ignored. The only exception is the `class` attribute which will always be taken over.

Attribute	Description
name	Name of the attribute to be taken over.
value	Value, which should be assigned to the attribute defined in `name`. If no value is defined, the value from the HTML element will be used.

Table 5.10. Attributes of the Attribute element

The following example shows how to extend the standard configuration with two changed rules. `<H1>` elements will be transformed into a paragraph (`<P>` element) with font size 20 and `<H2>` elements will be ignored.

```
 <Documents>
 <Document type="Article">
  <Property name="Content" editorClass="RichTextPane">
   <PasteTransformation>
    <TransformElement name="H1" to="P">
     <Attribute name="class" value="font-size--20"/>
    </TransformElement>
    <IgnoreElement name="H2" recursive="false"/>
   </PasteTransformation>
  </Property>
 </Document>
</Documents>
```

Example 5.2. Example of PasteTransformation

PlainXmlPropertyEditor

A component to edit raw XML of arbitrary grammar with a simple plain text editor. It's useful for debugging, troubleshooting and small modifications without comfort. The editor displays XML tags and content as simple text. When the user tries to save invalid XML, a detailed XML error message appears.

```
<Documents>
    <Document type="XmlExample">
       <Property name="Xml" editorClass="PlainXmlPropertyEditor"/>

    </Document>
</Documents>

<DocumentTypes>
    <DocumentType name="XmlExample">
         <PropertyType name="Xml">
            <ModelClass
```

Example 5.3. PlainXmlPropertyEditor configuration example

```
class="hox.corem.editor.toolkit.property.richtext.

impl.ConcurrentXmlPropertyModel"/>
        </PropertyType>
    </DocumentType>
</DocumentTypes>
```

Blob Editors

The following editors are available for blob fields:

Table 5.11. Editors for blob fields

Class	Description

AggregatingImageBlobEditor	This editor displays blobs of type image/gif, image/png or image/jpeg in the property as an image. In addition, you can define other properties which contain scaled versions of the blob. This blobs will be computed from the original blob by the CoreMedia Server. The size and type of the scaled images is defined in the `imageconverter.properties` file. The property `defaultExtension` in `imageconverter.properties` defines the type of the scaled image. jpeg for example would create a JPEG image, see the ImageMagick documentation for all supported formats. You can use the following attributes to define scaled versions of the blob:
	→ `onlineProperty="<PropertyName>"`: Sets the name of the property which should be converted when the blob contained in the property of the AggregatingImageBlobEditor changes. The size of the image is defined with the `onlineWidth` and `onlineHeigth` properties of the `imageconverter.properties` file.
	→ `originalProperty="<PropertyName>"`: Sets the name of the property which should be converted when the blob contained in the property of the AggregatingImageBlobEditor changes.
	→ `thumbnailProperty="<PropertyName>"`:Sets the name of the property which should be converted when the blob contained in the property of the AggregatingImageBlobEditor changes. The size of the image is defined with the `thumbnailWidth` and `thumbnailHeight` properties of the `imageconverter.properties` file.
	Example:
	`<Document type="Photo">`
	`<Property name="ScannedPhoto" editorClass="AggregatingImageBlobEditor" thumbnailProperty="Preview" onlineProperty="Online"/>`

Class	Description
	`</Document>`
BasicBlobEditor	Generic editor which only displays blob size and content type. Files from the file system can be loaded and saved. Using the property `WebDAVUriPrefix`, the blob can be opened directly in the associated WebDAV application. For this, the property has to be set to the URL prefix for the WebDAV server. If a prefix like `http://Server:8001/webdav/` is used on a blob in a document `/Inbox/Test.doc`, then an URL `http://Server:8001/webdav/Inbox/Test.doc` for WebDAV lookup is used.
ImageBlobEditor	Displays blobs of type image/gif, image/png or image/jpeg as an image. If the size of an image exceeds the document window size, it will be scaled down. You can use the attribute `imageScaledForDisplay="false"` to disable the behavior.
TextBlobEditor	Allows editing of blobs of type text/* in a text field.

LinkList Editors

The following editors are available for editing linklist fields:

Editors for LinkList fields

ComboBoxLinkListEditor

Allows selection of a document from a ComboBox, whose content consists of the documents of a folder. The `path` attribute determines the path of this folder. Automatically, only documents of appropriate type are shown. The `emptySelection` attribute configures the text displayed if no document is linked.

As with the GenericLinkListEditor, display is configured with a `LinkListRenderer`.

FolderLinkListEditor

A FolderLinkListEditor displays two `JLists` side by side. The left list contains the resources from the link list of the document. The right list contains resources that are determined by the `path` parameter which points to a CoreMedia folder. All documents in this folder can be selected and thus be inserted into the documents link list.

The following attributes can be used to configure the editor:

- `path`: The path to the folder whose content will be displayed in the folder list.
- `showFolderListOnCheckOut`: Indicates whether the folder list is shown when the document is checked out (true). If set to false, only the link list is displayed and the user must click a button to open the folder list.

- `listHeight`: Sets the height in pixel of the two lists. If the lists are longer, a scrollbar is displayed. The default value is 100 which displays approximately five list entries with a standard font size. For ten lines set the value to 180.

In addition, you can configure the lists using the following sub elements. Use the attribute `class` to define the classes to use:

- `LinkListRenderer`: Define a renderer which renders the content of the left link list.
- `FolderListRender`: Sets the renderer for the right folder list. If not set, a default render is used to display an icon and the document name.
- `FolderListPredicate` : Sets a predicate to filter the right folder list. If not set, all documents matching the given link list element type are displayed.
- `FolderListComparator` : Sets a comparator to sort the right folder list. If not set, the folder list is sorted by document name.

Example:

```
<Property name="TestLinkList" editorClass="FolderLinkListEditor"
          path="/" listHeight="180" >

  <LinkListRenderer class=
"hox.corem.editor.toolkit.property.ImageLinkListRenderer"
    property="TestBlob"/>

</Property>
```

GenericLinkListEditor

Displays a list of the linked documents and allows links to be added/deleted/moved. Display of the documents is determined by the `LinkListRenderer`. This can be determined with a child element `<LinkListRenderer class="Renderer class">`.

The default is DocumentTypeLinkListRenderer, which displays document type icon and document name.

Other renderers are DocumentStateLinkListRenderer and ImageLinkListRenderer. The DocumentStateLinkListRenderer displays a state icon and the document name. The ImageLinkListRenderer displays `Blob` properties of the linked documents. The `Blob` property which should be used is determined by the `property` attribute of the `LinkListRenderer` element.

Example:

```
<LinkListRenderer      class="ImageLinkListRenderer"      prop
erty="small" />
```

You can configure the file chooser and file creation dialogs of the `GenericLink ListEditor`. Use the child element <DialogSettings> with the two respective

child elements `<NewDocumentDialogSettings>` or `<DocumentChooserSettings>`. With the following attributes you can parameterize the dialogs:

Table 5.12. Attributes of NewDocumentDialogSettings and DocumentChooserSettings

Attribute	Value	Default	Description
`createPreselectedFolder`	true, false	false	If true, the preselected Folder defined with `preselectedResource` will be created if it does not exist yet. Ignored if `preselectedResourceId` is set.
`preselectedType`	Document type		Name of the preselected document type. If no preselected type is defined or it does not match the LinkList requirements, the first matching type will be used in a new dialog.
`preselectedResource`	Resource path		Absolute path or path relative to the current document, which defines the preselected resource. Alternative to the `preselectedResourceId` attribute. If both attributes are given, the `preselectedResourceId` takes precedence unless no resource with the given id exists.
`preselectedResourceId`	Integer		ID of the preselected resource. Alternative to the `preselectedResource` attribute. Additional feature available only using the id: Instead of specifying a folder id you may also specify a document id. In this case the document with the given id serves a kind of token where new documents will be created (or documents will be chosen from) because the document will be created in (chosen from) the very same directory where the referenced document is in.
`rootFolder`	Folder path		Name of the folder which defines the root of the file chooser dialog. Alternative use to the `rootFolderId` attribute. If both attributes are given, the `rootFolderId` takes precedence.
`rootFolderId`	Integer		ID of the folder which defines the root of the file chooser dialog. Alternative use to the `rootFolder` attribute.
`upperBound`	Document type		Configures the sub type of the shown document types. That is, you will only see document types which are super

Attribute	Value	Default	Description
			types of the defined document type (including the specified document type unless it is abstract). `typePredicate` overrides any bounds set.
`lowerBound`	Document type		Configures the super type of the shown document types. That is, you will only see document types which are sub types of the defined document type (including the specified document type unless it is abstract). `typePredicate` overrides any bounds set.
`typePredicate`	Class path		A class of type com.coremedia.common.util.Predicate<Object> with a no-argument constructor which filters the shown document types. If you set a `typePredicate` it overrides any `upperBound` or `lowerBound` set.

For the <NewDocumentDialogSettings> element you can also use the following two attributes:

Table 5.13. More attributes of NewDocumentDialogSettings

Attribute	Value	Default	Description
`resourceName`	Name		The name of the new document. If no resource name is defined, the name field will be empty. For all subsequent calls, the previously entered name will be used.
`openDocument`	true, false	true	Defines the state of the *Open document* check box in the dialog. If true, the newly created document will automatically be opened.

Example

```
<Document type="Dish">
  <Property name="pictures" editorClass="GenericLinkListEditor">
    <DialogSettings>
      <NewDocumentDialogSettings
      preselectedType="Picture"
      preselectedResource="/MenuSite/Fish"
      rootFolder="/MenuSite"
      upperBound="Picture"/>
    </DialogSettings>
  </Property>
</Document>
```

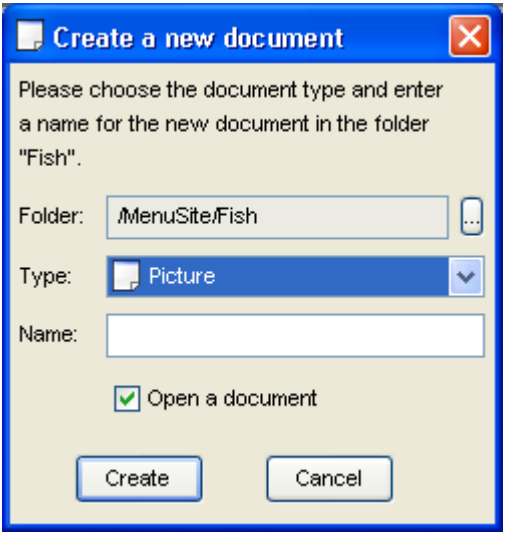


Figure 5.1. Configured file creation dialog

5.1.2 View Classes

```
<Documents>
 <Document type="Article" viewClass="mypackage.MyViewClass">
 .
 .
 .
 </Document>
</Documents>
```

Example 5.4. Example for the configuration of a document view

Document view classes define the look of the document window of the *Site Manager*. You can write own document view classes, which must be a subclass of hox.corem.editor.toolkit.document.AbstractDocumentView or hox.corem.editor.generic.GenericDocumentView. The following classes are predefined:

Table 5.14. View classes

Class	Description
TabbedDocumentView	The class hox.corem.editor.generic.TabbedDocumentView defines a tabbed document view with different tabs. Figure 5.2, "Example of a tabbed document view" [97] shows an example of a tabbed view. The tabs can be configured using the subelement `<Tab>`. Using the attribute `name`, a label can be attached to the tab. The properties of a document belonging to a tab can be configured using `<Property>` as a subelement of `<Tab>`. For an example, see the description of the element `<Tab>`. The position of the tabs can be configured using the attribute `tabPlace`

Class	Description
	`ment` of the element `<Document>`. The values top, bottom, left and right are allowed.

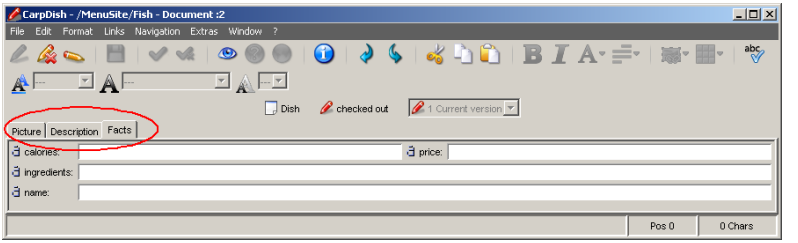


Figure 5.2. Example of a tabbed document view

5.1.3 Predicate Classes

```
<Filter name="deleted-filter">
        <Predicate class="DeletedPredicate"/>
</Filter>
```

Example 5.5. Example for the use of a filter

Predicate classes enable selective display of objects. Depending on the context in which the `<Predicate>` element is used, different object types can be selected:

- `<Filter>`

  If the `<Predicate>` element is used in a `<Filter>` element, the documents shown in the document overview of the *Site Manager* can be filtered, due to different conditions. Thus, the objects to be filtered are documents of the type hox.corem.editor.proxy.DocumentTypeModel.
- `<Treefilter>`

  If the `<Predicate>` element is used in a `<Treefilter>` element, the folders shown in the folder view of the *Site Manager* can be filtered. The objects to be filtered are folders of the type hox.corem.editor.proxy.ResourceHolder.
- `<DocumentTypes>`

  If a `<Predicate>` or `<DocumentTypePredicate>` element is used in a `<DocumentTypes>` element, the document types which can be created, moved, copied or selected in the document choosers of the *Site Manager* are filtered. Thus, the objects to be filtered are document types hox.corem.editor.proxy.DocumentTypeModel.
- `<Processes>`

If the `<Predicate>` element is used in a `<Processes>` element, the workflows offered for initiating in the Menu **File|New workflow...** are filtered. The objects to be filtered are workflows of the type `com.coremedia.workflow.WfProcess`.

→ `<Workflow>`

If the `<Predicate>` element is used in a `<Workflow>` element, the workflows and tasks shown in the sub views *My tasks*, *Offered tasks* and *My workflows* of the Workflow window can be filtered, if the `<Predicate>` element is used in a `<Workflow>` element. The objects to be filtered are com.coremedia.workflow.WfProcess and com.coremedia.workflow.WfTask.

Filtering can occur on both the client side and the server side. The predefined filters only run on the server side, while customized filters must run on the client side. This is determined by the `remote` attribute of the `<Filter>` element. The default value "true" indicates server-side selection.

Predicates for filtering document types

Default Predicate	Filtered objects	Description
GenericDocumentTypePredicate	Document types: hox.corem.editor.proxy.DocumentTypeModel	The default predicate filters the document types which are passed to the predicate class and sorts the remaining document types in alphabetic order. Own predicates must implement the interface com.coremedia.common.util.Predicate<Object>

Table 5.15. Predicate classes for filtering documents types.

Predicates for filtering folders

Default predicate	Filtered objects	Description
GenericTreePredicate	Folders: hox.corem.editor.proxy.ResourceHolder	The default comparator filters without read rights. Own predicates must implement com.coremedia.common.util.Predicate<Object>.

Table 5.16. Predicate classes for filtering folders

Predicates for filtering workflows and tasks

Default predicate	Sorted objects	Description
GenericProcessPredicate	Workflows: com.coremedia.workflow.WfProcess	The default predicate filters workflows for which no creation right is granted to the user. Own predicates must implement com.coremedia.common.util.Predicate<Object>.
DefaultWorklistPredicate	Workflows: com.coremedia.workflow.WfProcess Tasks: com.coremedia.workflow.WfTask	The default predicate filters tasks and workflows by the three categories: my tasks, offered tasks and my workflows Own predicates must implement `java.lang.Cloneable` and must extend hox.corem.editor.workflow.AbstractWorklistPredicate and should extend the default filter hox.corem.editor.workflow.DefaultWorklistPredicate for convenience.

Table 5.17. Predicate classes for filtering workflows

Predicates for filtering documents

The following filters are predefined and can be configured without stating the appropriate package. They can only be used for `<Predicate>` elements within the `<Filter>` element. For each predicate, you need to define an attribute `name` in the `<Filter>` element. Otherwise, no name for the filter will be shown in the **View|Filter** menu. You will find the names to use in the next table.

Class	Description

`DatePredicate`	Filters out documents which are older than the current date minus an offset which can be set, using the attribute `relativeOffset`. The unit of `relativeOffset` is milliseconds. You have to set the name of the document property to check, using the attribute `name`.
	
	Example:
	
	`<Explorer name="my-configurable-explorer">`
	
	`<Filter name="7-days-modified-date-filter>`
	
	`<Predicate class="DatePredicate" name="modificationDate_" relativeOffset="604800000"/>`
	
	`</Filter>`
	
	`...`

`MapPredicate`	Filters out specially defined documents. To do so, two sub elements of `<Predicate>` are provided. `<FilterMap>`

Table 5.18. Predicate classes for filtering documents

Class	Description

	In this element, the attribute `document` can be used to assign the document type which should be filtered and the attribute `property` can be used to assign the property which should be evaluated.
	
	`<FilterSet>`
	
	In this element, the attribute `filter` can be used to define the string which should be filtered. Only exact matches will be filtered, the check is case-sensitive.
	
	Example:
	
	`<Predicate class="MapPredicate">`
	
	`<FilterMap document="Article" property="Headline"/>`
	
	`<FilterSet filter="Sports"/>`
	
	`</Predicate>`
	
	All documents of type "Article" with exactly the string "Sports" in the property `Headline` would be filtered.
	
	If you define a name for this filter in the `<Filter>` element, you need to create a custom bundle (see Section 4.10, “Localization” [73] for localization) containing this name.

`UndeletedPredicate`	Filters deleted documents. You need to set the attribute `name` of the `<Filter>` element to undeleted-filter.
`PublishedPredicate`	Only displays published documents. You need to set the attribute `name` of the `<Filter>` element to "published-filter".
`UnapprovedUnpublishedPredicate`	Displays documents which have been neither published nor approved. You need to set the attribute `name` of the `<Filter>` element to "unapprovedunpublished-filter".
`ToBeApprovedPredicate`	Displays documents which have been moved, renamed, marked for deletion or where the latest version has not been approved. You need to set the attribute `name` of the `<Filter>` element to "tobeapproved-filter".
`ToBePublishedPredicate`	Displays documents which have been moved, renamed, marked for deletion or for which a new version exists. This has been approved but not published yet.

Class	Description
	You need to set the attribute `name` of the `<Filter>` element to "tobe-published-filter".

Own Predicates for filtering documents

Default predicate	Sorted objects	Description
None	Documents: hox.corem.editor.proxy.DocumentVersionHolder	Own predicates must implement com.coremedia.common.util.Predicate<Object>.

Table 5.19. Programming own predicates

5.1.4 Column Classes

```
<TableDefinition>
        <ColumnDefinition name="Text" class="StringColumn"
            weight="1.0">
                <DisplayMap document="Text" property="Headline"/>
                <DisplayMap document="Image" property="AltText"/>
                <DisplayMap document="*"        property="name_"/>
                .
                .
        </ColumnDefinition>
</TableDefinition>
```

Example 5.6. Example for the use of a column class

Column classes define how content is displayed in columns. A distinction can be made between column classes for different content:

- **Predefined content**

  These classes display the content of predefined properties of a document, such as DocumentTypeColumn, DocumentVersionColumn, ResourceDisplacementColumn and PathColumn.
- **Different fields according to the document type**

  These classes display the content of document properties which are filled by the user. The column classes are configured by means of multiple child elements `<DisplayMap document='document type' property='property name'/>`. The type '*****' specifies a field which should be shown for all document types for which no field is explicitly given.
- **Content for workflows**

These column classes display predefined content of workflows. The classes are shown in the following table.

Alternatively, a renderer can be set for each column class. The renderer carries out the display itself. For this purpose, an element, <Renderer class="Renderer class"/> must be embedded in the ColumnDefinition element. The renderer must be a subclass of the class hox.corem.editor.toolkit.columnrenderer.LayoutColumnRenderer. See the API documentation.

The standardized column classes (IntegerColumn, StringColumn, DateColumn, ...) as well as DocumentTypeColumn, DocumentVersionColumn, ResourceDisplacementColumn and PathColumn define which contents are held in the columns. Sort criteria can be chosen by a click on the column title bar, indicating the content type. Another click reverses the sort order.

Column classes for workflows

Table 5.20. Column classes for workflows

Class	Description
`hox.corem.editor.workflow.columns.WorklistDetailColumn`	Shows some information about the WfInstance (task or process) in the workflow list.
`hox.corem.editor.workflow.columns.WorklistProcessColumn`	Shows some process information in the workflow list. The process is either the WfInstance itself or the parent process in case the WfInstance is a task.

Column classes for predefined columns

Table 5.21. Column classes for predefined columns

Class	Field type
`DocumentTypeColumn`	Type of the document. Exists implicitly in every document; sorted according to document type.
`DocumentVersionColumn`	Version of the document. Exists implicitly in every document.
`ResourceDisplacementColumn`	State of the document. Exists implicitly in every document; sorted according to the importance.
`PathColumn`	Path of the document. Exists implicitly in every document.

Column classes for user defined document properties

Class	Field type
GenericColumn	Default class when no other class is defined. Is not sorted.
IntegerColumn, StringColumn, DateColumn	Basic properties; sorted according to value.
SgmlTextColumn	SgmlTextProperty; is not sorted.
BlobColumn	BlobProperty; sorted according to the size of the blob.
ImageColumn	BlobProperty with MIME type `Image`; sorted according to size.
LinkListColumn	LinkListProperty; sorted according to document type.

Table 5.22. Column classes for user defined document properties

Some of the properties which exist implicitly in all documents can be addressed using the field names in the `Property` attribute given in the following table:

Property name	Java type	Description
`id_`	Integer	Resource ID
`name_`	String	Name of the resource in its folder
`lastName_`	String	The last name of the resource, that is, the name before the last renaming
`folderId_`	Integer	ID of the folder in which the resource lies
`baseFolderId_`	Integer	The ID of the move delimiting folder from which this resource is a direct child of
`creationDate_`	Calendar	Creation date of the resource
`creator_`	UserModel	User who created the resource
`creatorId_`	Integer	The ID of the creator of the resource
`documenttype_`	String	The name of the document type of the document
`modificationDate_`	Calendar	Date of the last modification of the resource
`modifier_`	UserModel	User who last modified the resource
`modifierId_`	Integer	ID of the modifier
`isDeleted_`	Boolean	Is the resource marked as deleted?
`isCheckedOut_`	Boolean	Is the resource checked out for editing?
`version_`	Integer	Version number

Table 5.23. Implicit properties

Property name	Java type	Description
latestVersion_	Integer	Same as version_
isToBeWithdrawn_	Boolean	Denotes if the resource is to be withdrawn
isLive_	Boolean	Is true if the resource exists on the Live Server.
isNew_	Boolean	Is true if the resource has never been present on the Live Server.
isMoved_	Boolean	Is false if the resource is not in the recycle bin and FOLDER_ID is different from LAST_FOLDER_ID
isRenamed_	Boolean	Is true if NAME is not equal to LAST_NAME
isArchived_	Boolean	Is true if the FOLDER_ID is the recycle bin
isUnarchived	Boolean	Is true if the LAST_FOLDER_ID is the recycle bin and current FOLDER_ID is not in the recycle bin
latestApprovedVersion_	Integer	Dynamic property that is only valid for queries. Denotes the latest approved version.
latestPublishedVersion_	Integer	Dynamic property that is only valid for queries. Denotes the latest published version.
placeApprovalDate_	Calendar	The date of the place approval
placeApproverId_	Integer	The ID of the place approver
placeApprover_	UserModel	Denotes the place approver
isPlaceApproved_	Boolean	Denotes if the place of the resource is approved
syncDate_	Calendar	The date of the last synchronization, that is, publication to the Master Live Server
syncerId_	Integer	The ID of the syncer

5.1.5 Renderer Classes

```
<ColumnDefinition name="Image" class="ImageColumn">
  <Renderer class="ImageLayoutColumnRenderer" width="50"
   height="50"/>
  <DisplayMap document="Picture" property="thumbnail"/>
</ColumnDefinition>
```

Example 5.7. Example for the use of a renderer class

```
<ColumnDefinition name="Head">
  <Renderer class="StringPropertyLayoutColumnRenderer"/>
  <DisplayMap document="Article" property="headline"/>
</ColumnDefinition>
```

Renderer classes can be used to display the content of document fields in columns of the Explorer window, Query window and Resource chooser window. The predefined renderers are listed in the next table. For own renderer classes, you need to extent the class hox.corem.editor.toolkit.table.columnrenderer.LayoutColumnRenderer.

Table 5.24. Provided Renderer classes of CoreMedia CAP

Class	Description
GenericPropertyLayoutColumnRenderer	Combines the ImageLayoutColumnRenderer, LinkListPropertyLayoutColumnRenderer, SgmlTextPropertyLayoutColumnRenderer and StringPropertyLayoutColumnRenderer.
StringPropertyLayoutColumnRenderer	Usable for string, integer, Boolean and date fields. Displays plain text.
SgmlTextPropertyLayoutColumnRenderer	Usable for `SgmlText` fields. With the attribute `displayLength`, you can define the maximum length of the text to display. Default is "80".
BlobPropertyLayoutColumnRenderer	Usable for BlobProperty fields. The size of the blob in KB is shown.
ImageLayoutColumnRenderer	Usable for images of type `.jpeg` and `.gif` up to 10kB size. The width and height (in pixel) of the image can be defined with the attributes `width` and `height`.
LinkListPropertyLayoutColumnRenderer	Usable for LinkList fields. Two modes can be applied, using the attribute `displaySummary`. "True" (the default) will only show the number of links contained in the LinkList field. "False" will display one or more links included in the field. You can define the maximum number of shown links using the attribute `noRenderedLinks` (default is "1"). If you choose "0", all links will be shown.
DisplacementColumnRenderer	Shows the displacement status of a document.
ImpliedPropertyLayoutColumnRenderer	Usable for implied properties of the document (see the table in this chapter above). Display the property as plain text and optionally a state or document type icon using the attributes `displayStateIcon` and `displayTypeIcon` respectively. If you choose both icons to show, only the state icon will be displayed.

5.1.6 Initializer Classes

Example 5.8. Example for the use of an initializer

```
<DocumentType name="article">
  <PropertyType name="Author">
    <Initializer class="myInitializer" myattribute="MyValue"/>
  </PropertyType>
</DocumentType>
```

Initializer fill the fields of a newly created document with default values. Thus, an initializer is defined in the element `<PropertyType>` of the XML file. A `GenericInitializer` is provided which fills String, Integer, Date and SGML text fields with a default value. In addition, an own initializer class can be used which must be declared via the element `<Initializer>`. Own initializers can be written, which must implement the interface `hox.corem.editor.initialization.Initializer`.

Table 5.25. Initializer classes

Initializer	Description

GenericInitializer	This initializer fills String, Integer, Date and SGML text fields with a default value. The initializer is called implicitly, if the attribute `initialValue` of the element `< PropertyType>` is used. That is, the initializer is not explicitly specified via the attribute `class`. Thus, it is not allowed to use this initializer with the attribute `class`. If you want to initialize a Date field, you can use the following date formats:
	→ yyyy-MM-dd
	An ISO8601 format according to `java.text.SimpleDateFormat`.
	→ yyyy-MM-ddTHH:mm:ss
	An ISO8601 format according to `java.text.SimpleDateFormat`.
	→ dd.MM.yyyy HH.mm
	This format is locale dependent. It only works with a German locale.
	Example:
	A generic initializer that initializes the property "limit" with the value "20":
	`<DocumentType name="DynamicCollection">`
	`<PropertyType name="limit" initialValue="20"/>`
	`</DocumentType>`

LinkListInitializer	This initializer fills a LinkList field with a document. Either the document defined via the `path` attribute is used or - if you omit the document and

Initializer	Description
	enter only the folder - the first document with the appropriate document type found in the folder given via the attribute path is used. Example: `<Initializer class="LinkListInitializer" path="/default"/>`

5.1.7 Validator Classes

Example 5.9. Example for the use of a validator

```
<DocumentType name="article">
  <PropertyType name="Author">
    <Validator class="NotEmpty2"/>
  </PropertyType>
</DocumentType>
```

When a document is checked in, validators test whether certain conditions are fulfilled. At the present time, four validators are delivered with the *Site Manager*. These check, whether a field

- is filled,
- is filled with a certain pattern,
- contains an integer in a specified range
- A link list field can be checked for a minimal, maximal number of entries and for unique entries. In addition, the allocation of a linklist field can be checked.

Another class can be used to combine several validators and one always validates to "true".

In addition, own validator classes can be written. These classes implement the interface hox.corem.editor.validation.Validator2 and must be specified via the element `<Validator>`.

Table 5.26. Validator classes

Validator	Description
`SetValidator2`	Using this class several validators can be combined as subelements Validator of the element Validator. SetValidator2 uses the Validator2 interface and replaces the deprecated SetValidator class. Example:

Validator	Description

	`<DocumentType name="Text">`
	`<PropertyType name="Content">`
	`<Validator class="SetValidator2">`
	`<Validator class="my.very.own.TextValidator"/>`
	`<Validator class="my.very.own.SpellValidator"/>`
	`<Validator class="my.very.own.AddValidator"/>`
	`</Validator>`
	`</PropertyType>`
	`</DocumentType>`

`NotEmpty2`	This validator checks whether entries have been made in a field. If the field is empty, an error message is created and the document is not accepted for checking in. The validator must be named as the value in the `Attribute` class of the element `<Validator>`. NotEmpty2 uses the Validator2 interface and replaces the deprecated NotEmpty class.
`GenericValidator`	This validator checks whether a field is filled with the appropriate pattern. If the entry is wrong, an error message is created and the document is not accepted for checking in. The validator is called implicitly, if the attribute `validPattern` of the element `<PropertyType>` is used. As a valid Pattern, any regular expression can be used (see the element `<PropertyType>` in Section 5.2.3, “Configuring Document Types” [129]).
`MinMaxIntegerValidator`	This validator checks whether a field is filled with an integer in the appropriate range. If the entry is wrong, an error message is created and the document is not accepted for checking in. This validator must be set with the class attribute within the element `<Validator>`. The min and max values can be provided with the attributes `min` and `max`. Example: `<Validator class="MinMaxIntegerValidator" min="1" max="10"/>`
`LinkListValidator`	This validator checks whether a linklist field is filled with a minimum (attribute `minLength`) or a maximum (attribute `maxLength`) number of entries. It also checks if a field is filled with unique entries only (attribute `uniqueEntries`) and if the entries are all below a specified path (attribute `path`). Example: `<Validator class="LinkListValidator" minLength="1" path="/looks">`

Validator	Description
	Thus, at least one entry must be contained in the linklist field and the documents belonging to the entries must be located below the folder `/looks`.
`AlwaysTrueValidator`	This validator always validates to "true". If you use multiple editor XML configuration files (see Chapter 3, *Operation and Configuration* [18]), you can use the validator to override other validators.

5.1.8 Comparator Classes

Example 5.10. Example for the use of a Comparator

```
<Editor>
 <DocumentTypes>
   <Comparator class=
     "hox.corem.editor.generic.GenericDocumentTypeComparator"/>
  .
  .
 </DocumentTypes>
  .
 <Explorer name="FirstExplorer">
   <TreeSorter>
     <Comparator class="my.comparator"/>
   </TreeSorter>
 </Explorer>
</Editor>
```

A comparator is used for sorting objects. There are different comparators for sorting different object types:

- Document types shown in the **File|New resource** menu in the *Site Manager*.
- Columns
- Folders shown in the folder view of the *Site Manager*.
- Workflows shown in the **File|New workflow** menu in the *Site Manager*.

Comparators for sorting document types

Table 5.27. Comparators for document types

Default comparator	Sorted objects	Description
GenericDocumentTypeComparator	Document types: hox.corem.editor.proxy.DocumentTypeModel	The default comparator sorts the document types in alphabetic order. Own comparators must implement the interface `java.util.Comparator`

Server side comparators for sorting rows

For rows, there exists no default comparator. A row can be sorted according to the different columns which are shown. Own comparators must implement hox.corem.editor.toolkit.table.NamedDocumentVersionComparator. The following table shows the predefined column comparators which are implicitly used by the provided column classes.

Table 5.28. Server-side comparators for sorting rows

Comparator	Sort order
BlobComparator	Size of file, if equal the MIME type.
BooleanComparator	True > false
DateComparator	Chronological
DocumentVersionComparator	Version
IdComparator	Numeric
IntegerComparator	Numeric
LinkListComparator	Document type in alphabetic order, if equal the number of list entries.
NameComparator	Alphabetical
DocumentPathComparator	Alphabetical
ResourceDisplacementComparator	Sort order: 1. Published and deleted 2. Published and moved 3. Published and removed 4. New and marked for deletion 5. Moved out of trash 6. New 7. All others, if equal document type
StringComparator	Alphabetical
DocumentTypeComparator	Alphabetical

Client side comparators for sorting rows

For rows, there exists no default comparator. A row can be sorted according to the different columns which are shown. Own comparators must implement hox.corem.editor.toolkit.table.NamedDocumentVersionComparator. Client side comparators get objects of types hox.corem.editor.proxy.DocumentVersionHolder. The following table shows the predefined column comparators which can be used on the client side.

Comparator	Sort order
NameComparator	Alphabetical
NullComparator	Alphabetical
StringComparator	Alphabetical
TypeComparator	Alphabetical

Table 5.29. Client-side comparators for sorting rows

Comparators for sorting folders

Default comparator	Sorted objects	Description
GenericTreeComparator	Folders: hox.corem.editor.proxy.ResourceHolder	The default comparator sorts the folders in alphabetic order. Own comparators must implement `java.util.Comparator`.

Table 5.30. Comparators for sorting folders

Comparators for sorting workflows

Default comparator	Sorted objects	Description
GenericProcessComparator	Workflows: com.coremedia.workflow.WfProcess	The default comparator sorts the workflows in alphabetic order. Own comparators must implement `java.util.Comparator`

Table 5.31. Comparators for sorting workflows

5.2 Configuration Possibilities in the XML Files

The *Site Manager* can be configured with the settings in the XML files (see above). Their default location is the directory `<InstallDirectory>/properties/corem`. Customize the editor, by adjusting the following elements:

- Enter a user name and a password for the login. See element `<Editor>`.
- Select the language and country settings which should be used, preferably located in `editor-startup.xml`. See element `<Locale>`.
- Circumvent the standard login window with your own authentication factory. See element `<AuthenticationFactory>`.
- Determine which Web Extensions such as a preview should be used. See element `<WebContext>`.
- Determine which browser should be used. See element `<WebBrowser>`.
- Select and configure the appearance of the fields in the document overview of the main window, of the query window and of the selection window for internal links. See element `<Explorer>`.
- Determine the filters used on the documents in the document overview of the main window. See element `<Filter>`.
- Determine the filters and sorting algorithms affecting the folders in the folder overview. See elements `<TreeSorter>`, `<TreeFilter>`.
- Set a factory for client side properties. See element `<PropertyModelFactory>`.
- Define multiple views for the document overview of the main window, which can be selected with the menu item **View|Display**. See element `<Explorer>`.
- Determine which fields of a document type should be shown in the document window. See element `<Documents>`.
- Determine which document types should be shown in **File|New**. See element `<DocumentTypes>`.
- Define initializer and validators for the fields of newly created documents. See element `<DocumentTypes>`.
- Set certain conditions for these fields (editable, obligatory field ...). See element `<Property>`.
- Allocate certain editors to the fields and in this way, for example, define a selection field with certain preset values. See element `<Property>`.
- Enable or disable the spell checker. See element `<SpellChecker>`.
- Configure the class for language determination of a property used by the spell checker. See element `<PropertyLanguageResolverFactory>`.
- Configure the appearance of the workflow. See elements `<Workflow>`, `<Processes>`.
- Enable or disable the remote control of the editor. See element `<RemoteControl>`.

A formal description of the syntax of this XML file can be found in the corresponding DTD in `<InstallationDir>/lib/xml/coremedia-editor.dtd`. The XML files must obey the DTD, but are not validated against the DTD. Find the default `editor.xml` file in the `editor-components/editor` module in the development workspace of *CoreMedia Project*.

In the following section, the configuration of the *Site Manager* via the file `editor.xml` is described.

The BeanParser, that is used to parse the *Site Manager* configuration allows you to configure all bean properties of the beans that are introduced in the following. Since not all configuration hooks will be explained, it's always a good idea to consult the Javadoc and discover all configuration possibilities.

5.2.1 General Configuration

Using these elements, some general features can be configured.

<Editor>

Child elements: `<AuthenticationFactory>, <DocumentTableLayout>?, <Locale>?, <Preview>, <PropertyModelFactory>?, <RemoteControl>? <DocumentTypes>?, <Documents>?, <Explorer>*, <ResourceChooser>?, <Query>?, <Search>?, <SpellChecker>, <PropertyLanguageResolverFactory>, <ResourceNamingFactory>?, <Workflow>?, <Processes>?, <FrameFactory>?, <WebBrowsers>, <WebContext>*`

Parent elements:

Example 5.11. Example for the Editor element in editor-startup.xml

```
<Editor loginName="test" loginDomain="test"  loginPassword="test"
 loginImmediate="true">
       .
       .
       .
</Editor>
```

You can enter user name and password using the element `<Editor>`. The login window of the *Site Manager* is automatically filled with these data. If you set the attribute `loginImmediate="true"`, this login info will immediately accepted and the login will proceed. The settings for the Editor element must be located in `editor-startup.xml`.

Table 5.32. The attributes of the element Editor

Attribute	Description
`class`	This attribute is used to enter the editor class to use. Default is `hox.corem.editor.generic.GenericEditor`.
`loginName`	This attribute is used to enter the default name for login. If no name is entered, the name from the environment is used. You can always change the name during login, it is just a preset. If a login name should be predefined, it must be set in the `editor-startup.xml` file. If a global `editor.xml` file is used for all users it might be sensible to set the login name in the `editor.properties` file.
`loginPassword`	This attribute is used to enter the default password for login. If no password is entered, the login name is used. You can always change the password during login, it is just a preset. If a login password should be predefined, it must be set in the `editor-startup.xml` file. If a global `editor.xml` file is used for all users it might be sensible to set the login password in the `editor.properties` file.
`loginDomain`	This attribute is used to enter the default domain for login. You can always change the domain during login, it is just a preset. If a login domain should be predefined, it must be set in the `editor-startup.xml` file.
`loginImmediate`	If this attribute is set to "true", an attempt is made to connect directly to the server with the login data given above. The login window does not appear. The default value is "false".
`showCurrentUser`	If this attribute is set to "true", the name of the current user of the editor is shown at the top of the window. Default is "false", that is, the user name is not shown.
`startup`	This attribute defines the *Site Manager* window to start with. Possible values are "OpenExplorer" "OpenQuery", "OpenWorkflow", "OpenUserManager" which will open the respective window. By default, the *Site Manager* starts with the Explorer window ("normal" user) or with the user manager window ("administrator" user).
`startupMode`	This attribute defines the start-up mode for administrators. If set to "4.2", the super user with ID "0" always starts with the User Manager window. All other users will start with the window defined using "startup". If set to "5.0", the super user with ID "0" and all members of the administration group start with the User Manager window. All other users will start with the window defined using "startup". Default setting is "5.0".
`enableExplorer`	This attribute is used to disable the Explorer window of the *Site Manager* (false). Default is "true", so the Explorer window can be opened.
`enableDirectPublication`	This attribute is used to enable direct publication ("true"). Default is "false", so no direct publication icons and menu items are shown.

Attribute	Description
`enableWorkflow`	This attribute is used to disable all workflow related menu items and icons ("false"). Default is "true", therefore workflow features are enabled.
`removeEmptyParagraphs`	The *Site Manager* adds empty paragraphs around tables in order to enable the user to enter content before or after the table (this circumvents a Swing problem). By default, ("false") these empty paragraphs are saved on the server. If you set this attribute to "true", empty paragraphs without attributes will be removed in the following cases when writing rich text back to the server: → At the beginning of rich text, if a table follows: `<div><p/><table>` → At the end of rich text following a table: `</table><p/></div>` → At the beginning of a table cell, if a table follows: `<td><p/><table>` → At the end of a table cell following a table: `</table><p/></td>` Warning: If enabled, the representation of rich text on the server will be changed, if a document is saved in the *Site Manager*.
`mayChooseMemberFromOtherDomain`	If set to "false" (default) only members of the administrator group are allowed to search for users in other domains using the User Manager window. Non administrators can only search in their own domain. If set to "true", every user may choose a domain for search.

<AuthenticationFactory>

Child elements:

Parent element: `<Editor>`

You can use this element in order to set your own authentication factory. The element needs to be located in the `editor-startup.xml` file.

```
<Editor>
  <AuthenticationFactory
class="com.myFactory.OwnAuthenticationFactory"/>
.
.
.
</Editor>
```

This factory circumvents the standard login dialog and fetches principal and credentials by custom means.

Attribute	Description
class	The fully qualified name of your authentication factory. Your class must implement hox.corem.editor.AuthenticationFactory and needs a public no-argument constructor. See the Javadoc for details.

Table 5.33. Attribute of element <AuthenticationFactory>

<DocumentTableLayout>

Child elements:

Parent elements: <Editor>

You can globally define the appearance of document tables used in the explorer view, query view, publication view and resource choosers. By default, the tables are plain white without separators. See Section 5.2.5, "Configuring Table Views" [139] for more specific table configuration.

```
<Editor>
   <DocumentTableLayout horizontalLines="true"
     evenBgColor="FFFFFF" oddBgColor="FFCCCC"/>
   .
   .
   .
</Editor>
```

Example 5.12. Example of the DocumentTableLayout element

The background color settings do not apply to publication views, because the background colors of this view visualize the result categories of the publication

Attribute	Description
horizontalLines	With this attribute set to "true", you can enable horizontal separator lines between the table rows. Default is "false".
verticalLines	With this attribute set to "true", you can enable vertical separator lines between the table columns. Default is "false".
evenBgColor	With this attribute, you can set a background color for even table rows. Use hexadecimal values, representing RGB values, such as FFCCCC for a light red. Default is white. The numbering of rows starts with "0", so the first row is even.
oddBgColor	With this attribute, you can set a background color for odd table rows. Use hexadecimal values, representing RGB values, such as FFCCCC for a light red. Default is white.

Table 5.34. Attributes of the DocumentTableLayout element.

<Locale>

Child elements:

Parent elements: `<Editor>`

You can select the language and country settings which should be used by the *Site Manager* with the element `<Locale>`. These settings determine the language used in the GUI of the *Site Manager*. The locale that you set in `editor-startup.xml` will be used for the Login screen you can overwrite this setting with a `<Locale>` element in the `editor.xml` file. So you can define group specific localizations for example.

Example 5.13. Example for the Locale element

```
<Editor>
        <Locale language="de" country="DE"/>
        .
        .
        .
</Editor>
```

Using this element of the XML file, details of the localization of the *Site Manager* are given. If the element is not used, the environment settings are used. As a default, this element is used in the `editor-startup.xml` file.

Table 5.35. Attributes of element <Locale>

Attribute	Description
`language`	The language used in the program. At present, there are locales for English ("en") and German ("de"). The locales follow the usage in `java.util.Locale`.
`country`	Country-specific settings. At present, there are locales for the United States ("US") and Germany ("DE"). The locales follow the usage in `java.util.Locale`.

<Bundle>

Child elements:

Parent elements: `<Editor>`

Example 5.14. Example for the Bundle element

```
<Editor>
        <Bundle name="my/bundle"/>
        .
        .
        .
</Editor>
```

The `Bundle` element of the XML file defines the bundle file to use for localizing the *Site Manager* and for user defined properties. The file defined in the `Bundle` element, will be looked up by the *Site Manager* and will overwrite the default values. For the bundle shown in the example above, the following file has to be created in the classpath (`<CMInstallationDirectory>/classes`):

- `my/bundle.properties` for German localization
- `my/bundle_en.properties` for English localization

In this file, name/value pairs in the format `my-column-title=My Column title` are used. If "my-column-title" matches the value of an attribute `name` in the element `<ColumnDefinition>`, then "My Column title" would be the name of a column shown in the *Site Manager*. It is also possible to store bundle files for other languages simultaneously. For example, you can store the English names in a file `bundle_en.properties`. More details can be found in Section 4.10, "Localization" [73]. As a default, this element is used in the `editor-startup.xml` file.

Table 5.36. Attribute of the <Bundle> element

Attribute	Description
`name`	With this attribute, the name of the bundle file is entered. The name must correspond to a file in the Classpath. You must obtain the name of the bundle from your developers.

<Preview>

Child elements: `<Browser>*`

Parent elements: `<Editor>`

The `<Preview>` element is deprecated, use `<WebContext>` instead.

Example 5.15. Example for the Preview element

```
<Editor>
        <Preview host="zeus" port="8001"
          uriPath="coremedia/generator/goto"/>
        .
        .
        .
</Editor>
```

This element of the XML file is used to configure the *Content Application Engine* used for preview. The request to the generator then occurs via the URL (if no user defined pattern has been defined):

`http://<host>:<port>/<uriPath>`

Table 5.37. Attributes of the element Preview

Attribute	Description
`host`	The computer on which the *Content Application Engine* runs.
`port`	The port via which the *CAE* is accessed.

Attribute	Description
`uriPath`	The URI prefix for accessing the preview *CAE*. Default is `core media/generator/goto`.
`pattern`	You might configure individual URLs via a custom pattern. The following strings are replaced within the pattern: → %p the protocol to use → %h the host with the preview server → %n the port on the host with the preview server → %u the URI prefix of the resource locator URI → %i the numeric id of the resource locator URI → %v the version of the resource locator URI → %f combines %u and ?id=%i&Version=%v → %s combines %u and ?id=%i → %l returns the URL-encoded string id of the previewed resource Default setting is: `%p://%h:%n/%f`

<Browser>

Child elements:

Parent elements: `<Preview>`

The `<Browser>` element is deprecated. Use `<WebBrowsers>` instead.

Example 5.16. Example for the Browser element

```
<Preview host="zeus" port="8000"
                 uriPath="coremedia/generator/goto">
        <Browser name="Netscape Navigator"
                 command="c:\\Programme\\Netscape\\Communicator
                          \\Program\\netscape.exe %s"/>
        .
        .
</Preview>
```

You can select the browser for the preview using the element Browser. Multiple browsers can be entered. You can choose the browser to use from the **File|Preview** menu of the overview window. If you do not define any browser, the preview cannot be executed.

Table 5.38. Attributes of the element Browser

Attribute	Description
name	Name of the browser to start. Any number of names can be entered here.
command	Command for starting the browser. The string `%s` in the example is replaced by the URL of the document for display.
pattern	This attribute describes how the URL passed to the browser is constructed. `%p` Protocol `%h` Computer name `%n` Port number `%u` URI prefix `%i` URI postfix `%f` combined `%u` and `%i` Example: `pattern="wap://%h:%n/wap/%f"` Default: `%p://%h:%n/%f`
optional	Specifies whether this browser is optional when doing a preview with all configured browsers (for example by clicking the Preview button in the toolbar or by selecting **File|Preview|All**). The Editor only shows errors for non-optional browsers or if no browser could be started at all. Allowed values are `true` and `false`. Default is `false`.

<RemoteControl>

Child elements:

Parent elements: `<Editor>`

Example 5.17. Example for the RemoteControl element

```
<Editor>
        <RemoteControl enabled="true" port="44444"/>
        .
        .
</Editor>
```

This element is used to configure whether the CoreMedia can be remote controlled or not and to set the port where to listen for requests.

The remote control of the Site Manager allows you to execute

- all commands which may be executed on resources in the explorer view,

- all commands which may be executed on process and task instances in the workflow view and
- custom commands, which may be executed on resources, process and task instances, or external parameters given to the command.

Technically, the remote control is realized via HTTP by an embedded web server inside the Editor, which listens to remote control requests. Note that if you start two editors on the same computer which use the same configuration, remote control is disabled for the second editor, since it tries to use the same port. You could use custom configuration files for different users, specifying different ports, though.

Table 5.39. Attributes of the element RemoteControl

Attribute	Description
enabled	This attribute controls whether the *Site Manager* can be remote controlled ("true") or not ("false"). Default is "false".
port	This attribute determines the port which will be used for the remote requests. Default is "44444".

Editor Remote Control URLs

Requests have to be addressed to an URL of the pattern

```
http://localhost:<port>/coremedia/control?<parameters>
```

The port has to be the same as in the XML configuration.

The parameters determine, which command to execute and on which data to execute it. There are some well-known parameters which ease the usage:

Table 5.40. Parameters of the remote control URI

Parameter	Description
command	Allows you to specify the name of the command class, which is executed upon the request. Custom command classes have to implement the interface hox.corem.editor.toolkit.Command or one of its subinterfaces. If there is no dot in the command name, `hox.corem.editor.commands` is prepended.
`resourceId` or *`documentId`*	Allows you to specify one or more resources or documents, on which a hox.corem.editor.commands.ResourceCommand is executed.
`processInstanceId`	Allows you to specify one process instance ID, on which a hox.corem.editor.commands.ProcessInstanceCommand is executed.

Parameter	Description
taskInstanceId	In conjunction with a `processInstanceId`, it allows you to specify one task instance by its id, on which a hox.corem.editor.commands.TaskInstanceCommand is executed.

If the command class is a hox.corem.editor.commands.MapCommand, all the parameters are passed to the command as a Map.

Examples for remote control URLs are:

```
http://localhost:44444/coremedia/control?command=OpenResourceInExplorer&resourceId=4712
```

Opens the document with the id 4712 in the explorer view

```
http://localhost:44444/coremedia/control?command=OpenDocument&resourceId=4712
```

Opens the document with the id 4712 in a document view

```
http://localhost:44444/coremedia/control?command=ShowResourceInformation&resourceId=4712
```

Opens the resource information view for the resource with the id 4712

```
http://localhost:44444/coremedia/control?command=OpenWfInstanceInWorkflow&processInstanceId=1&taskInstanceId=2
```

Opens the task instance 2 from the process instance 1 in the workflow view

```
http://localhost:44444/coremedia/control?command=StoreProperties&documentId=4712&Text=Test
```

Stores "Test" in the property Text of the document with id 4712

```
http://localhost:44444/coremedia/control?command=CreateDocument&parentId=4711&type=Article&name=NewDocument&Text=Test
```

Creates a new document named NewDocument with the document type Article below the folder with id 4711 and stores "Test" in the property Text of the document.

```
http://localhost:44444/coremedia/control?command=CreateFolder&parentId=4711&name=NewFolder
```

Creates a new folder named NewFolder below the folder with id 4711

Prior to using the commands, you have to check the access control. Requests are only accepted, if

- their origin is the same computer as the one the Editor is running on and
- their command is activated in the remote control policy file.

The remote control policy file `$INSTALL_DIR/properties/policy/editor.policy` is a standard Java policy file and may be edited with the Java policy tool. It grants execute rights to commands by specifying the name and the package of the command.

<FrameFactory>

Parent elements: `<Editor>`

Example 5.18. Example of the FrameFactory element

```
<Editor>
  <FrameFactory explorerViewClass="my.ExplorerView"
  publishViewClass="my.PublishView"
  workflowViewClass="my.WorkflowView"/>
  .
  .
</Editor>
```

You can use this element to add your own `ExplorerView`, `PublishView`, `QueryView` or `WorkflowView` classes to the editor.

Table 5.41. Attributes of element <FrameFactory>

Attribute	Description
`explorerViewClass`	Use this attribute to define your own explorer view for the editor.
`publishViewClass`	Use this attribute to define your own publication view for the editor.
`queryViewClass`	Use this attribute to define your own query view for the editor.
`workflowViewClass`	Use this attribute to define your own workflow view for the editor.

<PropertyModelFactory>

Child elements: `%varies;`

Parent elements: `<Editor>`

Example 5.19. Example for the PropertyModelFactory element

```
<Editor>
  .
  <PropertyModelFactory class="my.propertyModelFactory"/>
  .
</Editor>
```

This element of the XML file is used to specify a class which implements the interface hox.corem.editor.proxy.PropertyModelFactory and which should be used in the *Site Manager*. Here an own PropertyModelFactory class can be programmed (see the API documentation) and invoked by the attribute `class`.

Attribute	Description
class	This attribute is used for selecting a PropertyModelFactory for use with the *Site Manager*.

Table 5.42. Attribute of element <PropertyModelFactory>

<ResourceNamingFactory>

Parent elements: `<Editor>`

```
<Editor>
 <ResourceNamingFactory class="MyResourceNames"/>
 .
 .
</Editor>
```

Example 5.20. Example of the ResourceNamingFactory element

You can use this element to define your own ResourceNamingFactory. This factory creates and modifies names of resources and folders. This is intended to enable customization of how resources and folders are named or renamed in different projects or to check for allowed resource names (see the API documentation for details). Own resource naming factory classes must implement ResourceNamingFactory or extend BasicResourceNamingFactory.

Attribute	Description
class	This attribute is used to enter the ResourceNamingFactory to use.

Table 5.43. The attributes of the <ResourceNamingFactory> element

<WebBrowsers>

Parent elements: `<Editor>`

Child elements: `<WebBrowser>`

```
<Editor>
  ..
    <WebBrowsers>
      ..
    </WebBrowsers>
..
</Editor>
```

You can use the `<WebBrowsers>` element to configure web browser definitions for Web Extensions such as the preview with the `<WebBrowser>` child element. The `<WebBrowsers>` element has no attributes.

<WebBrowser>

Parent elements: <*WebBrowsers*>

```
<WebBrowsers>
  <!-- Standard Windows IE installation -->
  <WebBrowser id="Internet Explorer" os="win"
  command="c:\\Program Files\\Internet Explorer\\Iexplore.exe %s"/>


  <!-- IE installation in german locale on Windows -->
  <WebBrowser id="Internet Explorer" os="win" language="de"
  command="c:\\Programme\\Internet Explorer\\Iexplore.exe %s"/>
</WebBrowsers>
```

This element configures web browser installations for a given locale of the *Site Manager* and operating system. Web extensions (see `<WebExtension>`) may open several web browsers (Preview) or the first matching web browser. Therefore, the order of `<WebBrowser>` elements is important.

The example above configures two Windows web browsers, one with language attribute set to 'de'. If a web extension running on German locale wants to select a browser, it should open the German browser. A precedence list defines which browser is selected.

1. os
2. language
3. country
4. no attribute

In the example above, for both browsers the `os` attribute has been set but the German browser is selected because it has a `language` attribute that matches the language of the German locale. If you delete the `os` attribute in the German browser configuration, the other browser will be opened.

In rare conditions a matching browser can not be opened. Take, for example, the configuration above and call a preview web browser from a *Site Manager* with a German locale on a French Windows system. The command `c:\\Programme\\Internet Explorer\\Iexplore.exe %s` can not be executed on the French system because "Programme" will not be found. In this case, the first browser is taken that can be opened, independently of any `os` or `language` settings.

Attribute	Description
`id`	The name of the browser, for example Internet Explorer. Use the same id for the same browser application, like FireFox for all Firefox configurations.
`os`	The name of the operating system. This string must be a substring of the value of the Java system property `os.name` (case-insensitive). This attribute is optional. If not set, the command must be executable on all operating systems your *Site Manager* runs on.
`language`	The language of the locale. The value must conform to a valid language in a Java `java.util.Locale` instance. For the English language the valid value is 'en' for the German language the valid value is 'de'. This attribute is optional.
`country`	The country of the locale. The value must conform to a valid country in a Java `java.util.Locale` instance. For the USA the valid value is 'US' for Germany the valid value is 'DE'. This attribute is optional.
`command`	The command to start a browser with a given URL on the configured operating system. For the Internet Explorer on an English Windows installation the command looks as follows: `c:\\Program Files\\Internet Explorer\\Iexplore.exe %s` The suffix `%s` is the placeholder for the URL to load in to the browser.
`optional`	Specifies whether this browser is optional. This feature is used by the Preview web extension when doing a preview with all configured browsers (for example by clicking the Preview button in the toolbar or by selecting **File|Preview|All**). The *Site Manager* only shows errors for non-optional browsers or if no browser could be started at all. Allowed values are true and false. Default is false

Table 5.44. The attributes of the <WebBrowser> element

5.2.2 Defining Group Specific Configuration Files

The *Site Manager* is configured with XML files. It is possible to define special configuration files for distinct groups or users of the CoreMedia system. To configure the usage of special configuration files you may adapt the following properties in the `editor.properties` file (see chapter "Defining XML Files for Configuration" in the *Administration and Operation Manual* for details):

→ `editor.startup.configuration`

- `editor.configuration`
- `group.configuration`
- `user.configuration`

If you only use `group.configuration`, you can define one specific configuration file for each group. To have multiple configuration files for one group, you may configure the set of files and in which order they are parsed in `editor-startup.xml` (default) or in the file configured by `editor.startup.configuration`. Mind that group configuration in `editor-startup.xml` overrides the mechanism one configuration file per group which especially means: If users are not member of any group configured in `<ConfigGroups>` no group configurations are applied to these users.

In both cases, that is either with one configuration file per group or with multiple configuration files per group you have to set the property `group.configuration` to point to configuration files with a path relative to `<CoreMediaHome>` or to the URL where to find the files. The path/URL defined has to contain a wildcard `{0}` which will be replaced either by the group name or by the names as defined in the `<Configuration>` element (see below).

Example:

```
group.configuration=properties/corem/editor-{0}.xml
```

The *Content Server* will look in the `properties/corem` directory for a file called `editor-<PlaceHolder>.xml` where `<PlaceHolder>` will be replaced by the values of the `name` attribute of the `<Configuration>` element described below or by the group name if no `<ConfigGroups>` element is used.

If a user is member of more than one group, the exact behavior reading group configuration files is undetermined. If multiple matching `<ConfigGroup>` exist, one of them is chosen by random. If `<ConfigGroups>` configuration is not used but direct mapping groups to configuration files all matching configuration files are read but in an undetermined order. To determine the exact behavior you have to implement your own selection scheme. Proceed as follows:

1. Extend GenericEditor
2. Override the `getConfigurationGroupNames(UserModel user)` method which is inherited from AbstractEditor with your own selection scheme. The default implementation of the method either returns the configuration file names as configured in the `<Configuration>` element (first case) and if no `<ConfigGroups>` element is used the unordered list of groups a user is member of. You might want to use the convenience method `getUserConfigGroups(UserModel user)` to create your own implementation. For further reference see the Javadoc.

3. Add your class to the `class` attribute of the `<Editor>` element in the `editor-startup.xml` file.

<ConfigGroups>

Child elements: `<ConfigGroup>`

Parent elements: `<Editor>`

```
<Editor>
  <ConfigGroups>
     .
     .
   </ConfigGroups>
</Editor>
```

This element combines the elements for the group configuration.

The element has no attributes. If `<ConfigGoups>` is not used but `group.configuration` is set, only the general editor configuration file (default: `editor.xml`) and the matching group specific configuration files will be applied. See the *Site Manager* chapter in the *Administration and Operations Manual* for details.

<ConfigGroup>

Child elements: `<Configuration>`

Parent elements: `<ConfigGroup>`

```
<ConfigGroups>
   <ConfigGroup name="editor" domain="main">
     .
     .
   </ConfigGroup>
</ConfigGroups>
```

This element defines for which group and domain the configuration should be used. It groups the `<Configuration>` elements.

Table 5.45. Attributes of the <ConfigGroup> element

Attribute	Description
name	The name of an existing group in the CoreMedia user management for which the configuration will be used.
domain	The domain of the group.

<Configuration>

Child elements:

Parent elements: `<ConfigGroup>`

```
<ConfigGroups>
   <ConfigGroup name="editor">
      <Configuration name="common"/>
      <Configuration name="special"/>
   </ConfigGroup>
</ConfigGroups>
```

This element defines the name with which the placeholder in `group.configuration` will be replaced and the order in which multiple configuration files are applied. In the example above the placeholder will first be replaced with "common" and then with "special", if the user is member of the "editor" group. This especially means that in case of conflicting settings the settings from the special file will override the settings in the common file.

Table 5.46. Attribute of the <Configuration> element

Attribute	Description
name	Name which will replace the placeholder in the `group.configuration` property of `editor.properties`. In general, this is not the name of an existing group, but it can be.

5.2.3 Configuring Document Types

Using these elements, the document types of *CoreMedia CMS* can be configured.

- The documents usable in the editor (creating by the menu **File|New Resource ...**, copy, move etc.**)** via the sub element `<DocumentTypePredicate>` or `<Predicate>`.
- The sorting of the document types shown in the query view, the resource chooser and in the menu **File|New Resource ...** via the sub element `<Comparator>`.
- The initiators and validators which should be used with document fields via the sub element `<PropertyType>`.

<DocumentTypes>

Child elements: `<DocumentType>*`, `<DocumentTypePredicate>?`, `<Predicate>?`, `<Comparator>?`

Parent elements: `<Editor>`

Example 5.21. Example for the DocumentTypes element

```
<Editor>
 <DocumentTypes>
  .
  .
 </DocumentTypes>
</Editor>
```

This element of the XML file is used to combine the elements for the document types configuration.

The element has no attributes. If no `<DocumentTypes>` element is defined, all document types for which the user has the appropriate rights will be shown (except of abstract document types) and will be arranged alphabetically.

<DocumentType>

Child elements: `<PropertyType>*`

Parent elements: `<DocumentTypes>, <DocumentTypePredicate>, <Predicate>`

Example 5.22. Example of a DocumentType element

```
<Editor>
 <DocumentTypes>
   <DocumentType name="Article">
     <PropertyType name="Text">
       .
       .
     </PropertyType>
       .
       .
   </DocumentType>
   .
   .
 </DocumentTypes>
</Editor
```

This element of the XML file designates the following features:

- The documents usable in the editor via the parent element `<Predicate>`.
- The initiators and validators which should be used with document fields via the sub element `<PropertyType>`.

The element has one attributes. If no `<DocumentType>` element is defined, all document types will be shown (except of abstract document types) and will be arranged alphabetically.

Table 5.47. Attribute of the DocumentType element

Attribute	Description
name	The name of the document type.

<PropertyType>

Child elements: `<Validator>?`, `<Initializer>?, <ModelClass>?`

Parent elements: `<DocumentType>`

```
<DocumentType name="Article">
  <PropertyType name="Source" initialValue="Internally">
    <Validator class="NotEmpty2"/>
  </PropertyType>
    .
    .
</DocumentType>
```

Example 5.23. Example of a PropertyType element

This element is used to provide initializers (or initial values) and validators (or valid pattern) for the properties of the document defined in `<DocumentType>`.

Attribute	Description
name	This attribute is used for configuring the name of the property which should be initialized or validated.
`initialValue`	Using this attribute, the value can be entered with which the property is initialized. See Section 5.1.6, “Initializer Classes” [106] for more details.
`validPattern`	This attribute is used for entering a regular expression against which the content of the property is checked.

Table 5.48. Attributes of the <PropertyType> element

The following table shows the regular expressions which can be used with the attribute `validPattern`.

Regular expression	Description
.	Matches any character except newline.
[a-z0-9]	Matches any single character of the set.
[^a-z0-9]	Matches any single character not in set.
\d	Matches a digit, that is, [0-9].
\w	Matches an alphanumeric character, that is, [a-zA-Z0-9_].
\W	Matches a non-word, that is [^a-zA-Z0-9_].
\metachar	Matches the character itself, that is, |, *, \+.
x?	Matches 0 or 1 x's, where x is any of the above.
x*	Matches 0 or more x's.
x+	Matches 1 or more x's.
x{m,n}	Matches at least m x's but no more than n.
foo|bar	Matches one of foo or bar.
(x)	Brackets a regular expression.

Table 5.49. Regular patterns to use with the attribute validPattern

<Validator>

Child elements: `%varies;`

Parent elements: `<PropertyType>`

```
<PropertyType name="Author">
 <Validator class="MyValidator" myattribute="myvalue"/>
 .
 .

</PropertyType>
```

Example 5.24. Example of the Validator element.

This element is used for setting validator classes which tests, when a document is checked in, whether certain conditions about the content of the document are fulfilled. It is possible to hand over parameters to the class via attributes of the element.

Table 5.50. Attributes of the element <Validator>

Attribute	Description
class	This attribute gives the name of the class which checks the content of the field for desired properties. This test is carried out when check in. See Section 5.1.7, "Validator Classes" [107] for predefined classes.
`%varies;`	This entity stands for further configuration possibilities which depend on the API of the `class` . The specific configuration possibilities must be obtained from your developers.

<Initializer>

Child elements: `%varies;`

Parent elements: `<PropertyType>`

```
<PropertyType name="Author">
 <Initializer class="myInitializer"
    myattribute="myvalue" />
 .
 .

</PropertyType>
```

Example 5.25. Example of the Initializer element

Initializer fill the fields of a newly created document with default values. With the element a class for initializing can be provided. Parameters can be handed to the class via attributes of the element (see the code example above).

Table 5.51. Attributes of the element <Initializer>

Attribute	Description
`class`	This attribute gives the class which presets the fields on initialization of the document. See Section 5.1.6, "Initializer Classes" [106] for predefined classes.
`%varies;`	This entity stands for further configuration possibilities which depend on the API of the `class` . The specific configuration possibilities must be obtained from your developers.

<ModelClass>

Child elements:

Parent elements: `<PropertyType>`

Example 5.26. Element ModelClass

```
<PropertyType name="Time">
 <ModelClass class="MyPropertyModel"/>
 .
 .
</PropertyType>
```

The `ModelClass` element allows you to configure the class from which instances for property values are created. The class attribute is the class name of the property model class, which must have a public no-arg constructor, that is, setting 'class' to 'xxx' corresponds to `Class.forName("xxx")` on which `newInstance()` is called to create new property models. An unqualified `ModelClass` class will be looked up in the package `hox.corem.editor.proxy`.

Table 5.52. Attribute of the element ModelClass

Attribute	Description
class	This attribute defines the class which is used to instantiate objects of the property value.

<Comparator>

Child elements: `%varies;`

Parent elements: `<DocumentTypes>`, `<ColumnDefinition>`, `<TreeSorter>`, `<Processes>`, `<NamedDocumentVersionComparator>`

Example 5.27. Example for sorting the offered document types and the folders in the folder view.

```
<Editor>
 <DocumentTypes>
   <Comparator
class="hox.corem.editor.generic.GenericDocumentTypeComparator"/>
  .
  .
 </DocumentTypes>
```

```
  .
 <Explorer name="FirstExplorer">
   <TreeSorter>
     <Comparator class="my.comparator"/>
   </TreeSorter>
 </Explorer>
</Editor
```

This element of the XML file is used for sorting items:

- The document types shown when creating a new document in the *Site Manager* when used in `<DocumentTypes>`.
- The folders shown in the folder view when used in `<TreeSorter>`.
- The elements shown in a column of the document view, when used in `<ColumnDefinition>`.
- The workflow menu entries when used in `<Processes>`.
- The worklist when used in `<ColumnDefinition>` of `<TableDefinition>` of `<Workflow>`.

Attribute	Description
class	Name of the class in which the sorting comparator is defined. The class must contain a public constructor without arguments and must implement an interface depending on the objects to sort (see the API documentation and Section 5.1.8, “Comparator Classes” [109]).

Table 5.53. Attribute of element <Comparator>

`<DocumentTypePredicate>`

Child elements: `<DocumentType>*, %varies;`

Parent elements: `<DocumentTypes>`

```
<DocumentTypes>
    <DocumentTypePredicate class="MyPredicate"/>
          .
          .
</DocumentTypes>
```

Example 5.28. Example for the DocumentTypePredicate element

You can configure the predicate for filtering document types with the `<DocumentTypePredicate>` element. The configured predicate defines the document types which can be used in the editor (which can be created, copied, moved, for example). Opposed to a predicate configured with the `<Predicate>` element in the `<DocumentTypes>` element, the `<DocumentTypePredicate>` also affects abstract types that can be selected in the editor's query and search views. Note, that you must not use both `<Predicate>` and `<DocumentTypePredicate>` elements in the `<DocumentTypes>` element.

Section 5.1.3, "Predicate Classes" [97] describes the provided classes to filter document types.

Table 5.54. Attributes of the DocumentType-Predicate element

Attribute	Description
class	Name of the class with the predicate for filtering. Own classes must implement the interface com.coremedia.common.util.Predicate<Object> to filter document types, which are represented by instances of class hox.corem.editor.proxy.DocumentTypeModel. If you enter no class attribute, the default predicate is used as described in Section 5.1.3, "Predicate Classes" [97].

<Predicate>

Child elements: `<DocumentType>*, %varies;`

Parent elements: `<Filter>, <TreeFilter>, <DocumentTypes>, <Processes>, <Workflow>`

Example 5.29. Example for the Predicate element used in a Filter element

```
<Filter name="deleted-filter">
        <Predicate class="UndeletedPredicate"/>
                .
                .
</Filter>
```

The predicate for filtering is entered with the `<Predicate>` element. The provided filter classes are described in Section 5.1.3, "Predicate Classes" [97]. Different objects can be filtered:

- The documents shown in the document overview of the *Site Manager* can be filtered, due to different conditions, if the `<Predicate>` element is used in a `<Filter>` element.
- The folders shown in the folder view of the *Site Manager* can be filtered, if the `<Predicate>` element is used in a `<Treefilter>` element.
- The document types which can be used in the editor (which can be, for example, created, copied, moved), if the `<Predicate>` element is used in a `<DocumentTypes>` element. That is the document types defined in the `<DocumentType>` element inside the `<Predicate>` element are no longer accessible (negative list).
- The workflows offered for initiating in the Menu **File|New workflow...** can be filtered, if the `<Predicate>` element is used in a `<Processes>` element.
- The workflows and tasks shown in the sub views *My tasks*, *Offered tasks* and *My workflows* of the Workflow window can be filtered by custom predicates, if the `<Predicate>` element is used in a `<Workflow>` element.

Attribute	Description
class	Name of the class with the predicate for filtering. Own classes must implement the interface com.coremedia.common.util.Predicate<Object>. Depending on the parent elements different object types will be filtered (see Section 4.6, “Program Own Predicate Classes” [60]). If you enter no class attribute the default predicates are used as described in Section 5.1.3, “Predicate Classes” [97].

Table 5.55. Attribute of the Predicate element

5.2.4 Configuring Document Windows

Using these elements, the appearance of the document window of the *Site Manager* can be configured.

<Documents>

Child elements: `<Document>*`

Parent elements: `<Editor>`

```
<Editor>
        <Documents>
        .
        .
        .
        </Documents>
        .
        .
        .
</Editor>
```

Example 5.30. Example for the Documents element

This element of the XML file designates the configuration of the document window.

Attribute	Description
autoCheck-Out	Setting this attribute to "false", you can disable the automatic checkout (start typing in a checked-in document and it will be checked-out automatically) functionality of the *Site Manager*. Default is "true".

Table 5.56. Attributes of the Documents element

<Document>

Child elements: `<Property>*, <Tab>*`

Parent elements: `<Documents>`

```
<Documents>
        <Document type="article">
```

Example 5.31. Example for the Document element

```
            .
        </Document>
        .
</Documents>
```

This element of the XML file is used for entering the document type for which the view is defined and the class used for the view.

Table 5.57. Attributes of element <Document>

Attribute	Description
historyIconCount	Here you can enter the number of versions for which the status icon is shown in the version history of the document window. For example: <Document type="Link" historyIconCount="3"/> The following values are possible: -1: show the icon for all versions (not recommended) 0: show no icons in the version history n: show icons for the last n versions Default value is 5 Note: The value should not be chosen greater than 10 otherwise the editor slows down.
type	Name of the document type for which the view is configured.
viewClass	Here you can enter the class which should be used for displaying the documents. Normally you would not enter anything, and therefore use the default hox.corem.editor.generic.GenericDocumentView. You can use the TabbedDocumentView class for a tabbed view of the properties (see Section 5.1.2, "View Classes" [96] and the description of element Tab below).
compact	The order of properties of a document in the *CoreMedia Editor* depends per default (compact=false) on the order defined in the documenttypes.xml or editor.xml file. If you set compact=true properties which do not use the full window width (int, date) will be shown consecutively.
%varies;	This entity stands for further configuration possibilities which depend on the API of the viewClass. The specific configuration possibilities must be obtained from your developers.

<Property>

Child elements: %varies;

Parent elements: <Document>, <Tab>

Example 5.32. Example for the Property element

```
<Document type="article">
      <Property name="Author" editorClass="ComboBoxStringEditor">
              <HistoryItem value="TextEditor"/>
      </Property>
<Document>
```

This element of the XML file is used for configuring some features of a property in the document window.

Table 5.58. Attributes of element <Property>

Attribute	Description
`name`	This attribute is used for entering the name of the field for configuration.
`visible`	This attribute determines whether the field is shown. Using "false", the field can be hidden. The default value is "true".
`editable`	This attribute determines whether the field can be edited. If "false" is entered, the field cannot be edited. The default setting is "true".
`editorClass`	This attribute is used for entering the class with which the field is edited.
`%varies;`	This entity stands for further configuration possibilities which depend on the API of the `editorClass`. The specific configuration possibilities must be obtained from your developers.

<Tab>

Child elements: `<Property>*`

Parent elements: `<Document>`

Example 5.33. Example for the Tab element

```
<Documents>
        <Document type="article" viewClass="TabbedDocumentView">
                <Tab name="MainData">
                        <Property name="Headline"/>
                        <Property name="Text"/>
                </Tab>
                <Tab name="Administration">
                        <Property name="Editor"/>
                </Tab>
                .
                .
        </Document>
</Documents>
```

This element of the DTD is used to define different tabs for the document view. The properties of a document shown in a tab are configured using the sub element `<Property>`. `<Tab>` can only be applied, when the view class TabbedDocumentView is used.

Table 5.59. Attributes of element <Tab>

Attribute	Description
`name`	Name of the tab which is shown as the label of the tab.

5.2.5 Configuring Table Views

Using these elements, for all windows of the *Site Manager* (except the workflow window) using table views or tree views, it can be configured which properties and how the properties should be shown. The following windows can be configured:

- Explorer window (element `<Explorer>`)
- Query window (element `<Query>`)
- Resource chooser window (element `<ResourceChooser>`)

<Explorer>

Child elements: `<TreeSorter>?`, `<TreeFilter>?`, `<Filter>*`, `<TableDefinition>`

Parent elements: `<Editor>`

Example 5.34. Example for the Explorer element

```
<Editor>
        <Explorer name="configurable-explorer-factory">
                .
                .
        </Explorer>
</Editor>
```

The explorer configuration which can be chosen via the menu item **View|Display** is defined within this element of the XML file.

Table 5.60. Attributes of the <Explorer> element

Attribute	Description
name	Name of the explorer configuration
class	With this attribute the class used for the appearance of the explorer is chosen. Own classes must implement the interface hox.corem.editor.explorer.ExplorerFactory. As a default, the class hox.corem.editor.generic.ConfigurableExplorerFactory is used, which allows customization. Another class which can be used is hox.corem.editor.generic.GenericExplorerFactory which shows the behavior known from CoreMedia CAP 3.2. The class is not configurable. See the API documentation for more details.

Attribute	Description
`extendedContextMenu`	Setting this attribute to "false", you can remove the menu items **Check in**, **Check out** and **Revoke Check out** from the context menu of the Explorer. Default is "true".
`combineSortingAndFiltering`	With this attribute set to "false", separated sorting and filtering of the documents shown in the document overview can be enabled. So it is possible, to use own Comparators with the predefined Predicates. By default, documents may be filtered and sorted in one step on the server. Remote filtering and sorting on the server is much faster than local operations which may slow down the server dramatically. So use local filtering and sorting with care.
`keepSelectionFocused`	Only use this attribute when `combineSortingAndFiltering=false`. If set to "true", this attribute replaces delete, insert event pairs with "content changed" events in the ExplorerTableResourceListChain. This change allows the Explorer `ResourceTable` to track entry updates (which it could not when receiving delete and insert events). Therefore, the focus in the document overview of the Explorer Window will not be lost. The default is "true".
`noLocalSortingAfterInsertAndUpdate`	Only use this attribute when `combineSortingAndFiltering=false`. If set to "false", this attribute will instruct the ListSorterImpl to insert (sort) new entries rather than adding them to the end of the entry list which is the existing behavior. Setting this property to false might imply performance degradation on editor and server, so use with care. If `keepSelectionFocused` is set to "true", you do not need to change the setting of `noLocalSortingAfterInsertAndUpdate`. The default is "true".

<ResourceChooser>

Child elements: `<TreeSorter>?`, `<TreeFilter>?`, `<Filter>*`, `<TableDefinition>`

Parent elements: `<Editor>`

Example 5.35. Example for the ResourceChooser element

```
<Editor>
        <ResourceChooser>
                .
                .
        </ResourceChooser>
</Editor>
```

Within this element of the XML file, the dialog for selecting an internal link is configured. The element has no attributes.

<Query>

Child elements: `<TableDefinition>`

Parent elements: `<Editor>`

Example 5.36. Example for the Query element

```
<Editor>
        <Query>
                .
                .
        </Query>
</Editor>
```

Within this element of the XML file, the document view in the query window is configured. The element has no attributes.

<Search>

Child elements: `<TableDefinition>`

Parent element: `<Editor>`

Example 5.37. Example of the Search element

```
<Editor>
  <Search>
     .
     .
  </Search
</Editor>
```

Within this element of the XML file, the result table of the full-text search is configured. The element has no attributes.

<TreeSorter>

Child elements: `<Comparator>`

Parent elements: `<Explorer>, <ResourceChooser>`

Example 5.38. Example for the Treesorter element

```
<Explorer name="configurable-explorer-factory">
   <TreeSorter>
    .
   </TreeSorter>
    .
</Explorer>
```

Within this element is defined, how the folder view is sorted. The element has no attributes. If an `<Explorer>` element contains no `<TreeSorter>` element, the folders are sorted in alphabetic order. In order to activate an own comparator class, a `<TreeSorter>` element must occur.

<TreeFilter>

Child elements: `<Predicate>`

Parent Elements: `<Explorer>`, `<ResourceChooser>`

Example 5.39. Example for the TreeFilter element

```
<Explorer name="configurable-explorer-factory">
   <TreeFilter>
    .
   </TreeFilter>
    .
</Explorer>
```

Within this element of the XML file, the configuration of the folder view is defined. The element has no attributes. If an `<Explorer>` element contains no `<TreeFilter>` element, the folders for which the user has no read rights are left out. In order to activate a filter class, a `<TreeFilter>` element with the appropriate `<Predicate>` element must occur.

<Filter>

Child elements: `<Predicate>`

Parent elements: `<Explorer>, <ResourceChooser>`

Example 5.40. Example for the Filter element

```
<Explorer name="configurable-explorer-factory">
        <Filter name="deleted-filter">
                .
                .
        </Filter>
</Explorer>
```

Using this element of the XML file, the filters in the explorer window are configured. You will find predefined filter predicates in Section 5.1.3, “Predicate Classes” [97].

Table 5.61. Attributes of element <Filter>

Attribute	Description
name	Use this attribute to enter the name of the filter. Using this name, the entry for the menu item **View|Filters** is looked up in the Bundle.
remote	Use this attribute to enter whether filtering occurs on the server ("true") or on the client. Filtering on the client must be executed with your own filters. Default is "true". If you use filtering on client side, you have to set the attribute `combineSortingAndFiltering` of the `<Explorer>` element to false. Remote filtering and sorting on the server is much faster than local operations which may slow down the server dramatically. So use local filtering with care.

<Predicate>

See the description in "Configuring document types" in this chapter. In contrast to this description, the sub element `<DocumentType>` can not be used here.

<Comparator>

See the description in Section 5.1.8, “Comparator Classes” [109].

<TableDefinition>

Child elements: `<ColumnDefinition>*`

Parent elements: `<Explorer>`, `<Query>`, `<ResourceChooser>`, `<Workflow>`

Example 5.41. Example for the TableDefinition element

```
<Explorer name="configurable-explorer-factory">
        <TableDefinition>
                .
                .

        </TableDefinition>
</Explorer>
```

Within this element of the XML file, the columns of the document table view are configured.

Table 5.62. Attribute of element <TableDefinition>

Attribute	Description
`rowHeight`	This attribute determines the height of a row in the table. The height is given in pixels.

<ColumnDefinition>

Child elements: `<DisplayMap>*`,`<Comparator>?`,`<Renderer>?`, `<NamedDocumentVersionComparator>?`

Parent elements: `<TableDefinition>`

Example 5.42. Example for the ColumnDefinition element

```
<TableDefinition>
        <ColumnDefinition class="StringColumn" weigth="1.0">
                .
                .

        </ColumnDefinition>
</TableDefinition>
```

Using this element, a column in the document table view is defined.

Table 5.63. Attributes of element <ColumnDefinition>

Attribute	Description
`name`	Name of the column which is shown in the header of the column.

Attribute	Description
`class`	This attribute is used for selecting a class for displaying the column (for example IntColumn, StringColumn see Section 5.1.4, “Column Classes” [101] for details). This determines the field type which can be displayed. Furthermore, the class sorts the contents of the column.
`width`	This attribute is used for defining the minimum width of the column in pixels. If the window width is smaller than the total sum of all column widths, a scroll bar appears. Scaling for a larger window is controlled with the attributes `weight` and `resizable`. The default value is 100 pixels.
`weight`	This attribute gives the relative weight of a column in the scaling. Rational numbers are entered. The default value for all columns is "1.0".
`resizable`	This attribute is used for defining whether a column is resized at all. The default setting is "true", that is, the column is enlarged. Resizing can be switched off with "false".
`searchField`	This attribute can only be used for column definitions in the `<Search>` element. Set it to the name of the Search Engine's index field that should be used for sorting. The field must be sortable in the *Search Engine*, and the *Content Feeder* must set its value accordingly. You can either use predefined index fields or define custom ones in the index profile of the *Search Engine*. For the latter case, see the *CoreMedia Search Manual* how to set custom fields with the *Content Feeder*.

<NamedDocumentVersionComparator>

Child elements: `<Comparator>`

Parent elements: `<ColumnDefinition>`

Example 5.43. Example for the NamedDocumentVersionComparator

```
<Explorer name="configurable-explorer"
          combineSortingAndFiltering="false">
  <Filter name="undeleted-filter">
    <Predicate class="UndeletedPredicate"/>
  </Filter>

  <TableDefinition>
    <ColumnDefinition class="StringColumn"
        name="documentname">
      <NamedDocumentVersionComparator remote="false">
        <Comparator class=
"hox.corem.editor.toolkit.clientoperation.comparator.
NameComparator"/>
      </NamedDocumentVersionComparator>
      <DisplayMap document="*" property="name_"/>
    </ColumnDefinition>
  </TableDefinition>
</Explorer>
```

This element of the XML file is used to define custom comparators. It defines if a comparator should be used on client or server side. If you combine a client side

comparator with a server side filter, the attribute `combineSortingAndFiltering` of the `Explorer` element must be "false".

Be aware that combined client and server filtering and sorting may slow down the server dramatically.

Table 5.64. Attributes of element <NamedDocumentVersionComparator>

Attribute	Description
remote	Defines whether the comparator should be used on client (false) or server side (true). Default is true.
name	Name of the comparator

<Renderer>

Child elements: `%varies;`

Parent elements: `<ColumnDefinition>`

Example 5.44. Example for the Renderer element

```
<ColumnDefinition class="ImageColumn">
  <Renderer class="ImageLayoutColumnRenderer" width="50"
   height="50"/>
  <DisplayMap document="Picture" property="thumbnail"/>
</ColumnDefinition>
```

This element is used to define a renderer class, which will be used instead of the predefined renderer of the column class (see example). You will find predefined renderer classes in Section 5.1.5, “Renderer Classes” [104].

Table 5.65. Attribute of the <Renderer> element

Attribute	Description
class	The renderer class which should be used to show the content of the column. For own renderer classes, the abstract class hox.corem.editor.toolkit.table.columnrenderer.LayoutColumnRenderer must be extended.

<DisplayMap>

Child elements:

Parent elements:`<ColumnDefinition>`

Example 5.45. Example for the DisplayMap element

```
<ColumnDefinition name="StringColumn" weight="1.0">
        <DisplayMap document="Bild" property="Name"/>
        <DisplayMap document="*" property="name_"/>
                .
                .
                .
```

```
</ColumnDefinition>
```

This element of the XML file is used for entering which field from which document type is displayed in a column of the document view.

Table 5.66. Attributes of the <DisplayMap> element

Attribute	Description
document	Document type from which the field is taken. Content_ can be used as a wildcard for all document types.
property	Property of the document type which should be displayed. Two types of properties exist for a document. The properties which are defined in the document types file and the predefined properties in all documents like `id_` or `name_`. See the Developer Manual for details.

5.2.6 Configuring the Spell checker

Using these elements, the configuration of the spell checker can be done.

<Spellchecker>

Child elements: `<MainDictionary>`, `<CustomDictionary>`

Parent elements: `<Editor>`

Example 5.46. Example of a Spellchecker element

```
<Editor>
  [...]
  <SpellChecker enabled="false" />

  <SpellChecker enabled="true" os="Windows">
    <MainDictionary class=
       "com.coremedia.spellchecker.Bridge2JavaWordDictionary"/>

    <CustomDictionary class=
       "hox.corem.editor.spellchecker.Dictionary"/>
  </SpellChecker>

  [...]
</Editor>
```

Use this element to enable or disable the spell checker. It's also the container element for the configuration of the spell checker.

Table 5.67. Attribute of the element SpellChecker

Attribute	Description
`enabled`	This attribute determines whether the spell checker should be used ("true") or should be disabled ("false"). By default, "false" is used.

Attribute	Description
os	Restricts the configured spell checker to a given operating system. The value is compared to Java's system property os.name. Common value is for example Windows. The configuration with the best (that is, longest) match wins. So for example if you have a spell checker configured with os="Windows" and another with os="Windows 7" and you are running on Windows 7 the second one will be taken. Default is to match all operating systems. So the example above says: Disable the spell checker on all operating systems but on Windows.

<MainDictionary>

Child elements:

Parent elements: <SpellChecker>

Example 5.47. Example of a MainDictionary element

```
<Editor>
        .
        <SpellChecker enabled="true">
              <MainDictionary
  class="com.coremedia.spellchecker.Bridge2JavaWordDictionary"/>
              .
        </SpellChecker>
        .
</Editor>
```

This element is used to determine the class used for integrating an external dictionary in the spell checker. As a default, the Microsoft Word dictionary will be used. Please keep in mind, that it is not possible to write from *CoreMedia CMS* to the Word *user* dictionary. If the Word user dictionary should be used for spell checking, you need to activate this option in Word itself (please refer to the Word documentation).

Table 5.68. Attributes of the MainDictionary element

Attributes	Description
class	With this attribute, the class used for determining the dictionary used in the spell checker is configured. By default, the class com.coremedia.spell checker.Bridge2JavaWordDictionary will be used. This class integrates the dictionary of a Microsoft Word installation. It takes the file properties/corem/language-mapping.properties to map the Java locale (de_DE, for example) to the Word locale (wdGerman, for example). The spell checker checks whether a language exists in Word or not. If the language does not exist, an error message will be shown and all properties using the language will not be checked (see the element PropertyLanguageResolverFactory). If you are sure that the language is installed with Word you might check the lan

Attributes	Description
	`guage-mapping.properties` file for the correct mapping. Maybe you need to add the appropriate mapping. For own classes, the interface com.coremedia.spellchecker.Dictionary must be implemented.

<CustomDictionary>

Child elements:

Parent elements: `<SpellChecker>`

Example 5.48. Example of a CustomDictionary element

```
<Editor>
        .
        <SpellChecker enabled="true">
                <CustomDictionary
                class="hox.corem.editor.spellchecker.Dictionary"/>
                .
        </SpellChecker>
        .
</Editor>
```

This element is used to determine the class used for integrating a customized dictionary in the spell checker. The entries of this dictionary will be used for spell checking and for suggestions.

Table 5.69. Attribute of the <CustomDictionary> element

Attributes	Description
class	With this attribute the class used for determining the customized dictionary used in the spell checker is configured. By default, the class `hox.corem.editor.spellchecker.Dictionary` will be used. This class uses two customized dictionary created in the folder tree of the CoreMedia System. For usage of these dictionaries see the User Manual. For own classes, the interface `com.coremedia.spellchecker.CustomDictionary` must be implemented.

<PropertyLanguageResolverFactory>

Child elements:

Parent elements: `<Editor>`

```
<Editor>
        .
        <PropertyLanguageResolverFactory
        class=
      "hox.corem.editor.DefaultPropertyLanguageResolverFactory"/>
        .
</Editor>
```

Example 5.49. Example of a PropertyLanguageResolver Factory element

This element is used to configure a class which determines the language used in a property. This information is used for the spell checker. For the default class hox.corem.editor.DefaultPropertyLanguageResolverFactory the additional attributes `language` and `country` can be used to override the settings of the element `Locale`.

Table 5.70. Attribute of the PropertyLanguageResolverFactory element

Attribute	Description
class	With this attribute the class used for determining the language of a property is configured. By default, the class hox.corem.editor.DefaultPropertyLanguageResolverFactory will be used. This class sets the language of all properties used for the spell checker to the value defined via the element `<Locale>`. You can override this setting with the attributes `language` and `country` of the default language resolver factory. For own classes, the interface hox.corem.editor.PropertyLanguageResolverFactory must be implemented.

5.2.7 Configuring the Workflow

Using these elements, the appearance of the worklist in the workflow window can be configured.

<Workflow>

Child elements: `<TableDefinition>`

Parent elements: `<Editor>`

```
<Editor>
        <Workflow>
                <TableDefinition>
                        <ColumnDefinition
  class="hox.corem.editor.workflow.columns.WorklistDetailColumn"/>
                        .
                        .
                </TableDefinition>
        </Workflow>
</Editor>
```

Example 5.50. Example of the Workflow element

This element is used to define, which information should be shown in the columns of the workflow list at the left side of the workflow window.

<TableDefinition>

See the description in "Configuring windows with table views".

<ColumnDefinition>

See the description in "Configuring windows with table views" and in Section 5.1.4, “Column Classes” [101].

<Processes>

Child elements: `<Process>*`, `<Predicate>?`, `<Comparator>?`

Parent elements: `<Editor>`

Example 5.51. Example for the Processes element

```
<Editor>
        <Processes>
                <Process name="Publication"/>
                        .
                        .
                </Process>
                .
                .
        </Processes>
</Editor>
```

This element is used to group the elements which define the view of the workflow variables.

<Comparator>

See the description in the section "Configuring document types".

<Predicate>

See the description in the section "Configuring document types".

<Process>

Child elements: `<View>?`, `<Task>*`, `<WorkflowStartup>?`

Parent elements: `<Processes>`

Example 5.52. Example for the Process element

```
<Processes>
                <Process name="Publication"/>
                        <View>
                                .
                                .
                        </View>
                        .
                </Process>
```

```
                .
                .
</Processes>
```

This element is used for the configuration of the variable editors for the process variable view (in the `<View>` element) and for each task (in the `<Task>` element).

Table 5.71. Attribute of element <Process>

Attribute	Description
`name`	Name of the process, for which the view should be configured.
`openWork-flowWindow`	Flag to control whether the workflow window should be opened on process creation. By default, "true" is used.

<View>

Child elements: `<Variable>*`, `<AggregationVariable>*`

Parent elements: `<Process>`

Example 5.53. Example for the Code element

```
<Process name="FourEyesProcess">
   .
   .
   <View>
           <Variable name="User" editorClass="UserChooserEditor"/>
           .
           .
           .
   </View>
</Process>
```

This element is used to group the elements which define the look of the workflow variables of a process. That is, with which editor the variables defined in the workflow XML file should be shown in the workflow window.

<Task>

Child elements: `<Variable>*`, `<AggregationVariable>*`

Parent elements: `<Process>`

Example 5.54. Example for the Task element

```
<Process name="FourEyesProcess">
        .
        .
        <Task name="approve">
              <Variable name="comment" editorClass="StringEditor"/>

                 .

                 .
        </Task>
</Process>
```

This element is used to group the elements which define the look of the variables of a task. That is, with which editor the variables defined in the workflow XML file for the task should be shown in the workflow window.

Table 5.72. Attribute of the <Task> element

Attribute	Description
name	Name of the task for which the definitions should be valid.

<WorkflowStartup>

Child elements:

Parent elements: `<Process>`

Example 5.55. Example for the WorkflowStartup element

```
<Process name="MyWorkflow">
        <WorkflowStartup class="MyWorkflowStarter"/>
        .
        .
</Process>
```

This element is used to define, how the workflow should be started. That is, how the variables should be filled.

Table 5.73. Attribute of <WorkflowStartup> element

Attribute	Description
class	This attribute defines the class which manages the startup of the workflow. Own classes must implement the interface hox.corem.editor.workflow.WorkflowStartup. As a default, the class hox.corem.editor.generic.GenericWorkflowStartup is used. It sets the resource variable with the selected resource(s) and opens a window for setting the other workflow variables.

<Variable>

Child elements:

Parent elements: `<View>`, `<Task>`

Example 5.56. Example for the Variable element

```
<Task name="approve">
       <Variable name="comment" editorClass="StringEditor"/>
       .
       .
       .
 </Task>
```

This element is used to define with which editor a workflow variable should be shown in the workflow window.

Table 5.74. Attributes of element <Variable>

Attributes	Description
`editorClass`	Editor class to be used for showing the variable. See Section 5.1.1, "Property Editors" [77] for a listing of the editor classes.
`name`	Name of the variable to be shown, as defined in the workflow definition.

<AggregationVariable>

Child elements:

Parent elements: `<View>`, `<Task>`

Example 5.57. Example for the AggregationVariable element

```
<Task name="approve">
        <AggregationVariable name="Resources"
        editorClass="ResourceChooserEditor"/>
        .

        .
</Task>
```

This element is used to define with which editor a workflow aggregation variable should be shown in the workflow window.

Table 5.75. Attributes of <AggregationVariable> element

Attributes	Description
`editorClass`	Editor class to be used for showing the variable. See Section "Workflow Editors" [77] for a listing of the editor classes.
`name`	Name of the aggregation variable to be shown, as defined in the workflow definition.

5.2.8 Configuring Web Extensions

A web extension is an addition to the *Site Manager*. A web extension is part of a standard web application in a servlet container. It displays one or more web pages and might return a result back to the invoking editor by opening a URL to the editor's remote control with the result of the web extension call as an URL parameter.

CoreMedia CMS contains the following web extension:

- Preview - Shows a preview of the selected document with different web browsers and different render URLs

‹WebContext›

Child elements: `<WebExtensions>`

Parent element: `<Editor>`

```
<Editor>
  ...
  <WebContext host="previewHost" port="40081" context="coremedia">
    ...
  </WebContext>
  ...
</Editor>
```

The ‹*WebContext*› element configures a web application in a Servlet container. Therefore, this element defines connection parameters. It is a container for ‹*WebExtension*› elements.

Table 5.76. Attributes of the WebContext element

Attribute	Description
`class`	Java class with no-arg constructor. Defaults to the value `hox.corem.editor.web.SingleSignOnWebContext`. Other classes must be sub classes of the class `hox.corem.editor.web.WebContext`.
`host`	The name of the hosting computer where the web application runs.
`port`	The port of the web application where the web extension runs.
`context`	The name of the web application where the web extension runs.
`protocol`	The protocol part of the URL. Defaults to 'http'.
`loginPort`	The port for the SSL connection which is used for single sign-on. This port is optional, if the web extension does not require authentication. Currently Preview does not require authentication.
`loginPath`	The path for the SSL connection which is used for single sign-on. Defaults to `/servlet/login`. Not needed if the web extension does not require authentication.

‹WebExtension›

Child elements: `<Pattern>`

Parent elements: `<WebContext>`

```
  <WebContext host="localhost" port="40081" context="coremedia">
    <WebExtension name="preview" path="/servlet/preview"/>
  </WebContext>
```

The `<WebExtension>` element configures a part of a web application which is used to add functionality to the Site Managers.

Table 5.77. Attributes of the <WebExtension> element

Attribute	Description
class	Java class with no-arg constructor. If not set, the class depends on the attribute `name`. If the name attribute is set to "preview", the class `hox.corem.editor.web.Preview` is used. Other names will produce an error (if class is not set).
name	The name of the web extension. Existing names are: → preview - the preview of documents or folders
path	The path to the web application without context and URL query part.
open	Use this attribute only with the preview web extension. It determines which browser(s) are opened, when you call the preview from the toolbar. If open is set to "all", then all defined browsers are opened. If you omit the attribute or set it to "last" (default), then the last selected browser is opened. If you have selected no browser before, the first configured browser is opened. Using the **Preview** menu item from the **File** menu, you can always choose the browser to use from all configured browsers. `<WebExtension name="preview" open="all">`

<Pattern>

Child elements:

Parent elements: `<WebExtension>`

```
<WebExtension name="preview">
    <Pattern id="Default Preview" browser="Internet Explorer"/>
    <Pattern id="Document-DE" browser="Internet Explorer"
      pattern="%p://%h:%n/de/%i.html"/>
    <Pattern id="Document-EN" browser="Internet Explorer"
      pattern="%p://%h:%n/en/%i.html"/>
</WebExtension>
```

The `<Pattern>`– element is optional and configures two things:

→ one or more browsers

→ one or more URL patterns

If no `<Pattern>` element is configured, the web extension uses one or more default browsers with the default pattern string. Preview uses all browsers that match the user's locale when all browsers should open.

Attribute	Description
id	The unique name of the pattern. Defaults to the `browser` attribute.
browser	The id of the browser to invoke. The value must match an `id` attribute in element `<WebBrowser>`.
pattern	A URL pattern string. Defaults to `%p://%h:%n/%f`. Every web extension may use different patterns with different placeholders. The place holders have the following meaning: **Common to all web extensions:** → p: protocol → h: host name → n: port number → f: rest of the URL → u: path including context name but without URL query **Preview:** → i: document id → v: document version → f expands to 'id=%i' if version = 0 otherwise to 'id=%i & version=v' → I: expands to the URL-encoded string ID of the document

Table 5.78. Attributes of the <Pattern> element

5.2.9 Example Configuration of the Document Overview

Using the file `editor.xml`, you can configure the document overview, that is, the tabular listing of documents in four windows:

- The explorer view in the Explorer window.
- The query view in the Query window.
- The document overview in the selection window for internal links to resources.
- The workflows in the workflow list.

Configuration is the same for all four windows. However, there is the additional possibility in the main window of filtering the documents in the document window.

The example configuration described here for the query window can simply be used for the other windows. Only the enclosing element has to be changed (`Query`, `Explorer`, `ResourceChooser`, `Search` and `Workflow` are possible).

Creating a query view in the query window

You want to create a query view in the query window which displays the following information:

- the document type
- the document name
- the content of a structured text field

These column definitions are created within the `<Query>` and `<TableDefinition>` tags. Note that the document types and fields given in the following examples do not necessarily exist in your CoreMedia installation.

Example 5.58. The query tags

```
<Query>
        <TableDefinition>
                .
                .
                .
        </TableDefinition>
</Query>
```

In the following examples, the attribute name is used in the <ColumnDefinition> element.

- **Definition of the table column for the document type**

  The column should have the name "Type". The minimum width should be 50 pixels and should scale with the width of the window. Enter the following lines in `editor.xml`:

Example 5.59. Example configuration for document type display

```
<ColumnDefinition name="Type" width="50"
                      class="DocumentTypeColumn"/>
```

  In this case it is necessary only to make a few entries, since most of the default values can be used. For example, the default value for `resizable` is "true", that is, the column is automatically scaled. It is also not necessary to make any `DisplayMap` entries, since the document type is an inherent property of all documents.

- **Definition of the table column for the document name**

  The column should have the name "Name". The minimum width should be 150 pixels and should scale with the width of the window. The document

name should be displayed in this column for each document. Enter the following lines in `editor.xml`:

```
<ColumnDefinition name="Name" width="150" class="StringColumn">
        <DisplayMap document="*" property="name_"/>
</ColumnDefinition>
```

Example 5.60. Example configuration for document name display

Mostly default values are also used in this case. However, because there is no special class for displaying the document name (the StringColumn class must be used), you must make `DisplayMap` entries. Since the name is defined in all document types, you can enter `document="*"`. The document name is accessed via `property="name_"`.

→ **Definition of the table column for the content of a structured text field**

The column should have the name "Content". The minimum width should be 250 pixels and should scale if the width of the window is increased. For a document of type "Dish", the content of the *description* field should be displayed; Enter the following lines in `editor.xml`:

```
<ColumnDefinition name="Content" width="250"
        class="SgmlTextColumn">
        <DisplayMap document="Dish" property="Description"/>
</ColumnDefinition>
```

Example 5.61. Example configuration for the structured text column

The SgmlTextColumn class is used here for displaying the structured text field (SGML field). One `DisplayMap` tags is created for the document type `Dish`. The next figure shows the configured query window.

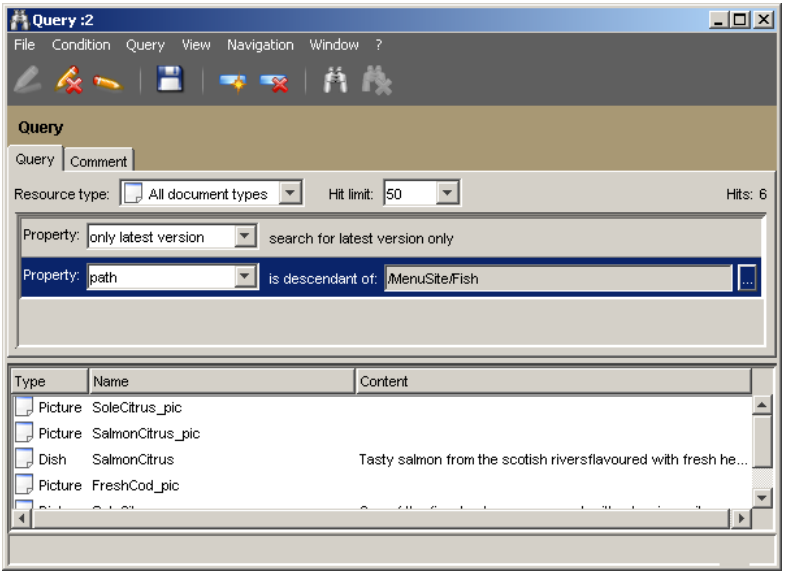


Figure 5.3. Configured Query window

Creating filters

You want to make filters for the following documents available via the menu point **View|Filter**:

- Published documents
- Deleted documents

Enter the following lines in the `editor.xml` file:

Example 5.62. Creation of two filters

```
<Explorer name="configurable-explorer">
        <Filter name="deleted-filter">
                <Predicate class="DeletedPredicate" />
        </Filter>
        <Filter name="unpublished-filter">
                <Predicate class="UnpublishedPredicate" />
        </Filter>
        .
        .
        .
</Explorer>
```

5.2.10 Example Configuration of the Document Window

With the help of the `editor.xml` file you can define the fields of a document type which should be displayed in the document window.

You can configure the following settings:

- Which fields are displayed?
- Which fields can be edited?
- Which fields must be filled?
- How is the content of a field displayed?
- The initial values of the fields of a newly created document.
- How the content of a field is validated at check in.

The configuration possibilities are shown using a document of type `Dish`. The following figure shows such a document without specially configured fields (a `Dish` document type could of course have completely different fields in your system, or not occur at all). The document has no fields of type `blob` and `integer`. However, the configuration of these fields can be carried out analogously to the configuration shown. The editors for these two field types can be found in Section 5.1.1, “Property Editors” [77].

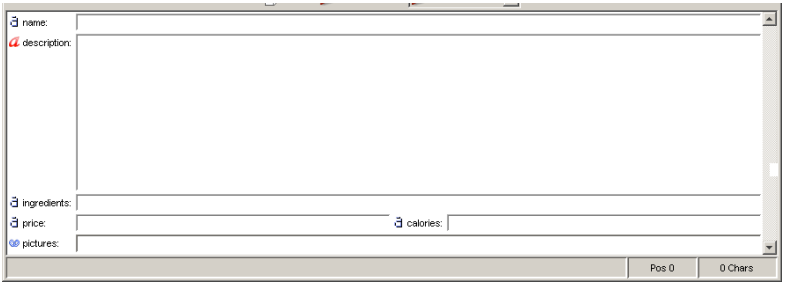


Figure 5.4. Dish document without special configuration

The fields of the document type `Dish` should now be configured as follows:

- Pictures

  A selection list should be presented here.
- Description

  This field must be filled, since it is necessary for a dish.
- price

  Three different prices should be made available for selection here.
- name

  This field should be filled with the initial value "New Dish".
- calories

  This field should only be displayed and filled with the initial value "200".

Once the configuration has been carried out, a document of type `Dish` appears as follows:

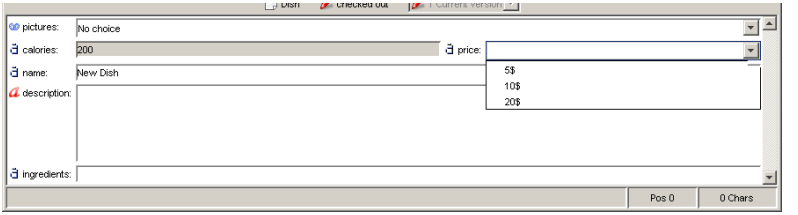


Figure 5.5. Dish document after the configuration

One of the two list fields is opened up here as example. If you try to save this document without entries in the fields *description*, an error message appears. An entry in the field *calories* is not possible. The field *name* has the default entry "New Dish". You achieve this configuration with the following settings in `editor.xml`:

Example 5.63. Code example for configuration of the editor

```
 <Documents>
.
.
 <Document type="Dish">
  <Property name="pictures" editorClass="ComboBoxLinkListEditor"
  path="/MenuSite/Fish"/>
  <Property name="calories" editable="false"/>
  <Property name="price" editorClass="ComboBoxStringEditor">
   <HistoryItem value="5$"/>
   <HistoryItem value="10$"/>
   <HistoryItem value="20$"/>
  </Property>
 </Document>
</Documents>

<DocumentTypes>
 <DocumentType name="Dish">
  <PropertyType name="description">
   <Validator class="NotEmpty2"/>
  </PropertyType>
  <PropertyType name="name" initialValue="New Dish"/>
  <PropertyType name="calories" initialValue="200"/>
 </DocumentType>
</DocumentTypes>
```

5.3 Configuration Possibilities in editor.properties

The file **editor.properties** contains settings for the following points:

- user login
- logging of the editor
- the URL of the XML configuration file of the editor
- the URL of the CSS file used for the richtext pane
- embedded document view
- the number of results from a user or group query in the user manager window

Table 5.79. editor.properties

Property	Value	Default	Description
`login.username`	`example`		With this parameter, the default setting of the user name when login in can be determined. If no value is entered, the login name of the user in the system is used.
`login.password`	`example`		With this parameter, the default setting of the password when logging in can be determined. If no value is entered, the user must enter the password.
`login.domain`	`example`		With this parameter, the default setting of the domain for login can be determined. If no value is entered, the user must enter the domain. If no domain is entered, the user must be an internal user.
`login.immediate`	`true, false`		If this parameter is set to "true", the editor tries to login the user with the "user name" and "password" configured above, without the login dialog box appearing.
`editor.startup.configuration`	`<Path relative to $COREM_HOME and name or URL of the XML startup.configuration file>`		The file defined in this property will be evaluated before login. Thus, if a user-defined editor class should be used, you must configure this class in the appropriate XML element of the file defined in `editor.startup.configuration`. If you want to change the language of the login dialog, the locale must be set in this file, too.

Property	Value	Default	Description
editor.configuration	<Path relative to <CoreMediaHome> and name or URL of the XML configuration file>		The main configuration file of the Site Manager, which will be loaded after login. As a default, the file properties/corem/editor.xml will be used.
group.configuration	<Path relative to <CoreMediaHome> and name or URL of the XML configuration file with wildcard {0}>		With this property, a group dependent editor.xml file can be configured. To do so, the wildcard {0} in the URL or path of the configuration file will be replaced by the group name of the user or by names defined in the <ConfigGroups> element of the editor-startup.xml file. If the user belongs to multiple groups, {0} will be replaced by one of these group names in an arbitrary order.
user.configuration	<Path relative to <CoreMediaHome> and name or URL of the XML configuration file with wildcard {0}>		With this property a user dependent editor.xml file can be configured. To do so, the wildcard {0} in the URL or path of the configuration file will be replaced by the name of the user.
editor.richtext.css.location	<Path relative to <CoreMediaHome> and name or URL of the CSS file>		With this property the CSS file for the look and feel of the richtext pane is defined.
editor.display.embedded.view	true, false		With this property, enable the embedded document view which shows the selected document in the overview window of the *CoreMedia Editor*. The default setting is "false".
usermanager.sear	integer value		You can limit the number of users or groups obtained by a query in the user manager

Property	Value	Default	Description
chResultSize			window. You should set it small, as the user manager GUI is not designed to be used as an LDAP browser. The value must be smaller than the user and group cache sizes of your CAP server set in the capserver.properties file. The default value is 490. The size may also be limited by your particular LDAP server (for example Active Directory: 1500).
editor.blob.mimetype.strict	true, false	false	If set to "true", a blob can only be loaded into a blob property when the MIME type of the blob meets the MIME type of the property. If set to "false", the user can decide to load a blob with the required MIME type even if the actual MIME type of the blob is different.
editor.query.allowUsers	true, false	true	If set to "false", all user related queries in the Query window (such as "Approved by") are restricted to the logged in user. If set to "true", the common user chooser button appears and a specific user can be selected. This property does not affect administration users, they can always choose a specific user.
show.content.change.popups	true, false	true	Show pop-up notifications for content changes ("checkin", "save" and "uncheckout") in opened document, that are performed in a different editor.

5.4 Configuration Possibilities in proxy.properties

In the file `proxy.properties` the resource cache for accelerating the *Site Manager* is configured. A large resource cache reduces the network load, but at the same time increases the memory required by the *Site Manager* on the local computer.

Table 5.80. proxy.properties

Property	Value	Default	Description
`proxy.cache.capacity`	<int>		With this parameter you can set the number of cached resources, that is documents and folders (default value: 2000). The default value of 2000 resources is set according to the maximum memory allocation of the Java environment of 128 Mbytes (see `bin/editor.jpif`). Furthermore, this value is based on the assumption that a document contains a normal sized image (JPEG, GIF, etc.) and these images are displayed in the overview as thumbnails. If the documents contain few images but their number is many thousands, the value for the cache can be, for example, doubled, while maintaining the memory size of the JVM. If large multimedia data is involved, the upper limit of the memory allocation for the Java environment must be increased, or the cache memory size reduced according to the larger size of the data.
`proxy.gc.limit`	<int>		Using this parameter, you can enter (in bytes) the size of the remaining free memory of the JVM before *Site Manager* objects start to be deleted, in order to avoid a memory deficiency (default value: 1,000,000). An increase of this value to ca. 3 ~ 5 million can lead to the resource cache being almost completely emptied very often, to maintain free memory for the program execution

Property	Value	Default	Description
			of the *Site Manager*. In this way, memory deficiency can be avoided, but the network load usually increases, since the resource cache must be refilled, leading to possible losses in speed when working with the *Site Manager*.
`proxy.status.interval`	<int>		Using this parameter, you can determine the interval (in seconds) at which the required memory is analyzed and, possibly, reduced, and messages about the resource cache and the memory use of the *Site Manager* are output. A value of 0 deactivates this function.

5.5 Configuration of The Site Manager in capclient.properties

The file `capclient.properties` contains information about the location of the IOR of the *CoreMedia Server*, about the home and system folder and global information on the timezone and on special elements in XML text Properties.

Table 5.81. capclient.properties

Property	Value	Default	Description
`cap.client.server.ior.url`	URL format `ht tp://<host>:<port>/coremedia/ior`		This property describes for the *Site Manager* the location where it gets the IOR of the *CoreMedia Content Server*. Since the editor is usually installed on a different computer from the *CoreMedia Content Server*, the property is given as HTTP URL. The editor then receives the IOR from the server with an HTTP query. In general, work is only carried out on the *Content Management Server*. In exceptional cases the administrator can access the *Master Live Server*. On the *MasterLive Server,* no resources should be changed with the editor, so that consistency of the *MasterLive Server* with the *Content Management Server* is ensured.

5.6 Configuration Possibilities in workflowclient.properties

The file `workflowclient.properties` defines configuration options for logging (not *Site Manager*!), user management for the workflow client, remote action handlers and the parameters necessary to connect to the *CoreMedia Workflow Server*. For more information about logging see the [Operations Basics Manual].

Table 5.82. Parameters of the workflowclient.properties file

Property	Value	Default	Description
`workflow.client.server.ior.url`	http://host-name:port/workflow/ior		Defines the URL where to get the IOR of the workflow server. The host name and port in the URL must match with the host name and port in the `workflowserver.properties` file.

`workflow.user`
 	
`workflow.password`
 	
`workflow.domain`
	`<workflow-user-name>`
	
	`<WorkflowUserPassword>`
	
	`<WorkflowUserDomain>`
			As a default, a workflow client (but not the *Site Manager*) connects to the *Content Server* as the workflow user. Here you can define a different user to connect with the server. Replace <*WorkflowUserName*>, <*WorkflowUserPassword*> and <*WorkflowUserDomain*> with your user name, domain and/or password.

5.7 Configuration Possibilities in language-mapping.properties

This file is necessary for the spell checker. It defines the mappings from Java Locale objects to Word language identifiers. For example,

```
de_DE=wdGerman
```

maps the locale `de_DE` to the Word language identifier `wdGerman`. The most common European languages are covered in the default `language-mapping.properties` file. The locale identifier can be taken from the `Locale.toString()` method (see Javadoc for details) and the Word language identifiers from the `WdLanguageID` enumeration of the Word object type library.

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, predefined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes 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

A

access resource, 51
API, 107
    Editor API, 77
attribute
    of Filter, 142
    of Process, 151
attributes
    of AggregationVariable, 153
    of Browser, 120
    of Bundle, 118
    of ColumnDefinition, 143
    of Comparator, 134
    of CustomDictionary, 148
    of DisplayMap, 146
    of Document, 137
    of DocumentTypes, 130
    of Explorer, 139
    of Initializer, 132
    of Locale, 117
    of MainDictionary, 147
    of Preview, 118
    of Property, 138
    of PropertyLanguageResolverFactory, 149
    of PropertyModelFactory, 124
    of RemoteControl, 121
    of Renderer, 145
    of ResourceChooser, 140
    of SpellChecker, 146
    of Tab, 138
    of TableDefinition, 143
    of Task, 152
    of TreeFilter, 142
    of Treesorter, 141
    of Validator, 132
    of Variable, 153
    of WorkflowStartup, 152

B

BeanParser, 15, 113

C

classes
    LinkListEditor, 92
comparators
    client side, 109
CoreMedia Editor , 17, 156
coremedia-editor.dtd (Elements)
    AggregationVariable, 153
    Browser, 119
    Bundle, 117
    ColumnDefinition, 143
    Comparator, 133
    CustomDictionary, 148
    DisplayMap, 145
    Document, 136
    Documents, 136
    DocumentType, 130
    DocumentTypes, 129
    Editor, 113
    Explorer, 139
    Filter, 142
    Initializer, 132
    Locale, 116
    MainDictionary, 147
    Predicate, 135
    Preview, 118
    Process, 150
    Processes, 150
    Property, 137
    PropertyLanguageResolverFactory, 148
    PropertyType, 130
    Query, 141
    regular expressions to use, 131
    Renderer, 145
    ResourceChooser, 140
    SpellChecker, 146
    Tab, 138
    TableDefinition, 143
    Task, 151
    TreeFilter, 141

TreeSorter, 141
Validator, 132
validPattern, 131
Variable, 152
View, 151
Workflow, 149
WorkflowStartup, 152

D

document fields (example), 159
DTD
coremedia-editor.dtd: generalconfiguration, 113

E

Editor classes
Column classes, 101
Column classes for workflows, 102
TabbedDocumentView, 96
Editor configuration
Predicate, 142
editor.xml, 112
execute, 64

G

GenericDocumentView, 68
getInitialValue, 52

H

HTML
copy and paste, 88

I

include, 60
Initializer class genericInitializer, 106

L

language-mapping.properties, 169
LanguageResolver, 56
localization, 73

N

NamedDocumentVersionComparator, 144
no start, 47

P

predefined Editor classes, 77
Property editors
blob fields, 91
date fields, 82
integer fields, 80
string fields, 77

S

SimpleValidationException, 55
Site Manager
Remote Control, 120
spellchecker, 45
style sheet group, 83

T

Tool bar , 36

U

unknow element: ROOT, 48

V

validate, 54

W

workflows, 77