

CoreMedia Digital Experience Platform 8
//Version 7.5.45-10



CoreMedia Content Application Developer Manual

COREMEDIA



CoreMedia Content Application 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. Introduction	1
1.1. Audience	2
1.2. Typographic Conventions	3
1.3. CoreMedia Services	5
1.3.1. Registration	5
1.3.2. CoreMedia Releases	5
1.3.3. Documentation	6
1.3.4. CoreMedia Training	8
1.3.5. CoreMedia Support	9
1.4. Change Chapter	12
2. Overview	13
2.1. Components and Use Cases	14
2.2. Architecture	15
2.3. Caching	16
2.3.1. Unified API Cache	16
2.3.2. Data View Cache	16
2.3.3. CacheKey Cache	16
2.4. The Spring Framework	18
3. Administration and Operation	19
3.1. Connecting and Caching	20
4. Development	22
4.1. Content Beans - Mapping content to objects	23
4.1.1. Generate Content Beans from the Content Type model	23
4.2. Data Views	31
4.2.1. Defining Data Views	32
4.2.2. Data View Design	33
4.2.3. Configuring Cache Sizes	46
4.2.4. Writing Cacheable Beans	47
4.3. The CAE Web Application	50
4.3.1. Handling Requests	50
4.3.2. Building Links	59
4.3.3. Views	65
4.3.4. Writing Templates	75
4.3.5. Adding Document Metadata	87
4.3.6. Working with Forms	93
4.3.7. Integrating with Spring Web Flows	102
4.3.8. Unit Testing a CAE Application	104
4.3.9. Dealing with Errors	107
4.4. Multi-Site and Localization Management	110
4.5. CAE Developer Toolbox	111

4.6. Image Transformation API	115
5. Appendix	121
5.1. Customizer	122
5.2. Aspects	125
5.3. Entity Resolver	128
5.4. Content Placeholders	129
5.5. Configuration Property Reference	132
5.6. Bean Definition Reference	135
Glossary	145
Index	152

List of Figures

4.1. Phases of a data view lifecycle	36
4.2. Example site structure	42
4.3. Entity Model	42
4.4. Dependencies of the Unified API cache	49
4.5. Processing chain of DispatcherServlet, handlers and view dispatcher	50
4.6. Processing chain of handlers and link schemes	59
4.7. View lookup sequence	67
4.8. Cache Statistics	111
4.9. Cache Browser	112

List of Tables

1.1. Typographic conventions	3
1.2. Pictographs	3
1.3. CoreMedia manuals	6
1.4. Log files check list	10
1.5. Changes	12
2.1. Components of the CAE framework	14
3.1. Connection properties	20
3.2. Properties for cache size settings	20
4.1. Parameters of the Beangenerator	24
4.2. Document property to Java property mappings	27
4.3. Association types	36
4.4. Bean Properties in the DataView Example	42
4.5. Implicit macros, functions and variables in FreeMarker templates	84
4.6. Example of image transformation strings	115
5.1. Configuration Properties	132
5.2. META-INF/coremedia/component-cae.xml in artifact cae-component	135
5.3. com/coremedia/cae/view-services.xml in artifact cae-viewservices-impl	136
5.4. com/coremedia/cae/view-error-services.xml in artifact cae-viewservices-impl	136
5.5. com/coremedia/cae/view-development-services.xml in artifact cae-viewservices-impl	137
5.6. com/coremedia/cae/view-freemarker-services.xml in artifact cae-viewservices-impl	138
5.7. com/coremedia/cap/common/uapi-services.xml in artifact cap-unified-api	138
5.8. com/coremedia/cae/uapi-services.xml in artifact cae-util	138
5.9. com/coremedia/cae/dataview-services.xml in artifact cae-contentbeanservices-impl	139
5.10. com/coremedia/cae/contentbean-services.xml in artifact cae-contentbeanservices-impl	139
5.11. com/coremedia/cache/cache-services.xml in artifact core-media-cache	139
5.12. com/coremedia/cae/link-services.xml in artifact cae-linkservices-impl	139

5.13. com/coremedia/id/id-services.xml in artifact coremedia-id	140
5.14. com/coremedia/cae/handler-services.xml in artifact cae-handlerservices-impl	140
5.15. com/coremedia/mimetype/mimetype-service.xml in artifact coremedia-common	141
5.16. com/coremedia/cae/security-services.xml in artifact cae-util	142
5.17. com/coremedia/transform/blob-transformer.xml in artifact coremedia-transform	142
5.18. com/coremedia/cae/controller-services.xml in artifact cae-handlerservices-impl	144

List of Examples

4.1. Auto completion example	38
4.2. Auto completion exclusion example	39
4.3. Bean property with custom dependency	48
4.4. Accessing a bean property with a custom dependency	48
4.5. Triggering an invalidation of a custom dependency	48
4.6. A link scheme	60
4.7. Defining a link scheme	60
4.8. Iterating over java.util.Map entries in FreeMarker templates	80
4.9. Code for Idea auto-completion	82
4.10. A DOM with Metadata and Generated Metadata Tree	87
4.11. Responsive Device Slider Metadata	88
4.12. Studio Specific CSS and JavaScript Metadata	89
4.13. Content With Property	92
4.14. Responsive Device Slider Metadata	92
4.15. Mixed preview and custom metadata in FreeMarker	93
4.16. Mixed preview and custom metadata in JSP	93
4.17. Adding the anti-CSRF header to jQuery Ajax requests	101
4.18. Forcing token creation from a login web flow	102
5.1. Add aspect support to content beans	125
5.2. Registering an aspects provider for content beans	126
5.3. Definition of an aspects provider for arbitrary Java beans	126
5.4. Annotating a Substitution method	130
5.5. Use of cm:substitute in CMAction.jsp	130
5.6. Registering a substitution programmatically	130

1. Introduction

This manual provides information on the administration and development of content applications using the *Content Application Engine (CAE)*.

- In [Chapter 2, Overview \[13\]](#) you will get an overview of the CAE and its concepts.
- In [Chapter 3, Administration and Operation \[19\]](#) you will learn some administrative tasks.
- In [Chapter 4, Development \[22\]](#) you will learn how to use the *Content Application Engine* for your own applications.

1.1 Audience

This manual is intended for developers of CoreMedia projects, people who set up and tune, who integrate and implement *CoreMedia CMS*. You'll find a description of ideas and concepts, building blocks, and detailed examples.

1.2 Typographic Conventions

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

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

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:




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

Table 1.2. Pictographs

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

1.3 CoreMedia Services

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

CoreMedia User Orientation for CoreMedia Developers and Partners

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

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



- [Section 1.3.1, “Registration” \[5\]](#) describes how to register for the usage of the services.
- [Section 1.3.2, “CoreMedia Releases” \[5\]](#) describes where to find the download of the software.
- [Section 1.3.3, “Documentation” \[6\]](#) describes the CoreMedia documentation. This includes an overview of the manuals and the URL where to find the documentation.
- [Section 1.3.4, “CoreMedia Training” \[8\]](#) describes CoreMedia training. This includes the training calendar, the curriculum and certification information.
- [Section 1.3.5, “CoreMedia Support” \[9\]](#) describes the CoreMedia support.

1.3.1 Registration

In order to use CoreMedia services you need to register. Please, start your [initial registration via the CoreMedia website](#). Afterwards, contact the CoreMedia Support (see [Section 1.3.5, “CoreMedia Support” \[9\]](#)) by email to request further access depending on your customer, partner or freelancer status so that you can use the CoreMedia services.

1.3.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

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

<http://releases.coremedia.com/dxp8>

Refer to our [Blueprint Github mirror repository](#) for recommendations to upgrade the workspace either via Git or patch files.



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

Maven artifacts

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

<https://repository.coremedia.com>

You have to add your CoreMedia credentials to your Maven settings file as described in section CoreMedia Digital Experience Platform 8 Developer Manual.

License files

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

1.3.3 Documentation

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

<http://documentation.coremedia.com/dxp8>

The manuals have the following content and use cases:

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 <i>CoreMedia Studio</i> for editorial and administrative work. It also describes the usage of the <i>Adaptive Personalization</i> and <i>Elastic Social</i> GUI that are integrated into <i>Studio</i> .

Table 1.3. CoreMedia manuals

Manual	Audience	Content
LiveContext for IBM WebSphere Manual	Developers, architects, administrators	<p>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 <i>Studio</i> extensions, folder and user rights concept and many more details. It also describes administrative tasks for the features.</p> <p>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.</p>
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 <i>Adaptive Personalization</i> , the CoreMedia module for personalized websites. You will learn how to configure the GUI used in <i>CoreMedia Studio</i> , 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 <i>Content Application Engine (CAE)</i> . 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 <i>Content Server</i> . 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 <i>Elastic Social</i> 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 <i>Importer</i> 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 <i>CoreMedia Search Engine</i> and the two feeder applications: the <i>Content Feeder</i> and the <i>CAE Feeder</i> .
Site Manager Developer Manual	Developers, architects, administrators	This manual describes the configuration and customization of <i>Site Manager</i> , the Java based stand-alone application for administrative tasks. You will learn how to configure the <i>Site Manager</i> with property files and XML files and how to develop your own extensions using the <i>Site Manager API</i> .
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of <i>CoreMedia Studio</i> . You will learn about the underlying concepts, how to use the development environment and how to customize <i>Studio</i> to your needs.
Unified API Developer Manual	Developers, architects	This manual describes the concepts and usage of the <i>CoreMedia Unified API</i> , 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 <i>Workflow Server</i> . This includes the administration of the server, the development of workflows using the XML language and the development of extensions.

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

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

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

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

<http://www.coremedia.com/training>

Contact the Training department at the following email address:

Email: training@coremedia.com

1.3.5 CoreMedia Support

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

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

<http://support.coremedia.com/>

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

Email: support@coremedia.com

Create a support request

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:

Support request

- 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.

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

Support checklist

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.

Log files

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.

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 pre-view	coremedia.log (Content Management Server) coremedia.log (Master Live Server)

Table 1.4. Log files check list

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) *.jpic files
	preview not running	coremedia.log (content server) preview.log
	website not running	coremedia.log (Content Management Server) coremedia.log (Master Live Server) coremedia.log (Replication Live Server) Blueprint.log capclient.properties license.zip
Server	not starting	coremedia.log (Content Management Server) coremedia.log (Master Live Server) coremedia.log (Replication Live Server) capclient.properties license.zip

1.4 Change Chapter

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

Table 1.5. Changes

Section	Version	Description
Content in multiple sections	7.5.41	Removed documentation of Persistent Cache and ProActive Engine. Some content has been moved into the Search Manual.

2. Overview

The overall goal of the *CoreMedia CAE (CAE)* framework is to provide a structure for any kind of web application that accesses the CoreMedia content repository. The declarative nature of this framework and the independence of the layers ensure fast development and maintainable application design. You may also encounter the term "ObjectServer" throughout APIs or configuration files. Please read *CoreMedia CAE* in that case. The overall application setup and web request processing are handled by the *Spring framework*, so it is useful to have a solid knowledge of Spring before developing *CAE* applications.

To represent the content objects in the repository, Java objects called *content beans* can be generated that directly reflect the repository structure. It is possible to extend these generated beans with any kind of custom business logic. On top of this data access object layer, a caching layer can be defined by simply declaring the cacheable properties of the content beans. The elements of the caching layer are views on the content beans and are therefore called *data views*. Based on the types of the content beans and/or data views, suitable views are chosen in order to render the back-end information. The object-oriented nature of the view registration and mapping subsystem harnesses the full power of inheritance and implementation relationships. Views may be defined in a supported template language, such as JavaServer Pages or FreeMarker, or in Java code.

The modular design makes it possible to extend and modify the *CAE* framework.

2.1 Components and Use Cases

The *Content Application Engine (CAE)* is a framework for the development of content applications. A content application, as defined by CoreMedia, is an application that takes content from several sources, transforms this content and delivers it to a target. This is a wide definition and comprises the "classical" task of a website delivered to a client, but also the editing and storing of content of the content management system.

The CAE is modularly built and offers components for different use cases. The following table lists the components of the CAE framework.

Table 2.1. Components of the CAE framework

Component	Description
<i>Content Application Engine web application</i>	The CAE web application offers a MVC model for content applications. It separates the view from the business logic and has declarative caching. It caches dependencies and contents in memory. It tracks invalidations and dependencies.
<i>Preview-based Editing</i>	A simple framework to make a preview website editable.

Highly Dynamic and Personalized Websites

The CAE web application is the basis for all content applications. It offers in-memory caching for highly dynamic websites. You can simply integrate third-party content into the web application. An example would be a website with personalized pages which includes content from an ERP system.

Content Push

The *CAE Feeder* is an application that calculates values from given objects triggered by the invalidation of these objects and that delivers these values to a receiver. The typical use-case of the *CAE Feeder* is to update a search engine index. However, it can also be used to push data to other external systems. See Section 5.5, "Integrating a Different Search Engine" in *CoreMedia Search Manual* for details.

2.2 Architecture

The *CoreMedia CAE* mainly comprises components from four sources:

- A servlet container that hosts the application
- The *Spring Framework* controls the application setup and main request control flow
- The *CoreMedia CAE Framework* provides content access and handles caching and rendering
- The *Application* is a custom implementation that typically provides custom request controllers, business logic, data view configuration for caching and templates that render the content.

The *CoreMedia CAE* strictly implements the MVC model for web applications:

- The controller part accepts a request and – depending on the request URI – dispatches it to an appropriate handler bean that executes the request using the model. The result is passed to the view layer for presentation. The *CoreMedia CAE* comes with a number of basic handler classes that provide out-of-the-box content display functionality and an easy starting point for customizations. Spring MVC 3.1 is fully supported. See [Section 4.3.1, “Handling Requests” \[50\]](#) for details.
- The model part comprises business entities stored in the content repository enriched with business logic. The *CoreMedia CAE* provides a framework for mapping content objects to generated and/or customized classes. Third-party repositories can be integrated as well. Business objects can be cached in this layer. See [Section 4.1, “Content Beans - Mapping content to objects” \[23\]](#) for details.
- The view engine is responsible for rendering objects into a presentation format, typically HTML. The *CoreMedia CAE* provides a flexible framework for object oriented template selection through the *ViewDispatcher*. See [Section 4.3.3, “Views” \[65\]](#) for details.

2.3 Caching

The *CoreMedia CAE* separates caching from business objects. Business objects are beans for content in the repository for example [ContentBean](#) implementations. They provide access to content properties and business logic computation results.

There are different caching layers that are used in the *CoreMedia CAE*. The lowest content caching layer is the Unified API. On top of that layer, [DataViews](#) and [CacheKeys](#) are cached. Both of these caching methods are used to cache results of computations from business related code.

2.3.1 Unified API Cache

All content access is routed through this cache, all content properties and metadata are cached. Its main purpose is to reduce server round-trips when content properties are accessed. This cache takes care of all configuration automatically, only cache sizes must be configured. The bigger the size, the less communication with the *Content Server* is needed during the lifetime of the application.

See [Section 5.5, “Configuration Property Reference” \[132\]](#) for more information about the property that configures the size of this cache: `repository.heapCacheSize`.

2.3.2 Data View Cache

All business objects that implement `AssumesIdentity` may be cached as *data views* by the CAE. Its main purpose is to cache results generated by business code getters. Data views are configured declaratively without direct modifications to the business objects. The properties of individual business objects and their aggregation and other forms of association can be defined. The *CoreMedia CAE* will automatically generate classes from that definition that are equivalent to your business objects with an additional cached state. The generation process is almost transparent and the generated classes comply with the same public interface(s) as the original classes. Although content properties are already cached in the *Unified API* caching layer, it is beneficial to additionally cache the relevant getter methods in the data view layer.

See [Section 4.2, “Data Views” \[31\]](#) for more information.

2.3.3 CacheKey Cache

The *Unified API* and data view provide a caching layer that is easy to configure but they both have their limitations. To overcome those limitations, [CacheKeys](#) classes allow caching of arbitrary computation results. Their API enables custom code to make full use of the [Cache](#).

See [CacheKey#evaluate in the API](#) for more information.

2.4 The Spring Framework

The Spring application framework is the frame that holds the *CoreMedia CAE* together. Much of the application's architecture is described in a Spring XML application context definition. Application specific extensions are easily plugged into the *CoreMedia CAE*, profiting from Spring's dependency injection features. Furthermore, the *CoreMedia CAE* makes use of the Spring MVC framework for its web request processing.

3. Administration and Operation

The *Content Application Engine (CAE)* is a framework for the development of content applications.

3.1 Connecting and Caching

This section covers configuration options related to the server communication and the basic cache configuration.

Connecting to the Content Server

In a CAE application there are a number of properties for setting up the connection to the *Content Server*, from which the *Content Application Engine* reads the content to be displayed. Configure the properties shown in the table to define the location of the *Content Server* and the identity of the user used to log in to the server.

Table 3.1. Connection properties

Property	Description
<code>repository.url</code>	The URL of the <i>Content Server</i>
<code>repository.domain</code>	The domain of the <i>Content Server</i>
<code>repository.user</code>	Define the user with which the CAE connects to the <i>Content Server</i> . Please note that the user must be permitted to use the <code>webserver</code> login service, which is only possible for the web server user, unless configured otherwise in the <code>jaas.conf</code> file of the <i>Content Server</i> .
<code>repository.password</code>	The password of the user.
<code>repository.workflow</code>	Use this property to disable the connection to the <i>Workflow Server</i> , because few content application require access to workflow data.

Configuring Cache Sizes

You can configure the size of the Unified API cache and of the disk cache for blobs using the properties defined in the table:

Table 3.2. Properties for cache size settings

Property	Description
<code>repository.heapCacheSize</code>	This property indicates the number of bytes used for the main memory cache of the <i>Unified API</i> embedded in the <i>Content Application Engine</i> . For 64 bit JVMs, the actual memory consumption may be up to twice the configured value. For 32 bit JVMs, the byte count is exact. When multiple CAEs run in a single application server, the caches are kept separate and the configured cache sizes add up.

Property	Description
<code>repository.blob-CacheSize</code>	This property defines the size of the disk cache for blobs.
<code>repository.blob-CachePath</code>	This property defines the location of the blob cache. Multiple CAEs may share the same directory for the blob cache. Again, the cache sizes add up. Make sure to provide enough disk space for caching.

Purging the Disk Cache after Forced Exits

When an application container is forced to shut down without stopping the web applications first, the CAE might not be able to clear its disk cache in time. This may happen when a Tomcat is shut down, which will invoke a process kill operation at system level, if the Tomcat does not shut down within eight seconds.

In order to avoid a buildup of left over cache files, it makes sense to purge the temporary file directory periodically during a planned downtime or every time at the start of the content application. Make sure not to purge the directory while it is in use by a CAE.

4. Development

The *CoreMedia CoreMedia CAE Framework* is intended for developing content applications with *CoreMedia CMS*. Its focus is set on web applications, yet the core frameworks are usable in other environments such as standalone clients, portal containers or web service implementations.

4.1 Content Beans - Mapping content to objects

The *CoreMedia CAE* defines a mapping framework to create application-specific "business" objects from generic content objects. In order to do that, application specific classes have to be written and they have to be registered with a factory that is used throughout the application whenever a content object needs to be converted into an application bean.

You do not need to write the beans from scratch. The *CoreMedia CAE* comes with a source code generator that reads your document type definition file and generates beans for each document type. The bean interfaces mirror the document properties, and the class hierarchy mirrors the document type hierarchy. After generating the code, you can modify and adapt it to your application's needs.

- `ContentBeans` can be used for other purposes than rendering, for example for implementing web services, for business logic deployed in the workflow server or in the CAE Feeder, or for custom standalone applications.
- `ContentBeans` can be cached in `DavaViews`. See [Section 4.2, "Data Views" \[31\]](#) for details.
- `ContentBeans` are rendered by the rendering layer. See [Section 4.3.3, "Views" \[65\]](#) for details.

4.1.1 Generate Content Beans from the Content Type model

The *Content Application Engine* comes with a source code generator that reads your content type definition file and generates beans for each content type. The bean interfaces mirror the content properties, and the class hierarchy mirrors the content type hierarchy. After generating the code, you can modify and adapt it to your application's needs.

Since you need the code generator only once (or even not at all, if you start over with *CoreMedia Blueprint*), there is no need for a dedicated application. Just add a temporary dependency

```
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>bean-generator</artifactId>
  <scope>runtime</scope>
</dependency>
```

to the POM file of your *CoreMedia CAE* module and build it. Then the code generator can be invoked as follows from within the `WEB-INF` directory of the *CoreMedia CAE* :

```
java -jar lib/bean-generator-<version>.jar
```

This gives you all available options of the generator as shown in [Table 4.1, “Parameters of the Beangenerator” \[24\]](#). You need to pass the location of your content type definition file(s), a package for the generated classes and a target directory for the source code. By default, the generator creates code with generics for link lists as comments in order to be compatible with legacy Java versions. When you have successfully created your content beans, you should delete the temporary dependency on `bean-generator` again so that your actual web application does not get unnecessarily bloated.

The meaning of the parameters of the Beangenerator is described in the next table.

Parameter	Variable	Description
<code>-d --document-types</code>	Name of the document types file to use	The URL of the document types file to use. You need at least one file.
<code>-baseonly</code>		Create only the base classes, but no implementations and no interfaces.
<code>-beanmapping</code>	Name of a file	The file into which the bean mapping will be written (see Section “Spring Configuration” [29]).
<code>-c</code>	Name of a properties file that contains the package information.	<p>Define specific packages for specific document types in order to organize your bean files properly. There are three possible configuration types:</p> <p>One package for all files of a document type</p> <p>Example:</p> <pre>doctype.Image.package= com.mycompany.image doctype.Article.package= com.mycompany.article</pre> <p>Different packages for the different files of a document type</p> <p>Example:</p> <pre>doctype.Article.package.base= com.mycompany.base doctype.Article.package.implementation= com.mycompany.impl</pre>

Table 4.1. Parameters of the Beangenerator

Parameter	Variable	Description
		<pre>doctype.Article.package.interface= com.mycompany</pre> <p>Settings for all document types</p> <p>Example:</p> <pre>package=com.mycompany package.base=com.mycompany.base package.implementation= com.mycompany.impl package.interface=com.mycompany</pre>
<i>-document</i>	Document type	Generate the code for the defined content type. It must be included in one of the document type files defined with the <i>-d</i> parameter.
<i>-e, --exclude</i>	Comma separated list of document types	Exclude these types from generation process.
<i>-enc, --encoding</i>	One of the defined Java encodings	Encode the output files with the defined encoding (default is Cp1252)
<i>-f</i>		Overwrite existing files
<i>-generics</i>		Use Java 1.5 Generics.
<i>-gensetter</i>		Deprecated parameter. Generates code with public setter methods.
<i>-avoidlist</i>		Generate methods with simple (non-list) parameters and return values for linklist types that have a maximum cardinality of '1'.
<i>-i, --include</i>	Comma separated list of document types	Document types that should be included in the generation process (default is all)
<i>-noimpl</i>		Only create base classes and interfaces but no implementations
<i>-o, -out</i>	Directory name	The directory where to generate the source code into. If it does not exist, it will be created.
<i>-p, --package</i>	Package name	The package name used for the generated beans. A setting defined with <i>-c</i> overwrites the package defined with <i>-p</i> . To avoid confusion only use <i>-p</i> or <i>-c</i> .

Example

If you use the following call from the lib directory of your content application web application,

```
java -jar beangenerator.jar
-d /contentserver/config/contentserver/doctypes/menu-doctypes.xml
-o /output/classes
-c /properties/package.properties
-beanmapping /properties/contentbeans.xml
-e Picture
-generics
```

with the following `package.properties` file,

```
doctype.Dish.package=com.menusite.dish
package=com.menusite
```

this will generate interfaces, base classes and implementation classes for all document types of the menu site example except for the `Picture` document type. The document types are read from the `/contentserver/config/contentserver/doctypes/menu-doctypes.xml` file and the generated classes are written to the `/output/classes` directory.

The `Dish` document type belongs to the `com.menusite.dish` package while the other types are in the `com.menusite` package, this information is taken from the `/properties/package.properties` file. The bean mapping is written to the `/properties/contentbeans.xml` file. Generics are used which results in code like the following:

```
public List<? extends Dish> getContent() {
    List/*<Content>*/ contents = getContent().getLinks("content");
    return (List<? extends Dish>) createBeansFor(contents);
}
```

Structure of the Generated Code

When you inspect the generated classes, you will find that three files per document type are generated:

- An interface with the same name as the document type
- An abstract class ending with "Base" and
- A concrete class ending with "Impl"

The interface is what you should use in other classes, "`*Base`" contains the repository access code and "`*Impl`" is the actual class that is instantiated. This class is the place for you to modify. When a document type inherits from another type, its "`*Base`" class inherits the "`*Impl`" class of its parent. This way, it inherits the custom extensions made for the supertype. For content types that do not have a parent,

the `**Base` class inherits from a framework class `AbstractContentBean` that defines the underlying content bean, factory, equality and hash code as well as a few convenience methods.

The generated code may not compile under some circumstances, due to naming conflicts, for example. A content property named 'content' will clash with the method `#getContent` inherited from `ContentBean`. In this case you should rename the generated getters in the interface and the `*Base` class.



The `**Base` class contains property getters for every user-defined property in the corresponding content type. Getters are not generated for metadata such as name or creation date. The property types are mapped to Java as follows:

Property Type	Java Type	Conversion
<code>IntProperty</code>	<code>int</code>	Simply the value from the underlying content object
<code>StringProperty</code>	<code>String</code>	Simply the value from the underlying content object
<code>DateProperty</code>	<code>Calendar</code>	Simply the value from the underlying content object
<code>XmlProperty</code> (with grammar "coremedia-struct-2008")	<code>Struct</code>	The parsed <code>Struct</code> value from the underlying content object
<code>XmlProperty</code>	<code>Markup</code>	The markup is transformed. Every internal xlink to a document or blob is transformed into the corresponding content bean id or blob id.
<code>BlobProperty</code>	<code>CapBlobRef</code>	This is the result of <code>#getBlobRef</code> of the underlying content object
<code>LinkListProperty</code>	<code>List</code>	Every content object in the link list is converted to a bean through the content bean factory

Table 4.2. Document property to Java property mappings

You are free to modify the generated code. For example, you can do the following:

- Add a new method to both the public interface and the `**Impl` class
- Remove a method from the public interface of a bean. It will still be available for the implementation classes but not for clients.
- Combine both to implement a derived property based on the now hidden content property, possibly with a different result type

- Implement additional interfaces that may crosscut the document type hierarchy

When extending, you should not declare any fields in a content bean except for read-only, immutable service references.

Patterns For Content Beans

A few important patterns are used by the generated content beans. Keep them in mind when you extend these classes:

- Construction

Content beans are both used to denote content and references (links) to content. A content bean used as a link must be cheap to construct. Thus, at construction time, a content bean should only set the information required to identify itself: its `contentBeanFactory` and content object (and maybe other required services like a DAO or a JDBC data source). No content should be retrieved. The source code generator fulfills this requirement by generating a default constructor and the two getters defined in the *CoreMedia CMS* interface.

When extending, you should not modify the default constructor in the generated beans.



- Identity, equality

Two content beans originating from the same factory for the same content object must be equal. They identify the same business identity. Generated content beans fulfill this requirement by inheriting `#equals` and `#hashCode` from `AbstractContentBean` which is defined in terms of the corresponding content methods.

When extending, you should not override `equals` or `hashCode`.



- Mutable state

A content bean must not store mutable information. Caching of mutable state is performed in other layers. All methods of a content bean should always modify the content object directly. This way, a content bean can never be invalid when the repository contents change.

When extending, you should not declare any fields in a content bean except for read-only, immutable service references.



- Mutable values

The results of getter methods of a content bean must not be modified by application code. Modification would lead to race conditions and break the data view framework. Getter methods of content beans should return immutable objects in order to prevent errors caused by illegal modification. In particular, they should not return arrays but immutable collections.

These patterns apply to any object that you design as a representative for data stored in an external data source and that you want to use within the data view caching framework. They ensure that the object is lightweight, interchangeable and always valid. With efficient data retrieval will be dealt at a later stage. Other designs are also possible, for example, stateful business objects directly loaded from a DAO, but they require a more complicated interaction with the caching framework that is not covered in this manual.

Spring Configuration

In order for the *CoreMedia CAE* to instantiate the right classes at runtime, they need to be configured with the factory. The engine's default factory implementation uses the Spring application context to instantiate content beans. This way content beans can participate in Spring's dependency injection mechanism - for example, they can receive references to other services without having to resort to service lookups in JNDI or the servlet context.

The content type to content beans mapping is defined using Spring's XML notation. It should contain a prototype definition for each class corresponding to a document type.

Prototype definitions follow a specific naming scheme. In order to be found by the factory, they must be given the same name as the factory, followed by a colon ':' and the name of the document type for which they were generated. For example, a class `com.company.Article` that represents Article documents is registered with the factory as follows:

```
<bean
  name="contentBeanFactory:Article"
  parent="abstractContentBean"
  scope="prototype"
  class="com.company.ArticleImpl" />
```

This line is a template for the content bean factory; it says:

- This is a definition for a content factory bean for the document type Article
- The bean might inherit configuration settings from a parent bean. This can simplify the configuration but is not mandatory.
- This definition is a prototype, not a singleton, it must be newly instantiated for every article document
- The implementation class is `com.company.ArticleImpl`

In short this reads as: "for documents of type **Article**, return a **new instance** of class `com.company.ArticleImpl`".

Important: using `scope="prototype"` is vital, otherwise Spring would cache one instance and return the same object every time.



Programmatic Access to Content Beans

In order to "bootstrap" yourself into the world of content beans from the *CoreMedia Unified API*, you need to use the content bean factory programmatically, for example from within a Controller. The factory API is simple, the most relevant method is [ContentBeanFactory#createBeanFor\(Content\)](#). For example:

```
Content content = ... // for example through a query
Article article =
    (Article) contentBeanFactory.createBeanFor(content);
```

The controller needs access to the content bean factory. Since the controller itself typically is a bean defined in the application context, you can inject the factory reference into the controller object:

```
<bean id="myController" class="...">
  <property name="contentBeanFactory" ref="contentBeanFactory"/>
  ...
</bean>
```

This fragment will invoke `#setContentBeanFactory` on the controller supplying an instance of the referenced factory.

4.2 Data Views

You've learned that the business objects should not store any of the information they receive from their data source. This task is performed by a dedicated caching layer.

Caching in the *CoreMedia CAE* has a number of important properties:

- Caching is defined outside the business objects.
- Caching is achieved by building a subclass of a business class, materializing properties into actual fields and storing an instance of this subclass.
- Cached objects have the same interface as the non-caching business objects so that one can develop against non-cached versions first and does not need to change the code later.
- A set of public bean properties of the business object is subject to caching.
- Cached objects can be aggregated: one cached object can store a direct object reference to another cached object. Once retrieved from the cache, this association can be navigated without further synchronization or cache lookups. This is important for fast rendering.
- It is possible to cache different sets of properties of the same business object; "more" or "different" properties of this object can be used in different contexts. Often it is not sensible to cache all properties of an object for two reasons: if one property set is significantly smaller than another or faster to compute (for example, only the metadata), it may be worth the overhead of caching two objects. The second – more important – reason is dependencies: if one representation acquires fewer dependencies than another and provides all properties needed in a certain context it should be cached separately (for example, "uses only content properties from the CMS but not the database"). Especially different amounts of aggregation are a concern here (for example, when the object cached in the parent property in turn depends on a different content object).
- Cache invalidation is dependency-driven, that is, a cache value has associated dependencies on external values and is invalidated if one of these changes. This happens automatically for dependencies to the content repository.

The *CoreMedia CAE* caching layer gives you the option to define several cached representations for one business object. It is possible to distinguish the following:

- The properties of the business object which are cached
- For properties that refer to other business objects, which cached representation, if any, should be aggregated

Such a definition is called a *data view* of an object.



Do not confuse this term with the term *view* used in rendering: a *data view* is an object that extracts and aggregates source data in the cache. A *view* is a method of rendering an object. Of course, data views and views are related: in order to render a view efficiently, the displayed object should provide its data sufficiently fast; possibly using a data view from the cache.

For example, you can define

- "fully cached" for display, for a data view that contains a page's description, its content and its parent page "for linking"
- "for linking", a data view of a page that only contains its description

4.2.1 Defining Data Views

Data views are defined declaratively using XML according to a schema `/META-INF/dataviews.xsd` which is located inside `cae-contentbeanservices-api.jar`. Behind the scenes, subclasses of the application classes are generated. This process is transparent, as the remainder of the application should be written to the application class interfaces. Looking at a data view object's class, however, it becomes obvious that it is actually an instance of a subclass of the original business class. How these classes behave, will be described later.

A sample XML data view definition using the example from above looks as follows:

```
<dataview appliesTo="com.company.PageImpl">
  <property name="name" />
  <property name="description" />
  <property name="content" />
  <property name="parent" associationType="composition">
    <dataview appliesTo="com.company.PageImpl" name="forLinking">
      <property name="name" />
    </dataview>
  </property>
</dataview>
```

This definition says: The default (no name attribute) data view of a `PageImpl` materializes the properties `name`, `description`, `content` and `parent` as fields where the latter is itself a bean of type `PageImpl` with data view `forLinking` (which is defined inline) applied. The association between the two data views is a composition. That means: the outer object embeds its private parent instance which is not shared with other beans, that is, the outer element owns the inner element exclusively. Specifically, no cache lookup is performed to retrieve the inner element, but it is always created when the outer element is created. The various association types will be described later.

This data view defines a view on Page documents that makes the following properties cached and quickly accessible:

- `page.{name,description,content,parent}`
- `page.parent.name`

All other properties are inherited from your `*Impl` classes and are therefore accessed dynamically. That does not mean that they are necessarily slow (there is a document cache after all).

To use the defined data views, the data view factory dynamically constructs two subclasses of `PageImpl`, one for each data view definition. When the default data view is loaded, the data view factory will look into the cache with a key `<Page content bean, null (default)>` (Remember that the `Page` content bean's equality is defined in terms of its content id). If the key is not in the cache, the factory will create an instance of the first subclass and load the properties `description`, `content` and `name` by invoking the business methods and storing the results. Furthermore, it will load `parent` (another lightweight `PageImpl`) and construct data view `forLinking` for it. To do so, it will not do a cache lookup but instantiate the corresponding second subclass of `PageImpl` directly. The result is stored as the materialized `parent` property of the result.

The generated code for the definition from above is roughly equivalent to the following:

```
class PageImpl$$ extends PageImpl {
  String name = super.getName();
  ...
  PageImpl parent =
    (PageImpl)dataviewFactory.lookupUncached(
      super.getParent(), "forLinking");
  ...
  Page getParent() {
    return this.parent;
  }
  ...
}
```

It is possible to define data views with the same name for different classes. During the lookup for that name, the class of the object determines which data view definition is chosen – a dynamic dispatch very much like for content bean creation or the templates. This way, it is possible to apply a data view to a property value with a varying runtime class.

The default data view has a special meaning: it is the data view that is loaded at the beginning of a request when rendering the bean referenced by the URI. So this data view should correspond to the properties that the default view and its included fragment views require.

4.2.2 Data View Design

This section describes concepts and guidelines for the design of data views.

Association Types

There are a number of design trade offs for data views. Consider the `forLinking` data view of the page, which is a composition and thus creates a private instance for each child. This design avoids a cache lookup. Caching has an overhead and allocating a cache entry for a parent object with only one string property would cost more than it saves.

On the other hand, since you defined a cacheable default data view of a page anyway, you could consider reusing the parent's default data view for the child:

```
<dataview appliesTo="com.company.PageImpl">
  <property name="name"/>
  <property name="description"/>
  <property name="content"/>
  <property name="parent" associationType="aggregation"/>
</dataview>
```

An aggregation is different from a composition in that a cache lookup is performed for this property. All children would therefore share the same parent instance (provided it is not evicted from the cache). In this definition, a `PageImpl` would aggregate its parent which would again recursively aggregate its parent ... until `null` is reached (any data view for `null` is `null`). Since you expect parents to be frequently accessed anyway, it is OK to have them pulled into the cache by their children. The generated code is basically equivalent to the following:

```
class PageImpl$$ extends PageImpl {
  // null is the default data view
  PageImpl parent =
    (Page) dataviewFactory.lookupCached(super.getParent(), null);

  public Page getParent() {
    return this.parent;
  }
  ...
}
```

However, you also have to take the cache's dependency tracking into consideration. When a data view reads a content object, a dependency is recorded. When a data view does a cache lookup for another data view, a dependency is recorded as well. Given the page definition above, a child page will therefore depend on its content object and onto its parent which itself has a dependency on its content object and so on. Thus, if you change the name of the root page, all page objects will be invalidated since they have transitively aggregated it.

There is an alternative solution. Instead of *embedding* the default data view of the parent, you can do the cache lookup on every access to the parent property. You avoid the dependency; instead you always read the latest version from the cache. This lazy lookup is achieved as follows:

```
<dataview appliesTo="com.company.PageImpl">
  <property name="name"/>
```

```
<property name="description"/>
<property name="content"/>
<property name="parent" associationType="static"/>
</dataview>
```

Defining a static association will make the caching system store *which* parent a page is associated with (the lightweight `PageImpl` instance that basically only holds the parent id), in place of its default data view (which contains the parent's state). Instead, a cache lookup is done for the default data view whenever the parent property is retrieved. In Java code, this behavior looks like this:

```
class PageImpl$$ extends PageImpl {
    PageImpl parent = super.getParent();
    ...
    Page getParent() {
        return (Page)dataviewFactory.lookupCached(
            this.parent, null);
    }
    ...
}
```

A cache lookup is reasonably efficient to make this definition possible. You should, however, keep an eye on the number of lookups. A cache lookup requires thread synchronization, and too many synchronization requests might lead to contention.

One last thing needs mentioning: Properties that should not be cached are simply omitted from the data view definition. But what, if you still want to apply a data view to the property value? For this case, a "dynamic" association can be defined:

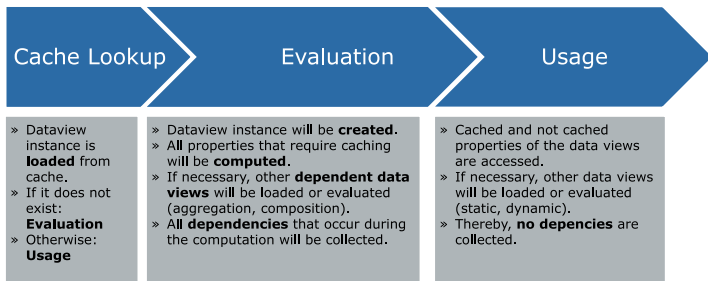
```
<property name="randomPage" associationType="dynamic"/>
```

With this definition, `#getRandomPage()` will be generated as follows:

```
class PageImpl$$ extends PageImpl {
    ...
    Page getRandomPage() {
        // invoke original impl, don't cache
        Page p = super.getRandomPage();
        // cache lookup
        return (Page)dataviewFactory.lookupCached(
            p, null);
    }
    ...
}
```

Figure 4.1, "Phases of a data view lifecycle" [36] shows, how data views are loaded and evaluated in the lifecycle of an HTTP request.

Figure 4.1. Phases of a data view lifecycle



To recapitulate, if a property is an association to another bean, it is possible to apply a data view to that bean as well. There are four ways to do that:

Association Type	Reference is stored in field	Data view is applied at ...	Cache Lookup	Implies Cache dependency to
composition	yes	creation time	no	Content Bean and Data View
aggregation	yes	creation time	yes	Content Bean and Data View
static (default)	yes	property access	yes	Content Bean
dynamic	no	property access	yes	none

Table 4.3. Association types

Guidelines For Data View Design

This section contains some guidelines or rules of thumb for the proper definition of data views.

Define the property configuration recursively

You have to ensure that a bean's data view configuration is recursively reachable from the root bean's data view configuration. For every property returning this bean, a "bridging" data view configuration entry needs to be added. In order to prevent the cache to be filled with unnecessary "bridge" properties, the association type *dynamic* might be used, for instance.

```

<dataview appliesTo="com.mycompany.PageBean">
  <property name="content" associationType="dynamic"/>
</dataview>
  
```

Why is this important?

From a data view's point of view, the process of rendering nested bean takes place as follows:

1. The controller computes the root bean (containing nested beans) from an incoming request
2. The controller invokes `DataViewFactory#loadCached(bean, name)` for this bean in order to apply a data view
3. The controller passes the bean to the rendering engine (and therefore to the view templates) where the bean's properties are accessed and rendered
4. When accessing a bean property which is returning further beans, a data view will be applied automatically to these sub beans

In other words, the initial appliance of a data view to the root bean leads to a recursive appliance of data views to all sub beans. Unfortunately, this is true in case that there is a data view configuration (`dataviews.xml`) for every relevant bean/property only. Let's say there is no such configuration for the root bean, then no data views will be applied to the sub beans automatically and these beans will be returned as they are. As a consequence, the sub beans wouldn't be cached even if there is a data view configuration available for them.

Example

There is a `PageBean` having a JSP template:

```
public interface PageBean {
    ArticleBean getContent();
}
<cm:include target="${self.content}"/>
```

The template includes the rendering of an `ArticleBean`

```
public interface ArticleBean {
    String getHeadline()
}
<c:out value="${self.headline}"/>
```

If there is a data view configuration for the (supposed "expensive") property "headline"

```
<dataview appliesTo="com.mycompany.Article">
  <property name="headline"/>
</dataview>
```

without defining a configuration for the root bean

```
<dataview appliesTo="com.mycompany.PageBean">
  <property name="content" associationType="static"/>
</dataview>
```

then there won't be any caching of the "headline" property.

Auto completing the data view configuration

In large projects, a recursive definition of data views might be a difficult and error-prone task. Unwanted gaps in the transitive closure and thus uncached beans may be the result. For this reason, there is a feature called "auto completion" which helps to achieve a complete transitive closure of data views.

Auto completion can be configured in the `dataviews.xml` like this:

```

    <dataviews>
    ...
    <autocomplete>
    <class
name="com.coremedia.objectserver.dataviews.AssumesIdentity"/>
    <class name="java.util.Map"/>
    <class name="java.util.List"/>
    </autocomplete>
    ...
</dataviews>

```

Example 4.1. Auto completion example

This configuration causes the `DataViewFactory` to implicitly use the association type `DYNAMIC` for all bean properties whose getter method's return type inherit from `AssumesIdentity`, `Map` or `List` and which are not already covered by a data view configuration. Not only properties of configured data views will be automatically completed but also those of beans that do not have a data view configuration at all.

Please note that only the getter method's return type is taken into account during auto completion, not the concrete type of an object returned from the getter at runtime.



As a consequence of this feature, you are able to design a lean data view configuration with only a few purposeful property configurations.

But there are also some drawbacks: If there are only a few data views explicitly declared, the `DataViewFactory` will have to create many transient ("uncached") data view objects in order to provide closure. Thus, lots of additional objects populate the java heap temporarily which mean more work for the garbage collector. In addition, some synchronization is required when accessing properties. This might reduce the application's performance. Choose the auto-completion types carefully so that all property return types are covered on the one hand, without being too generic on the other hand. As a rule of thumb, the super interface of your content beans (such as `AssumesIdentity`) together with `java.lang.List` and `java.lang.Map` might be a good starting point.

Of course, there might be properties which should not be automatically completed. For this reason, a pseudo association type `none` can be used to explicitly exclude a property from being automatically completed.

```
<dataview appliesTo="com.yourcompany.YourBean">
  <property name="userInfo" associationType="none"/>
</dataview>
```

Example 4.2. Auto completion exclusion example

The property `userInfo` of `YourBean` won't be ever automatically completed and will be treated as if there is no automatically completion and no data view configuration.

Let the controller apply a data view to its beans

A controller's contract is to compute a `ModelAndView` which contains one or more model beans to be passed to the rendering engine. In order to make the model beans cacheable, it's important to apply a data view to these beans within the controller.

Example

This example demonstrates a simple controller implementation snippet:

```
ModelAndView handleRequest(HttpServletRequest request,
    HttpServletResponse response) {
    // compute the model bean from the request
    MyBean modelBean = computeBean(request);
    // apply a data view to this bean
    MyBean cachedModelBean = (MyBean)
    getDataViewFactory().loadCached(modelBean, null);
    // construct the controller's result
    ModelAndView result = ControllerUtils.viewOf(cachedModelBean);
    return result;
}
```

Use caching only when it is reasonable

Caching with data views is for improving an application's performance: The results of property computations are stored in the heap memory in order to prevent a repeated computation when accessing the property the next time. The values are removed from the cache when they are becoming invalid or due to evictions.

The process of caching itself is not for free: Each cached entry consumes a bit of the (limited) heap space on the one hand. On the other hand, each cache read or write operation is synchronized by the cache which might lead to decreased concurrency. For this reason data view caching of a single property should be used purposeful, that is when it results in a better performance. Here are some situations where data view caching might not be worthwhile

- The computation of a property is cheap.
- The property value is already cached elsewhere. For instance, the *Unified API* is already caching its content properties: When simply delegating the content bean's property access to the content objects, the content beans need not to be cached again. Another example is a property which accesses another already cached property, for example a property `firstSentence` which performs a cheap string operation on a cached property `text`.
- A cached data view will be generally invalidated or evicted immediately after it is put into the cache without or rarely being accessed in the mean time.

Make sure that it is worthwhile from a performance point of view before enabling a property to be cached by a data view.

Avoid caching of large objects

Caching with data views is especially suited for properties that consume moderate memory. In opposite, large objects (such as binary objects) shouldn't be cached by data views since the heap memory is used disproportionately.

Choose the right association type

Properties can be separated into two groups from the data view's point of view

- Associating Properties: Properties which values are beans or collections of beans where data views can be applied on again.
- Simple Properties: All other properties with return values such as String, Int or other objects

You do not need to define an association type for a simple property. Instead, a data view configuration such as `<property name="propertyname" />` is sufficient. For an associating property you have to choose between the following association types which differ in terms of memory consumption, synchronization behavior and invalidation/eviction behavior.

- static
- composition
- aggregation
- dynamic

Despite this different behavior, these aspects doesn't need to be considered primarily when starting to create the data view configuration. For the beginning it is sufficient to choose "static" for a cacheable property and "dynamic" for a non-cacheable property in order to make another property recursively reachable (see above). As soon as you have finished your initial data view configuration, you can

do some optimizations by replacing specific association types with "aggregation" or "composition" in second step.

You can use the *CoreMedia Contribution* "CAE Console" to tweak your data view settings.

Do not implement property methods that use context data

In order to make a bean property cacheable you have to implement a public (non static and non final) getter method without parameters. Make sure that the method's implementation doesn't use any context data such as "current user", "current session" or similar stateful data. Otherwise, a property value is related to an arbitrary context when putting it into the cache. When reading it from the cache then, it might not fit to the reader's context.

The following example demonstrates a bad implementation where a list of content objects is filtered according to the current user's rights.

```
public List<ContentBean> getLinks() {
    List<Content> contents = getContent().getLinks("links");
    List<ContentBean> result = new ArrayList<ContentBean>();
    for (Content content : contents) {
        if (mayRead(content, getCurrentUser())) {
            // bad use of context data
            result.add(createBeanFor(content));
        }
    }
    return result;
}
```

Assume the property "links" to be cached when accessing it the first time: The cached result depends on the right of the user which accesses this property for the first time. Another user accessing this property afterwards will obtain a value which is not appropriate to the user's rights and therefore might have access to more or fewer contents than required.

Example Data View Design

This section illustrates the process of defining a data view configuration. For this example, a simple site with three pages is used. The first page consists of a brief overview of two articles that are completely shown on two separate pages. These article instances are shared between the overview page and the detail pages:

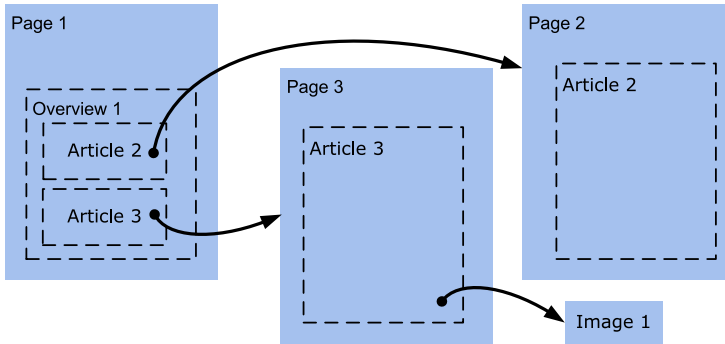


Figure 4.2. Example site structure

The entities are represented as beans and properties where the properties are assumed to have different costs: Some are expensive to compute while others are cheap.

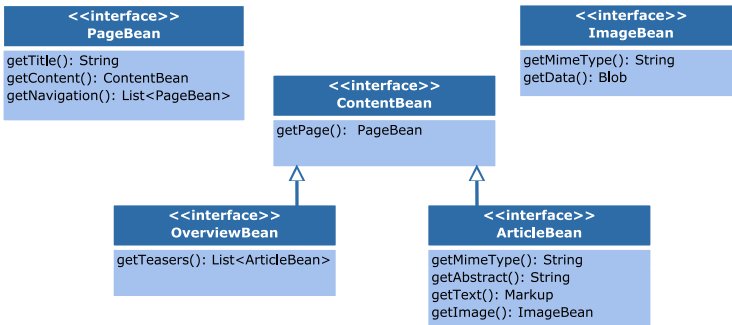


Figure 4.3. Entity Model

Bean	Property	Description	Expensive
PageBean	Title	The page's title.	No
	Content	The page content as a linked OverviewBean or ArticleBean.	No
	Navigation	All PageBeans to be rendered as navigation.	Yes
Content-Bean	Page	The PageBean which embeds this bean.	Yes

Table 4.4. Bean Properties in the DataView Example. (*) The computation of property "abstract" is not expensive by itself but the access of property "text" only.

Bean	Property	Description	Expensive
Overview-Bean	Teasers	A list of ArticleBeans to be rendered as teasers.	No
Article-Bean	Headline	The article's headline.	No
	Text	The article's text.	Yes
	Abstract	The article's abstract which is extracted from property Text automatically.	Yes*
	Image	An optional link to an image.	No
Image-Bean	MimeType	The image data's mime type.	Yes
	Data	The binary data.	No

The JSP templates for rendering the beans are modeled as follows:

PageBean.jsp

```
<html>
<head>
  <title><c:out value="\${self.title}"/></title>
</head>
<body>
  <div class="content"><cm:include self="\${self.content}"/>
  </div>
  <div class="navigation">
    <ul>
      <c:forEach items="\${self.navigation}" var="page">
        <li><a href="\<cm:link target='\${page}'/>">
          <c:out value="\${page.title}"/></a></li>
      </c:forEach>
    </ul>
  </div>
</body>
</html>
```

ArticleBean.jsp

```
<h1><c:out value="\${self.headline}"/></h1>
<div><c:out value="\${self.text}"/></div>
<c:if test="\${!empty self.image}">
  
</c:if>
```

OverviewBean.jsp

```
<c:forEach items="\${self.teasers}" var="article">
  <h2><c:out value="\${article.headline}"/></h2>
  <p>
```

```
<c:out value="${article.abstract}"/>
  [<a href="<cm:link target='${article.page}' />">more</a>]
</p>
</c:forEach>
```

Considering the above mentioned settings, the following `dataviews.xml` file can be derived:

```
<?xml version="1.0"?>
<dataviews xmlns=
"http://www.coremedia.com/2004/objectserver/dataviewfactory">

<dataview appliesTo="com.mycompany.PageBean">
  <property name="content" associationType="dynamic"/>
  <property name="navigation" associationType="static"/>
</dataview>

<dataview appliesTo="com.mycompany.ArticleBean">
  <property name="page" associationType="static"/>
  <property name="text"/>
</dataview>

<dataview appliesTo="com.mycompany.OverviewBean">
  <property name="teasers" associationType="dynamic"/>
</dataview>

<dataview appliesTo="com.mycompany.ImageBean">
  <property name="mimeType"/>
</dataview>

</dataviews>
```

All expensive associations (`PageBean#Navigation` and `ArticleBean#Page`) are declared to be data viewed using the default association type "static". Please note, that `OverviewBean#Page` is not marked here since this is not accessed by the templates. `PageBean#Content` and `OverviewBean#Teasers` are marked with the association type "dynamic" although they are not expensive: Instead they are used making `ArticleBean` recursively reachable from the `PageBean`. Finally, the non-associating but expensive properties `ArticleBean#Text` and `ImageBean#MimeType` are marked for caching as well. `ArticleBean#Abstract` is not marked here because it benefits from the already cached `ArticleBean#Text`.

Keep in mind that a perfect data view configuration depends on a lot of circumstances. Let's say that the underlying contents are updated very rarely on the one hand but accessed very often on the other hand. In order to reduce the number of cache read operations, some property associations might be switched to "composition". An additional "teaser" data view might be introduced in order to cache the `ArticleBean`'s different views (overview and detail) with separate objects.

```
<dataviews xmlns=
"http://www.coremedia.com/2004/objectserver/dataviewfactory">

<dataview appliesTo=
"com.coremedia.objectserver.dataviews.examples.PageBean">
  <property name="content" associationType="composition"/>
  <property name="navigation" associationType="static"/>
</dataview>
```

```

</dataview>

<dataview appliesTo=
"com.coremedia.objectserver.dataviews.examples.ArticleBean">
  <property name="text"/>
</dataview>

<dataview appliesTo=
"com.coremedia.objectserver.dataviews.examples.ArticleBean"
name="teaser">
  <property name="abstract"/>
  <property name="page" associationType="static"/>
</dataview>

<dataview appliesTo=
"com.coremedia.objectserver.dataviews.examples.OverviewBean">
  <property name="teasers" associationType="composition"
  dataview="teaser"/>
</dataview>

<dataview appliesTo=
"com.coremedia.objectserver.dataviews.examples.ImageBean">
  <property name="mimeType"/>
</dataview>
</dataviews>

```

Data Views for Experts

Data view design can be quite tricky. This section documents a very subtle pattern, injected aggregation, that should be omitted.

This problem occurs when you create beans that link to other beans that could be data views. Doing so, you will lose data view dependencies, because the data views are loaded outside of your bean.

Example

Take a `Page` bean, created in a controller and inject another content bean of type `Linkable`, called `childBean`. The `Page` bean has a getter method `getTitle()` that accesses the `Linkable` bean. The return value of the getter should be cached.

```

public class Page implements AssumesIdentity {
  private Linkable childBean;

  public void setLinkable(Linkable child) {
    this.childBean = child;
  }

  // not cached in dataview
  public Linkable getLinkable() {
    return this.childBean;
  }

  // cached in dataview!!!
  public String getTitle() {
    return this.childBean.getTitle();
  }
}

```

```

public boolean equals(Object o) {...}

public int hashCode() { ... }

public void assumeIdentity(Object bean) {
    this.childBean = ((Page) bean).getLinkable();
}
}

```

When the `Page` bean is created, it might be that the `Linkable` bean itself is a data view. If not, everything is fine. If you call `Page#getTitle()` a property dependency for `Linkable` is created. But, if the `Linkable` is a data view, no dependency is tracked:

The `Page` bean then acts like a data view that aggregates the `Linkable`. As a result, no property dependencies are generated if you call `Page#getTitle()`. Also, the `Linkable` is injected into the `Page` bean and therefore no data view dependency for the `Linkable` exists. As a result, the cached `Page` is not invalidated if the `Linkable` changes!

Solution

Do not access cached methods from a cached method or do not store the `Linkable` bean but the corresponding content object. Another method would be, to unwrap the `Linkable` data view into a normal `LinkableImpl`. You can use [DataViewHelpers](#) methods `#isDataView()` and `#getOriginal()` for that.

4.2.3 Configuring Cache Sizes

After defining the data views, make sure to configure the cache correctly, so that the data view objects are not evicted from the cache immediately. An indicator for this situation is the message "Unreasonable Cache Size null for java.lang.Object" in the log file.

To configure the cache, add a `<cacheSize>` element to the data view definition XML file, using attributes to specify the maximum number of cached instances and the object type this configuration should apply to. As a minimal solution, you can insert the line

```

<cacheSize
class="com.coremedia.objectserver.dataviews.AssumesIdentity"
size="10000"/>

```

This will allow a total of 10000 data view objects to be cached.

A more elaborate method would be to partition the cache according to the type of the cached objects. The type of an object is defined either by the Java type hierarchy or, if the object implements the interface `com.coremedia.dis`

`patch.HasCustomType`, by the result of the method `getCustomType()`. For ordinary content beans, the Java type hierarchy is used.

You can configure sizes for different types. If multiple types apply for a single cached object, the most specific type is used. For example

```
<cache size class="com.company.cms.SuperType" size="1000"/>
<cache size class="com.company.cms.SubType" size="100"/>
```

would allow the caching of up to 1000 direct or indirect instances of `SuperType` as long as these are not also direct or indirect instances of `SubType`. For `SubType`, at most 100 instances would be cached. This can make sense if instances of `SubType` consume a lot of main memory, so that 1000 instances might lead to an `OutOfMemoryError`.

Because data views extend their bean class, it is sufficient to configure cache sizes for the bean classes. You need not reference the class names of the automatically generated data view classes.

Please note that the configured cache sizes are directly forwarded to the cache of the *Unified API* in the CAE. That cache is an instance of the class [com.coremedia.cache.Cache](#). That class does not perform any type hierarchy analysis when caching objects. This is only done by the data view factory inside the CAE.

Please note that configured values for cache classes for data views may overwrite configured values for cache classes for cache keys, for example if `java.lang.Object` is configured. Make sure to always use `com.coremedia.objectserver.dataviews.AssumesIdentity` or classes higher in the class hierarchy if configuring cache classes for `DataViews`.



4.2.4 Writing Cacheable Beans

As mentioned above, the `DataViewFactory`'s caching mechanism takes care of dependencies. Any data view property may define one or more objects (called "dependencies") on which this property depends on. When caching a property, two things are stored in the cache: The property's value as well as its dependencies. In case that any dependent object becomes invalid (by modifications on it, for example) the dependent property value becomes invalid as well and will be removed from the cache automatically.

Example

A data view property "headline" is calculated from a row in a database table and so this row is defined as a dependency. When caching an instance of this property's value, the dependency is tracked as well. Changing the table's row causes the cached value to become invalid and this value to be removed from the cache.

Defining dependencies for a property value is done during the property's value computation by invoking the static method `com.coremedia.cache.Cache#dependencyOn(Object)` for each dependency. In order to notify the cache about a dependency invalidation, the method `invalidate(Object)` needs to be invoked on the `DataViewFactory`'s `Cache` instance. As a result, any cached item depending on this object is removed from the cache.

```
public class Bean {
    public String getHeadline() {
        Cache.dependencyOn(new String("mydependency"));
        return getHeadlineFromDatabase();
    }
}
```

```
DataViewFactory dataViewFactory = ...
Bean bean = new Bean();
Bean dataView = (Bean) dataViewFactory.loadCached(bean1);
String headline = dataView.getHeadline();
```

```
DataViewFactory dataViewFactory = ...
Cache cache = dataViewFactory.getCache();
cache.invalidate(new String("mydependency"));
```

Example 4.3. Bean property with custom dependency. Value of "headline" depends on dependency "mydependency".

Example 4.4. Accessing `getHeadline()` causes the property's value to be cached together with the dependency "mydependency" of type "String" in case caching is enabled for Bean's property "headline".

Example 4.5. Triggering an invalidation of the dependency "mydependency"

Types of dependencies

You may use any object as a dependency which is suitable as a key in a `HashMap`, typically by implementing the methods `equals(Object)` and `hashCode()` properly or by using the very same object as a dependency and for invalidation.

The class `com.coremedia.cache.Cache` already provides support for *timed dependencies* that invalidate automatically at a certain point in time. You may define these dependencies by invoking `Cache#cacheUntil(Date)` or `Cache#cacheFor(long)` during the evaluation of the cached property method. Have a look at `com.coremedia.cache.Cache`'s Javadoc for further details.

Dependency tracking and Content Beans

When using `ContentBeans` or (more generally) the *Unified API*'s content repository as the data source for your beans, you don't need to take care on the content's dependencies and invalidations: any access on the content repository's content objects causes appropriate dependencies to be tracked automatically. Further on, changes on the content objects leads to automatic invalidations. The only prerequisite (which is fulfilled by the default CAE configuration) is that the `DataViewFactory` and the *Unified API* share the same `Cache` instance.

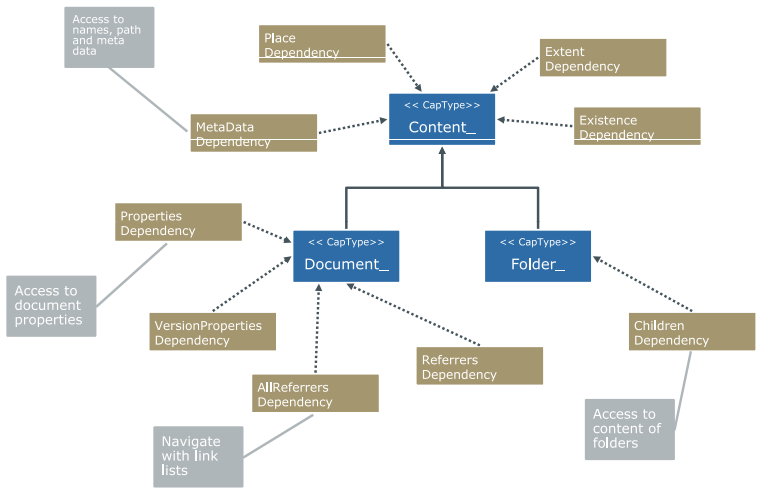


Figure 4.4. Dependencies of the Unified API cache

4.3 The CAE Web Application

The CAE web application framework provides services for building content based web applications. It is based on Spring Framework's [model-view-controller \(MVC\)](#) architecture.

4.3.1 Handling Requests

An incoming request is initially accepted by the `DispatcherServlet` and then delegated to a handler (also known as "controller") that is able to deal with the request. A handler's responsibility is to translate the request into a model and to provide a `ModelAndView` instance. This instance is passed to the view dispatching (or rendering engine respectively) which renders the model into some external representation such as HTML



Figure 4.5. Processing chain of `DispatcherServlet`, handlers and view dispatcher

There are several ways for implementing a handler, for example by implementing the interface `org.springframework.web.servlet.mvc.Controller` or by annotating a bean's method with `@RequestMapping`. Although any of these mechanisms can be used within a CAE web application, CoreMedia suggests using the `@RequestMapping` way because currently this is the most sophisticated way of writing handlers without the need to write reoccurring boilerplate code.

A simple content based handler might look as follows:

```

package com.mycompany;

import com.coremedia.objectserver.web.HandlerHelper;
import com.coremedia.objectserver.beans.ContentBean;

@RequestMapping
public class MyHandler {

    @RequestMapping(value="/content/{id}")
    public ModelAndView handleContent(
        @PathVariable("id") ContentBean bean) {

        if( bean == null ) {
            return HandlerHelper.notFound();
        }
        return HandlerHelper.createModel(bean);
    }
}
  
```

Such a handler can be registered by simply defining it as a bean:

```
<beans xmlns="http://www.springframework.org/schema/beans">
  <bean id="myHandler" class="com.mycompany.MyHandler"/>
</beans>
```

In this example, a request with an URI like `/context/servlet/content/1234` would be handled by service "myHandler" because the `@RequestMapping`'s URI pattern `/content/{id}` matches the full request URI's suffix `/content/1234`. The URI variable `{id}` is automatically bound to the method parameter `content Bean` so that the handler code can use it without parsing the request URI by itself and without converting the URI path segments into business objects manually. As a consequence, the remaining handler code is quite simple: It wraps the content bean into a `ModelAndView` and passes this to the rendering engine.

In order to get a numeric ID to be converted into a `ContentBean` automatically (and bound to the method parameter), it is necessary to register an adequate converter as follows:

```
<!-- required resources -->
<import
resource="classpath:/com/coremedia/cae/handler-services.xml"/>

<customize:append id="registerIdToContentBeanConverter"
bean="bindingConverters">
  <description>
    Registers a converter for transforming a
    numeric id ("1234", for instance) to a ContentBean
  </description>
  <set>
    <bean
class="com.coremedia.objectserver.web.binding.GenericIdToContentBeanConverter">
      <property name="contentBeanFactory" ref="contentBeanFactory"/>
      <property name="contentRepository" ref="contentRepository"/>
      <property name="dataViewFactory" ref="dataViewFactory"/>
    </bean>
  </set>
</customize:append>
```

Alternatively, the id could be passed to the handler method as an `Integer` object (for example `PathVariable("id") Integer id`) that is converted "manually" into a `ContentBean`, for example by using a `ContentBeanFactory`.

See <http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/mvc.html#mvc-ann-methods> for a list of possible argument types and different options of implementing a handler method.

Building the Model

As mentioned above, it's a handler's responsibility to provide a `ModelAndView` instance. A typical `ModelAndView` holds one or more named model beans. It also contains a view name (such as "rss") or, alternatively, a view implementation (of type `org.springframework.web.servlet.View`).

While building the model to be rendered by the CAE view dispatcher (see below) it is necessary to consider the following: At least a model bean with the name "self" needs to be added to the `ModelAndView`. This bean represents the "main" or "root" object of the model and will be used for looking up an adequate view. In addition, no View instance must be added to the `ModelAndView` because such an instance will be resolved automatically by the view resolving mechanism based on the type of the "self" bean in conjunction with the view name.

CoreMedia provides some convenience functions in `com.coremedia.objectserver.web.HandlerHelper` for building an adequate `ModelAndView`.

- `HandlerHelper.createModel(Object bean)`: Creates an instance with the given bean as the "self" object.
- `HandlerHelper.createModelWithView(Object bean, String viewName)`: Creates an instance with the given bean as the "self" object and a specific view name.

There are situations where a request must not result in a rendered page but should be answered with a special HTTP response code. E.g. a "bad request" (Status: 400) response should be returned in case that the request is malformed or a "not found" in case that the requested resource does not exist. Instead of sending such responses directly by using `HttpServletResponse`, it is also possible to return a `ModelAndView` containing a `com.coremedia.objectserver.web.HttpError` bean. The advantage of this approach is to let the view rendering decide how to handle a response like this. One way would be to use the programmed view (see below) `com.coremedia.objectserver.view.HttpErrorView` for writing the HTTP error to the response. Another approach is to render a comprehensive error page instead by using a template `com.coremedia.objectserver.view/HttpError.jsp`. The `HandlerHelper` utility provides helper methods for dealing with such situations:

- `HandlerHelper.notFound()`: Provides a `ModelAndView` that contains an `HttpError` with code 404.
- `HandlerHelper.badRequest()`: Provides a `ModelAndView` that contains an `HttpError` with code 400.

Finally, a handler might decide not to render a bean directly but send a "temporarily moved" response (Status: 302) instead. This is a typical use case when dealing with POST requests: After updating the application state, the user's web browser is redirected to a result page. This case is also supported by the `HandlerHelper`:

- `HandlerHelper.redirectTo(Object bean)`: Redirect to a page that is represented by the given bean. See [Section 4.3.2, "Building Links" \[59\]](#) for further information.

Post Processing the Model

Spring MVC includes a concept for preprocessing and post-processing a handler's execution. By implementing a `HandlerInterceptor` it is possible for example to modify the `ModelAndView` of all executed handlers.

Example:

```
import org.springframework.web.servlet.HandlerInterceptor;
public class MyInterceptor implements HandlerInterceptor {
    ...
    void postHandle(HttpServletRequest req, HttpServletResponse res,
        Object handler, ModelAndView modelAndView)
        throws Exception {
        // adds a new model object to the model and view
        modelAndView.addObject("message", "Hello World");
    }
    ...
}
```

A custom interceptor can be associated with all handlers by adding the interceptor bean to a global list bean named `handlerInterceptors` that defined by the CAE framework. A customizer might be used here, for example

```
<customize:append id="addMyInterceptor" bean="handlerInterceptors">
  <list>
    <bean class="com.mycompany.MyInterceptor"/>
  </list>
</customize:append>
```

See <http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/mvc.html#mvc-handlermapping> for more information about handler interceptors.

Best Practices

- When handler code isn't trivial, then this code should be considered to be moved to a separate service class. This makes the business code both better to test and reusable.

This is a simple example:

```
@RequestMapping(value="/service/{id}/{command}")
public ModelAndView handle(
    @PathVariable("id") Integer id,
    @PathVariable("command") String command)
{
    Object result = getService()
        .performComplexComputation(id, command);
    return HandlerHelper.createModel(result);
}
```

- It's possible to use Spring's mechanism for an annotation based automatic instantiation and autowiring of handlers and other beans. This requires the bean classes to be annotated with `@Controller`, `@Service`, `@Inject` etc. as well as using a `<context:component-scan>` declaration.

In contrast to this approach, CoreMedia suggests using the XML based way for defining and wiring beans. The reason is that in larger projects, using autowired beans may prove difficult to handle, especially when using external extensions.

When declaring beans in XML, a developer exerts much more direct control over the application context.

- Several handler methods may exist in the same class for the same URI path if they handle different request methods (such as GET or POST)
- The best practice for handling POST requests can be found here [Section "Handling POST requests" \[98\]](#)
- The best practice for handling redirects can be found here [Section "Handling redirects" \[98\]](#)

Handling Ajax Requests

Dealing with [Ajax](#) requests is quite simple when using the CAE together with Spring MVC features. The main difference of Ajax in comparison to "standard" request handling is the format of incoming and outgoing data. While standard requests typically provide an output format for end users such as HTML, Ajax requests mainly deal with machine readable formats like JSON and XML. The same applies to input formats: HTML based application have to deal with form input while Ajax request again make use of JSON/XML instead.

Spring MVC provides inbuilt converters for translating plain java beans ("POJOs") from/to XML or JSON. These converters can be easily used from within the CAE. When implementing an Ajax based handler, then no `ModelAndView` needs to be passed to the view engine but it is sufficient to provide the bean itself in conjunction with the `@ResponseBody` annotation.

Example

```
@RequestMapping(value = "/json/{id}", produces="application/json")
@ResponseBody
public MyPojo renderBeanAsJson(@PathVariable("id") String id) {
    MyPojo bean = getPojo(id);
    return bean;
}
```

In this example for an Ajax handler, a model bean is computed and simply returned as a "response body" rather than wrapping it into a `ModelAndView`. Due to the `produces="application/json"` attribute, the rendering engine knows that this bean should be automatically converted to JSON. This is internally done by recursively writing a JSON entry for all bean properties. When using `pro`

duces="text/xml" instead, then the bean will be converted to XML as long as the bean's class is annotated with `@javax.xml.bind.annotation.XmlRootElement`.

The automatic conversion is done by instances of `org.springframework.http.converter.HttpMessageConverters` that need to be registered before usage:

```
<customize:append id="registerHttpMessageConverters"
bean="httpMessageConverters">
  <list>
    <!-- converts request/response bodies from/to XML -->
    <bean class="org.springframework.http.converter.xml.
      Jaxb2RootElementHttpMessageConverter"/>
    <!-- converts request/response bodies from/to JSON -->
    <bean class="org.springframework.http.converter.json.
      MappingJacksonHttpMessageConverter"/>
  </list>
</customize:append>
```

The JSON converter `MappingJacksonHttpMessageConverter` requires the library `jackson-mapper-asl` which can be added to a Maven project like

```
<dependency>
  <groupId>org.codehaus.jackson</groupId>
  <artifactId>jackson-mapper-asl</artifactId>
</dependency>
```

Handling POST Data

Writing a handler that handles incoming data (typically sent with a HTTP POST request and formatted as JSON or XML) can be implemented nearly the same way. The only thing that has to be done is to pass an `@RequestBody` annotated parameter to the handler method like

```
@RequestMapping(value="/json/{id}", method=RequestMethod.POST,
  consumes="application/json",
  produces="application/json")
@ResponseBody
public MyResultPojo renderBeanAsJson(
    @PathVariable("id") String id,
    @RequestBody MyIncomingPojo data) {

  MyResultPojo bean = processData(id, data);
  return bean;
}
```

Building Links

Implementing and building links for Ajax handlers works the same way as for all other resources. An example link scheme implementation:

```
@Link(type=MyPojo.class, view="json", uri="/json/{id}")
public UriComponents buildJsonLink(MyPojo bean,
  UriComponentsBuilder uri) {
```

```
return uri.buildAndExpand(bean.getId());
}
```

A JavaScript snippet that can be embedded into a JSP might look like

```
<cm:link target="${myPojo}" view="json" var="pojoUrl"/>
<script type="text/javascript">
  var req = new XMLHttpRequest();
  req.open('GET', '${pojoUrl}', true);
  req.onreadystatechange = function() {
    // handle response ...
  };
  req.send();
</script>
```

Legacy Controllers

In past versions of the CoreMedia CMS, the preferred way of writing handlers was to implement an `org.springframework.web.servlet.mvc.Controller` rather than using annotations. These kinds of controllers can be still used in a CAE web application. They can be even coexist in conjunction with annotation based controllers. Keep in mind that `com.coremedia.objectserver.web.AbstractViewController` was removed in CM8.

Path Matching Details

The Spring documentation (<http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/mvc.html#mvc-ann-requestmapping>) describes in detail the request matching features provided by `@RequestMapping`. An important, if not the most important request matching criterion, is matching the request URI path against the URI templates defined by `@RequestMapping` annotations, a process performed by a `PathMatcher` implementation. There are two differences between Spring's default `AntPathMatcher` implementation and the `UriTemplatePathMatcher` provided by the CAE:

→ `@RequestMapping` supports the use of regular expressions in URI template variables, specified as `{variable:regex}`. An URL path will only be considered a match, if all the extracted URI template variable values match the corresponding regular expressions. If no regular expression is specified for a variable, the default is `"[^/]+?"`, that is, any non-empty sequence of any characters except a slash `'/'`. In other words, by default, a variable can match only one non-empty URI path segment. For instance, the URI template `{segment}` would match the URI path `/home`, but not `/news/breaking`.

If the regular expression allows for a slash character `'/'`, the CAE path matcher implementation can match multiple path segments for a single variable. This would not be possible with Spring's default path matcher. For instance, the URI template `{segments:.+}/index.html` would match the URI path `/one/two/index.html`, with variable `segment` bound to

"one/two". As a convenience and to simplify handler method implementations, an `@PathVariable` handler method argument representing a template variable can be of type `List<String>`. In this case, the variable value will be split into a list of path segments separated by slash characters `'/'`. In the previous example, the list `["one", "two"]` would be passed to the handler method.

- `UriTemplatePathMatcher` does not support Ant-style globs: `*`, `**`, and `?`. These characters should not be used in the literal part of URI templates, but only in regular expressions associated with template variables. Outside a template variable definition, they will be interpreted literally.

URI path matching behavior is not only influenced by `@RequestMapping` annotations, but also by some global Spring configuration parameters:

- `RequestMappingHandlerMapping.useTrailingSlashMatch` is `true` by default and causes any URI path with a trailing slash to be a match for a given URI template, if the template does not end with a slash, and the URI path without the slash would be a match. In effect, URIs will typically match a template, if they have a trailing slash, even if the template does not have a trailing slash. For instance, the URI template `{segment}` will match both `/home` and `/home/`.
- `RequestMappingHandlerMapping.useSuffixPatternMatch` is `true` by default and causes any URI path with an extra `.*` suffix (dot, plus some characters) to match a template, if the template does not contain any `'.'` characters. In effect, the URI path matching process will typically ignore extra path suffixes, if the template does not contain any dot characters. For instance, the URI template `{segment}/index` will match both `/home/index` and `/home/index.html`.
- `UrlPathHelper.urlDecode` is `true` by default and causes request URI paths to be percent decoded according to [RFC 3986](#), before they are matched against any URI template. This is usually the desired behavior and should not be changed as it relieves the application developer from taking into consideration percent encoding when defining URI templates. Any template variable regular expressions should therefore match the decoded form of reserved characters, if such characters are to be allowed in variable values. For instance, the URI template `/products/{name:[a-zäöü]+}` will match the request URI path `/products/m%C3%A4use` (assuming a request character encoding of UTF-8, see below). Note that the percent character `'%'` is not a valid name character as defined by the URI template. The matching process operates on the decoded URI path `/products/mäuse`.

As a consequence of this behavior, an application cannot differentiate during matching, whether the client sent a character percent encoded or not. Due to this ambiguity, an application should not generate URLs with path segments containing (percent encoded) slash characters `'/'`. Even though such

URLs are valid and can be generated, the matching process acting on the decoded path would treat such path segment as multiple segments. URLs with path segments containing encoded slash characters are considered unsound and should be avoided. Given the same example URI template as above, if the link scheme expanded the URI template with a `name` value of `"tablets/laptops"`, this would result in the valid URI path `/products/tablets%2Flaptops`. However, when dispatching a request for this path, it would be decoded and matched against URI templates as `/products/tablets/laptops`, and the template `/products/{name:[a-zäöü]+}` would not match.

- When percent decoding the request URI path, `UrlPathHelper` uses the request encoding (`HttpServletRequest#getCharacterEncoding`) or defaults to ISO-8859-1, if no request character encoding is available. Since this default character encoding is different from the `UriComponents` default encoding (UTF-8) during URL generation, it is recommended to force there request character encoding to UTF-8 by configuring an `org.springframework.web.filter.CharacterEncodingFilter` in the application's `web.xml`, with `encoding=UTF-8` and `forceEncoding=true`:

```
<filter>
  <filter-name>Character Encoding Filter</filter-name>
  <filter-class>
    org.springframework.web.filter.CharacterEncodingFilter
  </filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  <init-param>
    <param-name>forceEncoding</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
```

HTTP Method Overriding

Using the `@RequestMapping` annotation, it is straightforward to define REST APIs using a richer set of HTTP methods to specify the semantics of each operation, for example GET, POST, PUT, and DELETE.

To maintain compatibility with clients which support only GET and POST such as older browsers, Spring provides a filter `org.springframework.web.filter.HiddenHttpMethodFilter` to effectively tunnel any HTTP method through a POST request. If you intend to make use of HTTP methods other than GET and POST in your handler mappings, add this filter to your CAE web application's `web.xml` or `web-fragment.xml`:

```
<filter>
  <filter-name>HTTP Method Filter</filter-name>
  <filter-class>
```

```

    org.springframework.web.filter.HiddenHttpMethodFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>HTTP Method Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

```

With this filter in place, to signal the use of a particular HTTP method from the client, you may send a POST request with an additional parameter indicating the HTTP method to use. By default, the filter expects a parameter named `_method`. Note that only POST requests will be handled by this filter.

```

<form action="{url}" method="POST">
  <input type="hidden" name="_method" value="PUT"/>
  ...
</form>

```

Of course, clients supporting the HTTP method PUT, may send a PUT request directly, without adding the `_method` parameter.

4.3.2 Building Links

It has been already stated above that handlers are responsible for providing a model object named "self" that represents a page (or another resource). This page might be rendered as HTML or another output format. A typical page consists of links pointing to other pages that are handled by a handler again when requested by the client.

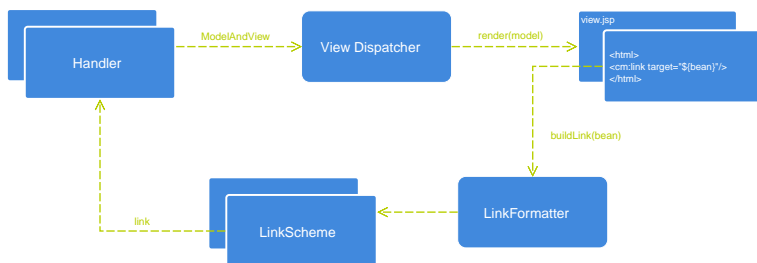


Figure 4.6. Processing chain of handlers and link schemes

In a CAE, links can be represented as model objects that can be translated into a URI by view technology specific mechanisms such as a JSP tag (`<cm:link target='${linkRepresentation}'/>`) or a Freemarker function (`<#assign imageSrc=cm.getLink(self.thumbnail)!'"/>`). Typically, the bean that is used for building the link is the same that is provided by the handler as a model. In the CAE there is a concept called "link scheme" that is used for translating an object (with an optional view name) into a URI string. A link scheme is logically bound to

a handler that is able to translate the URI back to a model. Link schemes are automatically collected by the CAE and exposed to the view technology specific link building facilities mentioned above.

Links created by a "@Link" link scheme are always relative to the servlet path. For adding servlet and context path, or making links absolute, instances of LinkPostProcessor should be used. (see below)

Example

```
package com.mycompany;

...
import java.net.URI;
import com.coremedia.objectserver.beans.ContentBean;
import org.springframework.web.util.UriComponents;
import org.springframework.web.util.UriComponentsBuilder;
import com.coremedia.cap.common.IdHelper;

@Link
public class MyLinkScheme {

    @Link(type = ContentBean.class, uri="/content/{id}")
    public UriComponents buildLink(UriComponentsBuilder uriTemplate,
        ContentBean bean) {

        Integer id = IdHelper.parseContentId(bean.getContent().getId());

        return uriTemplate.buildAndExpand(id);
    }
}
```

Example 4.6. A link scheme

```
<beans xmlns="http://www.springframework.org/schema/beans">
  <import resource="classpath:/com/coremedia/cae/link-services.xml"/>
  <bean id="myLinkScheme" class="com.mycompany.MyLinkScheme"/>
</beans>
```

Example 4.7. Defining a link scheme

This example demonstrates how to build links that point to the above mentioned handler. This link scheme is invoked for beans of type `ContentBean` only and uses the same URI pattern `/content/{id}` that is also in use by the handler. The link is generated by simply applying the value of the path variable `id` to the URI template.

Lookup

By annotating a bean's method with the `@Link` annotation, this method is turned into a link scheme. Typically, an application consists of several link schemes for different aspects as every handler is likely to have one or more link schemes as a counterpart. When a link generation is requested, by running, for example,

```
<cm:link target="{bean}" view="rss">
  <cm:param name="maxItems" value="10"/>
</cm:link>
```

from within a JSP template, the CAE needs to find a link scheme that matches best. This decision is made based on the information that is provided by the link generation invocation: The given target bean, the view name, any additional link parameters.

The parameters of the `@Link` annotation are used to determine methods that are link handler candidates. The parameters are turned into predicates which are evaluated against the arguments passed to the link generation request. In the following example, the annotated method is a candidate for beans of type `ContentBean` with views "rss" or "xml" and link parameter "maxItems":

```
@Link(type=ContentBean.class,
      view={"rss", "xml"},
      parameter="maxItems",
      order=10)
```

The predicates are evaluated in the following order to determine the ordering of the link handler candidates.

- **type**: The java class(es), that the given bean needs to match (either by class equality or by class super type relationship). Several types might be listed here but only a single type needs to match. If no type is specified, then the bean method parameter determines the type. Hence a link handler method with a parameter of type `ContentBean` would match every instance of `ContentBean` if no subclass of `ContentBean` is given as `type` parameter. A link handler method with the same parameters but a more specific type parameter in its `@Link` annotation would have a higher precedence, though.
- **view**: A list of supported view names. If this predicate is specified, the given view name needs to match one of the listed names. Omitting this predicates matches all view names. A view name "DEFAULT" matches the default ("null") view.
- **parameter**: A list of link parameters that need to be specified. In contrast to other predicates, all parameter predicates need to match here.
- **order**: A numeric order value to distinguish the precedence in case if more than one scheme matches all the criteria given above. A higher order value correlates here with a lower precedence. The default value is set to `Integer.MAX_VALUE`.

There might be situations where more than one link scheme matches the current link generation invocation. In this case, all matching schemes are invoked until one scheme returns a non-null result. The more specific a link scheme is, the earlier it is invoked.

Writing Link Schemes

The link scheme's method signature might contain several parameters (such as `bean`, `view`, `HttpServletRequest`, ...) that will be automatically bound by the

CAE framework on invocation. Furthermore, several classes are supported for the scheme's return type, for example `org.springframework.web.util.UriComponents` or even a `Map<String, Object>` that holds the URI variables only. See the Javadoc of the annotation `com.coremedia.objectserver.web.links.Link` for more details.

As a consequence, a link scheme can be implemented in several ways, for instance:

```
@Link(type = ContentBean.class, uri="/content/{id}")
public UriComponents buildLink(UriComponentsBuilder uriTemplate,
                               ContentBean bean) {

    Integer id = IdHelper.parseContentId(bean.getContent().getId());
    return uriTemplate.buildAndExpand(id);
}
```

or

```
@Link(type = ContentBean.class, uri="/content/{id}")
public Map<String, Object> buildLink(ContentBean bean) {

    Integer id = IdHelper.parseContentId(bean.getContent().getId());
    return Collections.singletonMap("id", id);
}
```

or

```
@Link(type = ContentBean.class)
public UriComponentsBuilder buildLink(ContentBean bean) {

    Integer id = IdHelper.parseContentId(bean.getContent().getId());
    return UriComponentsBuilder.newInstance()
        .pathSegment("content")
        .pathSegment(id.toString());
}
```

CoreMedia suggests using `org.springframework.web.util.UriComponentsBuilder` for building links since this utility provides convenience functions for manipulating URI parts as well as functions for substituting URI variables (such as `{id}`) by concrete values. In addition, an URI will be encoded (for example `/öffnungszeiten` to `/%C3%B6ffnungszeiten`) properly by using the `UriComponents#encode()` function. Moreover, CoreMedia suggests to return the resulting link as an `UriComponents`, `UriComponentsBuilder` or `Map<String, Object>` object. Post-processing (see below) of such values is much more efficient than for objects of type `String` or `URI`. As a side effect, it is not necessary to perform the encoding manually, because this is done by the framework.

Post Processing Links

Similar to handler interceptors, it is also possible to post process generated links. A common use case is to prepend a prefix (such as context and servlet path) to the URI when the link schemes are used to generate the link suffixes only.

```

package com.mycompany;
import com.coremedia.objectserver.view.ViewUtils;
import org.springframework.web.util.UriComponentsBuilder;
import org.springframework.web.util.UriComponents;
import com.coremedia.objectserver.web.links.UriComponentsHelper;
...
@LinkPostProcessor
public class MyLinkPostProcessor {
    @LinkPostProcessor
    public UriComponentsBuilder prependPrefix(UriComponents
originalUri,
                                           HttpServletRequest
request) {

        String baseUri = ViewUtils.getBaseUri(request);
        return UriComponentsHelper.prependPath(baseUri, originalUri);
    }
}

```

```

<beans xmlns="http://www.springframework.org/schema/beans">
  <import resource="classpath:/com/coremedia/cae/link-services.xml"/>

  <bean id="myLinkPostProcessor"
        class="com.mycompany.MyLinkPostProcessor"/>
</beans>

```

This example demonstrates how the base URI (context path and the servlet path) is prepended to an URI that has been built by an annotated link scheme. Writing a post processor is quite similar to writing a link scheme. The main difference is that the original link needs to be passed to the post processor method as a parameter of type `UriComponents` or `UriComponentsBuilder`. All other parameters bindings as well as the possible return types are the same. Just like the `@Link` annotation, the `@LinkPostProcessor` supports an optional `type` element which restricts the post-processor to links for the particular bean types.

Best Practices

It's a good idea to put handler, corresponding link implementations and post-processors into the same class since these are strongly related. Also, the URI pattern used in `@RequestMapping` and in `@Link` can be shared by a constant like

```

private static final String URI = "/content/{0}";

@RequestMapping(value=URI, ...);
public ModelAndView handle(...) { ... }

@Link(uri=URI, type=MyBean.class, ...)
public UriComponents buildLink(MyBean myBean, ...) {...}

@LinkPostProcessor(type=MyBean.class, ...)
public UriComponents prefixLink(UriComponents originalUri, ...)
{...}

```

The `PostProcessorPrecedences` class provides some constants to control the order of post-processors. All the Blueprint's default post-processors are ordered

by these constants. You can use the constants for additional independent post-processors or use other values in order to apply subsequent post-processors in between.

Legacy Link Schemes

In past versions of the CoreMedia CMS, the preferred way for writing handlers was to implement a `LinkScheme` interface rather than using the `@Link` annotation. This kind of link scheme can still be used in a CAE web application. It can even coexist in conjunction with annotation based link schemes. Keep in mind that `com.coremedia.objectserver.web.links.AbstractLinkScheme` was removed in CM8.

External Link Placeholder

It is possible to use placeholder tokens in External Link content and external links within a Rich Text editor in *Studio*. For example: an external link that contains a hostname for the Perfect Chef shop which is linked from the corporate site can be written as follows:

```
http://{perfectchef.host}/blueprint/servlet/perfectchef
```

The tokens are enclosed with curly braces. The `TokenReplacingLinkTransformer` class acts as a `LinkTransformer` class to replace these tokens in URLs. A list of token resolvers can be registered and each resolver is called per token. The first token Resolver that returns a replacement (`value != null`) wins. After that no other token resolver will be asked anymore for the current token. The following token resolvers are preconfigured in the given order:

→ `SettingsTokenResolver`

This token resolver is used to resolve tokens with values coming from setting documents.

For example: the following URL

```
http://localhost?connectionId={livecontext.connectionId}
```

will be transformed to the URL

```
http://localhost?connectionId=wcs1
```

The `SettingsTokenResolver` resolves the token of the URL to the value "wcs1" because the property `livecontext.connectionId` is defined as `wcs1` in a Setting document, which is accessible via the settings service starting from the current External Link content.

→ `StoreContextTokenResolver`

This token resolver is used to resolve tokens with values coming from the LiveContext Store Context.

For example: the following URL

```
http://localhost/wcs/resources/store/{storeId}
```

in the PerfectChef page will be transformed to the URL

```
http://localhost/wcs/resources/store/10302
```

The `StoreContextTokenResolver` is only available if the livecontext extension is active.

→ `SpringPropertyTokenResolver`

This token resolver is used to resolve tokens with values coming from Spring properties.

The placeholder and their values can be defined by using the prefix "urlToken." in a properties file as follows:

```
urlToken.perfectchef.host=${blueprint.host.helios}
urlToken.aurora.host=${livecontext.apache.wcs.host}
urlToken.aurorab2b.host=${livecontext.apache.wcs.host}
```

Within the URL attribute of an External Link content the token will be defined without the "urlToken." prefix. It will be replaced with the defined value. For example: the following URL

```
//{perfectchef.host}/blueprint/servlet/perfectchef
```

will be transformed to the PerfectChef homepage.

4.3.3 Views

In a Model-View-Controller (MVC) architecture, it is the responsibility of views to present the model to the end-user. In the CAE context, content beans are the models and views are typically implemented in one of the supported templating languages, *JavaServer Pages (JSP)* or *FreeMarker*. Views may also be implemented in Java code, but programmatic views are usually reserved for special cases, such as XML output. It is important to note, that central view concepts are the same, regardless of how a particular view is implemented: view dispatching, accessing the model, including other views, and linking back to controllers.

This chapter will demonstrate how to apply these concepts in both of the supported templating languages. It is not a tutorial or complete reference of either *JavaServer Pages* or *FreeMarker*.

View Repository

The CAE uses a concept called ViewRepository to organize its views. A ViewRepository can be understood as a store that contains JSP or FreeMarker templates for beans of certain types.

Template Views

The default implementation `ResourceViewRepository` looks up templates for a given type at a location `<package>/<class>.<fileextension>` below a configured base location such as `/WEB-INF/templates`. For instance, a JSP template for a bean of type `com.company.Article` is looked up at a location `/WEB-INF/templates/com.company/Article.jsp`. A template for the same bean but with a specific view name `asTeaser` is looked up at location `/WEB-INF/templates/com.company/Article.asTeaser.jsp`.

Note that the type's package name isn't mapped to a template location containing nested directories (like `com/company/`) but to a single directory (like `com.company/`).



The file extension must match a supported view engine, that is, `.jsp` for a JSP template or `.ftl` for a FreeMarker template.

Programmed Views

Besides templates, a resource view repository might also contain so called "programmed views". These are view instances implemented in Java rather than in a template language. To write a programmed view, implement `ServletView` or `TextView`. If a programmed view is added to the predefined Map "programmedViews", it will be used for rendering.

For example, this is a simplified version of a programmed view implementation that renders `com.coremedia.xml.Markup` as plain text:

```
/**
 * Programmed view that renders a given Markup as plain text
 */
public class PlainView implements TextView {

    @Override
    public void render(Object bean, String view, Writer writer,
        HttpServletRequest request, HttpServletResponse response) {

        Markup markup = (Markup) bean;
        // create serializer instance for scripts
        PlainTextSerializer handler = new PlainTextSerializer(writer);

        // transform and flush markup
        markup.writeOn(handler);
    }
}
```

This is how a programmed view is added to view repositories with a customizer:

```
<!-- programmed view to render plain markup -->
<bean id="plainView" class="com.company.PlainView"/>

<!-- add programmed views to predefined map "programmedViews" -->
<customize:append id="customProgrammedViews" bean="programmedViews">

  <map>
    <entry key="com.coremedia.xml.Markup#plain"
value-ref="plainView"/>
  </map>
</customize:append>
```

View Lookup

Looking up a view for a given bean is performed by a service called [ViewDispatcher](#). It computes the bean's type hierarchy by taking its super types, interfaces, and even [HasCustomType](#) implementations into account. Then it asks the underlying view repositories to provide a template (or view, respectively) by passing the bean's type. If a view repository cannot provide such a view, then it will be asked iteratively for the bean's super type until a matching view can be provided.

Example:

Assume a class `com.company.Base` that is extended by `com.company.Article`. If during a view lookup for a bean of type `com.company.Article` there is no template `com.company/Article.jsp` available, but a template `com.com pany/Base.jsp` can be found, then the latter template is used.

The view dispatcher is invoked whenever a bean is rendered. This happens at least once per request. When a controller has returned with a `ModelAndView` instance, then the bean `self` is extracted and used to find the root view for the request. While executing a template, it might happen that a child bean is rendered by another template. When passing this bean to `<cm:include>` another view lookup and rendering is triggered.

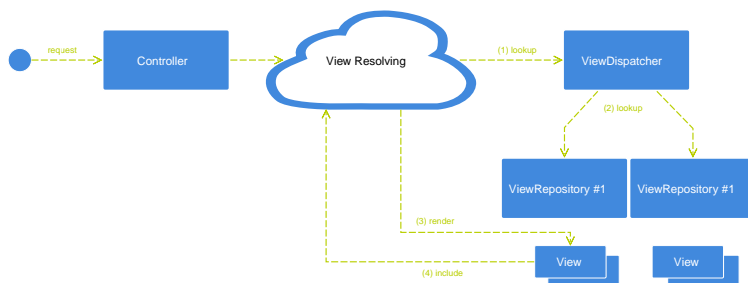


Figure 4.7. View lookup sequence



Although it is possible for the CAE to look up all types, it is encouraged to write templates for interfaces only. While View lookups are cached, it may not always be desirable to cache lookups indefinitely, also caches are not filled every time a CAE is started. Going through the hierarchy of all types for every view lookup can be very costly, and a production CAE easily reaches a 6-digit number of View lookups (100.000+) until all views are cached.

To limit CAE lookups to certain types, set the Spring property `filter.viewlookup.by.predicate` to true. Types ending on "Impl", "Base" and a few technical types will be removed from the type hierarchy before doing the View lookup. This reduces the number of lookups dramatically (up to 80%).

If you cannot adhere to the CoreMedia naming conventions and need a view lookup, for example for a class that ends on "Impl", you can add exceptions to this rule to the `viewlookupPredicate` property `includes`.

This is an example on how to add class names that should be included in the View lookup in addition to all interfaces.

```
<customize:append id="addMyViewlookupIncludes"
bean="viewlookupPredicate" property="includes"
enabled="${filter.viewlookup.by.predicate}">
  <description>
    Override the predicate's exclusion patterns for these classes.
  </description>
  <list>
    <value>my.package.MyViewRelevantBeanImpl</value>
  </list>
</customize:append>
```

Using Multiple View Repositories

In a smaller project it might be sufficient to use a single view repository only.

When hosting several sites with different template sets in a single CAE, multiple view repositories may be used. The CAE provides a mechanism for choosing a set of view repositories dynamically per request.

This mechanism is separated into two services that are implementations of [ViewRepositoryNameProvider](#) and [ViewRepositoryProvider](#) respectively.

ViewRepositoryNameProvider

The `ViewRepositoryNameProvider` is responsible for providing the names of the view repositories to be used for resolving templates for the current request. For instance, if a page is requested that is located in a "sports" subsite within a larger site, a list `[sports, site]` might be returned where "site" refers to a common template sets that is used when the more special set "sports" does not provide a matching template. If another request is sent for a "politics" page, then a list

[politics,site] might be returned so that the output is rendered differently due to the use of different templates.

A default implementation [StaticViewRepositoryNameProvider](#) returns a list of predefined view repository names. Another default implementation [CompoundViewRepositoryNameProvider](#) returns the view repository names from several view repository name providers. Applications that require more flexibility must implement the interface `ViewRepositoryNameProvider` to return a project specific list of view repository names.

ViewRepositoryProvider

The `ViewRepositoryProvider` is responsible for providing a `ViewRepository` instance for a given name. A default implementation [TemplateViewRepositoryProvider](#) is included. It inserts the repository name into a configured base path format pattern, for example, a name "sports" with a format `/WEB-INF/templates/%s` provides a `ViewRepository` instance with a base path `/WEB-INF/templates/sports`.

The following example configuration registers a custom `ViewRepositoryNameProvider` and a `TemplateViewRepositoryProvider` to locate view repositories using the pattern `/WEB-INF/templates/sites/<viewRepositoryName>`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:customize="...">

  <!--
    Instance of the project specific viewRepositoryNameProvider
  -->
  <bean id="customViewRepositoryNameProvider"
        class="com.company.CustomViewRepositoryNameProvider">
    ...
  </bean>

  <!--
    Register the view repository name provider
  -->
  <customize:append id="addCustomViewRepositoryNameProvider"
                   bean="viewRepositoryNameProviders">
    <list>
      <ref bean="customViewRepositoryNameProvider"/>
    </list>
  </customize:append>

  <!--
    Create an instance of TemplateViewRepositoryProvider
  -->
  <bean id="customViewRepositoryProvider"
        class="com.coremedia.objectserver.view.resolver.TemplateViewRepositoryProvider">
    <property name="templatesLocationFormat"
              value="/WEB-INF/templates/sites/%s"/>
  <!-- configure predefined beans -->
  <property name="viewDecorators" ref="viewDecorators"/>
  <property name="viewEngines" ref="viewEngines"/>
  <property name="loader" ref="templatesResourceLoader"/>
  <property name="programmedViews" ref="programmedViews"/>
  </bean>
</beans>
```

```

</bean>
<!--
  Register the view repository provider
-->
<customize:append id="addCustomViewRepositoryProvider"
  bean="viewRepositoryProviders">
  <list>
    <ref bean="customViewRepositoryProvider"/>
  </list>
</customize:append>
</beans>

```

Loading Templates from the Content Repository

Templates can be loaded by the `TemplateViewRepositoryProvider` from a blob property in the content repository instead of a folder in the file system. This may be useful if for example a small campaign site should be launched or a template needs patching but there isn't time to redeploy all CAEs.

This feature only works with FreeMarker templates.



In order to store templates in the content repository, sets of templates must be put into a JAR container. The templates in the JAR must have the same directory structure as if the templates were located in the file system, for example `templates.jar/com.company/Base.ftl` but may be stored in an arbitrary subfolder if the path is specified in the pattern as described below. The JAR can then be uploaded to an arbitrary content type with a blob property.

A specially formatted value for the properties `templateLocations` or `templateLocationPatterns` must be used. The value must start with the prefix `"jar:id:contentproperty:"`, add the absolute path to the content containing the templates JAR (ending with the name of the property), and add `"!/"` to separate the content path from the path within the JAR.

For instance, to use a JAR in the "data" blob property of content `"/Sites/templates/<repository name>"` as the base for a view repository, set the following format string: `jar:id:contentproperty:/Sites/templates/%s/data!/"`.

```

<customize:replace id="customizeTemplateLocationPatterns"
  bean="templateLocationPatterns">
  <list>
    <value>jar:id:contentproperty:/path/to/document/%s/blobPropertyName!/</value>
  </list>
</customize:replace>

```



It is recommended to use a dedicated document type for storing the template JAR. The document type(s) may be added to the list of `viewLookupTypeTriggers` provided in `classpath:/com/coremedia/cae/view-services.xml`. The CAE will automatically invalidate internal view caches when a document of one of the types is added, modified, or a property is changed. (On live servers, publication and deletion of such a document leads to the cache invalidation)

Loading Templates from an Arbitrary Directory

When working on a new version of templates that have not yet been uploaded to the content repository, the templates location for this view repository can be overwritten in a local CAE configuration using a customizer:

```
<customize:replace id="overrideTemplateLocation"
                  bean="templateLocations">
  <map>
    <!-- the key is the logical name of the view repository -->
    <entry key="customViewRepository"
          value="file:///C:/path/to/template-module/src/main/webapp/WEB-INF/templates"/>
  </map>
</customize:replace>
```

Loading Templates from a JAR in Classpath

When using *Servlet 3.0*, resources may be stored in JARs, and so can Templates. In order for that to work, templates must be stored under the path `/META-INF/resources/WEB-INF/templates`. The application container will automatically resolve that path as if it were in the file system.

The same JAR may be used inside a web application and from the content repository if the configured path matches the path inside the JAR. Following the example above, the format would have to be: `jar:id:contentproperty:/Sites/templates/%s/data!/META-INF/resources/WEB-INF/templates/`

Debugging

If you observe an error on a page, it is sometimes not obvious which view has rendered the particular fragment of the page. In order to ease debugging, you can set the flag `view.debug.enabled=true` in the `application.properties` file of your preview web application. If this flag is set, the CAE renders comments with meta information about the content bean and the view before and behind each fragment of a page. The output looks like this:

```
<li class="titlestory first" >
<!--
BEGIN
```

```

com.coremedia.blueprint.cae.contentbeans.CMArticleImpl$$[id=454]
asTitleStory webapp resource

view[/WEB-INF/templates/sites/media/com.coremedia.blueprint.common.contentbeans/CMTeasable.asTitleStory.jsp]
-->
<a href="/blueprint/servlet/media/diving/454">
<div class="img-box">
<!--
  BEGIN
  com.coremedia.blueprint.cae.contentbeans.CMPictureImpl$$[id=446]
  null webapp resource

view[/WEB-INF/templates/sites/media/com.corcom.coremedia.blueprint.common.contentbeans.Picture.jsp]

-->
<script type="text/javascript">
  com_coremedia_createContextInfo('PBE4',446,null,'mouse',true,null);
</script>
<span data-image-rwd="image-rwd" class="image-rwd">
  <span data-max-width="320" data-src="..."> </span>
</span>
<!--
  END
  com.coremedia.blueprint.cae.contentbeans.CMPictureImpl$$[id=446]
  null webapp resource

view[/WEB-INF/templates/sites/media/com.coremedia.com.coremedia.blueprint.common.contentbeans]

-->
</div>
<h4>Scuba diving the underwater adventure</h4>
</a>
<!--
  END
  com.coremedia.blueprint.cae.contentbeans.CMArticleImpl$$[id=454]
  asTitleStory webapp resource

view[/WEB-INF/templates/sites/media/com.coremedia.blueprint.common.contentbeans/CMTeasable.asTitleStory.jsp]
-->
</li>

```

View Decorators

With a `ViewDecorator` you can wrap your Views in order to modify the behavior. `ViewDecorators` are useful for conditional aspects.

In the last section you learned how to enhance the generated HTML pages with debugging comments by simply setting a flag. Implementing these comments directly in the templates would be hard to maintain, hard to understand and distract from the actual functionality of the template. A `ViewDecorator` solves the problem much more effectively. It can be switched on and off in the preview and live CAE, respectively, and it has no impact on template development.

Configuration

`ViewDecorators` are declared as Spring beans and appended to the `viewDecorators` list in the CAE's `view-services`. E.g. the configuration for the `DebugViewDecorator` looks like this:

```

<bean id="debugDecorator"
class="com.coremedia.objectserver.view.DebugViewDecorator">
  <description>
    Decorates view fragments with debug comments
  </description>
</bean>

```



```

    </description>
  </bean>

  <customize:append id="addCAEDebugDecorator" bean="viewDecorators"
    enabled="${view.debug.enabled}">
    <description>
      Registers debug decorator
    </description>
    <list>
      <ref bean="debugDecorator"/>
    </list>
  </customize:append>

```

The activation of a `ViewDecorator` is controlled by the `enabled` flag of the customizer. For the `DebugViewDecorator` the `view.debug.enabled` flag is by default set to `true` in the preview web application and to `false` in the live web application.

Implementation

The actual `ViewDecorator` interface consists of a single method

```
View decorate(View view)
```

While this interface is very flexible, it would be cumbersome to implement a decorating view from scratch. You would have to deal with `ServletView`, `TextView` and `XmlView` arguments and preserve the particular types for your decorating result view. In order to simplify this, the CAE provides the abstract `ViewDecoratorBase` which handles these type issues. If you extend the `ViewDecoratorBase`, you only have to implement `getDecorator` and return a custom `Decorator`. A `Decorator` consists of three `decorate` methods for the `View` interfaces `ServletView`, `TextView` and `XmlView`. The default implementations simply delegate to the render methods of the original views. Custom overriding can enhance or replace this behavior. For example, a `decorate` method for `ServletViews` might look like this:

```

@Override
public void decorate(ServletView view, Object self, String viewName,
  HttpServletRequest request, HttpServletResponse response) {
  try {
    Writer out = response.getWriter();
    out.write("<!-- Decoration before rendering -->");
    view.render(self, viewName, request, response);
    out.write("<!-- Decoration after rendering -->");
  } catch (IOException e) {
    throw new RuntimeException("Cannot decorate", e);
  }
}

```

View Hooks

View Hooks provide a means to define extension points in *JSP* and *FreeMarker* templates. Project Extensions can make use of these extension points to add their

own functionality at the respective locations in the resulting website without having to change the core templates.

In the past you either directly implemented the functions in your content beans and templates or you implemented a plugin by means of an [Section 5.2, “Aspects” \[125\]](#) to achieve this. Both solutions are feasible however content beans and Aspects should only accomplish basic tasks based on the content defined by the editor and View Hooks are more loosely coupled and as such improve your project's code quality.

Required Configuration

View Hooks are not enabled by default. In order to use them in your templates you have to append the Spring bean `viewHookEventView` to the list of existing [Section “View Repository” \[66\]](#).

```
<customize:append id="customProgrammedViews" bean="programmedViews">
  <map>
    <entry key="com.coremedia.objectserver.view.events.ViewHookEvent"
           value-ref="viewHookEventView"/>
  </map>
</customize:append>
```

Instead of using a customizer you can also add the `viewHookEventView` to the existing map of `programmedViews`.

Example Implementation

Assuming there is a content bean `CMArticle` which represents an editorial article and a corresponding template called `CMArticle.detail.jsp`. The template defines an extension point with the id `articleEnd`.

```
<div class="detailView">
  <h1><c:out value="\${self.title}"/></h1>
  <cm:include self="\${self.text}" view="detailText"/>
  <cm:hook id="articleEnd" self="\${self}"/>
</div>
```

A project extension now wants to add a list of user generated comments at the end of the article. Instead of changing the `CMArticle.detail.jsp` in the core modules directly, you only need to add an implementation of the [com.coremedia.objectserver.view.events.ViewHookEventListener](#) to the Spring application context.

An implementation of this interface could look as follows:

```
@Named
public class CommentsViewHookEventListener implements
ViewHookEventListener<CMArticle> {
```

```

@Inject
private CommentsService commentsService;

@Override
public RenderNode onViewHook(ViewHookEvent<CMArticle> event) {
    if("articleEnd".equals(event.getId())) {
        CommentsResult commentsResult =
commentsService.getCommentsResult(event.getBean());
        return new RenderNode(commentsResult);
    }

    return null;
}

@Override
public int getOrder() {
    return DEFAULT_ORDER;
}
}

```

The resulting `com.coremedia.objectserver.view.RenderNode` contains the object and the view name that will finally be passed to the [Section “View Lookup” \[67\]](#). The view lookup is responsible for identifying and rendering the corresponding template or programmed view. Returning `null` tells the application to skip this listener.

4.3.4 Writing Templates

A template accesses variables in its current environment that have been provided by the controller. In a *CoreMedia Content Application Engine* template, the property `self` has a special meaning: it denotes the target object on which the template was invoked. It is the equivalent of the `this` object reference in Java methods. A simple FreeMarker template to display the `title` property of a target object of type `com.company.Article` and set the `Content-Type` HTTP response header looks as follows:

```

<cm.responseHeader name="Content-Type" value="text/html;
charset=UTF-8"/>
<#-- @ftlvariable name="self" type="com.company.Article" -->
${self.title}

```

While the `@ftlvariable` comment is not necessary, it serves as a hint for the IntelliJ IDEA development environment to support code completion for the `self` variable.

Template Output Escaping for HTML

To prevent output that allows cross-site scripting (XSS) attacks, the CAE switches on HTML escaping for all FreeMarker templates by default. More precisely, each FreeMarker template is automatically wrapped in an `#escape` directive:

```
<#escape x as x?html>
...
</#escape>
```

This `#escape` directive leads to all FreeMarker "interpolations" (`{...}` expressions) being HTML-escaped when written to the output stream. All literal output of the template is of course not escaped. See FreeMarker online documentation for details.

In special cases, it might be necessary to disable escaping. For this purpose, FreeMarker provides the directive `#noescape`. However, this directive is only allowed nested inside an `#escape` directive, so because the outer `#escape` directive is implicit, an IDE could regard this an error.

To always have valid templates and to also be able to change the escaping type to something other than HTML, the CAE only wraps templates that do not contain any `#escape` directives themselves. Thus, to switch off escaping, you have to make the outer escaping explicit:

```
<#escape x as x?html>
...
  <#noescape>...</#noescape>
...
</#escape>
```

Note that disabling HTML escaping can lead to cross-site scripting (XSS) vulnerabilities if a templates outputs unchecked data like user input that may contain scripts.



The same applies to the use case of changing the escaping type to something other than HTML:

```
<#escape x as x?xml>
...
</#escape>
```

Like said above, specifying any `#escape` directive in your template completely disables any automatic escaping normally added by the CAE. Otherwise, specifying custom escaping would have lead to double escaping, which is not desired.

Template Inclusion

Other templates can be included via FreeMarker's `<#include>` directive. However, in this case the view dispatcher is not involved in determining the included file. In order to involve the view dispatcher, you need to use the `include` macro from the *Content Application Engine's* FreeMarker library `cae.ftl`. This library is auto-imported under the namespace `cm`. In FreeMarker, custom macros are invoked using `<@namespace.macro>`. The macro `@cm.include` requires an attribute

`self` to determine the target object for the view. The following code will find the appropriate template named "teaser" for `anObject` and include its output into the current page. Inside that template, `self` is temporarily bound to `anObject`:

```
<@cm.include self=anObject view="teaser"/>
```

Assuming that `anObject` is of type `Article`, the template `Article.teaser.ftl` will be included. The view attribute is optional; the default template (in this example, `Article.ftl`) will be chosen in case it is omitted. When no template for the view name "teaser" is found, the search will end with a failure - the default template is **not** used as a fallback! Also, the include will fail if `anObject` is `null` (unless you specify a default value of `cm.UNDEFINED` for `self`, see reference).

A template including the *teaser* views of all objects in its *articles* property would look as follows. Within each teaser template, `self` will be bound to the respective article object. Note the use of FreeMarker's built-in `#list` directive:

```
<#list self.articles as article>
  <@cm.include self=article view="teaser"/>
</#list>
```

When looking for the appropriate template, the *Content Application Engine* performs the same steps as in an object-oriented language. If no template is defined for a target bean type, it will be inherited from its super type: the CAE will look for the template upwards in the inheritance hierarchy. It also considers interfaces, so you can register templates for interfaces, too.

Rendering Markup

Markup properties are also rendered by including them. Assuming `self` has a method `getText` returning a `com.coremedia.xml.Markup`, this template snippet will render the text value using the default markup view.

```
<@cm.include self=self.text/>
```

The *CoreMedia* CAE defines a default view for objects of type `com.coremedia.xml.Markup` that converts CoreMedia richtext to XHTML. See [Section "Rendering Markup" \[78\]](#) for details.

Template Parameters

CAE includes allow handing over parameters from the calling template to the included one. This is implemented by temporarily setting a request scope attribute and resetting it to its old value after the included fragment returns.

In a FreeMarker template, the `include` macro and the `getLink` function support such parameters by using a hash-valued parameter named `params`. Macro `include` also allows adding parameters as additional attribute-value pairs to the macro itself. These two includes are equivalent:

```
<@cm.include self=article view="teaser"
    params={ "images": false }/>
<@cm.include self=article view="teaser" images=false/>
```

Within the "teaser" template, the variable `images` will be set to `false` and will revert to its original value (if any) afterwards.

Linking

Like `include`, linking also works with objects. To compute a URL to an object and a view, you can use the CAE FreeMarker library function `getLink()`:

```
<a href="{cm.getLink(article, "teaser")}">more</a>
```

This function consults the [LinkFormatter](#) strategy to compute a URL and hands in its first parameter as the target object and its second parameter as the (optional) view identifier. The link formatter strategy requires a link scheme that is able to handle the class of the object. All generated content beans implement the [Content-Bean](#) interface for which a link scheme exists; so there is no need to implement another one. It is necessary for beans that originate from other sources.

Using the function in an expression (FreeMarker: "interpolation"), the formatted URL is written directly to the page. If the URL is used several times within the template or if you feel that the actual template code looks cleaner when separating URL computation and usage, use FreeMarker's `#assign` directive to assign the resulting URL to a variable:

```
<#assign teaserLink><@cae.link target=article
view="teaser"/></#assign>
<a href="{teaserLink}">more</a>
```

You can hand over parameters to the [LinkFormatter](#) as an optional third parameter of the `getLink()` function, specified as a FreeMarker hash of name-value pairs. If you do not want to specify a view, you can also hand over parameters as the second parameter. Do not forget to quote the keys and *not* quote the values (unless they are strings, of course).

Rendering Markup

Render objects of type [com.coremedia.xml.Markup](#) by including them from a FreeMarker template using:

```
<@cm.include self=self.text/>
```

This uses the class [XmlMarkupView](#) as a default view, which converts richtext to XHTML applying the following transformations:

- internal links are converted to URIs pointing back into the *CoreMedia CAE*
- links (`href` attributes in the `xlink` namespace) without protocol and server are URL encoded
- anchor and image elements with `xlink href` attributes are converted to XHTML a `href` and `img src`.
- the CoreMedia richtext namespace is dropped from the elements

If you want to use your own transformations you have to proceed as follows:

1. Define your own view, *plain* for example, using a Customizer:

```
<customize:append id="addMarkupView"
                  bean="programmedViews">
  <map>
    <entry key="com.coremedia.xml.Markup#plain">
      <bean ..../>
    </entry>
  </map>
</customize:append>
```

2. Use [XmlMarkupView](#) as the implementation of the view, but apply a custom filter factory which creates a SAX filter chain per output. Proceed as follows:

- Let your filter factory extend [RichtextToHtmlFilterFactory](#).
- Overwrite `#createFilters` and append your own transformations before `super.createFilters`.

```
public List createFilters(HttpServletRequest req,
                        HttpServletResponse res, Markup markup, String view) {

  List result = new ArrayList();
  result.add(new MyFilterForRichtext());
  result.addAll(super.createFilters(req, res, markup, view));
}
```

3. Configure your filter factory in `cae-views.xml` as follows:

```
<entry key="com.coremedia.xml.Markup#plain">
  <bean class="com.coremedia.objectserver.web.XmlMarkupView">
    <property name="xmlFilterFactory">
      <bean class="com.coremedia.objectserver.web.
        MyRichtextToHtmlFilterFactory">
        <property name="idProvider" ref="idProvider"/>
        <property name="linkFormatter" ref="linkFormatter"/>
      </bean>
    </property>
  </bean>
</entry>
```

Advanced Patterns for FreeMarker Templates

Working with Maps in FreeMarker Templates

FreeMarker supports variables of type *hash*, which are unordered mappings of *strings* to other models, and provide the built-in `?keys` and `?values` to expose the key and value sets as *sequences*. In order to support maps with key types other than strings, the CAE FreeMarker view engine does not map Java objects of type `java.util.Map` to FreeMarker *hashes*. Instead, `java.util.Map` methods will be available on such models. In order to access the entry, key, or value sets, call the respective methods on the model object. Any such set is a FreeMarker *sequence* and thus compatible with the `#list` directive.

```
<#list map.entrySet() as entry>
  ${entry.key} is mapped to ${entry.value}
</#list>
```

Example 4.8. Iterating over java.util.Map entries in FreeMarker templates

Using JSP Tag Libraries in FreeMarker Templates

FreeMarker templates can access functionality provided by a JSP tag library, assuming that the tag library is deployed in the web application as specified in JavaServer Pages 2.2 and up. Import the tags exposed by a JSP tag library into a named hash by using its URI as a key into the implicit `JspTaglibs` map. The imported tags will be available as custom directives in the named hash.

```
<#assign fmt=JspTaglibs["http://java.sun.com/jsp/jstl/fmt"]>
<@fmt.formatNumber value=self.someValue/>
```

`JspTaglibs` only exposes tags, not static methods exposed by a JSP tag library as functions.

Accessing Static Methods in FreeMarker Templates

To give a FreeMarker template access to public static methods of a Java class, you have to implement a "facade" Java singleton that provides non-static methods that delegate to the static methods.

```
public final class FreemarkerFacaceExample {
    public static final FreemarkerFacaceExample INSTANCE = new
    FreemarkerFacaceExample();

    private FreemarkerFacaceExample() {
    }

    /**
     * Provides non-static access to static method.
     */
    public String nonStaticDefaultString(String text) {
        return StringUtils.defaultString(text);
    }
}
```



```
}

```

Then, add this singleton as a shared variable to the CAE's FreeMarker configuration, and access the methods using the singleton in any CAE FreeMarker template.

The following listing shows an example Spring configuration to add a custom shared FreeMarker variable, assuming the facade singleton class is called `com.company.cae.MyFreemarkerFacade` and the variable should be exposed as `myFreemarkerFacade`.

```
<import
resource="classpath:/com/coremedia/cae/view-freemarker-services.xml"/>

<customize:append id="myFreemarkerSharedVariablesCustomizer"
bean="freemarkerSharedVariables">
  <map>
    <entry key="myFreemarkerFacade">
      <bean class="com.company.cae.MyFreemarkerFacade">
        <!-- inject services etc. here! -->
      </bean>
    </entry>
  </map>
</customize:append>
```

Auto-Import of Freemarker Functions and Macros

In order to expose functions, macros, or common configuration to all templates, you need to add the corresponding Freemarker file to the `freemarkerConfigurer` bean's property `autoImports`. The following listing shows an example Spring configuration that exposes all functions of `custom-functions.ftl` with the name `cufu`.

```
<import
resource="classpath:/com/coremedia/cae/view-freemarker-services.xml"/>

<customize:append id="myFreemarkerAutoImportsCustomizer"
bean="freemarkerConfigurer" property="autoImports">

  <map>
    <entry key="cufu"
value="/lib/custom/freemarker/custom-functions.ftl"/>
  </map>
</customize:append>
```

All functions are now available to your Freemarker templates. However, the IDE will most likely not recognize these functions and the name defined in your Spring configuration. Adding a `freemarker_implicit.ftl` as shown in [Example 4.9, "Code for Idea auto-completion" \[82\]](#) to `src/main/resources/META-INF/resources/` within your Maven module's directory will add auto-completion to the IntelliJ IDEA development environment.

```
[#ftl]
[#-- @implicitly included --]
[#import "/lib/custom/freemarker/custom-functions.ftl" as cufu]
```

Error Handling

The views rendered for a particular page can be thought of as a tree of views, with the outermost (top-level) at the root of the tree, and each include operation adding another "child". In this nested hierarchy of views, exceptions may be thrown at any time: either because one of the templates has a syntax error and cannot be compiled, because of an I/O error when loading content from the content repository, or for any other reason which may cause exceptions at runtime. By default, exceptions thrown while rendering views are passed all the way "up" the inclusion stack. Exceptions not handled at any level will eventually be handled by the servlet container by forwarding the request to the appropriate error page, if configured appropriately.

In addition to this default exception processing, the *CoreMedia CAE* provides an [ExceptionHandlerViewDecorator](#) to handle exceptions at different levels of the view hierarchy. Using this feature, view exception messages may be shown in the context of the page on which they occurred which is useful to find and fix issues in a development or preview environment. In production environments, the same decorator can simply remove the output of the view causing the error, thus leading to fewer error pages presented to end users at the price of not showing some content on the page.

Activating View Exception Handling

The view exception handling decorator is activated by default. To deactivate it, set

```
view.errorhandler.enabled=false
```

Determining How to Handle a View Exception

By default, "handling an exception" means that the output of the view subtree producing the error will be discarded. Note that this mechanism will use additional output buffering, so as - always - it is a good idea to watch out for potential negative effects on temporary heap usage or garbage collection times. However, in most cases this should not be an issue. To render the error message and the exception stack trace on the page (replacing the output of the view subtree producing the error), set the following property in preview or development environments:

```
view.errorhandler.output=true
```

The output can be styled using an appropriate CSS style sheet to match the visual appearance of the surrounding page. For instance, a minimal style sheet could

show a red box containing the error message while hiding the stack trace, which may become very long:

```
table.cae-rendererror {
  border: #FF0000 solid 3px;
  color: #000000;
}

table.cae-rendererror .cae-rendererror-stacktrace {
  display: none;
}
```

To render view exceptions on a page, a fallback view is provided in the `fallbackViewRepository`. To use a custom exception rendering template rather than the fallback view, add your own view - such as a FreeMarker template - for `com.coremedia.objectserver.view.ViewException`.

Choosing Where to Handle Exceptions

Regardless of whether you suppress output in a production environment or show an error message in a preview or development environment, it is necessary to control where on the page exceptions will be handled. A page usually consists of many nested inline and block elements, all rendered by views in the view tree. It usually makes sense to handle an exception at a certain block level, where it is semantically acceptable to discard erroneous view output or replace it with an error message.

As an example, assume a page with a side bar rendering each item in a collection of content beans using the view name "teaser". The same "teaser" views may also be used in other areas of the page, and each such view again includes many smaller views to include images, video previews, text, metadata and so on. For such application, it is useful to handle exceptions at the "teaser" level, which means that any exception thrown in any of the views making up that teaser view, will be passed up to the "teaser" level for exception handling. In this case, if the "metaData" view included from within the "teaser" threw an exception, the output of the "teaser" view would be discarded completely or replaced completely with an error message, instead of just the "metaData" output.

To control which views should handle exceptions thrown by themselves or views they include, the `ExceptionHandlerViewDecorator` is configurable with `accept` and `reject` lists for bean types as well as view names. Each list may be configured by an appropriate customizer. To continue with the above example, assume you decide to handle exceptions at the "teaser" level for any `com.example.content.beans.base.CMObject`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:customize="http://www.coremedia.com/2007/coremedia-spring-beans-customization">
  <customize:append id="addCustomExceptionHandlerDecoratorAcceptBeanClasses"
```

```

        bean="exceptionDecoratorAcceptBeanClasses">
    <list value-type="java.lang.Class">
        <value>com.company.contentbeans.base.CMObject</value>
    </list>
</customize:append>
<customize:append id="addCustomExceptionDecoratorRejectBeanClasses"

        bean="exceptionDecoratorRejectBeanClasses">
    <list value-type="java.lang.Class">
<!-- do not add anything -->
    </list>
</customize:append>
<customize:append id="addCustomExceptionDecoratorAcceptViews"
        bean="exceptionDecoratorAcceptViews">
    <list value-type="java.util.regex.Pattern">
        <value>teaser</value>
    </list>
</customize:append>
<customize:append id="addCustomExceptionDecoratorRejectViews"
        bean="exceptionDecoratorRejectViews">
    <list value-type="java.util.regex.Pattern">
<!-- do not add anything -->
    </list>
</customize:append>
</beans>

```

In this example, any exceptions thrown will be passed up the view hierarchy to a view "teaser" rendered for a bean of type `com.example.contentbeans.base.CMObject`, where it will be handled. The *reject* lists may be used as a restriction: a view will only handle an exception, if both *accept* conditions and no *reject* conditions match.

You might instead add `java.lang.Object` to `exceptionDecoratorAcceptBeanClasses` and `*` to `exceptionDecoratorAcceptViews`, if you wanted any view to handle an exception. In that case, you should reject beans of type [com.coremedia.cap.common.Blob](#), to avoid breaking binary content.

Reference for FreeMarker Templates

The macros, functions and variables described in this section as well as any request, session, and servlet context scope attributes are implicitly available in any FreeMarker template view rendered by the CAE.

Macro / Function	Parameters	Description
@cm.include	<i>self</i> : Object (required), <i>view</i> : String (optional), <i>params</i> : Hash (optional)	Render the object passed as "self" in the given view at this position in the output. Requires a template/view to be defined for such an object. Examples:

Table 4.5. Implicit macros, functions and variables in FreeMarker templates

Macro / Function	Parameters	Description
		<pre><@cm.include self=self.navigation/></pre> <pre><@cm.include self=self view="frame"/></pre> <pre><@cm.include self=self view="overview" params={"details": false}/></pre>
@cm.hook	<i>id</i> : String (required), <i>self</i> : Object (optional, defaults to <i>self</i> object from template context), <i>params</i> : Hash (optional)	<p>Renders the results of all com.coremedia.objectserver.view.events.ViewHookEventListener implementations that match the given type of <i>self</i> and that support the given <i>id</i> and the parameters.</p> <p>Examples:</p> <pre><@cm.hook id="htmlHead"/></pre> <pre><@cm.hook id="pictureSubHeadline" self=self.picture/></pre> <pre><@cm.hook id="teaserContainer" self=containerItem params={"isFirstInList": true}/></pre>
cm.getLink()	<i>target</i> : Object (required), <i>view</i> : String (optional, defaults to <code>cm.UNDEFINED</code>), <i>params</i> : Hash (optional): additional parameters given as a hash	<p>Create a link to the object passed as "target" in the given view and return the URL as a string. Requires a link scheme to be defined for the target object. If the target object is <code>cm.UNDEFINED</code>, an empty string is returned.</p> <p>Examples:</p> <pre>#{cm.getLink(article, {"foo": 1})}</pre> <pre>#{cm.getLink(self, "asTeaser")}</pre> <pre>#{cm.getLink(article, "asTeaser", {"foo": 1})}</pre>
cm.getId()	<i>self</i> : Object (required)	<p>Determine this object's id through the <code>IdProvider</code> and return the id as a string.</p> <p>Examples:</p>

Macro / Function	Parameters	Description
		<pre><input name="teaser" type="hidden" value="\\${cm.getId(teaser)}"/></pre>
@cm.responseHeader	<i>name</i> : String (required), <i>value</i> : String (required)	<p>Set an HTTP response header. If the response is already committed, the macro will fail. Provided as part of the CAE library because this feature is missing in the standard FreeMarker Web integration.</p> <p>Example:</p> <pre><@cm.responseHeader name="Content-Type" value="text/html; charset=UTF-8"/></pre>
cm.getRequestHeader()	<i>name</i> : String (required)	<p>Get an HTTP request header. Provided as part of the CAE library because this feature is missing in the standard FreeMarker Web integration.</p> <p>Example:</p> <pre><#if cm.getRequestHeader("Accept")?contains("application/xml")>...</pre>
@cm.metadata	<i>data</i> : * (required)	<p>Create a metadata attribute with a value containing a serialization of the given data. This function is covered in depth in section Section "Metadata Support in FreeMarker Templates" [89].</p> <p>Example:</p> <pre><div class="foo">@cm.metadata self.content/>...</pre>
@cm.previewScripts		<p>Enables the usage of the @cm.metadata macro and has to be included in the template once, before @cm.metadata can be used. For details see Section "Metadata Support in FreeMarker Templates" [89].</p>

Supported Standards and Template Language Versions

FreeMarker templates are expected to comply with the FreeMarker 2.3.x syntax. See the [FreeMarker documentation \(http://freemarker.sourceforge.net/docs/index.html\)](http://freemarker.sourceforge.net/docs/index.html) for details.

The *CoreMedia CAE* web application and tag library support the Servlet 2.5/JavaServer Pages 2.2 standards.

4.3.5 Adding Document Metadata

In order to hand over information rendered by the *CAE* to *Studio* you can include metadata in your HTML documents. To allow attaching metadata to a specific DOM element, it is added as a custom HTML 5 data attribute called `data-cm-metadata`. For each DOM element, metadata may consist of complex data structures in terms of (nested) maps and lists that hold primitive data objects like strings or integers but also application objects if corresponding serializers are available. Several serializers are predefined, in particular one for Content objects.

Metadata nodes are assumed to be nested corresponding to the DOM hierarchy of the elements they are attached to. From all metadata nodes found in the HTML document, a metadata tree is built according to the following rules:

- There is an artificial metadata tree root node.
- For a metadata node *m* found in a DOM node *d*, look for the first parent DOM node that also has a metadata node assigned (say *m'*) and add *m* as a child of *m'*. If no such parent node is found, add *m* as a child of the root node.
- If a DOM node has a list of metadata nodes assigned, these are interpreted as hierarchical nodes in the metadata tree, that is, children are assigned to the last node of the list and the first node of the list is assigned as a child to the metadata parent node.

Example 4.10, “A DOM with Metadata and Generated Metadata Tree” [87] shows an example DOM tree with metadata attached to its elements (→). Note that the list of metadata at the topmost div element is mapped to a hierarchy of metadata nodes in the metadata tree.

DOM with Metadata	Metadata Tree
<pre> <html> <body> -> "S" <div> -> ["A", "x"] -- <div> -> "B" <div> -> "y" -- -> "C" </pre>	<pre> root slider metadata "S" content "A" property "x" -- content "B" property "y" -- content "C" </pre>

Example 4.10. A DOM with Metadata and Generated Metadata Tree

*S: slider metadata
A, B, C: content objects
x, y: properties*

When the preview page is shown inside *Studio*, the resulting metadata tree is serialized and sent to the containing *Studio*, where it is deserialized and used by the built-in preview integration.

Supported Metadata

If metadata refers to a [Content](#) object, *Studio* shows a context menu that allows the editor to interact with this document (open it in a document tab, for instance) when the editor right-clicks inside the preview panel on the corresponding DOM element to which the metadata has been attached.

Similarly, string metadata is interpreted as a property path starting at the document specified by the parent metadata node. If this document is the same as the one shown in the document form, right-clicking the DOM element to which the property metadata has been attached (or any of its subelements) focuses the corresponding property field in the document form. This even works for link list properties. If the property belongs to another document, right-clicking on the property DOM element delegates to the parent node, that is, it opens a context menu that offers actions for that document.

Since Preview Shortcuts refer to `Content`, not content beans, note that all custom properties have to be specified with a `properties.` prefix. Only `Content` meta properties like `modificationDate` are specified without this prefix.

The third kind of metadata which is supported in *Studio* is device slider metadata, which is used to render a device slider for responsive websites that can be used to switch between different target resolutions of the site. The device slider metadata is a structured object consisting of two properties: `cm_responsiveDevices` which is basically a map from device name to resolution and `cm_preferredWidth` which tells the width for the full-width mode of the *Studio* preview.

```
{
  "cm_preferredWidth": 1280,
  "cm_responsiveDevices": {
    "mobile": {"width": "320", "height": "480", "order": "1",
    "isDefault": true},
    "tablet": {"width": "600", "height": "800", "order": "2"},
    "notebook": {"width": "1024", "height": "768", "order": "3"}
  }
}
```

Due to the tight integration of *CoreMedia Studio* and the embedded preview it might be preferable to block animations or certain behavior inside the embedded preview. In order to do so a previewed documents can provide metadata with additional style sheet and JavaScript URLs. These URLs are only loaded when the document is displayed in the context of the embedded preview. The metadata specifying these URLs has to be attached to the `head` element of the previewed document.

Supported Metadata:

Content Objects

Property Paths

Slider Metadata

Example 4.11. Responsive Device Slider Metadata

Studio Specific CSS and JavaScript


```
{
  "cm_studioPreviewCss": ["css-url-1", "css-url-2"],
  "cm_studioPreviewJs": ["js-url-1", "<js></js>-url-2"]
}
```

Example 4.12. Studio Specific CSS and JavaScript Metadata

The built-in *Studio* preview integration renders borders around highlighted preview DOM elements to indicate where metadata is available (gray border on mouse hover) and which DOM elements carrying metadata have been focused (blue border on right-click or focus). Usually, these borders are rendered by absolutely positioned line overlays. Occasionally, these lines interfere with the web page's mouse hover behavior, for example when the web page uses pop-up menus for navigation.

Controlling the highlight border rendering strategy

For such cases, you can tell *Studio* to use an alternative highlight border rendering strategy by adding the metadata property `cm_highlightStrategy` with a value of "css" to a DOM element. Then, for all metadata of this DOM element or any transitive child elements, highlight borders are rendered by adding a generated style class that sets an inner border (more precisely, an inset box shadow). This rendering strategy does not interfere with mouse hover events, but its visibility on different kinds of DOM elements (images, for instance) is less reliable.

If you have to combine standard metadata and `cm_highlightStrategy`, consider [Section "Advanced Metadata Usage" \[92\]](#) about using the default property "_" (underscore).

It is also possible to attach custom metadata to the preview and implement a *Studio* plugin that accesses the metadata tree. For details, see [Section "Advanced Metadata Usage" \[92\]](#).

Custom Metadata

Enabling Metadata Support

In order to include metadata in your documents, you have to explicitly enable it globally. Metadata is usually only enabled in a preview *CAE*, not in a live (production) *CAE*.

To enable metadata inclusion globally, you have to set the `metadata.enabled` property in the `WEB-INF/application.properties` file of your *CAE* application.

```
...
metadata.enabled=true
...
```

Metadata Support in FreeMarker Templates

If you want to add metadata to an HTML document from within a FreeMarker template, make sure the FreeMarker macro `@cm.previewScripts` is called in a template rendered once anywhere on the generated HTML page. You can then call the macro `@cm.metadata` with the metadata that is to be assigned to an HTML DOM node. To allow assigning multiple metadata nodes to the same DOM node,

you can call `@cm.metadata` with an array, where each array element generates a metadata node.

The macro call `<@cm.metadata . . . >` renders an HTML fragment, namely a custom HTML 5 attribute named `data-cm-metadata` (all custom HTML 5 attributes have to start with `data-`) with the serialized metadata as its value.

There are essentially two ways to attach metadata to an HTML element: directly or through a local variable.

The inline metadata macro call looks like so:

```
<div class="page"<@cm.metadata data=self.content/>>Hello world!</div>
```

Since `data` is the only parameter of the `@cm.metadata` macro, FreeMarker allows omitting its name and the equal sign, resulting in this even shorter variant:

```
<div class="page"<@cm.metadata self.content/>>Hello world!</div>
```

Note that macro `@cm.metadata` outputs a complete HTML attribute name and value, including a leading space. When metadata output is disabled, nothing is written, so leaving out the leading space leads to a bit less readable template, but to cleaner output - your choice.

You can use FreeMarker's object literal notation to specify more complex metadata. If the metadata expression is more extensive, if metadata is reused for multiple DOM nodes, or if you just want a very clear separation of metadata and HTML output, it is recommended to assign metadata to a variable using FreeMarker's `#assign` directive and hand over the variable to `@cm.metadata` inside the HTML tag:

```
<#assign sliderMetadata={
  "cm_preferredWidth": 1280,
  "cm_responsiveDevices": {
    "mobile_portrait": {
      "width": 320,
      "height": 480,
      "order": 1,
      "isDefault": true
    },
    ...
  }
}>
...
<body id="top"<@cm.metadata sliderMetadata />>
```

In a normal CAE FreeMarker template, `self` refers to the current content bean. Each content bean has a property `content` that refers to the underlying `Content`, so typical Preview Shortcut metadata looks like so:

```
<div<@cm.metadata self.content/>>...</div>
```

As an example, assume the current content bean provides the content properties `title` and `text`, these properties are written by the template as heading and block text, and you want to add metadata to tell *Studio* about the used content properties. Here is an example of a FreeMarker template fragment that adds the correct metadata:

```
<div<@cm.metadata self.content/>>
  <h1<@cm.metadata "properties.title"/>>${self.title}</h1>
  <div<@cm.metadata "properties.text"/>>${self.text}</div>
</div>
```

Note how the containing document is only attached once to a surrounding DOM element. If this is not possible because of the given DOM structure (which you usually do not want to change to avoid layout problems), you can use `@cm.metadata` with an array parameter specifying multiple metadata nodes:

```
<h1<@cm.metadata [self.content,
"properties.title"]/>>${self.title}</h1>
<div<@cm.metadata [self.content,
"properties.text"]/>>${self.text}</div>
```

As mentioned above, you can define CSS and JavaScript that is to be loaded in a preview inside *Studio* only. In a FreeMarker template the corresponding metadata object can be created via the convenience function `cm.getStudioAdditionalFilesMetadata()` that takes two list parameters. The first list provides additional style sheets, the second one additional JavaScripts. Each list can either contain content beans of an appropriate type or URL strings.

Adding Metadata for Studio Specific CSS and JavaScript

```
<#assign studioMetadata=
cm.getStudioAdditionalFilesMetadata(CSS_LIST, JS_LIST)/>
<head <@cm.metadata studioMetadata/>>
```

Metadata Support in JSP Templates

If you want to add metadata to an HTML document from within a JSP template, include the JSP tag `cm:previewScripts` in a template that is called once for each HTML page. You can then use the tag `cm:metadata` each time metadata is to be assigned to an HTML DOM node.

The tag `cm:metadata` checks whether metadata rendering is enabled (either globally or locally for this tag occurrence). If enabled, the given metadata is serialized as a JSON string. In the rendered document, this string is escaped accordingly and output as the value of the custom HTML attribute `data-cm-metadata` of the HTML element that the metadata is attached to.

Example:

```
<cm:metadata value="${self.content}" />
```

To allow assigning multiple metadata nodes to the same DOM node, multiple nested `cm:object` tags have to be used instead of the `value` attribute. `cm:object` has only a `value` attribute and is used for list elements.

```
<cm:metadata>
  <cm:object value="\${self.content}"/>
  <cm:object value="properties.title"/>
</cm:metadata>
```

Example 4.13. Content With Property

The tag `cm:property` can be nested into `cm:metadata`, `cm:object` or `cm:property` to create a name-value pair. Again, the value can be specified either as an attribute or through nested tags.

```
<cm:metadata>
  <cm:property name="cm_preferredWidth" value="1280"/>
  <cm:property name="cm_responsiveDevices">
    ...
    <cm:property name="mobile portrait">
      <cm:property name="width" value="320"/>
      <cm:property name="height" value="480"/>
      <cm:property name="isDefault" value="\${true}"/>
      <cm:property name="order" value="1"/>
    </cm:property>
  </cm:property>
</cm:metadata>
```

Example 4.14. Responsive Device Slider Metadata

Advanced Metadata Usage

For *Studio* preview integration, you usually use content and property paths as metadata to specify the source of generated HTML output. As convenience, the metadata macro / tag automatically converts object and string parameters to metadata nodes with a single "default" property named `"_"` (underscore), containing the given data. You only need to specify this default property explicitly if you want to add custom metadata to the same metadata node.

The *Studio* preview integration only evaluates content objects and properties in the `_` property, the properties `cm_preferredWidth` and `cm_responsiveDevices` which are used for the device slider, and additionally the property `cm_highlightStrategy` to control the highlight border rendering strategy.

All metadata using other property names will be handed through to *Studio*, but is not interpreted by the built-in preview integration. To take advantage of such custom metadata, you have to implement a *Studio* plugin that accesses and interprets this metadata. For details, see Chapter 1, *Introduction* in *CoreMedia Studio Manual*.

Adding custom metadata

Here is an example of the same combination of preview metadata and custom metadata in both template languages, FreeMarker and JSP.

```
<@cm.metadata [self.content, {" ": "properties.title",
    "custom-key": "custom-value"}] />
```

Example 4.15. Mixed preview and custom metadata in FreeMarker

```
<cm.metadata>
  <cm.object value="\${self.content}"/>
  <cm.object>
    <cm.property name=" " value="properties.title"/>
    <cm.property name="custom-key" value="custom-value"/>
  </cm.object>
</cm.metadata>
```

Example 4.16. Mixed preview and custom metadata in JSP

4.3.6 Working with Forms

Often times, users need to interact with a website. Be it searching, editing a profile or signing up for a newsletter. These use cases are commonly implemented using a form based solution. Since the CAE integrates deeply with the Spring Framework, this description focuses on using Spring Forms and using a Spring Web MVC 3.x handler.

Form rendering

In order to render a form with Spring Forms, several things must be done:

1. A simple model Java bean (POJO) with properties for each form field is used as a back end and to represent the form.

This is a simple example for such a backing bean:

```
public class MyForm {

    private String email;
    private String emailRepeat;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmailRepeat() {
        return emailRepeat;
    }

    public void setEmailRepeat(String emailRepeat) {
        this.emailRepeat = emailRepeat;
    }
}
```

2. The form backing bean must be added to the model that is rendered.

To add the form backing bean to the model, add a method to the handler class, annotated with [@ModelAttribute](#)

```
@ModelAttribute("nameOfForm")
public MyForm createMyForm() {
    return new MyForm();
}
```

- To render the front end, Spring provides a tag library to create HTML forms in JSPs, accessing the form bean in the model.

This is a simple example for such a form, see [Spring form tag library documentation](#) for details on how the form taglib may be used.

```
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>

<form:form method="POST"
    action="handlerUri"
    commandName="nameOfForm">

    <!-- render form fields -->
    <form:input path="email"/>
    <form:input path="emailRepeat"/>

    <input type="submit" value="Subscribe"/>

</form:form>
```

Using Ids for Encoding Objects in Form Fields

Under some circumstances, you will need to write down a string representation of the identity of a bean, for example "the content bean for content 22". This is typically necessary in intermediary XML documents or when you want to refer to a bean in an HTML hidden input field.

For this purpose, the *CoreMedia CAE* contains a generic ID facility that allows you to convert selected bean types to a string and back. The ID API basically consists of two methods `#getId` and `#parseId` in the class `com.coremedia.id.IdProvider`. Note that this is *not* an object serialization. This facility is only useful to capture an id of a stateless object that represents an external business entity, as outlined in [Section "Patterns For Content Beans" \[28\]](#). The default implementation comes with id support for content beans and blob properties. Other bean types can be supported by writing a new implementation of `com.coremedia.id.IdScheme` and plugging it into the id resolver using a `Customizer`.

In order to encode an object id into a form field in a template, as well as to decode it back on a form submission, the *CoreMedia CAE* comes with a custom tag `<cm:id>` as well as an implementation of the `java.beans.PropertyEditor` interface that you can use in Spring to parse form fields back into bean references.

The following example shows how to encode the id of a bean `feature` into an HTML form:

```

<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>

<cm:id self="${feature}" var="id"/>
<form:form method="POST"
    action="handlerUri"
    commandName="nameOfForm">
  <!-- render form fields -->
  <input name="feature" value="${id}" type="hidden" />
  <form:input path="email"/>
  <form:input path="emailRepeat"/>

  <input type="submit" value="Subscribe"/>
</form:form>

```

In this example, a regular `<input>` field was used to render the id. Because of this, the id will not be bound to the backing bean, but the value can be retrieved by the controller using the command `request.getParameter("feature")`

Form submission

A form submission can be handled with Spring MVC means. The form backing bean is automatically filled with the posted values of the form. When a responsible handler is found for a request, the form bean is passed as a method argument to the handler method if a method parameter is annotated with `@ModelAttribute`.

```

public ModelAndView handleFormSubmit(
    @ModelAttribute("nameOfForm") MyForm form, ...)

```

Form validation

Spring provides a general concept for form/bean validation in the back end.

Validators

In order to validate a form, an [org.springframework.validation.Validator](#) can be implemented for arbitrary form backing beans. The validation method populates an [org.springframework.validation.Errors](#) object with error messages, see [MyForm-Validator Example \[96\]](#) for a complete example.

```

if(form.getEmail() == null) {
    errors.rejectValue(
        "email",
        "error-email-missing",
        "The email address is missing."
    );
}

```

The first argument passed to `Errors#rejectValue()` denotes the form bean property (here: "email") that is invalid. The following arguments are an error code (to be defined in a resource bundle) and a default message.

Global errors affecting the entire form instead of a single property are supported, too.

Associate a validator with a form bean

To validate a form bean with a validator in the context of a handler, add an `@InitBinder` annotated method to the handler:

```
@InitBinder("nameOfForm")
protected void initBinder(WebDataBinder binder) {
    binder.setValidator(new MyFormValidator());
}
```

Do not forget the form name, otherwise the validator will be applied to any `@ModelAttribute` or `@PathVariable` arguments.



To actually validate the form bean, annotate the method parameter with `@Valid`.

```
public ModelAndView handleFormSubmit(
    @ModelAttribute("nameOfForm") @Valid MyForm form, ...)
```

This is an example validator that implements all necessary methods for the example use case of validating the `MyForm` example shown before:

```
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import java.util.regex.Pattern;

/**
 * Validator for {@link MyForm}
 */
public class MyFormValidator implements Validator {

    /**
     * this pattern matches an email address such as "test@test.com"
     */
    private static final Pattern EMAILADDRESS_PATTERN =
Pattern.compile("\\b[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,4}\\b");

    @Override
    public boolean supports(Class<?> clazz) {
        return MyForm.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {

        MyForm form = (MyForm) target;

        //use Spring Utility to validate if form field is empty
        ValidationUtils.rejectIfEmptyOrWhitespace(
            errors,
            "email",
```



```

        "error-email-missing",
        "Email is missing");

    //if form field has content, validate if format matches email
    pattern
    if (!errors.hasErrors()) {

        if (!isValidEmail(form.getEmail())) {
            errors.rejectValue(
                "email",
                "error-email-format",
                "Not a valid email address");
        }
        //and if form field contents match each other.
        else if (!form.getEmail().equals(form.getEmailRepeat())) {
            errors.reject(
                "error-email-no-match",
                "Emails are not equal");
        }
    }
}

/**
 * @return true if email matches the pattern
 */
protected boolean isValidEmail(String email) {
    return EMAILADDRESS_PATTERN.matcher(email).matches();
}
}

```

Error handling in the handler method

When errors during binding should be handled within a handler method, an optional [BindingResult](#) method parameter must be added to the handler method to be able to access any validator errors added during binding.

The method parameter `BindingResult` *MUST* follow the validated parameter immediately!



```

public ModelAndView handleFormSubmit(
    @ModelAttribute("nameOfForm") @Valid MyForm form,
    BindingResult formBindingResult,
    ...)

```

`BindingResult#hasErrors()` can be used to check for errors in the handler method.

`BindingResult#reject()` can be used to add errors (as a result of a business transaction, for example) in the handler method.

Presenting form errors

The Spring form tag lib contains tags to display global or field specific error messages:

```

<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>

<form:form method="POST"
    action="handlerUri"
    commandName="nameOfForm">

    <!-- render global error message if available -->
    <form:errors cssClass="notification error"/>

    <!-- render form fields with
    specific error messages, if available. -->
    <form:input path="email"/>
    <form:errors path="email" cssClass="notification error"/>

    <form:input path="emailRepeat"/>
    <form:errors path="emailRepeat" cssClass="notification error"/>

    <input type="submit" value="Subscribe"/>
</form:form>

```

See `<form:errors>` tag [documentation](#).

Handling POST requests

When handling POST requests, these steps should be done in the handler method:

1. Consume POST data
2. Update application state (for example update external database, send data to external service, ...)
3. Send a 302 "moved temporarily" response and redirect to the page the request came from so that a page reload won't change the application state again. See [Section "Handling redirects" \[98\]](#)
4. If needed, status information can be transferred from the handler to the following (redirected) request using flash attributes, see [Section "Preserving attributes in a redirect" \[99\]](#)

Handling redirects

Sometimes it's necessary to return a redirect from a handler method. The CoreMedia CAE supplements Spring MVC in order to support this use case.

Redirecting to a (content) bean

The API provides a convenience method for redirecting to a page that is represented by a model bean: [HandlerHelper#redirectTo\(bean\)](#)

Redirecting to an external URL

When redirecting to an (external) URL, a [RedirectView](#) may be used for the `ModelAndView` that is returned from the handler method, for example:

```

RedirectView redirectView = new
RedirectView ("http://www.my-website.com/");
redirectView.setStatusCode(HttpStatus.MOVED_PERMANENTLY);

return new ModelAndView (redirectView);

```

Preserving attributes in a redirect

Sometimes it is necessary to display status information (a confirmation message, for instance) as result of a POST handler. Spring MVC provides the concept of "Flash Attributes": Attributes that can be passed to the handler receiving a redirected request, for example:

```

public ModelAndView handleRequest(..., RedirectAttributes
redirectAttributes) {

    // handle request

    redirectAttributes.addFlashAttribute(
        "status",
        "Everything is fine.");

    // send redirect using
    // HandlerHelper#redirectTo() or
    // a org.springframework.web.servlet.view.RedirectView
}

```

Also, see [this post on Tikal.com](#).

Because of [SPR-10516](#) any beans added as objects to ModelAndView are converted to Strings (and might require to add a converter to `bindingConverters` bean (see [Section 4.3.1, "Handling Requests" \[50\]](#)) as soon as request handler specifies `RedirectAttributes` as parameter (and only then). This might prevent link handlers to be found by bean type. In order to work around this issue it is recommended to use `HandlerHelper#redirectBuilder(bean)` and specify the `redirectAttributes` which as a result when building the model and view will receive the model bean in addition to ModelAndView.



Protecting against Cross Site Request Forgery

Cross-site request forgery (CSRF) is a trivial attack on a web application, which - if vulnerable to this attack - allows an attacker to perform a state-modifying operation on behalf of an authenticated, honest user. Depending on the nature of the web application and the operations an authenticated user may perform, the potential damage may be significant. For instance, a vulnerable application may allow an attacker to take over an honest user's account by changing that user's email address to his own.

A variation on CSRF is "login CSRF", which is an attack tricking an honest user to log into a vulnerable application with an account owned by the attacker. An unsus-

pecting user who fell victim to this attack may add valuable information, such as his address or payment information to the account, resulting in a leak of sensitive user data to the attacker.

More information on cross-site request forgery can be found at the [Open Web Application Security Project: CSRF](#).

To reduce a CAE application's risk of vulnerability to CSRF attacks, the CAE provides a blanket protection against CSRF and login CSRF. As long as the application adheres to a set of conventions, this protection is mostly transparent. If enabled in an application which does not comply with the necessary conventions, the protection may be ineffective or state-modifying operations (for example POST requests made by authenticated users) may incorrectly be treated as attacks and rejected.

Enabling CSRF protection

To enable the CSRF protection mechanism, set the following property to `true` in your CAE application (default is `false`):

```
security.csrf-prevention.enabled=true
```

By enabling this feature, a random token will be stored in each non-anonymous user session. The client is expected to send the user's token value with every "unsafe" HTTP request (POST, PUT, DELETE). Any such "unsafe" request without a matching token will be treated as unauthorized and rejected.

To be effective, the anti-CSRF token must be kept as secret as the HTTP session ID.



In particular, mind the following:

- A secure, random anti-CSRF token will be created for a user session, whenever a non-anonymous user authenticates with the application. The token value can be trusted for embedding in HTML attributes or text nodes without HTML escaping.
- The token is made available as a model attribute `_CSRFToken` and can be used from a view.
- A hidden input field named `_CSRFToken` will be added automatically to any HTML form created using the Spring Form tag library. In future releases, this behavior may be changed to add the token only to forms with "unsafe" methods (such as POST).
- An interceptor will validate an anti-CSRF token sent with a request against the one stored in the user session. This validation only applies to "unsafe" request methods (POST, PUT, DELETE). If no request token is sent or the two token values do not match, an [org.springframework.security.access.Access-](#)

`DeniedException` will be thrown. To support Ajax requests, the request token will be taken from the custom HTTP request header "X-CSRF-Token", if available. Otherwise, the request token will be taken from the request parameter `_CSRFToken`.

In order for this mostly transparent protection to work properly, an application must fulfill these simple requirements:

- As recommended by [RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1](#), the application must use "unsafe" HTTP methods (POST, PUT, DELETE) for any state-modifying HTTP requests.
- Spring Security must be used for authentication.
- The application must create a new HTTP session after successful authentication. This prerequisite is implemented by Spring Security's [SessionFixationProtectionStrategy](#) and is necessary to protect against session fixation attacks anyway.
- Spring Security's [AuthenticationTrustResolver](#) is used to recognize anonymous users, so if an application implements custom [Authentication](#) objects to represent anonymous users, they should be compatible with the application's [AuthenticationTrustResolver](#).
- Spring's Form tag library should be used to create HTML forms, in order to add the anti-CSRF token automatically. Otherwise, forms to be sent via an unsafe HTTP method such as POST need to specify this field explicitly:

```
<input type="hidden" name="_CSRFToken" value="${_CSRFToken}"/>
```

- Ajax XMLHttpRequests sent with an unsafe HTTP method such as POST, PUT, or DELETE, must include the token value: either as a `_CSRFToken` request parameter (when the request is built from form values, for instance) or as a custom request header "X-CSRF-Token". For instance, an application which makes heavy use of XMLHttpRequest via POST, could add the token value in a global HTML5 data attribute (or hidden form field or any other hidden DOM element):

```
<html data-csrf-token="${_CSRFToken}"> ... </html>
```

When using jQuery, the token can be added to every POST Ajax request by registering an appropriate global `beforeSend` handler:

```
$.ajaxSetup({
  beforeSend: function ( xhr, settings ) {
    settings.headers[ "X-CSRF-Token" ] =
      $( "html" ).data( "csrfToken" );
  }
});
```

Example 4.17. Adding the anti-CSRF header to jQuery Ajax requests

- The application should be prepared to handle the [org.springframework.security.access.AccessDeniedException](#) thrown by an interceptor, for instance by mapping it to an appropriate error code such as 403 (FORBIDDEN). See [Section 4.3.9, “Dealing with Errors” \[107\]](#) for details.

Login CSRF prevention requires special treatment, because no (non-anonymous) user is authenticated when showing the login form.

To protect the login process, an application should force the creation of an anti-CSRF token before rendering the login form by calling [com.coremedia.security.web.csrf.CsrfPreventionManagement#forceToken\(HttpServletRequest\)](#). The bean implementing this interface is available in a CAE application context as [csrf-TokenManagement](#). Method `forceToken` has no effect, if no HTTP session exists or it already contains a token, so the application must make sure that a session has been created before calling this method.

Note that upon successful authentication, a new user session and a new token value will be created, as explained above.

If the login process is implemented as a Spring Webflow, the web flow framework will typically take care of creating a HTTP session to manage web flow state. Since all application context beans are available to expressions in a web flow definition, token creation may be forced in the `<on-entry>` phase of a `<view-state>` rendering the login form:

Example 4.18. Forcing token creation from a login web flow

```
<on-entry>
  <evaluate expression="csrfTokenManagement.forceToken(
    flowRequestContext.externalContext.nativeRequest)"/>
</on-entry>
```

4.3.7 Integrating with Spring Web Flows

[Spring Web Flow](#) is a framework for building complex form based web applications. Since it is based on Spring MVC, it can be easily integrated into any existing CAE web application.

CoreMedia provides an integration for merging Web Flows into a content based CAE application: a typical page that is delivered by a CAE application is composed of several hierarchical structured content beans, each of them representing a certain fragment of the page. Typically, a (Web Flow) form application should be embedded in a page as a fragment only.

In other words: Spring Web Flow result beans need to be merged into the CAE bean model.

Embedding Web Flows

First of all, creating web flows for the CAE does not differ from creating "standard" web flows: writing flow definitions, form beans etc. is exactly the same in the CAE.

The main difference lies in the way the flow execution is controlled: The standard [org.springframework.webflow.mvc.servlet.FlowController](#) takes over the control of the request including the rendering of the model. It uses an [org.springframework.webflow.context.servlet.FlowUrlHandler](#) for building and parsing adequate URLs pointing to this controller.

The CAE integration works in a slightly different way: the request can be still controlled by a custom controller which builds its `ModelAndView` traditionally. After that, it temporarily delegates the request to the Web Flow engine (by invoking [FlowRunner#run](#)). This runner executes the Web Flow logic and returns an enriched model consisting of the original model merged with the Web Flow model, a form and binding results, for instance. This merged model can be passed to the view rendering process (for instance the templates) that render the entire page containing the fragment with the flow results.

Example

A typical handler/controller method may look like this:

```
// step#1: build content model
ModelAndView modelAndView = ...;

// step#2: fetch flow id similar to
// FlowUrlHandler#getFlowId(HttpServletRequest)
String flowId = ...;

// step#3: run flow and enrich model
ModelAndView mergedModelAndView = flowRunner.run(
    flowId,
    modelAndView,
    request,
    response);

// step#4: pass merged model to rendering engine.
// Note, that it might be null in case that webflow has handled
// the response directly, e.g. by sending a 302 redirect
return mergedModelAndView;
```

Configuration

In order to use the Web Flow integration, the artifact dependency `coremedia-webflow` as well as a Spring bean configuration `<import resource="classpath:/com/coremedia/cae/webflow/webflow-services.xml"/>` must be added to the application. The latter contains CAE specific web flow infrastructure setup as well as the bean `flowRunner`. This bean can be used by custom handler in the way described above.

Finally, custom flow definitions still need to be registered:

```
<webflow:flow-registry id="flowRegistry"
    flow-builder-services="flowBuilderServices">
  <webflow:flow-location-pattern
    value="classpath:/com/mycompany/**/*-flow.xml" />
</webflow:flow-registry>
```

4.3.8 Unit Testing a CAE Application

In order to promote a test-driven approach for development and to make testing of services implemented with the CAE application framework easier, CoreMedia ships an ease to use test add-on to be used in your tests based on [Spring Testing](#).

Differing from the unit testing approach, it doesn't focus on testing single classes only but helps to test services in a larger context and therefore brings the tests closer to the real world.

This approach enables to develop system tests at unit test level as there is no need for running external systems such as a content server or a servlet engine. The basic idea is to use a Spring application context that is composed from the same Spring bean declaration files that are used in the project.

Note that this requires the project Spring bean declaration in general to be self-contained and independent from each other. Otherwise, the application context could become too unhandy for testing when too many declarations have to be included recursively.



The add-on provided by CoreMedia supports an easy and convenient setup of an application context providing especially an in-memory content repository for your tests.

Below you will find two examples. For more examples, usage information and templates you might want to use in your IDE have a look at [XmlRepoConfiguration](#).

Examples

Testing Link Schemes

This example demonstrates how to set up an infrastructure that can be used for testing project link schemes. In the project's bean declaration `myproject-link-schemes-beans.xml` several link schemes are defined, as well as some CAE basic infrastructure such as the `LinkFormatter` bean. It is very useful to load exactly this file into a test application context, in order to...

1. test the contents of the file itself, for example detect whether there a syntactical or wiring problems

2. test the service instances with a configuration that is (nearly) equal to the configuration used in the project
3. test the service (in this example: the links scheme) in interaction with similar services, for example make sure that a certain link scheme is addressed for certain parameters and not a different link scheme instance.

Use the configuration pattern to construct the application context with the desired configuration:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = MyTest.LocalConfig.class)
@ActiveProfiles(MyTest.LocalConfig.PROFILE)
public class MyTest {
    @Configuration
    @ImportResource({
        value = {
            XmlRepoResources.LINK_FORMATTER,
            "classpath:/com/mycompany" +
            "/myproject/myproject-linkschemes-beans.xml"
        },
        reader = ResourceAwareXmlBeanDefinitionReader.class
    })
    @Import(XmlRepoConfiguration.class)
    @Profile(PROFILE)
    public static class LocalConfig {
        public static final String PROFILE = "MyTest";
    }

    // ...
}
```

Using a local test-only profile is recommended if you are using component scan to find your beans. If not using the `ActiveProfile`, `Profile` annotation pair `LocalConfig` classes of other tests might be found through component scan.

Now you can just inject the `LinkFormatter` and use it as in production code:

```
@Inject
LinkFormatter linkFormatter;

String link = linkFormatter.formatLink(
    new MyPage(123),
    "myView",
    new MockHttpServletRequest(),
    new MockHttpServletResponse(),
    false);

Assert.assertEquals("/123?view=myView", link);
```

Testing Handlers

A controller/handler's behavior strongly depends on the concrete setup of the application context. For instance, the registered `Converters` or `PropertyEditors` might have an influence on its behavior as well as the currently used `HandlerMapping`. Thus, it might be useful to take this environment into account when testing a handler. Spring provides `MockMvc` for emulating servlet requests and by capturing a handler's `ModelAndView` result. See corresponding [JavaDoc](#) for details.

```

@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(classes = MyTest.LocalConfig.class)
@ActiveProfiles(MyTest.LocalConfig.PROFILE)
public class MyTest {
    @Configuration
    @ImportResource(
        value = {
            XmlRepoResources.HANDLERS,
            "classpath:/com/mycompany" +
            "/myproject/myproject-handlers-beans.xml"
        },
        reader = ResourceAwareXmlBeanDefinitionReader.class
    )
    @Import(XmlRepoConfiguration.class)
    @Profile(LocalConfig.PROFILE)
    public static class LocalConfig {
        public static final String PROFILE = "MyTest";

        @Bean
        @Scope(SCOPE_SINGLETON)
        MockMvc mockMvc(WebApplicationContext wac) {
            return MockMvcBuilders.webAppContextSetup(wac).build();
        }
    }

    @Inject
    private MockMvc mockMvc;

    @Test
    public void test() throws Exception {
        Object expectedModelBean = ...;
        mockMvc
            .perform(
                MockMvcRequestBuilders
                    .get("/context/servlet/123")
                    .servletPath("/servlet")
                    .contextPath("/context")
            )
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers
                .model()
                .attribute(
                    HandlerHelper.MODEL_ROOT,
                    Matchers.equalTo(expectedModelBean)
                )
            );
    }
}

```

Mind the test annotation `@WebAppConfiguration` which is required to have a `WebApplicationContext` available to build the `MockMvc` object.

`MockMvcResultMatchers` provides several matchers for validating the response. For more sophisticated analysis you can end the validation with `andReturn()` and get for example the `ModelAndView` from the returned `MvcResult`.

More information

Take a look at the javadoc of [XmlRepoConfiguration](#) for getting more examples and how to use for example a custom in memory content repository.

4.3.9 Dealing with Errors

In any application, error handling is an important part of a consistent user experience. In a web application the goal is to return a useful response to the client in the case of an error condition, including an appropriate HTTP status code, an informative error page, a redirection, and often a combination of these.

Although the details of how particular errors are dealt with may differ from case to case, this section gives an overview of the different application components involved and best practices on how to implement error handling strategies.

Errors during request processing usually arise in one of two forms: expected and unexpected errors.

- *Expected errors* are often the outcome of validating input sent with the request such as the URL path, parameters or cookies. Request input is typically interpreted by a controller to construct a model and determine the view, so this is where such errors should be handled.
- *Unexpected errors* can - by definition - occur at any time during request processing. In addition to explicit error handling in controllers, it is therefore necessary to implement uncaught exception handling in an application.

Explicit error handling in controllers

Spring's `DispatcherServlet` is responsible for finding and executing a handler and rendering the view. A handler is first located by matching the request properties. Then the request will be bound to a handler method, including locating and calling appropriate type converters. Then the handler itself will be called to construct a `ModelAndView`. As mentioned above, the handler is the place for the application to decide whether a request is valid or should be answered with an error response.

To keep controllers and views separate, it is good practice to return a model representing the error case instead of generating the error response in the controller itself. For this purpose the CAE provides the `HttpError` class and utility methods in `HandlerHelper` to create error models. A default view for `HttpError` will set an appropriate HTTP status code and can be overwritten to generate more sophisticated error pages. See [Section “Building the Model” \[51\]](#) for details.

Uncaught exceptions while executing a handler

The `DispatcherServlet` will catch any unhandled exception thrown while executing handlers or handler interceptors and delegate them to `HandlerExceptionResolvers` to map the unhandled exception to a `ModelAndView`. Spring throws different unchecked exceptions when the `DispatcherServlet` is unable to resolve a request to a controller or fails to bind the request to it, for example because no matching type converter is defined. The default `HandlerExceptionResolver` simply maps these exception types to HTTP status codes such as `404 (NOT FOUND)`

or *400 (BAD REQUEST)*. A list of the default status code mappings is included in the [Spring documentation](#). For consistent error pages, it is recommended to define a custom exception resolver and map unhandled exceptions to [HttpError](#) models to share error views with explicit exception handling.

Uncaught exceptions while rendering a view

During this last stage of request processing the response may already have been committed and the status code set. Falling back to an error page is therefore not always possible. The CAE can react to unhandled exceptions during view rendering by dropping parts of the view or rendering an error message as part of the generated page. See [Section “Error Handling” \[82\]](#) for details.

Fallback error pages

So far it is assumed that a request will be handled by the `DispatcherServlet` and error handling can be implemented as part of the application. This is not always true, either because the web server forwards requests to the servlet container which do not map to the application or the `DispatcherServlet`, or because any of the components in the request processing chain becomes unavailable, or cannot communicate with the next component.

As a fallback for these cases, static error pages should be installed in all components in the request processing chain for a consistent user experience:

- Static default error pages can be configured in the application's deployment descriptor itself as described in [Java Servlet Specification 3.0](#). For instance, these will respond to otherwise unhandled error conditions or requests to unmapped URLs. Tomcat will only fallback to these defaults error pages, if the application does not handle an exception or sets an error HTTP status code with an empty body in the response.
- A web server configured as a reverse proxy to forward requests to a servlet container should at least be configured to return static error pages for cases when a request cannot be forwarded, the servlet container is not available, or there is a timeout. Apache HTTP Server provides the [ErrorDocument](#) directive for this purpose.
- In a more elaborate setup with load balancers, HTTP accelerators, or content delivery networks, each such stage should be able to deliver static error pages should the downstream stage become unavailable.

Best practices for error pages

- Error pages should set an appropriate HTTP status code: 4xx for client errors such as invalid requests and 5xx for server errors.

- HTTP error codes will prevent upstream components from caching the response. Heavyweight error pages which rely on upstream response caching should therefore be avoided.
- Invalid requests should be detected early and be rejected quickly, without spending much CPU resources on them.
- For security reasons, error messages and error pages should not reveal information about the application or its infrastructure. For instance, avoid sending stack traces to untrusted users.

4.4 Multi-Site and Localization Management

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

Configuration

The CoreMedia site model is defined via the bean `siteModel`. Refer to the Section 6.5, “Localized Content Management” in *CoreMedia Digital Experience Platform 8 Developer Manual* to know, how CoreMedia has designed multi-site and multi-language.

SitesService

To access all the features of multi-site and multi-language, you can use the `SitesService` defined as `siteService` Bean via the `bpbase-multisite-services.xml` Spring Bean Declaration.

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

4.5 CAE Developer Toolbox

The *CAE Developer Toolbox* is a web based set of tools that can help developers when working with CAE web applications. It connects to a CAE's remote (or local) MBean server, gathers and aggregates data and provides this via a user interface. The toolbox comes in two flavors: As a *CoreMedia Studio* plugin and as a standalone variant that can be integrated into any CAE web application.

The *Toolbox* GUI uses Studio UI components and therefore provides the same user experience. For example, detailed information is displayed in tooltips, columns can be added, removed or sorted. The *Toolbox* contains two tabs, *Cache Statistics* and *Cache Browser*.

Cache Statistics

Cache Statistics provides an overview of the current state of the memory cache.

There are several measurements that report the current state and behavior of the cache keys. For instance, the cache utilization and the hit rate is displayed. Displayed values are reloaded periodically. If the CAE monitored by the tool serves requests while the *Toolbox* is open, a developer can watch the memory cache utilization in order to determine if the application works correctly.

Cache class	Capacity	Miss rate	Removal rate	Eviction rate	Utilization
j1.Object	10000	0%	0%/min	0%/min	3%
c.c.s.StructParserCacheKey	5000	0%	0%/min	0%/min	2%
c.c.c.unlimited	2147483647	100%	0%/min	0%/min	0%
c.c.c.Heap	20000000	0%	0%/min	0%/min	19%
c.c.c.disk	10737418240	0%	0%/min	0%/min	0%
c.c.b.c.p.Aspect	1000	0%	0%/min	0%/min	0%
c.c.b.c.n.c.RootNavigationOnlySegmentCacheKey	10	100%	0%/min	0%/min	10%
c.c.b.c.e.Page	5000	0%	0%/min	0%/min	1%
c.c.b.c.s.s.SqlQueryCacheKey	100	40%	0%/min	0%/min	1%
c.c.b.c.p.ProductExcelSheetCacheKey	100	0%	0%/min	0%/min	0%
c.c.b.c.l.PageGridImpl	200	0%	0%/min	0%/min	0%
c.c.b.c.l.ContentBoardBackedPageGridPlacement	800	0%	0%/min	0%/min	2%
c.c.b.c.c.SettingsImpl	5000	0%	0%/min	0%/min	2%

Figure 4.8. Cache Statistics

Cache Browser

A tool for browsing through the memory cache.

Starting with list of all cache key classes currently present in the memory cache, you can look into single cache entries and analyze the cached contents as well as the dependencies and dependents. The view is split up into three areas, a list of

all `CacheKeys` on the left, a list of instances of a particular `CacheKey` class (if selected in the list of all `CacheKeys`) in the upper right and the detailed view of one `CacheKey` (if selected in the list of `CacheKeys` in the upper right) in the lower right portion of the screen. Values displayed in the three areas are not reloaded automatically so that you can spend more time looking at a snapshot of the memory cache. If one of the areas is reloaded (or loaded for the first time), the contents are requested new from the memory cache, so value counts may vary. All content can be reloaded by clicking the reload button in the upper right corner.

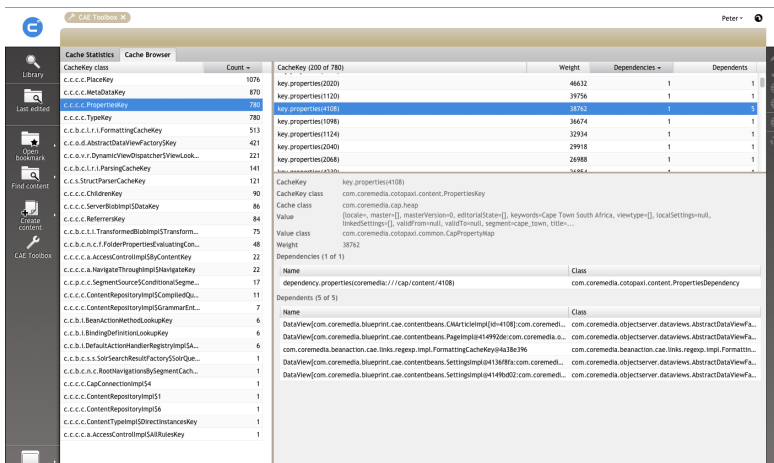


Figure 4.9. Cache Browser

Installation

This section describes the two ways the *Toolbox* can be installed.

Studio Web Application Plugin

Follow these installation steps to run the Toolbox as a plugin within a Studio web application:

1. Add this dependency to your Studio web application

```
<dependency>
  <groupId>com.coremedia.cae</groupId>
  <artifactId>cae-toolbox-studio-component</artifactId>
  <scope>runtime</scope>
</dependency>
```

2. Configure the JMX URL of the CAE web application to be connected to. Add a property `toolbox.jmx.url=service:jmx:the-jmx-url` to the application, to `WEB-INF/application.properties`, for instance. If your JMX connector

is password protected, you will also need to configure the properties `toolbox.jmx.user` and `toolbox.jmx.password`.

After login to *CoreMedia Studio* with administrator privileges, there is a new button "CAE Toolbox" available in the Favorites Bar that opens the toolbox.

CAE web application Integration

In order to run the Toolbox within a CAE web application (without the need of a running Studio web application) you simply need to add this dependency to your CAE web application.

```
<dependency>
  <groupId>com.coremedia.cae</groupId>
  <artifactId>cae-toolbox-component</artifactId>
  <scope>runtime</scope>
</dependency>
```

After starting the CAE web application, the Toolbox connects to the local MBean server which is available at:

```
http://<host>:<port>/<context>/<servlet>/toolbox (for example
http://localhost:8001/webappName/servlet/toolbox)
```

Working with the Toolbox

The following two examples show how the *Toolbox* can be used.

Example: How to find cache capacity problems

Problem:

- When the capacity configured for a cache class is too low relative to the working set, then even frequently used cache entries will be evicted (high eviction rate). This will also result in a high cache miss rate, as frequently used values do not remain in the cache for very long.
- If the cache key evaluation is sufficiently "expensive", this inefficient use of the cache may result in a poor performance.

Finding the cause:

- In the Cache Statistics tool check the utilization for all entries.
- An entry that has a red circle (means a utilization $\geq 90\%$) needs to be observed in more detail. Note that a high utilization isn't a problem per se.
- Check the eviction rate of the suspicious entry: If it has a high value ($> 20\%$) over a longer period then you should consider increasing the capacity. A value of 20% for "Eviction rate" means that roughly 20% of the values (measured by their weight, not by their number) are removed per minute.

- Increasing the capacity can be easily done by double-clicking into the capacity field and enter the new value. Note that this change is not made persistent: After a restart, the old value will be used.
- You can copy the fully qualified cache class name into the system clipboard for further examination in other tools like IDEs, say. Simply select the corresponding row and press CTRL-C.

Example: How to find cache entries with many dependents

Problem:

- A cache entry (such as a cached piece of content) having many dependents (such as many data views) can cause a high invalidation rate that requires lots of potentially expensive evaluations.
- This might happen when this cache entry is invalidated frequently and thus triggers transitive invalidation of its dependents.

Finding the cause:

- In the Cache Statistics tool check the "Removal Rate". An entry having a high value (> 10%) over a longer period should be analyzed, because this is a sign for continuous invalidation activity. If those cache entries are continuously evaluated to insert them into the cache again, you will also see a relatively high "Miss Rate" (number of cache misses divided by lookups) and "Insertion Rate" (approx. amount of cache values computed and inserted into the cache, relative to the capacity, per minute).
- Switch to the Cache Browser and click the "Dependents" tab in the upper right corner so that this area's entries are sorted by highest "dependents" values.
- Walk through all entries in column "Cache key class" (left area) and check the "Dependents" column (upper right area) for high values.
- Cache entries with large numbers of dependents are potential candidates for optimization.

4.6 Image Transformation API

Both the *CAE* and *CoreMedia Studio* support the specification and rendering of named variants of images. These variants are specified by a string which describes the transformation steps necessary to compute the variant. This feature is used extensively for rendering images, obviating the need to store image variants and renditions as distinct blobs within the CMS.

The transformation strings are stored in a map-like data structure within the image document settings. For example, an image document may contain the following variants:

Variant Name	Transformation String
"landscape_ratio4x3"	"crop;x=0;y=0;width=2285;height=1714"
"landscape_ratio5x2"	"crop;x=478;y=581;width=1807;height=725"
"portrait_ratio1x1"	"crop;x=570;y=0;width=1715;height=1714"

Table 4.6. Example of image transformation strings

A transformation is specified by a string with a syntax conforming to the hierarchical part of URIs (see RFC 3986: URI Generic Syntax). It is basically a sequence of path segments separated by slashes ('/'), each defining a single transformation operation. Each operation is applied to the binary data step by step, from left to right. The segment path denotes the name of the operation, and optional path parameters denote operation parameters.

An operation has a name, an optional alias, and an optional set of parameters. Each parameter may have a default value associated with it. Parameters are identified by name rather than ordinal position in the argument list.

For example, `a;x=1;y=2/b/c;r=q` is interpreted as the operation sequence `a(x="1",y="2")`, `b()`, `c(r="q")`.

Here is a slightly more complex example of an image transformation string:

```
rotate;angle=23/brightness;amount=70/box;width=121;height=121;upscale=false
```

Transformation operators and parameters may have shorter alias names, and parameters may have default values. Exploiting these, the example above might be rewritten as:

```
r;a=23/b;a=70/bo;w=121;h=121
```

Image Operations

Image transformations are implemented in the package `com.coremedia.transform.image` and subpackages. `com.coremedia.transform.image.ImageOperations` specifies a set of frequently needed image manipulation operations. These operate on an image representation specified by the type parameter `Image`. The package `com.coremedia.transform.image.java2d` contains an implementation of these operations based on the `javax.imageio` package which is part of the Java runtime environment.

The following operations are currently implemented. For details, see `com.coremedia.transform.image.ImageOperations`.

- `scale (alias: s)`: Scales the image.
- `fit (f)`: Fits the image into a rectangle.
- `box (bo)`: Scales the image to the target size, preserving the aspect ratio. An empty area on the sides will be filled with the background color, specified in the AARRGGBB (alpha red green blue) format. The default (0) is fully transparent.
- `crop (c)`: Uses only a specified area of the image, altering its dimensions.
- `flip (m)`: Mirrors the image horizontally or vertically.
- `rotate (r)`: Rotates the image around its geometrical center. A background color can be specified to fill the corners (see "box" operation).
- `gamma (g)`: Applies gamma correction.
- `brightness (b)`: Changes the brightness.
- `convert`: Produces an output image in the specified format.
- `gif`, `png` and `jpeg`: shortcuts for `convert` with the respective format. `jpeg` accepts a quality parameter in the range 0.0 to 1.0, where 0.0 represents the lowest and 1.0 the highest quality.
- `defaultJpegQuality (djq)`: Sets the JPEG compression quality to be used should the output image be a JPEG and no explicit quality parameter has been given. There is a configuration parameter `defaultJpegCompressionQuality` that allows you to specify this value if this operation is not executed.
- `removeMetadata (rm)`: Removes any metadata that might be associated with the image, such as EXIF or IPTC information. There is a configuration

parameter `preserveMetadata` that allows you to specify whether metadata should be kept if this operation is not executed.

- `progressiveMode (p)`: Sets the threshold (image size in pixel) at which the image should be encoded in progressive (JPEG) resp. interlaced (GIF, PNG) mode for faster perceived image display. There is a configuration parameter `defaultProgressiveThreshold` that allows you to specify this value if this operation is not executed.
- `unsharpMask (usm)`: Sharpen the image using an unsharp mask.

CMYK Images

JPEG images using the CMYK color model are converted to sRGB before further processing. The conversion utilizes a ICC color profile in order to map the CMYK colors to the sRGB color space as accurate as possible. When there is a suitable color profile embedded within the source image, that color profile is used for conversion. It is highly recommended to save a CMYK JPEG image with an embedded color profile before uploading it into the CMS.

If there is no embedded color profile, conversion falls back to a platform specific "generic" CMYK color profile. If the resulting colors are not acceptable there is the possibility to specify a custom ICC color profile for converting CMYK images w/o embedded color profile. All that is needed is to put a properties file

```
com/twelvemonkeys/imageio/color/icc_profiles.properties
```

into the classpath and define the key "GENERIC_CMYK" with the path to your profile, e.g.

```
GENERIC_CMYK=/usr/share/color/icc/MyGenericCMYKProfile.icc
```

Writing CMYK images is not supported. Moreover, writing image metadata is not supported for images originating from CMYK source images. Any metadata is removed before writing the image, as if the `removeMetadata (rm)` operation has been applied. This is done in the Blueprint CAE anyway in order to generate small and compact images.

A General Blob Transformation Framework

Image transformations make use of a more general binary object transformation framework. Within this framework, it is possible to implement any transformation on blobs you may think of. Transforming images is just a special case, albeit an important one.

The transformation framework resides within the package [com.coremedia.transform](#) and subpackages which define the framework API and contain an implementation

for image transformations. The central interface is the [BlobTransformer](#) with the `transformBlob` method:

```
public interface BlobTransformer {
    TransformedBlob transformBlob(Blob blob, String operations)
        throws IOException;
    boolean accepts(MimeType mimeType);
}
```

CAE Component Beans

Within the CAE Spring application context, a bean implementing the `BlobTransformer` interface is defined with the id `blobTransformer`. It is capable of transforming image blobs with the operations defined within the [ImageOperations](#) interface. This is done with the help of a [DispatchingBlobTransformer](#) bean with the id `imageTransformer`. This is the place where you can add your own image operations as described in the next section.

The `blobTransformer` also caches the transformed images on disk using a [CachingBlobTransformer](#). Moreover, it performs some load control so that many concurrent image transformation requests do not blow up the heap (see [ThrottlingBlobTransformer](#)). Please refer to the [Bean Definition Reference](#) for some more information.

Extending the Set of available Image Operations

The [DispatchingBlobTransformer](#) class is an implementation of the [BlobTransformer](#) interface. It consists of an [InputAdapter](#), a list of so called *processor* objects, and an [OutputAdapter](#). The [InputAdapter](#) converts the input blob into an internal representation (type parameter `State`) that is suitable for performing the desired transformations. The processors operate on this representation to perform their tasks. Finally, the [OutputAdapter](#) renders the internal representation back into an object implementing the [Blob](#) interface. This blob is then wrapped with a [TransformedBlob](#) which also remembers the original blob and the transformation string.

Processors are objects that perform the transformation operations. Processors implement one or more interfaces. Within these interfaces, methods providing the transformation operations are marked with the [@Operation](#) annotation.

By convention, the first parameter of methods implementing operations is the transformation state object (created by the [InputAdapter](#)). Operation methods manipulate this state object to perform their transformation task.

Any parameters of an operation are specified as additional method parameters and must be annotated with the [@Param](#) annotation. This annotation tells the [DispatchingBlobTransformer](#) about the name of the parameter (recall that operation parameters are specified by name rather than position). Furthermore, it allows you to specify a default value for the parameter, making it optional, and an alias as a

shorthand name. The `@Operation` annotation may optionally specify an alias for the operation.

For each operation within the transformation string, a `DispatchingBlobTransformer` tries each of its processors in turn and invokes the first one in the list that implements the operation. This way it's easy to extend the set of operations understood by a `DispatchingBlobTransformer` by simply adding another processor to the list that implements some new operations. And it is also possible to override some specific operation with a custom implementation by adding a custom processor at an earlier position in the list.

Let's assume you would like to extend the set of predefined image operations with a `sharpen` operation. You would start implementing the processor interface as follows:

```
package com.mycompany.transform;

import com.coremedia.transform.image.ImageTransformerState;
import com.coremedia.transform.dispatch.Operation;
import javax.imageio.IIOImage;

public interface SharpenerOperations {

    @Operation(alias="sh")
    void sharpen(ImageTransformerState<IIOImage> state,
        @Param(name="centerWeight", alias="cw", defaultValue = "1.0")
        float cw,
        @Param(name="neighbourWeight", alias="nw", defaultValue = "0.0")
        float nw
    );
}
```

Then you would implement this interface using the `javax.imageio` library:

```
package com.mycompany.transform;

import com.coremedia.transform.image.ImageTransformerState;
import javax.imageio.IIOImage;
import java.awt.image.BufferedImage;
import java.awt.image.ConvolveOp;
import java.awt.image.Kernel;
import java.util.Map;

public class Sharpener implements SharpenerOperations {

    @Override
    public void sharpen(ImageTransformerState<IIOImage> state,
        float cw,
        float nw) {
        BufferedImage img = (BufferedImage)
            state.getImage().getRenderedImage();
        float data[] = {
            nw, nw, nw,
            nw, cw, nw,
            nw, nw, nw
        };
        Kernel kernel = new Kernel(3, 3, data);
        ConvolveOp convolve = new ConvolveOp(kernel,
            ConvolveOp.EDGE_NO_OP, null);
        img = convolve.filter(img, null);
    }
}
```

```

    state.getImage().setRenderedImage(img);
  }
}

```

The final step is to add this processor to the list of processors in the Spring configuration:

```

<customize:append id="imageProcessorCustomizer"
                  bean="imageProcessors">
  <list>
    <bean class="com.mycompany.transform.SharpenImpl" />
  </list>
</customize:append>

```

Now you may use the sharpen operation within a transformation string:

```
r;a=23/g;a=2/b;a=70/sharpen:centerWeight=3.0;neighbourWeight=-0.25"/png
```

Exploiting operation and parameter aliases, the same transformation would read:

```
r;a=23/g;a=2/b;a=70/sh;cw=3.0;nw=-0.25"/png
```


5. Appendix

5.1 Customizer

A Customizer is a mechanism, which enables you to change an existing bean definition without touching the actual configuration file of the bean. Technically speaking, a Customizer is a `BeanPostProcessor` bean, which adjusts the bean during creation of the `ApplicationContext`. The declaration of a Customizer uses the "Extensible XML Authoring" (see <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/extensible-xml.html> for details) which enables you to write compact bean definitions.

Examples:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:customize="http://www.coremedia.com/2007/
    coremedia-spring-beans-customization"
  xsi:schemaLocation="
    http://www.coremedia.com/2007/
    coremedia-spring-beans-customization
    http://www.coremedia.com/2007/
    coremedia-spring-beans-customization.xsd">

  <bean id="someService" class="com.mycompany.Service">
    <property name="enable" value="false"/>
  </bean>
  <customize:replace id="enableSomeService" bean="someService"
    property="enable"
    custom-value="true"/>

</beans>
```

Here, the property `enable` of the bean `someService` is set to "true".

```
...
<customize:append id="addEntriesToSomeMap" bean="someMap">
<map>
  <entry key="key1" value="value1"/>
  <entry key="key2" value="value2"/>
</map>
</customize:append>
...
```

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

```
<bean id="myLoginInterceptor"
  class="my.interceptors.MyLoginInterceptor"/>
...
<customize:replace id="registerMyLoginInterceptor"
  bean="loginInterceptor"
  custom-ref="myLoginInterceptor"/>
```

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

Syntax

The syntax to define a Customizer are as follows (id attribute omitted):

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

or

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

or

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

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

The following operations are supported:

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

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

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

```
<customize:wrap id="wrapService" bean="service"
  wrapper-property="delegate">
```

```
<bean class="com.mycompany.WrapperService"/>
</customize:wrap>
```

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

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

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

5.2 Aspects

Aspects are a feature that allows you to add new functionality to existing content beans without modifying the content bean source code itself, either because the content bean source code is not available, or to create a reusable extension. When access to the content bean source code is available, using aspects is usually not necessary.

Terminology

Beans designed for extension by aspects are called *aspect aggregators* and implement the [com.coremedia.cae.aspect.AspectAggregator](#) interface. Typically, these will be content beans, extending [com.coremedia.cae.aspect.contentbean.AbstractAspectAggregatorContentBean](#), but this is not a requirement. In the sections to follow, it is assumed your aspect aggregators are content beans.

An *aspect* is a bean to be "attached" to or associated with an *aspect aggregator*. Aspects have a name, and an aspect aggregator instance can have at most one aspect bean with a given name associated with it.

Setting up the Aspect Infrastructure

Aspect aggregators must implement the [com.coremedia.cae.aspect.AspectAggregator](#) interface. To make existing content beans aspect aware, make sure that they inherit from [AbstractAspectAggregatorContentBean](#) (rather than [AbstractContentBean](#)) and adjust your parent content bean definitions. Also, aspect aggregators need an [AspectsProvider](#). The [CompoundAspectsProvider](#) in this example serves as a registry for plugins adding aspects to your content beans.

```
<bean id="aspectsProviders"
class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
    <list
value-type="com.coremedia.cae.aspect.provider.AspectsProvider"/>
  </property>
</bean>

<bean id="aspectsProvider"
class="com.coremedia.cae.aspect.provider.CompoundAspectsProvider">
  <property name="aspectsProviders" ref="aspectsProviders"/>
</bean>

<bean id="contentBeanBase" abstract="true"
class="AbstractAspectAggregatorContentBean">
  <property name="aspectsProvider" ref="aspectsProvider"/>
</bean>

<bean id="contentBeanFactory:YourType"
class="com.yourcompany.YourTypeContentBean" parent="contentBeanBase">
  . . .
</bean>
```

Example 5.1. Add aspect support to content beans

Registering Aspects

To create a new aspect, implement the [Aspect](#) interface and add the new behavior to this class, by adding bean properties, for instance. Choose an adequate `AspectProvider` to provide instances of these beans. For instance, for an aspect to be added to content beans, choose the [ContentBeanAspectsProvider](#). This provider needs its own content bean factory instance whose configuration will determine which content types the aspect should apply to.

```

<!-- factory to create aspect bean instances -->
<bean id="myAspectContentBeanFactory"

class="com.coremedia.objectserver.beans.SpringContentBeanFactory"/>

<!-- configuration to map MyDoctype to MyDoctypeAspectContentBean
-->
<bean name="myAspectContentBeanFactory:MyDoctype"
class="com.mycompany.MyDoctypeAspectContentBean"
scope="prototype">
  <property name="aggregatorContentBeanFactory"
ref="contentBeanFactory"/>
</bean>

<!-- aspects provider for the new aspect -->
<bean id="myAspectsProvider"

class="com.coremedia.cae.aspect.provider.ContentBeanAspectsProvider">

  <property name="contentBeanFactory"
ref="myAspectContentBeanFactory"/>
</bean>

<!-- register the aspects provider -->
<customize:append id="addMyAspectProvider" bean="aspectsProviders">

  <list>
    <ref bean="myAspectsProvider"/>
  </list>
</customize>

```

Example 5.2. Registering an aspects provider for content beans

On the other hand, to add aspects to arbitrary Java beans which are not content beans, use a [BeanFactoryAspectProvider](#) instead. The aspect implementation class should also implement the [AspectAggregatorAware](#) interface to have access to the aspect aggregator. Define your aspect bean as a prototype bean with name like `classNameOfAspectProvider:classNameOfAggregatorBean` where "classNameOfAggregatorBean" is the fully qualified class name of the bean the aspect will be applied to. You might use super classes or interfaces here as well.

```

<bean id="myAspectsProvider"

class="com.coremedia.cae.aspect.provider.BeanFactoryAspectsProvider"/>

<bean name="myAspectContentBeanFactory:com.mycompany.MyBean"
class="com.mycompany.MyBeanAspectImpl" scope="prototype">

```

Example 5.3. Definition of an aspects provider for arbitrary Java beans

```
...  
</bean>
```

Working with Aspects

Aspects added to aggregator beans by an aspects provider are available in code by calling the `getAspects` or `getAspectsByName` methods:

```
Collection<? extends Aspect> aspects = contentBean.getAspects();
```

```
Map<String, ? extends Aspect> aspectsByName =  
contentBean.getAspectsByName();
```

In templates, use the map returned by `getAspectsByName` to access an aspect of the aggregator by its name:

```
self.aspectByName['myAspect'].myProperty
```

5.3 Entity Resolver

Documents, such as templates, document type definitions or other XML files, need to address third-party DTDs, Schemas or Entities (in the following summarized as entities). In order to prevent problems with slow websites and to enhance offline functionality, CoreMedia XML utility classes in the `com.coremedia.xml` package (see the API documentation for details) support proxies in the classpath for such entities.

That is, you can simply use the original URL of an entity in your XML data, for example `http://www.w3.org/1999/xlink`, but the CoreMedia utilities will try to resolve against the classpath first. Proxies have to be stored with their original path, `/www.w3.org/1999/xlink.xsd`, in this example. CoreMedia provides some third-party proxies in the `cap-schema-bundle.jar` file. CoreMedia entities on the other hand are stored directly in `cap-schema-bundle.jar!/xml` due to backwards compatibility.

To keep it short, follow the following rules for entity resolving:

- Provide classpath proxies for external entities with a path mapping as described above.
- Provide your own entities via classpath.
- Use the CoreMedia XML utilities (especially `MarkupFactory` and `XMLUtil15`) because they offer out-of-the-box entity proxy support for third-party entities and class path support for project entities.

5.4 Content Placeholders

The pages of a typical CAE based website are composed of several content objects where each page fragment corresponds to one or more contents. For example, a teaser area on a page may be modeled from teaser documents that are placed in a link list property. When rendering the page, then the entire content structure is rendered by recursively applying the content beans to matching templates, for example a content of type `Teaser` is translated to a content bean `Teaser.class` that is rendered by a template `Teaser.jsp`.

There are situations where it may not be adequate to add a new content type for every piece of functionality that should be used on a website. This may be true when there is only one or a very few content instances of this type.

Example: Consider a website function "Current Weather" that displays the weather forecast for the user's current location. Another example would be a "Login" form that enables the user to login to or log out from the website. In order to enable an editor to add, remove or replace such functionality in a page, it is necessary to represent it as a content item. On the other hand it would be a huge overhead to add a content type "Weather" and a content type "Login".

Such functionality can be easily added to an application using the Substitution API. The basic idea behind the API is, that there is a generic content type that serves as a kind of placeholder. Content of this type must have a string property containing an identifier (for example `com.mycompany.weather`) that is internally used to render the real information that is represented by the document. This identifier is an arbitrary string, linking the content object to the substitution implementing the intended behavior. To avoid name clashes of logical identifiers, for instances with future project extensions, it is recommended to adopt the naming convention known from Java packages as shown here.

Example: Let's say that there is placeholder content type called `Action`

```
<DocType Name="Action">
  <StringProperty Name="id" Length="128"/>
</DocType>
```

with a corresponding content bean implementing this interface:

```
public interface Action {
    String getId();
}
```

Instead of rendering the `Action` bean using a template `Action.jsp`, a more special bean `Weather` could be rendered using a matching template `Weather.jsp`. This kind of substitution (for example an `Action` with an id `com.mycompany.weather` is substituted by an instance of bean `Weather`) is supported by the Substitution

API. Note that the bean resulting from the substitution can be of an arbitrary type, and does not need to implement any particular interface.

In order to define such substitution, simply add an `@Substitution` annotated method to any bean in the application context:

```
package com.mycompany.weather.handlers;
import com.coremedia.objectserver.view.substitution.*;
public class WeatherHandler {
    // ...
    // Substitution ID "com.mycompany.weather" is arbitrary,
    // but uses package naming conventions to avoid name
    // clashes.
    // It must match the property value in the
    // corresponding content object, whose content bean will
    // be substituted with this Weather bean during rendering
    // by the ${cm:substitute} function.
    @Substitution("com.mycompany.weather")
    public Weather createWeatherBean(Action original,
        HttpServletRequest request) {
        return new Weather(original,
            getCurrentWeather(request.getSession()));
    }
}
```

Example 5.4. Annotating a Substitution method

The (generic) template `Action.jsp` can perform this substitution by calling the JSP Expression Language function `cm:substitute` and dispatching the substitution result to its responsible template (for example `Weather.jsp`).

```
<%@ taglib prefix="cm"
    uri="http://www.coremedia.com/2004/objectserver-1.0-2.0"
%>
<!--@elvariable id="self" type="com.mycompany.Action"-->
<cm:include self="${cm:substitute(self.id, self,
pageContext.request)}"/>
```

Example 5.5. Use of cm:substitute in CMAAction.jsp

Using the `@Substitution` annotation isn't the only way to register a substitution. Consider a login example that requires a handler to perform the login action:

```
import com.coremedia.objectserver.view.substitution.*;
public class LoginHandler {
    // ...
    @RequestMapping("/{id}/login")
    public ModelAndView handleLogin(
        @PathVariable("id") Page page,
        @RequestParam("user") String user,
        @RequestParam("password") String password,
        HttpServletRequest request) {
```

Example 5.6. Registering a substitution programmatically

```
    LoginState state=processLogin(user, password,
request.getSession());
    ModelAndView result=HandlerHelper.createModel(page);
    SubstitutionRegistry.register("com.mycompany.login",
                                state, result);
    return result;
}
```

This example demonstrates the substitution from within a handler. The advantage in comparison to the annotation based approach is the fact that form data can be handled conveniently using the binding of Spring MVC.

In fact, the different approaches can be used in conjunction. An explicitly registered substitution (using the `SubstitutionRegistry` service) has precedence over the annotation approach. Thus, `@Substitution` can be used as a fallback in case that there hasn't been a registration by a handler.

Spring Forms

When using the Spring Form tag library, then it is necessary to have the form beans stored under certain names (other than `self`) in the request scope. For this reason, an optional `ModelAttribute` can be specified in the `@Substitution` annotation. When this is done, then the substituted bean is stored under this name in the request. Example: An annotation `@Substitution(value="com.mycompany.weather", ModelAttribute="weatherBean")` will cause the substituted bean to be stored in the request as an attribute `weatherBean`.

5.5 Configuration Property Reference

Table 5.1. Configuration Properties

Property	Value	Default	Description
repository.url	string		This property determines where to get the IOR of the <i>Content Server</i> (format: <code>http://<server>:<port>/coremedia/ior</code>). <code><server></code> must be the name of the <i>Content Server</i> host. For <code><port></code> you have to set the server's web server HTTP port.
repository.user	string		The user in whose name the connection to the <i>Content Server</i> is established. By default, only the user <code>webserver</code> is allowed to use the <code>webserver</code> login service required for a <i>Content Application Engine</i> .
repository.domain	string		The domain of the user indicated above
repository.password	string		The password of the user indicated above
repository.workflow	true/false	false	Defines if a connection with the <i>Workflow Server</i> should be established. For most content application, this is not necessary.
repository.workflow.url	string		This property specifies the IOR of the <i>Workflow Server</i> , if the IOR configured at the <i>Content Server</i> should not be used (format: <code>http://<server>:<port>/workflow/ior</code>).
repository.heapCacheSize	number	20000000	The total number of bytes used by the main memory cache of the <i>Content Application Engine</i> . For 32 bit JVMs this value is exact, for 64 bit JVMs, the actual memory consumption may be up to 2 times the configured value.
repository.blobCacheSize	number	32000000	The total number of bytes used by the disk cache of the <i>Content Application Engine</i> . This cache is used for

Property	Value	Default	Description
			storing blobs downloaded from the <i>Content Server</i> . The default size is 32 MB. The configured directory for the blob cache must be large enough to hold all cached blobs and potentially leftover files from earlier forced shutdowns of the <i>Content Application Engine</i> web application.
reposit- ory.blob- StreamingS- iz- eThreshold	number	131072	The minimum size of streamed blobs in bytes. blobs less than or equal to this size will be downloaded completely to disk before the first byte can be read. Larger blobs will be downloaded in the background.
reposit- ory.blob- Streaming- Threads	number	2	The number of threads reserved for streaming blob.
reposit- ory.max- CachedBlob- Size	number		The maximum size of blobs that are cached on the local disk. Larger blobs are downloaded from the <i>Content Server</i> on every request.
reposit- ory.blob- CachePath	string	the Java tempor- ary directory	The directory in which cached blobs are stored. Make sure that the file system for this directory is large enough. Note that forced shutdowns of the <i>Content Application Engine</i> web application may result in leftover files in this directory, which should be cleared while the CAE is down. The configured directory may be shared with other CAEs, because the actual cache content is placed in dynamically allocated subdirectories.
viewdis- patch- er.cache.en- abled	true/false	true	Defines if the caching of view lookups is enabled. Disabling might be useful when developing templates.
secur- ity.csrf- preven-	true/false	false	Enables protection against cross-site request forgery attacks by generating anti-CSRF tokens for authenticated users, adding such tokens to

Property	Value	Default	Description
tion.enabled			all forms created with the Spring Form tag library, and validating tokens for unsafe requests.

5.6 Bean Definition Reference

All following files are loaded into the application context automatically with `cae-component`, except for `controller-services.xml`, which is provided for backwards compatibility.

CAE Component Configuration

Service or Extension Point Definition	Type
<code>richtextMarkupView</code>	<code>XmlMarkupView</code>
<code>blobView</code>	<code>MultiRangeBlobView</code>
<code>viewHookEventView</code>	<code>ViewHookEventView</code>
<code>programmedViews</code>	<code>Map<String, View></code> Extension point to register programmed views, initialized to (<code>Markup := richtextMarkupView</code> , <code>Blob := blobView</code> , <code>ViewHookEvent := viewHookEventView</code>) by a customizer with order 100.
<code>fallbackViewRepository</code>	Default <code>ViewRepository</code> implementation, loading templates from <code>/WEB-INF/templates-fallback</code> and using <code>richtextMarkupView</code> , <code>blobView</code> , <code>errorView</code> , <code>viewExceptionRenderer</code> , and <code>viewEngines</code> .
<code>templateLocations</code>	<code>Map<String, String></code> Extension point to register additional template locations with <code>templateViewRepositoryProvider</code> , initialized to "default" := <code>/WEB-INF/templates</code> .
<code>templateLocationPatterns</code>	<code>List<String></code> Extension point to register additional template location path patterns with <code>templateViewRepositoryProvider</code> . In each pattern, "%s" will be replaced with the view repository name to resolve a location.
<code>templateViewRepositoryProvider</code>	Default <code>ViewRepositoryProvider</code> implementation, initialized with <code>programmedViews</code> , <code>viewDecorators</code> , and <code>viewEngines</code> . It will lookup view repositories using <code>templateLocations</code> and <code>templateLocationPatterns</code> . View repository name "fallback" will be resolved to the <code>fallbackViewRepository</code> . <code>viewRepositoryProviders</code> are initialized to try view repository names "default" and "fallback", if no view is found.

Table 5.2. META-INF/coremedia/component-cae.xml in artifact cae-component

Views

Service or Extension Point Definition	Type
viewEngines	Map<String, ViewEngine > Extension point to register custom view engines for template file extensions, initialized to ("jsp" := WebappResourceViewEngine , "ftl" := freemarkerViewEngine) by a customizer with order 100.
viewDecorators	List< ViewDecorator > Extension point to register custom view decorators, initialized to exceptionDecorator , if view.errorhandler.enabled=true , and debugDecorator , if view.debug.enabled=true .
viewRepositoryNameProviders	List< ViewRepositoryNameProvider > Extension point to register custom view repository name providers, initialized to an implementation returning "default" and "fallback".
viewRepositoryProviders	List< ViewRepositoryProvider > Extension point to register custom view repository providers, initialized to templateViewRepositoryProvider .
renderNodeDecoratorProviders	List< RenderNodeDecoratorProvider > Extension point to register custom render node decorator providers.
viewResolverAttributes	Map<String, Object> Extension point to register custom view resolver attributes, which will be copied into the request attributes for each request, before rendering a view.
viewResolver	ModelAwareViewResolver

Table 5.3. *com/core-media/cae/view-services.xml* in artifact *cae-viewservices-impl*

Service or Extension Point Definition	Type
viewingHandlerExceptionResolver	ViewingHandlerExceptionResolver
errorView	ErrorView

Table 5.4. *com/core-media/cae/view-error-services.xml* in artifact *cae-viewservices-impl*

Service or Extension Point Definition	Type
viewExceptionRenderer	ViewExceptionRenderer
httpErrorView	HttpErrorView
exceptionDecorator	ExceptionHandlerViewDecorator , will only be registered with <code>viewDecorators</code> , if property <code>view.errorhandler.enabled=true</code> .
exceptionDecoratorAcceptBeanClasses	List<Class> Configuration for <code>exceptionDecorator</code> , empty by default.
exceptionDecoratorRejectBeanClasses	List<Class> Configuration for <code>exceptionDecorator</code> , empty by default.
exceptionDecoratorAcceptViews	List<java.util.regex.Pattern> Configuration for <code>exceptionDecorator</code> , empty by default.
exceptionDecoratorRejectViews	List<java.util.regex.Pattern> Configuration for <code>exceptionDecorator</code> , empty by default.

Service or Extension Point Definition	Type
debugDecorator	DebugViewDecorator , will only be registered with <code>viewDecorators</code> , if property <code>view.debug.enabled=true</code> .
debugDecoratorAcceptBeanClasses	List<Class> Configuration for <code>debugDecorator</code> , empty by default.
debugDecoratorRejectBeanClasses	List<Class> Configuration for <code>debugDecorator</code> , empty by default.
debugDecoratorAcceptViews	List<java.util.regex.Pattern> Configuration for <code>debugDecorator</code> , empty by default.
debugDecoratorRejectViews	List<java.util.regex.Pattern> Configuration for <code>debugDecorator</code> , empty by default.

Table 5.5. `com/core-media/cae/view-development-services.xml` in artifact `cae-viewservices-impl`

Service or Extension Point Definition	Type
freemarkerViewEngine	ViewEngine to render FreeMarker templates.
freemarkerModels	Map<String, Class> Extension point to register additional <code>TemplateModel</code> implementations, which will be made available to FreeMarker templates. Initialized to the set of CAE-defined directives.

Table 5.6. *com/core-media/cae/view-free-marker-services.xml* in artifact *cae-viewservices-impl*

Unified API

Service or Extension Point Definition	Type
connectionParameters	Map<String, Object> Configuration for the <code>connection</code> bean.
connection	CapConnection
contentRepository	ContentRepository
userRepository	UserRepository
workflowRepository	WorkflowRepository
worklistService	WorklistService

Table 5.7. *com/core-media/cap/common/uapi-services.xml* in artifact *cap-unified-api*

Service or Extension Point Definition	Type
contentIdScheme	ContentIdScheme
contentBlobIdScheme	ContentBlobIdScheme
memberIdScheme	MemberIdScheme
contentTypeIdScheme	ContentTypeIdScheme
contentPropertyIdScheme	ContentPropertyIdScheme

Table 5.8. *com/core-media/cae/uapi-services.xml* in artifact *cae-util*

Data Views

Service or Extension Point Definition	Type
dataViewFactory	ConfigurableDataViewFactory implementation, loading its data view definitions from <code>dataViewDefinitionLocations</code>
dataViewDefinitionLocations	List<String> Extension point to register data view factory configuration file patterns, initialized to <code>classpath:/framework/dataviews/**/*.xml, /WEB-INF/dataviews/**/*.xml</code> .

Table 5.9. `com/core-media/cae/dataview-services.xml` in artifact `cae-contentbeanservices-impl`

Content Beans

Service or Extension Point Definition	Type
contentBeanFactory	ContentBeanFactory , creating content beans from prototype beans with name "contentBeanFactory:<content_type>".
contentBeanIdScheme	ContentBeanIdScheme

Table 5.10. `com/core-media/cae/content-bean-services.xml` in artifact `cae-contentbeanservices-impl`

Caching

Service or Extension Point Definition	Type
cache	Cache instance created for <code>connection</code>

Table 5.11. `com/core-media/cache/cache-services.xml` in artifact `coremedia-cache`

Link Generation

Service or Extension Point Definition	Type
linkSchemes	List< LinkScheme > Extension point to register link schemes with <code>linkFormatter</code> .

Table 5.12. `com/core-media/cae/link-services.xml` in artifact `cae-linkservices-impl`

Service or Extension Point Definition	Type
linkTransformers	List<LinkTransformer> Extension point to register link transformers with linkFormatter.
linkFormatter	LinkFormatter

IDs

Service or Extension Point Definition	Type
idProvider	IdProvider , initialized with the registered idSchemes.
idSchemes	List<IdScheme> Extension point to register ID schemes with idProvider. Initialized to contentIdScheme, contentBeanIdScheme, contentBlobIdScheme, memberIdScheme, contentTypeIdScheme, and contentPropertyIdScheme by a customizer with order 100.

Table 5.13. *com/core-media/id/id-services.xml* in artifact *coremedia-id*

Handlers

Service or Extension Point Definition	Type
bindingConverters	Set<?> (Converter or GenericConverter) Extension point to register custom converters to bind request path variables to handler method parameters.
httpMessageConverters	List< HttpMessageConverter > Extension point to register custom HTTP message converters to parse HTTP request body content or generate HTTP response body content.
bindingPropertyEditorRegistrars	List< PropertyEditorRegistrar > Extension point to register custom property editor registrars (which in turn will register property editors) to bind form fields to bean properties.

Table 5.14. *com/core-media/cae/handler-services.xml* in artifact *cae-handlerservices-impl*

Service or Extension Point Definition	Type
<code>handlerInterceptors</code>	List< HandlerInterceptor > Extension point to register handler interceptors, which will be applied to all handlers.
<code>idContentBeanConverter</code>	Converter to convert numeric IDs to method parameters of type ContentBean . An application must register this bean with <code>bindingConverters</code> explicitly, in order to use it.
<code>idGenericContentBeanConverter</code>	GenericConverter to convert between numeric IDs and subtypes of ContentBean . This converter subsumes the functionality provided by <code>idContentBeanConverter</code> . An application must register this bean with <code>bindingConverters</code> explicitly, in order to use it.
<code>idContentBeanPropertyEditor</code>	<code>java.beans.PropertyEditor</code> to convert between numeric IDs and ContentBeans . An application must register this bean with <code>bindingPropertyEditorRegistrars</code> explicitly using a PropertyEditorRegistrar , in order to use it.

MIME Type Mappings

Service or Extension Point Definition	Type
<code>mimeTypeService</code>	MimeTypeService , providing methods related to MIME types and file extensions. Note: The implementation class of this bean is deprecated and will be replaced by <code>TikaMimeTypeService</code>
<code>mimePropertiesFileLocations</code>	List< Resource > Extension point to register locations of property files containing MIME type mappings. <code>classpath:/com/coremedia/mimetype/mime-default.properties</code> and <code>/WEB-INF/mime.properties</code> . This bean is deprecated and will be removed in a future release.
<code>tikaMimeTypeService</code>	TikaMimeTypeService , providing methods related to MIME type detection and mapping to file extensions. The implementation is based on Apache Tika. Configured with the following properties:

Table 5.15. `com/coremedia/mimetype/mime-type-service.xml` in artifact `coremedia-common`

Service or Extension Point Definition	Type
	<p><code>mimeTypeResourceNames</code> A comma-separated list of resource names of Tika Mime-Info configuration files. Set this property by Spring Environment property <code>mimeTypeService.mimeTypeResourceNames</code>.</p> <p><code>tikaConfig</code> An optional custom Tika configuration. Set this property by Spring Environment property <code>mimeTypeService.tikaConfig</code>. The value of this property must be a Spring Resource location (e.g. <code>file:/path/to/local/file</code>), or null. If a custom Tika configuration is set, the <code>mimeTypeResourceNames</code> configuration has no effect.</p>

Security

Service or Extension Point Definition	Type
<code>csrfTokenManagement</code>	CsrfPreventionManagement , providing methods handling anti-CSRF tokens.

Table 5.16. `com/core-media/cae/security-services.xml` in artifact `cae-util`

Image Transformations

Service or Extension Point Definition	Type
<code>blobTransformer</code>	CachingBlobTransformer , handling concurrent image transformation requests, caching image transformation results. Delegates cache misses to the bean <code>throttlingTransformer</code> .
<code>throttlingBlobTransformer</code>	ThrottlingBlobTransformer , handling concurrent image transformation requests, performing some basic load control. Delegates the actual transformation work to the bean imageTransformer .
<code>imageTransformer</code>	DispatchingBlobTransformer , the blob transformer actually performing image transformations. Holds a list of processor objects, initialized with the list bean imageProcessors .
<code>imageProcessors</code>	List<?>, extension point to register your own image processors, implementing additional image operations. Initialized with a single processor, imageOperations .

Table 5.17. `com/core-media/transform/blob-transformer.xml` in artifact `coremedia-transform`

Service or Extension Point Definition	Type
imageOperations	Java2DImageOperations , implementing the ImageOperations interface using the <code>javax.imageio</code> library. You may reuse this bean and its operations when implementing your own image operations.
imageTransformerInputAdapter	<p>Java2DInputAdapter, input adapter used by imageTransformer.</p> <p>Since the reading and decoding of an image consumes a significant amount of processing time, this input adapter caches loaded images in memory. When another variant is then requested for the same image, this variant can be computed much faster.</p> <p>Configured with the following properties:</p> <p><code>cache</code> the cache instance to use, configured with 100MB heap capacity for the configured cache class. You may overwrite this with the property <code>com.coremedia.transform.loadedImageCacheCapacity</code> within your <code>application.properties</code>.</p> <p><code>cacheClass="com.coremedia.transform.image.java2d.LoadedImageCacheKey"</code></p>
imageTransformerOutputAdapter	<p>Java2DOutputAdapter, output adapter used by imageTransformer. Configured with the following properties:</p> <p><code>preserveMetadata=false</code></p> <p><code>defaultProgressiveThreshold=10000</code></p> <p><code>defaultJpegCompressionQuality=0.75</code></p>
imageTransformerConversionService	<code>org.springframework.core.convert.ConversionService</code> , conversion service used by the imageTransformer to convert operation arguments to the required method parameter type. Extension point to register your own type converters.

Controllers

`controller-services.xml` is not imported automatically and is only provided for backwards compatibility with existing controller implementations.

Table 5.18. `com/core-media/cae/controller-services.xml` in artifact `cae-handlerservices-impl`

Service or Extension Point Definition	Type
<code>controllerMappings</code>	<code>Map<String, Controller></code> Extension point to register controllers handling requests with the given path prefix.
<code>controllerInterceptors</code>	Alias for <code>handlerInterceptors</code> , which will be applied to all controllers in <code>controllerMappings</code> .

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 <i>CAE Feeder</i> makes content beans searchable by sending their data to the <i>Search Engine</i> , which adds it to the index.
Content Application Engine (CAE)	<p>The <i>Content Application Engine (CAE)</i> is a framework for developing content applications with <i>CoreMedia CMS</i>.</p> <p>While it focuses on web applications, the core frameworks remain usable in other environments such as standalone clients, portal containers or web service implementations.</p> <p>The CAE uses the Spring Framework for application setup and web request processing.</p>
Content Bean	A content bean defines a business oriented access layer to the content, that is managed in <i>CoreMedia CMS</i> and third-party systems. Technically, a content bean is a Java object that encapsulates access to any content, either to <i>CoreMedia CMS</i> content items or to any other kind of third-party systems. Various <i>CoreMedia</i> components like the <i>CAE Feeder</i> 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	<p>The <i>Content Delivery Environment</i> is the environment in which the content is delivered to the end-user.</p> <p>It may contain any of the following modules:</p> <ul style="list-style-type: none">→ <i>CoreMedia Master Live Server</i>→ <i>CoreMedia Replication Live Server</i>→ <i>CoreMedia Content Application Engine</i>→ <i>CoreMedia Search Engine</i>→ <i>Elastic Social</i>

	<ul style="list-style-type: none"> → <i>CoreMedia Adaptive Personalization</i>
Content Feeder	The <i>Content Feeder</i> is a separate web application that feeds content items of the CoreMedia repository into the <i>CoreMedia Search Engine</i> . Editors can use the <i>Search Engine</i> to make a full text search for these fed items.
Content item	In <i>CoreMedia CMS</i> , 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 <i>Content Management Environment</i> is the environment for editors. The content is not visible to the end user. It may consist of the following modules: <ul style="list-style-type: none"> → <i>CoreMedia Content Management Server</i> → <i>CoreMedia Workflow Server</i> → <i>CoreMedia Importer</i> → <i>CoreMedia Site Manager</i> → <i>CoreMedia Studio</i> → <i>CoreMedia Search Engine</i> → <i>CoreMedia Adaptive Personalization</i> → <i>CoreMedia CMS for SAP Netweaver® Portal</i> → <i>CoreMedia Preview CAE</i>
Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.
Content Repository	<i>CoreMedia CMS</i> 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	<i>Content Server</i> is the umbrella term for all servers that directly access the CoreMedia repository: <p><i>Content Servers</i> are web applications running in a servlet container.</p> <ul style="list-style-type: none"> → <i>Content Management Server</i> → <i>Master Live Server</i> → <i>Replication Live Server</i>

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 <i>CoreMedia CMS</i> . 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	<i>Controm Room</i> is a <i>Studio</i> plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other <i>Studio</i> users.
CORBA (Common Object Request Broker Architecture)	<p>The term <i>CORBA</i> 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.</p> <p>CORBA programs communicate using the standard IIOP protocol.</p>
CoreMedia Studio	<p><i>CoreMedia Studio</i> 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.</p> <p>As a modern web application, <i>CoreMedia Studio</i> is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.</p>
Dead Link	A link, whose target does not exist.
DTD	<p>A Document Type Definition is a formal context-free grammar for describing the structure of XML entities.</p> <p>The particular DTD of a given Entity can be deduced by looking at the document prolog:</p> <pre><!DOCTYPE coremedia SYSTEM "http://www.coremedia.com/dtd/coremedia.dtd"</pre> <p>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.</p>
Elastic Social	<i>CoreMedia Elastic Social</i> is a component of <i>CoreMedia CMS</i> that lets users engage with your website. It supports features like comments, rating, likings on your website. <i>Elastic Social</i> is integrated into <i>CoreMedia Studio</i> so editors can moderate user generated content from their common workplace. <i>Elastic Social</i> 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 <i>Best current practice</i> (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, <i>Interoperable Object Reference</i> refers to the name with which a CORBA object can be referenced.
Jangaroo	<i>Jangaroo</i> 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 <i>Master Live Server</i> is the heart of the <i>Content Delivery Environment</i> . It receives the published content from the <i>Content Management Server</i> and makes it available to the CAE. If you are using the <i>CoreMedia Multi-Site Management Extension</i> you may use multiple <i>Master Live Server</i> 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	<p>In relation to CoreMedia, properties have two different meanings:</p> <p>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.</p> <p>In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.</p>
Replication Live Server	The aim of the <i>Replication Live Server</i> is to distribute load on different servers and to improve the robustness of the <i>Content Delivery Environment</i> . The <i>Replication Live Server</i> is a complete Content Server installation. Its content is an replicated image of the content of a <i>Master Live Server</i> . The <i>Replication Live Server</i> updates its database due to change events from the <i>Master Live Server</i> . You can connect an arbitrary number of <i>Replication Live Servers</i> to the <i>Master Live Server</i> .
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 <i>Active Delivery Server</i> . 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	<p>A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In <i>CoreMedia CMS</i> a site especially consists of a site folder, a site indicator and a home page for a site.</p> <p>A typical site also has a master site it is derived from.</p>

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 <code>CMsite</code> .
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	<p>In CoreMedia, JSPs used for displaying content are known as Templates.</p> <p>OR</p> <p>In <i>Blueprint</i> a template is a predeveloped content structure for pages. Defined by typically an administrative user a content editor can use this template to quickly create a complete new page including, for example, navigation, pre-defined layout and even predefined content.</p>
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes web application	The <i>User Changes</i> web application is a <i>Content Repository</i> listener, which collects all content, modified by <i>Studio</i> users. This content can then be managed in the <i>Control Room</i> , 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	<p>In general <i>CoreMedia CMS</i> always guarantees link consistency. But links can be declared with the <i>weak</i> attribute, so that they are not checked during publication or withdrawal.</p> <p>Caution! Weak links may cause dead links in the live environment.</p>
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 <i>CoreMedia Workflow Server</i> 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. <i>CoreMedia Studio</i> allows you to export content items in the XLIFF format and to import the files again after translation.

Index

A

- architecture, 15
- aspects, 125
 - register, 126
 - setup, 125
- association types, 36

B

- Beangenerator, 23
 - structure of generated code, 26
- blob transformation, 117
- blobTransformer, 118

C

- cache, 16
- cache sizes, 46
- caching
 - overview, 16
- CAE, 19
 - architecture, 15
 - components, 14
 - connecting with Content Server, 20
 - MVC model, 15
 - purging disk cache, 21
 - use cases, 14
- CAE Developer Toolbox, 111
- CAE web application
 - Ajax requests, 54
 - cache browser, 111
 - cache statistics, 111
 - error pages, 108
 - errors, 107
 - link schemes, 61
 - links, 59
 - multiple view repositories, 68

- properties, 132
- request handling, 50
- template inclusion, 76
- template output escaping, 75
- uncaught exceptions, 107
- unit testing, 104
- views, 65
 - writing templates, 75
- CMYK, 117
- content bean, 23, 94
 - equality, 28
- content beans, 23
 - dependencies, 48
 - generation, 23
 - pattern, 28
- customizer, 122
 - append bean, 123
 - replace bean, 123

D

- data view, 16, 31
- data view cache, 16
- data views, 31
 - association types, 36
 - auto completion, 38
 - definition, 32
 - design guidelines, 36
 - lifecycle, 35

E

- entity resolver, 128

H

- handler, 50

I

- image transformation, 115
 - adding own operations, 118
 - format, 115
 - supported operations, 116
- include, 76

L

- link, 59

[LinkListProperty](#), 27

M

[MVC model](#), 15

P

[placeholders](#), 129

S

[Spring configuration](#), 29

[Spring framework](#), 18, 29

[Substitution API](#), 129

T

[test framework](#), 104

U

[Unified API cache](#), 16

[using data views](#), 34