COREMEDIA CONTENT CLOUD

Content Application Developer Manual

COREMEDIA

List of Figures

List of Tables

List of Examples

1. Preface

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

- In Chapter 2, *Overview* [14] you will get an overview of the CAE and its concepts.
- In Chapter 3, *Administration and Operation* [20] you will learn some administrative tasks.
- In Chapter 4, *Development* [29] 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 Command line entries Parameter and values Class and method names Packages and modules	Courier new	`cm systeminfo start`
Menu names and entries	Bold, linked with |	Open the menu entry **Format|Normal**
Field names CoreMedia Components Applications	Italic	Enter in the field *Heading* The *CoreMedia Component* Use *Chef*
Entries	In quotation marks	Enter "On"
(Simultaneously) pressed keys	Bracketed in "<>", linked with "+"	Press the keys <Ctrl>+<A>
Emphasis	Italic	It is *not* saved
Buttons	Bold, with square brackets	Click on the [OK] button
Code lines in code examples which continue in the next line	\	`cm systeminfo \` `-u user`

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

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

Table 1.2. Pictographs

1.3 CoreMedia Services

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

NOTE   ⓘ

CoreMedia User Orientation for CoreMedia Developers and Partners

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

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

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

1.3.1 Registration

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

1.3.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

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

https://releases.coremedia.com/cmcc-12

Refer to our Blueprint Github mirror repository for recommendations to upgrade the workspace either via Git or patch files. You will also find how-tos for upgrading the system on our documentation website.

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

Maven artifacts

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

https://repository.coremedia.com

You have to add your CoreMedia credentials to your Maven settings file as described in section Section 3.1, "Prerequisites" in *Blueprint Developer Manual* .

npm packages

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

https://npm.coremedia.io

Your pnpm client first needs to be logged in to be able to utilize the registry (see Section 3.1, "Prerequisites" in *Blueprint Developer Manual*).

License files

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

1.3.3 Documentation

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

https://documentation.coremedia.com

The manuals have the following content and use cases:

Manual	Audience	Content
Adaptive Personalization Manual	Developers, architects, administrators	This manual describes the configuration of and development with *Adaptive Personalization*, the CoreMedia module for personalized websites, and Client-Side Personalization. You will learn how to configure the GUI used in *CoreMedia Studio*, how to use predefined contexts and how to develop your own extensions.
Analytics Connectors Manual	Developers, architects, administrators	This manual describes how you can connect your CoreMedia website with external analytic services, such as Google Analytics.
Blueprint Developer Manual	Developers, architects, administrators	This manual gives an overview over the structure and features of *CoreMedia Content Cloud*. It describes the content type model, the *Studio* extensions, folder and user rights concept and many more details. It also describes administrative tasks for the features. It also describes the concepts and usage of the project workspace in which you develop your CoreMedia extensions. You will find a description of the Maven structure, the virtualization concept, learn how to perform a release and many more.
Connector Manuals	Developers, administrators	This manuals gives an overview over the use cases of the eCommerce integration. It describes the deployment of the Commerce Connector and how to connect it with the CoreMedia and eCommerce system.

Manual	Audience	Content
Content Application Developer Manual	Developers, architects	This manual describes concepts and development of the *Content Application Engine (CAE)*. You will learn how to write Freemarker templates that access the other CoreMedia modules and use the sophisticated caching mechanisms of the CAE.
Content Server Manual	Developers, architects, administrators	This manual describes the concepts and administration of the main CoreMedia component, the *Content Server*. You will learn about the content type model which lies at the heart of a CoreMedia system, about user and rights management, database configuration, and more.
Deployment Manual	Developers, architects, administrators	This manual describes the concepts and usage of the CoreMedia deployment artifacts. That is the deployment archive and the Docker setup. You will also find an overview of the properties required to configure the deployed system.
Elastic Social Manual	Developers, architects, administrators	This manual describes the concepts and administration of the *Elastic Social* module and how you can integrate it into your websites.
Frontend Developer Manual	Frontend Developers	This manual describes the concepts and usage of the Frontend Workspace. You will learn about the structure of this workspace, the CoreMedia themes and bricks concept, the CoreMedia Freemarker facade API, how to develop your own themes and how to upload your themes to the CoreMedia system.
Headless Server Developer Manual	Frontend Developers, administrators	This manual describes the concepts and usage of the *Headless Server*. You will learn how to deploy the Headless Server and how to use its endpoints for your sites.
Importer Manual	Developers, architects	This manual describes the structure of the internal CoreMedia XML format used for storing data, how you set up an *Importer* application and how you define the transformations that convert your content into CoreMedia content.

Manual	Audience	Content
Multi-Site Manual	Developers, Multi-Site Administrators, Editors	This manual describes different otions to desgin your site hierarchy with several languages. It also gives guidance to avoid common pitfalls during your work with the multi-site feature.
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of *CoreMedia Studio*. You will learn about the underlying concepts, how to use the development environment and how to customize *Studio* to your needs.
Studio User Manual	Editors	This manual describes the usage of *CoreMedia Studio* for editorial and administrative work. It also describes the usage of the *Adaptive Personalization* and *Elastic Social* GUI that are integrated into *Studio*.
Studio Benutzerhandbuch	Editors	The Studio User Manual but in German.
Supported Environments	Developers, architects, administrators	This document lists the third-party environments with which you can use the CoreMedia system, Java versions or operation systems for example.
Unified API Developer Manual	Developers, architects	This manual describes the concepts and usage of the *CoreMedia Unified API*, which is the recommended API for most applications. This includes access to the content repository, the workflow repository and the user repository.
Utilized Open Source Software & 3rd Party Licenses	Developers, architects, administrators	This manual lists the third-party software used by CoreMedia and lists, when required, the licence texts.

Manual	Audience	Content
Workflow Manual	Developers, architects, administrators	This manual describes the *Workflow Server*. This includes the administration of the server, the development of workflows using the XML language and the development of extensions.

Table 1.3. CoreMedia manuals

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

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

CoreMedia's training department provides you with the training for your CoreMedia projects either on our online learning platform (CoreMedia Enablement, live online 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 Core-Media 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:

Support checklist

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

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

Log files

Which Log File?

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

Where do I Find the Log Files?

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

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

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

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

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

1.4 Changelog

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

Section	Version	Description

Table 1.4. Changes

2. Overview

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.

Content Beans

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*.

Caching and data views

Based on the types of the content beans and/or data views, suitable views are chosen in order to render the backend 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.

Views

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.

Component	Description
Content Application Engine web application	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.
Preview-based Editing	A simple framework to make a preview website editable.

Table 2.1. Components of the CAE framework

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 *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" [55] 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" [30] 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" [69] 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" [131] 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" [35] 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. This section covers *CAE* configuration options related to the server communication and the basic cache configuration.

3.1 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.

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

Table 3.1. Connection properties

3.2 Configuring Cache Sizes

For every *CAE* component you can configure the following caches:

3.2.1 In-memory caching (UAPI Cache)

You can configure the size of the in-memory Unified API cache and set the temporary folder on the disk to be used for holding cached blobs. Note, that the blobs saved in this folder are accessed and used solely by the UAPI cache itself and therefore their lifecycle is no longer than of UAPI cache, that is, blob files in this folder become obsolete after system shutdown.

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

Property	Description
`reposit ory.heapCacheSize`	This property indicates the number of bytes used for the main memory cache of the *Unified API* embedded in the *Content Application Engine*. 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 *CAE*s run in a single application server, the caches are kept separate and the configured cache sizes add up.
`repository.blob CacheSize`	This property defines the size of the disk cache for blobs. The blobs are cached temporarily and are garbage collected if no more needed. On the *CAE* restart the cached data is getting lost. Make sure to provide enough disk space for caching.
`repository.blob CachePath`	This property defines the location of the blob cache. Multiple *CAE*s may share the same directory for the blob cache. Again, the cache sizes add up.

Table 3.2. Properties for UAPI and temp blob cache

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*.

For more details on UAPI Cache implementation please consult Section 4.10, "Caching" in *Unified API Developer Manual* for details.

3.2.2 File System Cache for Transformed Image Blobs

> **CAUTION**
>
> This cache can only be used by one JVM. So, you cannot share the cache between several CAEs.

You can configure a separate cache where results of image transformations are stored persistently and can survive *CAE* restarts. Enabling this cache helps to solve the problem of high CPU load in the first hours of operation after restart. This could happen because the transformation of large image blobs to compute scaled crops can be quite expensive. After system restart, the in-memory cache (UAPI cache) is empty, and without this file system cache all needed images would have to be transformed again.

> **NOTE**
>
> Due to the specifics of file locking on Windows, the timely and correct deletion/rename of files and directories on this operation system cannot be guaranteed. Therefore, the *file system transformed blob cache* must not be used on Windows systems.

The configuration properties for the *transformed blob cache* are listed in the table:

Property	Description
`com.coremedia.trans form.blobCache.size`	The maximum allowed size that the *transformed blobs cache* can occupy on the disk. Note that the file system overhead for storing the files does not

Property	Description
	count towards this value. So the physical space that has to be reserved on the disk for the cache has to be slightly higher than value of this configuration property.
	If several concurrent threads write large blobs at the same time, the deletion of the folder with the old unused files can be postponed for later, thus this is the second reason why the maximum allowed cache size can grow slightly higher than this configuration property. The size of such deviation depend on the blobs size as well as the amount of parallel threads.

`com.coremedia.trans`
`form.blob`
`Cache.basePath`
	The path to the transformed blobs cache. If not set, then this cache is deactivated and the results of blob transformations are stored using *UAPI cache*. It is recommended to set this property to a folder that is not cleared during *CAE* restart, for example `/var/cache/coremedia/persistent-`
	`transformed-blobcache`.

Table 3.3. Properties for transformed blob cache

Pay attention, not to purge the directory with the *transformed blobs cache* on *CAE* restarts.

3.3 Configuring HTTP Cache-Control

HTTP Caching improves the website performance by instructing CDNs and clients to reuse previously fetched resources. The Cache-Control HTTP header offers fine-grained instructions for CDNs and HTTP clients on how to cache. Websites and eCommerce integrations always need to balance between efficiency and accuracy. With the CoreMedia Cache Control API and default implementation, projects have full control over caching behavior of content delivered by *CoreMedia Content Cloud*.

Aim of caching

HTTP Cache-Control headers generated by the CAE can be configured very precisely. See Section 3.1.3, "Http Cache Control Properties" in *Deployment Manual* for all configuration options.

Additionally, the Cache-Control header's `max-age` directive is computed taking the content validity settings (`validFrom/validTo`) into account so that the response expires at the next scheduled content change date. These validity-based calculation works by recording and caching of validity data during rendering. Thus, for the first request towards a URL a default, short `max-age` for the content type is used. During rendering the validity information is recorded for all content beans that are involved. For the next request, the recorded, more accurate validity-based expiration date is available and used for a more precise `max-age` value.

Overriding Cache Settings

By adapting the configuration or implementing custom code, projects can also tune Cache-Control headers according to their needs. If the function of the provided `DefaultCacheControlStrategy` does not fit your needs it is also possible to implement your own `CacheControlStrategy`.

> **NOTE**
>
> If your CAE extensions already add the HTTP Cache-Control header for some resources to the responses, the CoreMedia Cache-Control component would not overwrite these values. If you want to use the CoreMedia Cache-Control component, make sure your customization do not handle Cache-Control header itself.

You can configure Cache-Control defaults for certain object types or for URL pattern. Object type specific configuration and URL pattern based configuration can be used side by side.

Configure Cache Control

If both URL based and type based configuration match, URL pattern based configuration takes precedence over type based configuration.

The configuration options include those defined by Cachecontrol. The most important property in this context is the `max-age` property.

3.3.1 Object Type based Configuration

The object type based configuration allows you to configure default Cache-Control header by object type. The object type is the class of the model that is resolved by the handler mapping of the CAE. The configuration is type specific, but can also benefit from abstraction along the class hierarchy. That means that you may simply configure a common value for a super class instead of taking care about all the child classes. The type information part of the property name should be the simple name of the class (no package information, see Example 3.1, "Type based Cache-Control Configuration" [26]) and in lowercase. Configuration of classes with the same name in different packages is not supported.

```
# Cache Control Configuration
cae.cache-control.for-type.cmlinkable.max-age=1m

# time to cache blobs: "forever", aka 180 days
cae.cache-control.for-type.blob.max-age=180d
cae.cache-control.initial-max-age-for-type.blob=180d

# Fallback if nothing else specified
cae.cache-control.initial-max-age-for-type.object=1m
cae.cache-control.for-type.object.max-age=1m
```

Example 3.1. Type based Cache-Control Configuration

If you want to omit that the Cache-Control Header is added for an object type, you can set `cae.cache-control.for-type.myobject.max-age=-1s`. Otherwise, the fallback configuration for `cae.cache-control.for-type.object.max-age=1m` would be used.

In case your request mapping resolves, for example, a combined object like an article page, consisting of a navigation and a main article, the lookup mechanism uses the type of the article first to produce a configuration match and then tries the type hierarchy of the page itself. This can be customized to project needs by defining a primary bean of type `Function<Object, Stream<Type>>`.

3.3.2 URL Pattern based Configuration

The URL pattern based configuration allows you to configure default Cache-Control header based on URL pattern or specific URLs. The URL pattern may be in Ant path style or in Spring URI template pattern style. In case that more than one pattern matches, the more specific pattern wins.

```
# Use small max-age value, since catalog images urls do not change on editorial
 change.
```

```
cae.cache-control.for-url-pattern[/catalogimage/**].max-age=5m

# do not cache dynamic requests
cae.cache-control.for-url-pattern[/dynamic/**].max-age=0
cae.cache-control.for-url-pattern[/dynamic/**].no-store=true
```

Example 3.2. Type based Cache-Control Configuration

The Preview/Live CAE App comes with a default configuration. Especially the Live CAE configuration should be reviewed and adapted to your projects needs.

See Section 3.1.3, "Http Cache Control Properties" in *Deployment Manual* for details.

3.4 JMX Management

The *Content Application Engine* provides JMX access for management and monitoring.
Read the following chapters for further information:

- Section 4.9, "JMX Management" in *Operations Basics* chapter with general information about JMX and its configuration in CoreMedia applications.
- Read Section 5.7, "Managed Properties" [143] in order to get an overview of the managed attributes of the *CAE*.

> **CAUTION**
> Note that configuration changes made via JMX are not persisted, they are effective only until the next server restart.

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.

> NOTE
>
> For local development you should enable the Spring profiles `dev`, `local`, and `disableDataViews`. The `dev` profile activates Spring boot actuators, for example, while `local` configures the CAE in a way that it assumes to run on a local developer machine. The `disableDataViews` profile prevents code generation for data views.

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.

> **NOTE**
>
> Blueprint comes with content beans. If your project does not use Blueprint but uses content types similar to the Blueprint content types, then you may derive your content beans from the Blueprint content beans. You only have to write content beans from scratch if the content types of your project differ significantly from the Blueprint content types.

- `ContentBeans` are rendered by the rendering layer. See Section 4.3.3, "Views" [69] for details.
- `ContentBeans` can be cached in `DataViews`. See Section 4.2, "Data Views" [35] for details.
- `ContentBeans` can be used for other purposes than rendering, for example for implementing web services, for business logic deployed in the CAE Feeder, or for custom standalone applications.

4.1.1 Structure of the Content Bean

This section examines the structure of Blueprint content beans. The structure of these content beans shows best practices to be considered when writing or modifying content bean classes.

The Blueprint content bean classes comprise three files per content type:

- An interface with the same name as the content type, for example `CMArticle`.
- An abstract class ending with "Base", for example `CMArticleBase`, and
- A concrete class ending with "Impl", for example `CMArticleImpl`.

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 add business logic. When a content 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 "*Base" class contains property getters for every user-defined property in the corresponding content type. Getters are not used for metadata such as name or creation date. The property types are mapped to Java as follows:

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

Table 4.1. Content property to Java property mappings

4.1.2 Patterns For Content Beans

A few important patterns are used by the content beans. Keep them in mind when you write or modify content bean classes.

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.

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 `content-BeanFactory` and content object (and maybe other required services like a DAO or a JDBC data source). No content should be retrieved. The content bean source code fulfills this requirement by defining a default constructor and the two getters defined in the *CoreMedia CMS* interface.

> NOTE
>
> Content bean classes need a default constructor.

Identity, equality

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

> NOTE
>
> Content bean classes must not override `equals` or `hashCode` of `AbstractCon tentBean`.

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.

> **NOTE**
>
> Content bean classes must not declare any fields 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.

4.1.3 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 content 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 content type for which they were used. For example, a class `com.company.Article` that represents Article content items 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 content 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 content item
- The implementation class is `com.company.ArticleImpl`

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

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

i

4.1.4 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.

> **CAUTION**
>
> 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.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 content items 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 content item cache after all).

To use the defined data views, the data view factory dynamically constructs two sub-classes 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.

4.2.2.1 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" [40] shows, how data views are loaded and evaluated in the lifecycle of an HTTP request.
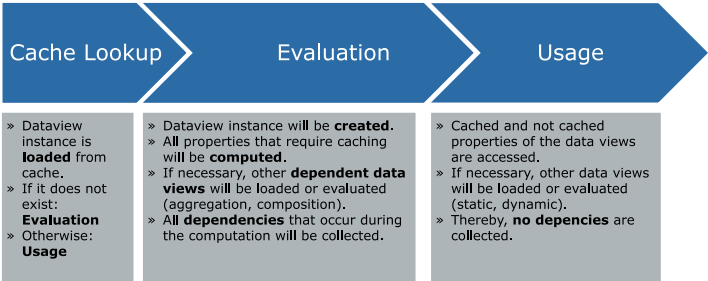
Figure 4.1. Phases of a data view lifecycle

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 4.2. Association types

4.2.2.2 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 Freemarker template:

```
public interface PageBean {
  ArticleBean getContent();
}
<@cm.include self=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.

> **CAUTION**
> 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.

4.2.2.3 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:
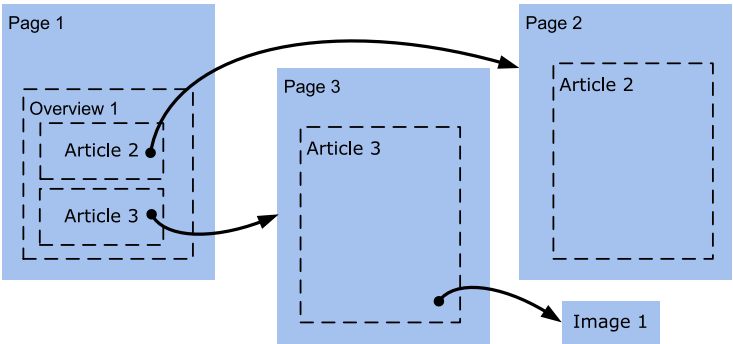
Figure 4.2. Example site structure

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.
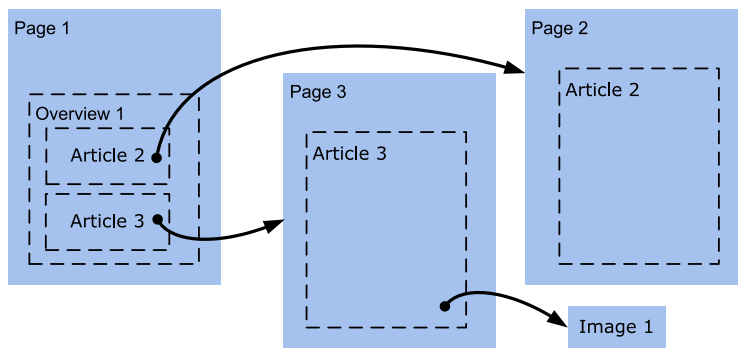
Figure 4.3. Entity Model

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
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

Bean	Property	Description	Expensive
Image Bean	MimeType	The image data's mime type.	Yes
	Data	The binary data.	No

Table 4.3. Bean Properties in the DataView Example. [*] The computation of property "abstract" is not expensive by itself but the access of property "text" only.

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

PageBean.ftl

```
<#-- @ftlvariable name="self" type="com.mycompany.PageBean" -->
<html>
  <head>
    <title>${self.title}</title>
  </head>
  <body>
    <div class="content"><@cm.include self=self.content/>
     </div>
    <div class="navigation">
      <ul>
      <#list self.navigation![] as page>
        <li><a href="${cm.getLink(page)}">${page.title!""}</a></li>
      </#list>
      </ul>
    </div>
</body>
</html>
```

ArticleBean.ftl

```
<#-- @ftlvariable name="self" type="com.mycompany.ArticleBean" -->
<h1>${self.headline!""}</h1>
<div>${self.text!""}</div>
<#if self.image?has_content>
  <#assign imageUrl= cm.getLink(self.image)/>
  <img src="${imageUrl!""}" alt="image"/>
</#if>
```

OverviewBean.ftl

```
<#-- @ftlvariable name="self" type="com.mycompany.OverviewBean" -->
<#list self.teasers as article>
  <h2>${article.headline!""}</h2>
  <p>
    ${article.abstract!""}
    [<a href="${cm.getLink(article.page)}">more</a>]
```

```
   </p>
</#list>
```

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 `Image Bean#MimeType` are marked for caching as well. `ArticleBean#Abstract` is not marked here because it benefits from the already cached `Article Bean#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 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>
```

4.2.2.4 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.

```java
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 `DataViewHelper`s 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.dispatch.HasCustom`

`Type`, 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

```
<cachesize class="com.company.cms.SuperType" size="1000"/>
<cachesize 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.core-media.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.

> **WARNING**
>
> 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.core media.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();
  }
}
```

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

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

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".

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

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 `ContentBean`s 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.
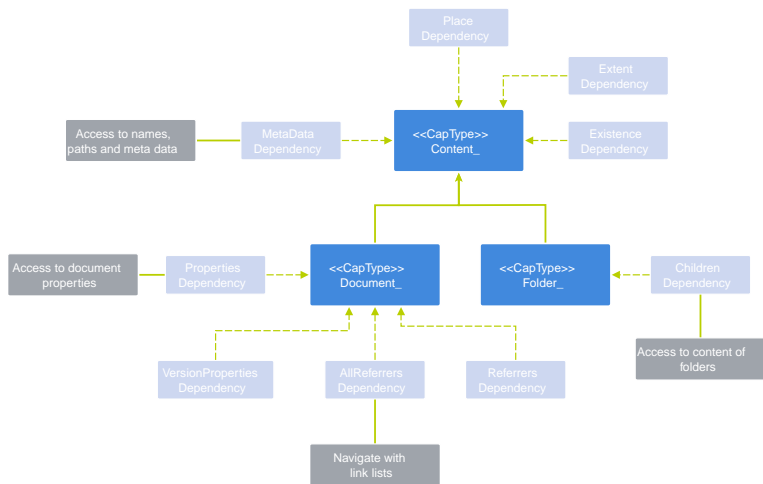
Figure 4.4. Dependencies of the Unified API cache

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 -->
<bean
class="com.coremedia.objectserver.web.config.CaeHandlerServicesConfiguration"/>

<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 Spring Framework documentation for a list of possible argument types and different options of implementing a handler method.

4.3.1.1 Building the Model

As mentioned above, it's a handler's responsibility to provide a `ModelAndView` in-stance. 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 ne-cessary 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 the 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.object server.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.core media.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.ftl`. 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" [64] for further information.

4.3.1.2 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 Spring Framework documentation for more information about handler interceptors.

4.3.1.3 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 instanti-ation 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.
- 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 4.3.6.4, "Handling POST requests" [100]
- The best practice for handling redirects can be found here Section 4.3.6.5, "Handling redirects" [100]

4.3.1.4 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 implement-ing 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 `@Respon seBody` 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 `pro duces="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 `produces="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.
                 MappingJackson2HttpMessageConverter"/>
  </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 buildings 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 Freemarker template might look like

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

4.3.1.5 Legacy Controllers

In past versions of the CoreMedia CMS, the preferred way of writing handlers was to im-
plement 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.AbstractView`
`Controller` was removed in CM8.

4.3.1.6 Path Matching Details

The Spring Framework documentation 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 `AntPathMatch`
`er` 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 `segments` 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 the request character encoding to UTF-8. Blueprint CAEs are configured with `spring.http.encoding.force=true`.

4.3.1.7 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, configure the `HiddenHttpMethodFilter` via Spring.

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.

4.3.1.8 Solution for the Same-Origin Policy Problem

Access-Control-Allow-Origin

This solution is built into the CoreMedia Blueprint workspace, so you may use it out of the box. The idea is to customize the same origin policy by setting the `Access-Control-Allow-Origin` HTTP header accordingly. The allowed origins can be configured via the properties `cae.cors.allowed-origins-for-url-pattern[*]`.

```
cae.cors.allowed-origins-for-url-pattern[{path\:.*}]= \
    http://my.site.domain1,https://my.site.domain2
```

To fine-tune the configuration for Cross-Origin Resource Sharing (CORS), use the provided `cae.cors` configuration properties. See `CaeHandlerServicesConfiguration#caeCorsConfigurations(CaeCorsConfigurationProperties,ObjectProvider)`, `CaeCorsConfigurationProperties` and Section 3.1.4, "CORS Properties" in *Deployment Manual*.

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.
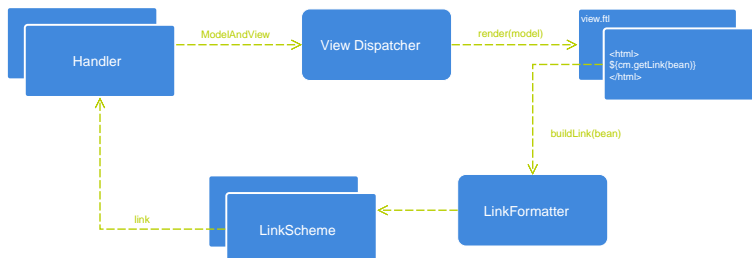
Figure 4.6. Processing chain of handlers and link schemes

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 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">
  <bean
class="com.coremedia.objectserver.web.links.CaeLinkServicesConfiguration"/>
  <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.

4.3.2.1 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,

```
<#assign rssLink=cm.getLink(bean, "rss", {
"maxItems": 10
})/>
```

from within a Freemarker 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 gener- ation 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.

1. `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 determ- ines 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 preced- ence, though.

2. `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.

3. `parameter`

A list of link parameters that need to be specified. In contrast to other predicates, all parameter predicates need to match here.

4. `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.

4.3.2.2 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.UriCompon`
`entsBuilder` 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 `/öffnung`
`szeiten` to `/%C3%B6ffnungszeiten`) properly by using the `UriCompon`
`ents#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 ne-
cessary to perform the encoding manually, because this is done by the framework.

4.3.2.3 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">
  <bean
class="com.coremedia.objectserver.web.links.CaeLinkServicesConfiguration"/>
  <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.

4.3.2.4 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 `PostProcessorPrecendences` 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.

4.3.2.5 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.core media.objectserver.web.links.AbstractLinkScheme` was removed in CM8.

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 *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 *FreeMarker*.

4.3.3.1 View Repository

The *CAE* uses a concept called ViewRepository to organize its views. A ViewRepository can be understood as a store that contains 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 Free-marker template for a bean of type `com.company.Article` is looked up at a location `/WEB-INF/templates/com.company/Article.ftl`. 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.ftl`.

> **CAUTION** ⚠
>
> 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 `.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>
```

4.3.3.2 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.ftl` available, but a template `com.company/Base.ftl` 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.
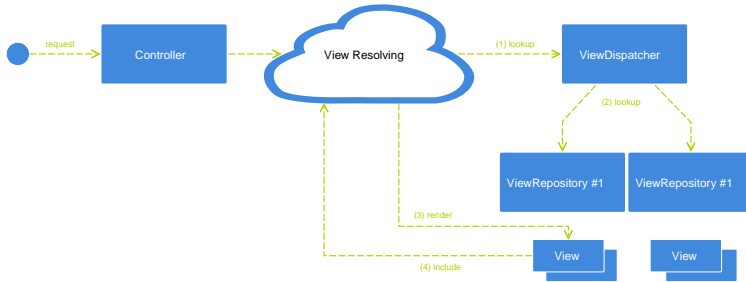
Figure 4.7. View lookup sequence

> **CAUTION**
>
> 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 `cae.view.filter-lookup-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="${cae.view.filter-lookup-by-predicate:true}">
  <description>
    Overrule the predicate's exclusion patterns for these classes.
  </description>
  <list>
    <value>my.package.MyViewRelevantBeanImpl</value>
```

```
  </list>
</customize:append>
```

4.3.3.3 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 `Compound-ViewRepositoryNameProvider` 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
<?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>

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

4.3.3.4 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.

> **CAUTION**   ⚠
>
> The property `delivery.local-resources` must be set to "false".

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 `tem plates.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 `template LocationPatterns` 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>
```

> **CAUTION**   ⚠
>
> It is recommended to use a dedicated content type for storing the template JAR. The content 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 content item of one of the types is added, modified, or a property is changed. (On live servers, publication and deletion of such a content item leads to the cache invalidation)

4.3.3.5 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>
```

> ## CAUTION
>
> This feature only works with FreeMarker templates and for templates which are not available in the site's theme. The site must be configured to use the view repository (e.g. `customViewRepository` in the example above). In addition, the property `delivery.local-resources` must be set to "true".

4.3.3.6 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/re sources/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/tem plates/%s/data!/META-INF/resources/WEB-INF/templates/`

4.3.3.7 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

`cae.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 comment behind each fragment also contains the time it took (in milliseconds) to render the fragment including its children. 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.ftl]
-->
  <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.ftl]

  -->
    <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$$[id=446]
    null webapp resource
    view[/WEB-INF/templates/sites/media/com.coremedia.com.coremedia.blueprint.common.contentbeans]
    took 31ms
  -->
  </div>
  <h4>Scuba diving the underwater adventure</h4>
  </a>
<!--
  END
  com.coremedia.blueprint.cae.contentbeans.CMArticleImpl$$[id=454]
  asTitleStory webapp resource

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

  took 58ms
-->
</li>
```

4.3.3.8 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 effective. 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 `viewDec` `orators` list in the CAE's `view-services`. E.g. the configuration for the `De` `bugViewDecorator` looks like this:

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

<customize:append id="addCAEDebugDecorator" bean="viewDecorators"
enabled="${cae.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 `cae.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);
```

```
  }
}
```

4.3.3.9 View Hooks

View Hooks provide a means to define extension points in *FreeMarker* templates. Project Extensions in *Blueprint Developer Manual* 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" [124] 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 programmed views.

```
<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.ftl`. The template defines an extension point with the id `articleEnd`.

```
<div class="detailView">

  <h1>${self.title}</h1>
  <@cm.include self=self.text!cm.UNDEFINED view="detailText" />

  <@cm.hook id="articleEnd"/>
</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.ftl` in the core modules directly, you only need to add an implementation of the `com.coremedia.ob-`

`jectserver.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 4.3.3.2, "View Lookup" [71]. 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.com pany.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. The default output format for all templates is set to HTML. See FreeMarker online documentation for details.

In special cases, it might be necessary to disable escaping. For this purpose, FreeMarker provides the directive `<#noautoesc/>` or built-in for Strings `?no_esc`.

> **CAUTION**
>
> 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.

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 4.3.4.1, "Rendering Markup" [83] 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`.

```
<@cm.include self=article view="teaser"
             params={ "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 `ContentBean` 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=cm.getLink(article, "teaser") />
<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).

4.3.4.1 Rendering Markup

Render objects of type `com.coremedia.xml.Markup` by including them from a FreeMarker template using:

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>
```

4.3.4.2 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

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 FreemarkerFacadeExample {

  public static final FreemarkerFacadeExample INSTANCE = new
FreemarkerFacadeExample();

  private FreemarkerFacadeExample() {
  }

  /**
   * 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.com pany.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_implict.ftl` as shown in Example 4.9, "Code for Idea auto-completion" [86] 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]
```

Example 4.9. Code for Idea auto-completion

4.3.4.3 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 `ExceptionHandlingViewDecorator` 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

```
cae.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:

```
cae.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 `fallback ViewRepository`. To use a custom exception rendering template rather than the fallback view, add your own view - such as a FreeMarker template - for `com.core-media.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 *ExceptionHandlingViewDecorator* 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 excep-

tions at the "teaser" level for any `com.example.contentbeans.base.CMOb ject`:

```
<?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="addCustomExceptionDecoratorAcceptBeanClasses"
                    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.content beans.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 `exceptionDecoratorAc ceptBeanClasses` 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.

4.3.4.4 Reference for FreeMarker Templates

The macros, functions and variables available in any FreeMarker template view rendered by the CAE are documented in the Section 6.5.1, "CoreMedia (cm)" in *Frontend Developer Manual*.

4.3.4.5 Supported Standards and Template Language Versions

FreeMarker templates are expected to comply with the FreeMarker 2.3.x syntax. See the FreeMarker documentation (https://freemarker.apache.org/docs/index.html) for details.

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 " [89] 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.

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

DOM with Metadata	Metadata Tree
`<html>`	`root`

DOM with Metadata	Metadata Tree

```
     |
<body> -> "S"
     |
<div> -> ["A", "x"]
     |
     |-- <div> -> "B"
     |     |
     |    <div>
     |     |
     |    <span> -> "y"
     |
     `-- <img> -> "C"
```

```
     |
slider metadata "S"
     |
content "A"
     |
property "x"
     |
     |-- content "B"
     |     |
     |    property "y"
     |
     `-- content "C"
```

Example 4.10. A DOM with Metadata and Generated Metadata Tree

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.

Preview Slider Setting: sliderMetaData

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.

```
{
  "sliderMetaData": {
    "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"}
    }
  }
}
```

Example 4.11. Responsive Device Slider Metadata

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.

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 4.3.5.3, "Advanced Metadata Usage" [94] 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 4.3.5.3, "Advanced Metadata Usage" [94].

Custom Metadata

4.3.5.1 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 `cae.pre view.metadata-enabled` property in the `WEB-INF/application.prop erties` file of your *CAE* application.

```
...
cae.preview.metadata-enabled=true
...
```

4.3.5.2 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 `@preview.previewScripts` is called in a template rendered once anywhere on the generated HTML page. You can then call the macro `@preview.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 `@preview.metadata` with an array, where each array element generates a metadata node.

The macro call `<@preview.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"<@preview.metadata data=self.content/>>Hello world!</div>
```

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

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

Note that macro `@preview.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 `@preview.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"<@preview.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<@preview.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<@preview.metadata self.content/>>
  <h1<@preview.metadata "properties.title"/>>${self.title}</h1>
  <div<@preview.metadata "properties.text"/>>${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 `@preview.metadata` with an array parameter specifying multiple metadata nodes:

```
<h1<@preview.metadata [self.content, "properties.title"]/>>${self.title}</h1>
<div<@preview.metadata [self.content, "properties.text"]/>>${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.getStudioAdditionalFiles Metadata()` 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= preview.getStudioAdditionalFilesMetadata(CSS_LIST,
 JS_LIST)/>
<head <@preview.metadata studioMetadata/>>
```

4.3.5.3 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_highlight Strategy` 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 *Studio Developer Manual*.

Adding custom metadata

Here is an example of the same combination of preview metadata and custom metadata in FreeMarker.

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

Example 4.13. Mixed preview and custom metadata in FreeMarker

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 6.x handler.

4.3.6.1 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();
   }
   ```

3. To render the front end, Spring provides macros to create HTML forms in Freemarker Templates, accessing the form bean in the model.

   This is a simple example for such a form, see Spring documentation for details on how a form view may be used.

   ```
   <form action="handlerUri" method="post">
     <label for="email">Email:</label>
       <@spring.bind path="nameOfForm.email"/>
       <@spring.formInput path="nameOfForm.email"/>

     <label for="emailRepeat">Repeat Email:</label>
       <@spring.bind path="nameOfForm.emailRepeat"/>
       <@spring.formInput path="nameOfForm.emailRepeat"/>

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

sary 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 4.1.2, "Patterns For Content Beans" [31]. 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 macro `cm.getId` 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:

```
<#assign featureId=cm.getId(feature)/>
<form action="handlerUri" method="post">
  <@spring.bind path="nameOfForm.feature"/>
  <input type="hidden"
         name="${spring.status.expression}"
         value="${featureId}"/>
  <label for="email">Email:</label>
    <@spring.bind path="nameOfForm.email"/>
    <@spring.formInput path="nameOfForm.email"/>

  <label for="emailRepeat">Repeat Email:</label>
    <@spring.bind path="nameOfForm.emailRepeat"/>
    <@spring.formInput path="nameOfForm.emailRepeat"/>

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

4.3.6.2 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, ...)
```

4.3.6.3 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 MyFormValidator Example [98] 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());
}
```

> **NOTE**
> 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:

```java
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.

> **NOTE**
>
> 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 macros contains macros to display global or field specific error messages:

```
<form action="handlerUri" method="post">
  <label for="email">Email:</label>
    <@spring.bind path="nameOfForm.email"/>
    <@spring.formInput path="nameOfForm.email"/>
    <@spring.showErrors "<br>", "notification error"/>

  <label for="emailRepeat">Repeat Email:</label>
    <@spring.bind path="nameOfForm.emailRepeat"/>
    <@spring.formInput path="nameOfForm.emailRepeat"/>
    <@spring.showErrors "<br>", "notification error"/>

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

See `showErrors` macro documentation.

4.3.6.4 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 4.3.6.5, "Handling redirects" [100]
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" [100]

4.3.6.5 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 `ModelAnd View` 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
}
```

> **NOTE**
>
> Because of SPR-10516 any beans added as objects to ModelAndView are converted to Strings (and might require adding a converter to `bindingConverters` bean (see Section 4.3.1, "Handling Requests" [55]) as soon as request handler specifies `Redir ectAttributes` 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 `redirectAt tributes` which as a result when building the model and view will receive the model bean in addition to ModelAndView.

4.3.6.6 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 unsuspecting 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 makes use of the Spring Security CSRF protection.

The Spring Security CSRF protection for the CAE is configured in `CaeWebSecur-ityAutoConfiguration#securityFilterChain(HttpSecurity)`. For customizations see Section 4.3.8, "Spring Security" [106] and Spring Security documentation on CSRF support for Servlet Environments.

To provide CSRF protection for web forms, add this to the templates:

```
<#-- @ftlvariable name="_csrf"
type="org.springframework.security.web.csrf.CsrfToken" -->
<form>
  <#if _csrf?has_content>
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}">

  </#if>
  ...
</form>
```

> **CAUTION**
>
> The name of the `ftlvariable` for the `CsrfToken` in the FreeMarker templates must match the name of the configured CSRF token parameter. Changing the parameter name (using HttpSessionCsrfTokenRepository.html#setParameterName(String)) requires the name of the `ftlvariable` in the FreeMarker templates to be changed likewise.

CSRF Tokens in Multipart Forms

Spring Security cannot check the CSRF token, when it is provided as (hidden) parameter in multipart forms. See Spring Security documentation on considerations for CSRF protection for multipart forms. To solve this for the registration form, the Elastic Social extension for the CAE registers the MultipartFilter to run before the Spring Security filter chain to enable CSRF for `multipart/form-data` POST requests. Projects that don't use the Elastic Social extension can also register the filter:

```
package com.coremedia.blueprint.component.cae.csrf;

import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;
import org.springframework.web.multipart.support.MultipartFilter;

@Configuration(proxyBeanMethods = false)
public class CaeCsrfMultipartConfiguration {

  private static final int ORDER_MULTIPART_FILTER =
          Ordered.HIGHEST_PRECEDENCE + 247_483_648; // == -1_900_000_000

  @Bean
  public FilterRegistrationBean<MultipartFilter>
multipartFilterRegistrationBean() {
    var registrationBean = new FilterRegistrationBean<>(new MultipartFilter());
```

```
    registrationBean.setOrder(ORDER_MULTIPART_FILTER);
    return registrationBean;
  }
}
```

Example 4.14. Configuring support for CSRF tokens in multipart forms

Backward Compatibility for CSRF Tokens in Legacy Templates

When updating to CoreMedia CMCC from an older version to 2007 or newer, there may be custom templates (other ones than those that are provided with Blueprint) that cannot instantly be updated to using the org.springframework.security.web.csrf.Csrf-Token instead of the former _CSRFToken string. To allow such legacy templates to still work (for a migration period) with the Spring Security CSRF implementation that is now used by CoreMedia CMCC, the following code snippets show how to add backward compatibility to the project.

To allow the legacy templates to still render the _CSRFToken parameters with the string value, a HandlerInterceptor has to be added that provides the _CSRFToken request attribute to the templates:

```
package com.coremedia.blueprint.component.cae.csrf;

import org.springframework.security.web.csrf.CsrfToken;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

class CsrfLegacyTokenSetter implements HandlerInterceptor {

  @Override
  public void postHandle(HttpServletRequest request, HttpServletResponse
response,
                         Object handler, ModelAndView modelAndView) {
    CsrfToken csrfToken = (CsrfToken)
request.getAttribute(CsrfToken.class.getName());
    if (csrfToken != null) {
      request.setAttribute("_CSRFToken", csrfToken.getToken());
    }
  }
}
```

Example 4.15. Implementing a CsrfLegacyTokenSetter

To verify the token, Spring Security expects the CSRF token to be provided with different parameter and header names. To allow Spring Security to also verify tokens that are sent by the legacy templates, a filter has to be added that wraps the HttpServletRequest with one that gets the token using the old parameter or header name when it is not provided with the new name:

```
package com.coremedia.blueprint.component.cae.csrf;

import org.springframework.web.filter.OncePerRequestFilter;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletRequestWrapper;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;

class CsrfLegacyTokenGetterFilter extends OncePerRequestFilter {

  @Override
  protected void doFilterInternal(HttpServletRequest request,
                                  HttpServletResponse response,
                                  FilterChain filterChain)
          throws ServletException, IOException {
    filterChain.doFilter(new CsrfLegacyTokenRequestWrapper(request), response);

  }

  private static class CsrfLegacyTokenRequestWrapper extends
HttpServletRequestWrapper {

    public CsrfLegacyTokenRequestWrapper(HttpServletRequest request) {
      super(request);
    }

    @Override
    public String getParameter(String name) {
      String value = super.getParameter(name);
      if (value == null && "_csrf".equals(name)) {
        value = super.getParameter("_CSRFToken");
      }
      return value;
    }

    @Override
    public String getHeader(String name) {
      String value = super.getHeader(name);
      if (value == null && "X-CSRF-TOKEN".equals(name)) {
        value = super.getHeader("X-CSRFToken");
      }
      return value;
    }
  }
}
```

Example 4.16. Implementing a CsrfLegacyTokenGetterFilter

Both classes have to be added to the application context for the CAE:

```
package com.coremedia.blueprint.component.cae.csrf;

import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;
import org.springframework.web.servlet.handler.MappedInterceptor;

@Configuration(proxyBeanMethods = false)
public class CaeCsrfBackwardCompatibilityConfiguration {

  private static final int ORDER_CSRF_LEGACY_TOKEN_FILTER =
          Ordered.HIGHEST_PRECEDENCE + 347_483_648; // == -1_800_000_000

  @Bean
```

```
  public MappedInterceptor csrfLegacyTokenSetter() {
    // Register the token setter for all paths
    return new MappedInterceptor(null, new CsrfLegacyTokenSetter());
  }

  @Bean
  public FilterRegistrationBean<CsrfLegacyTokenGetterFilter>
csrfLegacyTokenGetterFRB() {
    var registrationBean = new FilterRegistrationBean<>(
            new CsrfLegacyTokenGetterFilter());
    // Register the filter before the Spring Security filter chain
    registrationBean.setOrder(ORDER_CSRF_LEGACY_TOKEN_FILTER);
    return registrationBean;
  }
}
```

Example 4.17. Configuring CSRF backward compatibility

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 re source="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 Spring Security

The CAE security implementations are established using Spring Security.

CAE Security Configura-tion

The configuration classes for the CAE security are located in the package `com.coremedia.cae.security`. All beans for the CAE security are created by the `CaeWebSecurityAutoConfiguration` auto configuration. For customiz-

ations, each of these beans can be replaced by an equally typed bean in a custom configuration class.

In Spring Security, the main bean to realize the HTTP web security is the org.springframe-work.security.web.SecurityFilterChain. When the application context is created, it is configured and build using the org.springframework.security.config.annotation.web.build-ers.HttpSecurity prototype bean. For the CAE, the default configuration is done in the `CaeWebSecurityAutoConfiguration` auto configuration. To customize the HTTP web security for the CAE in case that the externalized configuration possibilities are not sufficient, extend the `com.coremedia.cae.security.CaeHttpSe-curityConfigurer`, and provide it as a bean.

HTTP Web Security

With Spring-Security an `HttpFirewall` is configured.

HTTP Firewall

For CoreMedia CAE, the `StrictHttpFirewall` is configured in `CaeWebSecur-ityAutoConfiguration.html#httpFirewall`. It uses the CAE http firewall configuration options to enable selective removal of its default rejections. In the default CAE (without any extensions), none of the default rejections are removed. If a rejection has to be removed for an extension, the regarding `cae.http-firewall.allow-*` property has to be set to `true` in the extensions component properties file.

4.3.9 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
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`.

4.3.9.1 Example - 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-linkschemes-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:

```
@SpringJunitConfig(MyTest.LocalConfig.class)
@ActiveProfiles(MyTest.LocalConfig.PROFILE)
class MyTest {
  @Configuration(proxyBeanMethods = false)
  @ImportResource(
          value = {
                  "classpath:/com/mycompany" +
                  "/myproject/myproject-linkschemes-beans.xml"
          },
          reader = ResourceAwareXmlBeanDefinitionReader.class
  )
  @Import({
          CaeLinkServicesConfiguration.class,
          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 `Local Config` classes of other tests might be found through component scan.

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

```
@Autowired
LinkFormatter linkFormatter;
```

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

Assertions.assertThat(link).isEqualTo("/123?view=myView");
```

4.3.9.2 Example - 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 org.springframework.test.web.servlet.MockMvc for details.

```
@SpringJunitWebConfig(MyTest.LocalConfig.class)
@ActiveProfiles(MyTest.LocalConfig.PROFILE)
class MyTest {
  @Configuration(proxyBeanMethods = false)
  @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();
    }
  }

  @Autowired
  private MockMvc mockMvc;

  @Test
  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`.

4.3.10 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.

4.3.10.1 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 `Handler-Helper` 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 4.3.1.1, "Building the Model" [57] for details.

4.3.10.2 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 RE-QUEST)*. 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.

4.3.10.3 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 4.3.4.3, "Error Handling" [86] for details.

4.3.10.4 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.

4.3.10.5 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 5.5, "Localized Content Management" in *Blueprint 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 `sitesService` 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 content type the Site Indicator is. For a detailed understanding, you are asked to read the API documentation as well.

4.5 Image Transformation API

> ### Functionality only for Self-Managed Installation
>
> This section describes the Image Transformation API which was initially designed for the built-in Java implementation. When using the cloud-only image transformation with support for the WebP format, the specification of the transformation string stays the same, but it is for example not possible to add a custom BlobTransformer or custom image transformations.

The *CAE*, the *Headless Server* 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 content settings. For example, an image content 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 4.4. 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.trans-form.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 in the range 0.0 to 1.0 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 a generic default if this parameter is not included.

- `defaultWebPQuality (dwq)`: Sets the WebP compression quality in the range 0.0 to 1.0 to be used should the output image be a WebP and no explicit quality parameter has been given.

  This value is ignored in the built-in image transformation and may only be used with the CoreMedia Image Transformation Service.

- `defaultAvifPQuality (daq)`: Sets the AVIF compression quality in the range 0.0 to 1.0 to be used should the output image be a AVIF and no explicit quality parameter has been given.

  This value is ignored in the built-in image transformation and may only be used with the CoreMedia Image Transformation Service.

- `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 his highly recommended saving 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, for example,

```
GENERIC_CMYK=/usr/share/color/icc/MyGenericCMYKPro
file.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.trans-form` 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);
}
```

The Blob Transformer Bean

In 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 in the `ImageOperations` interface. You can use this bean in your custom code to transform images, so you do not need to create an own `BlobTransformer` instance (which is way more complicated than just a constructor call).

Customizing the Blob Transformer

The image transformation is done with the help of a `DispatchingBlobTrans-former` 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:

```java
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:

```java
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.SharpenerImpl" />
  </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. Reference

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 startup of the `ApplicationContext`. The recommended way is to use Spring Java configuration and to annotate beans acting as customization sources using the annotation class `com.coremedia.springframework.customizer.Customize`. However, a customizer can also be declared in a Spring XML file.

Examples

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

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

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

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

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

XML Syntax

> **NOTE**
>
> **Important:** The recommended way to customize beans provided by CoreMedia applications is to use the first mechanism of the following list that can be applied to the target bean.
>
> - Use external configuration (for example, system properties) to configure the bean.
> - Override injection of the target bean by using Spring's `Primary` annotation (or XML attribute).
> - Define a source bean in Spring Java configuration style (for example, using `Component`/ComponentScan or `Configuration/Bean`) and add the `Customize` annotation.
> - Only use the XML customizers if none of the above can be used in your project.

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

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

or

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

or

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

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

The following operations are supported:

- **Replace** - Depending on the context, a bean will be replaced by another bean 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.

> **NOTE**
>
> Since 1907.1, Aspects are deprecated. This applies to the classes and interfaces provided by module `cap-contentbean-aspect`. Aspects of the content type model remain untouched. Although content bean classes are shipped with Blueprint and can be changed directly to expose new content properties (added via content type Aspects), the recommended way is to keep custom code in extensions. Content beans can be customized as described in Section 5.1, "Customizer" [121] to replace existing content beans with project-specific ones.

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 `Aspects
Provider` 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 `BeanFactoryAspectsProvider` 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 `beanNameOfAspectProvider: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">
  ...
</bean>
```

Example 5.3. Definition of an aspects provider for arbitrary Java beans

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, content type definitions or other XML files, need to ad-
dress 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 `XMLUtil5`)
  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 content items 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.ftl`.

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 content item. 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.ftl`, a more special bean `Weather` could be rendered using a matching template `Weather.ftl`. 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.ftl can perform this substitution by calling the Freemarker function cm.substitute() and dispatching the substitution result to its responsible template (for example Weather.ftl).

```
<#-- @ftlvariable name="self" type="com.mycompany.Action" -->
<@cm.include self=cm.substitute(self.id!"", self) />
```

Example 5.5. Use of cm.substitute() in CMAction.ftl

For more information on FreeMarker usage, see Section 6.5.1, "CoreMedia (cm)" in *Frontend Developer Manual*.

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 LoginHandler {

  // ...

  @RequestMapping("/{id}/login")
  public ModelAndView handleLogin(
    @PathVariable("id")Page page,
    @RequestParam("user")String user,
    @RequestParam("password")String password,
```

```
    HttpServletRequest request){

    LoginState state=processLogin(user, password, request.getSession());
    ModelAndView result=HandlerHelper.createModel(page);
    SubstitutionRegistry.register("com.mycompany.login",
                                  state, result);
    return result;
  }
}
```

Example 5.6. Registering a substitution programmatically

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 Freemarker integration, 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

Different aspects of the *Content Application Engine* can be configured with different properties. All configuration properties are bundled in the Deployment Manual (Chapter 3, *CoreMedia Properties Overview* in *Deployment Manual*). The following links contain the properties that are relevant for the CAE:

- Section 3.1.1, "General CAE Properties" in *Deployment Manual* contains properties for the general configuration of the CAE, such as the View Dispatcher or preview features.

- Section 3.1.2, "Delivery CAE Properties" in *Deployment Manual* contains mostly properties for development mode.

- Section 3.1.3, "Http Cache Control Properties" in *Deployment Manual* contains properties for the configuration of the HTTP Cache of the CAE.

- Section 3.1.4, "CORS Properties" in *Deployment Manual* contains properties for the configuration of Cross-Origin Resource Sharing (CORS) of the CAE.

- Section 3.1.5, "Blob Transformation properties" in *Deployment Manual* contains properties for the configuration of blob transformation feature.

- Section 3.1.6, "Renamed CAE Properties" in *Deployment Manual* contains an overview of old and new names of renamed CAE properties.

- Section 3.11, "UAPI Client Properties" in *Deployment Manual* contains properties for UAPI clients which can also be used by the CAE.

5.6 Bean Definition Reference

The following Spring beans are loaded into the application context automatically with the `cae-component` Maven artifact, except for beans defined in the Spring XML config file `controller-services.xml`, which is provided for backwards compatibility.

CAE Component Configuration

Service or Extension Point Definition	Type
`fallbackViewRepository`	Default `ViewRepository` implementation, loading templates from `/WEB-INF/templates-fallback` and using `rich textMarkupView`, `blobView`, `errorView`, `viewExceptionRenderer`, and `viewEngines`.
`templateViewRepository Provider`	Default `ViewRepositoryProvider` implementation, initialized with `programmedViews`, `viewDecorators`, and `viewEngines`. It will lookup view repositories using `templateLocations` and `templateLocationPatterns`. View repository name "fallback" will be resolved to the `fallbackViewRepository`. `viewRepositoryProviders` are initialized to try view repository names "default" and "fallback", if no view is found.

Table 5.1. Beans in artifact com.coremedia.cms: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 `["ftl" := freemarkerViewEngine]` by a customizer with order 100.
`viewDecorators`	List<`ViewDecorator`>

Service or Extension Point Definition	Type
	Extension point to register custom view decorators, initialized to `ex ceptionDecorator`, if `cae.view.errorhandler.en abled=true`, and `debugDecorator`, if `cae.view.debug-enabled=true`.
`viewRepositoryNamePro viders`	List<`ViewRepositoryNameProvider`>
	Extension point to register custom view repository name providers, initialized to an implementation returning "default" and "fallback".
`viewRepositoryPro viders`	List<`ViewRepositoryProvider`>
	Extension point to register custom view repository providers, initialized to `templateViewRepositoryProvider`.
`renderNodeDecoratorPro viders`	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.
`modelAwareViewResolver`	`ModelAwareViewResolver`
`richtextMarkupView`	`XmlMarkupView`
`blobView`	`MultiRangeBlobView`
`viewHookEventView`	`ViewHookEventView`
`programmedViews`	Map<String, `View`>
	Extension point to register programmed views, initialized to [`Markup` := richtextMarkupView, `Blob` := blobView, `Vie-wHookEvent` := viewHookEventView] by a customizer with order 100.
`templateLocations`	Map<String, String>

Service or Extension Point Definition	Type
	Extension point to register additional template locations with `tem plateViewRepositoryProvider`, initialized to "default" := `/WEB-INF/templates`.
`templateLocationPat terns`	List<String>
	Extension point to register additional template location path patterns with `templateViewRepositoryProvider`. In each pattern, "%s" will be replaced with the view repository name to resolve a location.
`viewingHandlerExcep tionResolver`	`ViewingHandlerExceptionResolver`
`errorView`	`ErrorView`
`viewExceptionRenderer`	`ViewExceptionRenderer`
`httpErrorView`	`HttpErrorView`
`exceptionDecorator`	`ExceptionHandlingViewDecorator`, will only be registered with `viewDecorators`, if property `cae.view.errorhand ler.enabled=true`.
`exceptionDecoratorAc ceptBeanClasses`	List<Class>
	Configuration for `exceptionDecorator`, empty by default.
`exceptionDecoratorRe jectBeanClasses`	List<Class>
	Configuration for `exceptionDecorator`, empty by default.
`exceptionDecoratorAc ceptViews`	`List<java.util.regex.Pattern>`
	Configuration for `exceptionDecorator`, empty by default.
`exceptionDecoratorRe jectViews`	`List<java.util.regex.Pattern>`
	Configuration for `exceptionDecorator`, empty by default.
`debugDecorator`	`DebugViewDecorator`, will only be registered with `viewDec orators`, if property `cae.view.debug-enabled=true`.

Service or Extension Point Definition	Type
`debugDecoratorAccept BeanClasses`	`List<Class>` Configuration for `debugDecorator`, empty by default.
`debugDecoratorReject BeanClasses`	`List<Class>` Configuration for `debugDecorator`, empty by default.
`debugDecoratorAc ceptViews`	`List<java.util.regex.Pattern>` Configuration for `debugDecorator`, empty by default.
`debugDecoratorRe jectViews`	`List<java.util.regex.Pattern>` Configuration for `debugDecorator`, empty by default.
`freemarkerViewEngine`	`ViewEngine` to render FreeMarker templates.

Table 5.2. Beans in artifact com.coremedia.cms:cae-viewservices

Unified API

Service or Extension Point Definition	Type
`connectionParameters`	`Map<String, Object>` Configuration for the `connection` bean.
`connection`	`CapConnection`
`contentRepository`	`ContentRepository`
`userRepository`	`UserRepository`
`workflowRepository`	`WorkflowRepository`
`worklistService`	`WorklistService`

Table 5.3. Beans in artifact com.coremedia.cms:cap-unified-api

Data Views

Service or Extension Point Definition	Type
dataViewFactory	`ConfigurableDataViewFactory` implementation, loading its data view definitions from `dataViewDefinitionLocations`
dataViewDefinitionLocations	List<String> Extension point to register data view factory configuration file patterns, initialized to `classpath:/framework/dataviews/**/*.xml`, `/WEB-INF/dataviews/**/*.xml`.
contentBeanFactory	`ContentBeanFactory`, creating content beans from prototype beans with name "contentBeanFactory:<content_type>".
contentBeanIdScheme	`ContentBeanIdScheme`
contentIdScheme	`ContentIdScheme`
contentBlobIdScheme	`ContentBlobIdScheme`
memberIdScheme	`MemberIdScheme`
contentTypeIdScheme	`ContentTypeIdScheme`
contentPropertyIdScheme	`ContentPropertyIdScheme`

Table 5.4. Beans in artifact com.coremedia.cms:cae-contentbeanservices

Caching

Service or Extension Point Definition	Type
cache	`Cache` instance created for `connection`

Table 5.5. Beans in artifact com.coremedia.cms:coremedia-cache

Link Generation

Service or Extension Point Definition	Type
linkSchemes	List<`LinkScheme`> Extension point to register link schemes with `linkFormatter`.
linkTransformers	List<`LinkTransformer`> Extension point to register link transformers with `linkFormatter`.
linkFormatter	`LinkFormatter`

Table 5.6. Beans in artifact com.coremedia.cms:cae-linkservices

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`, `contentTypeIdS`

Service or Extension Point Definition	Type
	cheme, and contentPropertyIdScheme by a customizer with order 100.

Table 5.7. Beans in artifact com.coremedia.cms: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.
bindingPropertyEditor Registrars	List<PropertyEditorRegistrar> Extension point to register custom property editor registrars (which in turn will register property editors) to bind form fields to bean properties.
handlerInterceptors	List<HandlerInterceptor> Extension point to register handler interceptors, which will be applied to all handlers.
idContentBeanConverter	Converter to convert numeric IDs to method parameters of type ContentBean. An application must register this bean with bindingConverters explicitly, in order to use it.
idGenericContent BeanConverter	GenericConverter to convert between numeric IDs and subtypes of ContentBean. This converter subsumes the functionality provided by idContentBeanConverter. An application must register this bean with bindingConverters explicitly, in order to use it.
idContentBeanProper tyEditor	java.beans.PropertyEditor to convert between numeric IDs and ContentBeans. An application must register this bean

Service or Extension Point Definition	Type
	with `bindingPropertyEditorRegistrars` explicitly using a PropertyEditorRegistrar, in order to use it.

Table 5.8. Beans in artifact com.coremedia.cms:cae-handlerservices

MIME Type Mappings

Service or Extension Point Definition	Type
`mimeTypeService` (alias `tikaMimeTypeService`)	`MimeTypeService`, providing methods related to MIME type detection and mapping to file extensions. The implementation is based on Apache Tika. Configured with the following properties: • `mimeTypesResourceNames` A comma-separated list of resource names of Tika Mime-Info configuration files. Set this property by Spring Environment property `mimeTypeService.mimeTypesResourceNames`. • `tikaConfig` An optional custom Tika configuration. Set this property by Spring Environment property `mimeTypeService.tikaConfig`. The value of this property must be a Spring Resource location (file:/path/to/local/file, for example), or null. If a custom Tika configuration is set, the `mimeTypesResourceNames` configuration has no effect.

Table 5.9. Beans in artifact com.coremedia.cms:coremedia-common

Image Transformations

Service or Extension Point Definition	Type
blobTransformer	An opaque `BlobTransformer`, handling concurrent image transformation requests.

Table 5.10. Beans in artifact com.coremedia.cms:coremedia-transform

Image Transformations, Built-In

These beans are available only if you are using the built-in Java image transformation of the CoreMedia Content Cloud applications.

Service or Extension Point Definition	Type
blobTransformer	In case of the built-in transformation, the `blobTransformer` bean is a `CachingBlobTransformer`, handling concurrent image transformation requests, caching image transformation results. Delegates cache misses to the bean `throttlingTransformer`.
throttlingBlobTrans former	`ThrottlingBlobTransformer`, handling concurrent image transformation requests, performing some basic load control. Delegates the actual transformation work to the bean imageTransformer.
imageSizePermit Strategy	`ImageSizePermitStrategy`, the permit strategy for the `throttelingBlobTransformer`. For images, it reads the image header and estimates the required in-memory size of the image. Finally, a safety factor is applied to the result.
memoryGuard	`JvmMemoryGuard`, checks the available JVM memory prior to allocating large chunks of memory during loading and transforming blobs.
imageTransformer	`DispatchingBlobTransformer`, the blob transformer actually performing image transformations. Holds a list of processor objects, initialized with the list bean imageProcessors.

Service or Extension Point Definition	Type
imageProcessors	List<?>, extension point to register your own image processors, implementing additional image operations. Initialized with a single processor, imageOperations.
imageOperations	Java2DImageOperations, implementing the ImageOperations interface using the javax.imageio library. You may reuse this bean and its operations when implementing your own image operations.
imageTransformerInputAdapter	Java2DImageInputAdapter, input adapter used by imageTransformer. Since the reading and decoding of an image may consume 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. Configured with the following properties: cache the cache instance to use, configured with 100MB heap capacity for the configured cache class. You may overwrite this with the property cache.capacities.com.coremedia.transform.image.java2d.LoadedImageCacheKey.
imageTransformerOutputAdapter	Java2DImageOutputAdapter, output adapter used by imageTransformer. Configured with the following properties: preserveMetadata=false defaultProgressiveThreshold=10000 defaultJpegCompressionQuality=0.75
imageTransformerConversionService	org.springframework.core.convert.ConversionService, conversion service used by the imageTransformer to convert operation arguments to the required method parameter type. Extension point to register your own type converters.

Table 5.11. Conditional beans in artifact com.coremedia.cms:coremedia-transform

Controllers

`controller-services.xml` is not imported automatically and is only provided for backwards compatibility with existing controller implementations.

Service or Extension Point Definition	Type
`controllerMappings`	Map<String, Controller> Extension point to register controllers handling requests with the given path prefix.
`controllerInterceptors`	Alias for `handlerInterceptors`, which will be applied to all controllers in `controllerMappings`.

Table 5.12. com/coremedia/cae/controller-services.xml in artifact cae-handlerservices

5.7 Managed Properties

In this section, you will find tables with several important properties and actions manageable via JMX.

File System Transformed Image Blob Cache

This is a file system cache where the results of image transformations are persisted in a dedicated directory and can survive *CAE* restarts. Internally it is represented as two subfolders "*new*" and "*old*". Transformed images are written to the "*new*" folder until it exceeds `NewGenerationCacheSize`. When that happens, the "*new*" folder is renamed to "*old*", and a new "*new*" folder is created. The process of renaming the "*new*" folder to "*old*" is called a *rotation*. If the "*old*" folder is not empty at rotation, than its files are regarded to be obsolete and get deleted. During a cache lookup, a content is first searched for in the "*new*" folder. If it not found there, the "*old*" folder is searched instead. In case it exists in "*old*", it is regarded to be "*still needed*" and moved back to the "*new*" folder (*recalled* from "*old*").

The content of both the "*old*" and the "*new*" folder are kept across system restarts.

The JMX attributes `Level`, `NewGenerationLevel` and `OldGeneration Level` show how much data is stored in bytes in the whole cache and its "*new*" and "*old*" subfolders at a certain point of time. They correspond to the values of internal counters and are also written every 5 seconds to "*level.txt*" files residing in the "*new*" and "*old*" folders. Upon restart, the internal counters are initialized from these files.

> **NOTE**
>
> It is possible to manipulate "*level.txt*" files, or the values of internal counters by setting JMX attributes `NewGenerationLevel` and `OldGenerationLevel`, as well as manually delete some cached image blob files. Make sure you know what you are doing when performing such manipulations and take care to restore consistency between the physical size of "*new*" and "*old*" folders, the state of internal counters and the values persisted in "*level.txt*".

The following table shows the relevant JMX attributes. They are also documented in the Javadoc of `TransformedBlobCacheManager`.

Attribute	Type	Description
BasePath	Read-only	The configured path used to store cache files.

Attribute	Type	Description
CacheSize	Read/Write	The maximum allowed cache size in bytes for the whole cache consisting of two generation folders "*new*" and "*old*". The cache level of both generations is rarely higher and typically significantly lower than this value. Note that the file system overhead for storing the files does not count towards this value. So the physical space that has to be reserved on the disk for the cache has to be slightly higher than this value.
NewGenerationCacheSize	Read-only	Cache size of the *new* generation in bytes, that is, the size at which the *new* generation gets rotated into the *old* generation. The size of *new* generation is hard-coded to be exactly one half of the value of CacheSize. Note that the file system overhead for storing the files does not count towards this value.
Level	Read-only	The cache level in bytes as a sum of files in the *new* generation and *old* generation folders. The value of level is the sum of NewGenerationLevel and OldGenerationLevel. Note that the file system overhead for storing the files does not count towards this value.
NewGenerationLevel	Read/Write	The cache level of the *new* generation in bytes, that is, the total size of all files in the *new* generation. Note that the file system overhead for storing the files does not count towards this value.
OldGenerationLevel	Read/Write	The cache level of the *old* generation in bytes, that is, the total size of all files in the *old* generation.
RotateCount	Read-only	The count of *rotate* operations since system start.
AccessCount	Read-only	The count of blob *accesses* since system start.
RecallCount	Read-only	The count of *recalls* since system start. During a *recall*, a blob is recovered from the *old* generation and moved back into the *new* generation.
FaultCount	Read-only	The count of *faults* since system start. During a fault, a blob is computed and put into the *new* generation.

Attribute	Type	Description
RecallSizeSum	Read-only	The sum of sizes in bytes of all blobs *recalled* since system start. During a *recall*, a blob is recovered from the *old* generation and moved back into the *new* generation.
FaultSizeSum	Read-only	The sum of sizes in bytes of all blobs *faulted* since system start. During a fault, a blob is computed and put into the *new* generation.
InitialLevel	Read-only	The initial cache level for the whole cache consisting of two generation folders "*new*" and "*old*". Note that the file system overhead for storing the files does not count towards this value.
NewGenerationIni tialLevel	Read-only	The initial cache level of the *new* generation at system start, that is, the total size of all files in the *new* generation. Note that the file system overhead for storing the files does not count towards this value.
OldGenerationIni tialLevel	Read-only	The initial cache level of the *old* generation at system start, that is, the total size of all files in the *old* generation. Note that the file system overhead for storing the files does not count towards this value.
StartTime	Read-only	The time of the system start in milliseconds since the Epoch.
Uptime	Read-only	Uptime of the cache in milliseconds.

Table 5.13. TransformedBlobCacheManager attributes

Glossary

Blob	Binary Large Object or short blob, a property type for binary objects, such as graphics.
CaaS	Content as a Service or short caas, a synonym for the CoreMedia Headless Server.
CAE Feeder	Content applications often require search functionality not only for single content items but for content beans. The *CAE Feeder* makes content beans searchable by sending their data to the *Search Engine*, which adds it to the index.
Content Application Engine [CAE]	The *Content Application Engine* [*CAE*] is a framework for developing content applications with *CoreMedia CMS*.
	While it focuses on web applications, the core frameworks remain usable in other environments such as standalone clients, portal containers or web service implementations.
	The CAE uses the Spring Framework for application setup and web request processing.
Content Bean	A content bean defines a business oriented access layer to the content, that is managed in *CoreMedia CMS* and third-party systems. Technically, a content bean is a Java object that encapsulates access to any content, either to CoreMedia CMS content items or to any other kind of third-party systems. Various CoreMedia components like the CAE Feeder or the data view cache are built on this layer. For these components the content beans act as a facade that hides the underlying technology.
Content Delivery Environment	The *Content Delivery Environment* is the environment in which the content is delivered to the end-user.
	It may contain any of the following modules:
	- *CoreMedia Master Live Server* - *CoreMedia Replication Live Server* - *CoreMedia Content Application Engine* - *CoreMedia Search Engine* - *Elastic Social* - *CoreMedia Adaptive Personalization*

Content Feeder	The *Content Feeder* is a separate web application that feeds content items of the CoreMedia repository into the *CoreMedia Search Engine*. Editors can use the *Search Engine* to make a full text search for these fed items.
Content item	In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.
Content Management Environment	The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules:

- *CoreMedia Content Management Server*
- *CoreMedia Workflow Server*
- *CoreMedia Importer*
- *CoreMedia Studio*
- *CoreMedia Search Engine*
- *CoreMedia Adaptive Personalization*
- *CoreMedia Preview CAE*

Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.
Content Repository	*CoreMedia CMS* manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.
Content Server	*Content Server* is the umbrella term for all servers that directly access the Core-Media repository:

Content Servers are web applications running in a servlet container.

- *Content Management Server*
- *Master Live Server*
- *Replication Live Server*

Content type	A content type describes the properties of a certain type of content. Such properties are for example title, text content, author, ...
Contributions	Contributions are tools or extensions that can be used to improve the work with *CoreMedia CMS*. They are written by CoreMedia developers - be it clients, partners or CoreMedia employees. CoreMedia contributions are hosted on Github at ht-tps://github.com/coremedia-contributions.
Control Room	*Control Room* is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
CORBA (Common Object Request Broker Architecture)	The term *CORBA* refers to a language- and platform-independent distributed object standard which enables interoperation between heterogenous applications over a network. It was created and is currently controlled by the Object Management Group (OMG), a standards consortium for distributed object-oriented systems.

CORBA programs communicate using the standard IIOP protocol.

CoreMedia Studio	*CoreMedia Studio* is the working environment for business specialists. Its functionality covers all the stages in a web-based editing process, from content creation and management to preview, test and publication.
	As a modern web application, *CoreMedia Studio* is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.
Dead Link	A link, whose target does not exist.
Derived Site	A derived site is a site, which receives localizations from its master site. A derived site might itself take the role of a master site for other derived sites.
DTD	A Document Type Definition is a formal context-free grammar for describing the structure of XML entities.
	The particular DTD of a given Entity can be deduced by looking at the document prolog:

```
<!DOCTYPE    coremedia    SYSTEM    "http://www.core
media.com/dtd/coremedia.dtd"
```

	There're two ways to indicate the DTD: Either by Public or by System Identifier. The System Identifier is just that: a URL to the DTD. The Public Identifier is an SGML Legacy Concept.
Elastic Social	*CoreMedia Elastic Social* is a component of *CoreMedia CMS* that lets users engage with your website. It supports features like comments, rating, likings on your website. *Elastic Social* is integrated into *CoreMedia Studio* so editors can moderate user generated content from their common workplace. *Elastic Social* bases on NoSQL technology and offers nearly unlimited scalability.
EXML	EXML is an XML dialect used in former CoreMedia Studio version for the declarative development of complex Ext JS components. EXML is Jangaroo 2's equivalent to Apache Flex (formerly Adobe Flex) MXML and compiles down to ActionScript. Starting with release 1701 / Jangaroo 4, standard MXML syntax is used instead of EXML.
Folder	A folder is a resource in the CoreMedia system which can contain other resources. Conceptually, a folder corresponds to a directory in a file system.
FTL	FTL (FreeMarker Template Language) is a Java-based template technology for generating dynamic HTML pages.
Headless Server	CoreMedia Headless Server is a CoreMedia component introduced with CoreMedia Content Cloud which allows access to CoreMedia content as JSON through a GraphQL endpoint.
	The generic API allows customers to use CoreMedia CMS for headless use cases, for example delivery of pure content to Native Mobile Applications, Smart-

	watches/Wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of *Best current practice* (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese, or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.
Importer	Component of the CoreMedia system for importing external content of varying format.
IOR (Interoperable Object Reference)	A CORBA term, *Interoperable Object Reference* refers to the name with which a CORBA object can be referenced.
Jangaroo	*Jangaroo* is a JavaScript framework developed by CoreMedia that supports TypeScript (formerly MXML/ActionScript) as an input language which is compiled down to JavaScript compatible with Ext JS. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net. Jangaroo 4 is the Action-Script/MXML/Maven based version for CMCC 10. Since CMCC 11 (2110), Jangaroo uses TypeScript and is implemented as a *Node.js* and *npm* based set of tools.
Java Management Extensions (JMX)	The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
Locale	Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server	The *Master Live Server* is the heart of the *Content Delivery Environment*. It receives the published content from the *Content Management Server* and makes it available to the *CAE*. If you are using the *CoreMedia Multi-Master Management Extension* you may use multiple *Master Live Server* in a CoreMedia system.
Master Site	A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites.
MIME	With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multi-media emails and of web documents is standardised.
MXML	MXML is an XML dialect used by Apache Flex (formerly Adobe Flex) for the declarative specification of UI components and other objects. Up to CMCC 10 (2107), CoreMedia Studio used the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 7. Starting

with CMCC 11 (2110), a new, Node.js and npm based version of Jangaroo is used that supports standard TypeScript syntax instead of MXML/ActionScript, still compiling to Ext JS 7 JavaScript.

Personalisation

On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.

Projects

With projects you can group content and manage and edit it collaboratively, setting due dates and defining to-dos. Projects are created in the Control Room and managed in project tabs.

Property

In relation to CoreMedia, properties have two different meanings:

In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type.

In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.

Replication Live Server

The aim of the *Replication Live Server* is to distribute load on different servers and to improve the robustness of the *Content Delivery Environment*. The *Replication Live Server* is a complete Content Server installation. Its content is an replicated image of the content of a *Master Live Server*. The *Replication Live Server* updates its database due to change events from the *Master Live Server*. You can connect an arbitrary number of *Replication Live Servers* to the *Master Live Server*.

Resource

A folder or a content item in the CoreMedia system.

ResourceURI

A ResourceUri uniquely identifies a page which has been or will be created by the *Active Delivery Server*. The ResourceUri consists of five components: Resource ID, Template ID, Version number, Property names and a number of key/value pairs as additional parameters.

Responsive Design

Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.

Site

A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site.

A typical site also has a master site it is derived from.

Site Folder

All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.

Site Indicator

A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.

Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Template	In CoreMedia, FreeMarker templates used for displaying content are known as Templates.
	OR
	In *Blueprint* a template is a predeveloped content structure for pages. Defined by typically an administrative user a content editor can use this template to quickly create a complete new page including, for example, navigation, predefined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes web application	The *User Changes* web application is a *Content Repository* listener, which collects all content, modified by *Studio* users. This content can then be managed in the *Control Room*, as a part of projects and workflows.
Variants	Most of the time used in context of content variants, variants refer to all localized versions within the complete hierarchy of master and their derived sites (including the root master itself).
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.
Weak Links	In general *CoreMedia CMS* always guarantees link consistency. But links can be declared with the *weak* attribute, so that they are not checked during publication or withdrawal.
	Caution! Weak links may cause dead links in the live environment.
Workflow	A workflow is the defined series of tasks within an organization to produce a final outcome. Sophisticated applications allow you to define different workflows for different types of jobs. So, for example, in a publishing setting, a document might be automatically routed from writer to editor to proofreader to production. At each stage in the workflow, one individual or group is responsible for a specific task. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process.
Workflow Server	The *CoreMedia Workflow Server* is part of the Content Management Environment. It comes with predefined workflows for publication and global-search-and-replace but also executes freely definable workflows.
XLIFF	XLIFF is an XML-based format, standardized by OASIS for the exchange of localizable data. An XLIFF file contains not only the text to be translated but also metadata about the text. For example, the source and target language. *CoreMedia Studio*

allows you to export content items in the XLIFF format and to import the files again after translation.

Index

P

placeholders
    substitution, 128

S

Spring configuration, 33
Spring framework, 19, 33
Substitution API, 128

T

test framework, 107

U

Unified API cache, 17
using data views, 38