Search Manual

CoreMedia Content Cloud - v13

List of Figures

List of Tables

List of Examples

1. Preface

This manual describes the concepts of the *CoreMedia Search Engine* and how data is indexed with *Content Feeder*, *CAE Feeder* and *Elastic Social*. You will learn how to configure and operate these applications and how to customize them.

1.1 Audience

This manual is intended for all administrators and developers that use the *CoreMedia Search Engine*. If you want to use the *CAE Feeder*, you should also read the Content Application Developer Manual in order to become familiar with the *Content Application Engine*. For searching in *Elastic Social* you should also read the Elastic Social Manual.

1.2 Typographic Conventions

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

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

Code lines in code examples which continue in the next line	\	`cm systeminfo \`
		`-u user`

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

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

Table 1.2. Pictographs

1.3 CoreMedia Services

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

NOTE

CoreMedia User Orientation for CoreMedia Developers and Partners

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

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

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

1.3.1 Registration

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

1.3.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

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

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

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

NOTE

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

Maven artifacts

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

https://repository.coremedia.com

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

npm packages

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

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

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

Manual	Audience	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.
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*.

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

Table 1.3. CoreMedia manuals

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

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

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

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

https://www.coremedia.com/training

Contact the training department at the following email address:

Email: training@coremedia.com

1.3.5 CoreMedia Support

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

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

https://support.coremedia.com/

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

Email: support@coremedia.com

Create a support request

Support request

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

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

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

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

Support checklist

To put it in a nutshell, CoreMedia needs:

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

Log files

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

Which Log File?

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

Where do I Find the Log Files?

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

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

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

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

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

1.4 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 *CoreMedia Search Engine* adds full-text search capabilities to the *CoreMedia CMS*. You can use it to quickly find content of a *CoreMedia Content Server*, content beans of a *CoreMedia CAE* and social data such as users and comments of *CoreMedia Elastic Social*. It is possible to search for text in binary data of many supported formats.

You can search for content in *Studio*. You can also integrate search functionality into your website and custom applications.

The *CoreMedia Search Engine* is based on *Apache Solr* and comes with some *CoreMedia* specific extensions for content processing. It maintains indices and provides full-text search capabilities. Chapter 3, *Search Engine* [15] describes the *Search Engine* in more detail.

The *CoreMedia CMS* is delivered with different Feeder applications, which send data to the *Search Engine*.

- The Content Feeder sends content to the *Search Engine* for indexing. This makes it possible to search for content in the *Studio* and custom content applications.

  Chapter 4, *Searching for Content* [43] describes concepts, configuration and operation of the required components in detail.

- Content applications often require search functionality not only for content items but for content beans of a *CoreMedia CAE*. The *CoreMedia CAE Feeder* makes content beans searchable by sending their data to the *Search Engine*.

  Chapter 5, *Searching for CAE Content Beans* [80] describes concepts, configuration, operation and developing for the *CAE Feeder* in detail.

- *Elastic Social* worker applications send social data such as created comments and users to the *Search Engine*. Worker applications are *Elastic Social* applications configured with property `taskqueues.worker-node=true`.

  The *Elastic Social* Plugin for *CoreMedia Studio* allows searching for comments and users.

  See the Elastic Social Manual for more information.

A *Search Engine* index contains index documents. Each of these index documents carries a unique String identifier and multiple fields with values. Applications can search for index documents that match a given query, for example index documents that contain a specific word in one field. Index document fields and field types can be configured in the index schema as required by the application.

When using the *Content Feeder*, an index document represents a *CoreMedia* content. When using the *CAE Feeder*, an index document represents a content bean. With *Elastic Social*, an index document represents a comment or a user.

Multiple *Content Feeder* applications, *CAE Feeder* applications and *Elastic Social* tenants can use the same *Search Engine* but require separate indices. An index is a group of index documents for a specific application and with similar structure. Search requests use a specific index to retrieve results for the specific application. Each index can use different fields for its index documents as configured in the index schema.

3. Search Engine

The *CoreMedia Search Engine* is based on *Apache Solr*. It is a server application that receives search and indexing requests via HTTP. Solr provides two modes of operation: as standalone Solr instance with optional leader/follower index replication, or as SolrCloud cluster.

Solr manages multiple indices with possibly different configurations. Each of these indices is stored as a *Lucene index* on disk. In Solr terminology, an index managed by a standalone Solr server is called a *Solr Core* (or shortly a core) while an index managed by a SolrCloud cluster is called a *Solr Collection* (or shortly a collection). This documentation uses these terms interchangeably.

You can download *Apache Solr* from http://solr.apache.org. Make sure to download version 9.10.0, which is the supported version for *CoreMedia Content Cloud*.

You can find the Solr Reference Guide at https://solr.apache.org/guide/solr/9_10/index.html. Make sure to read the sections about system requirements and taking Solr to production.

This chapter describes configuration and operational tasks specific to the integration of *Apache Solr* as *CoreMedia Search Engine*.

3.1 Starting

You can start Solr by running "`bin/solr start`" from the Solr distribution directory. If you're using Windows, you'll have to use "`bin\solr.cmd start`" instead.

The Solr start script takes additional parameters such as `-p` to specify the port. Enter "`bin/solr start -help`" for an overview of parameters. Further configuration options can be specified as environment variables in `bin/solr.in.sh`, or `bin\solr.in.cmd` for Windows. For details, have a look at the Solr reference guide, for example at Solr Reference Guide: Solr Control Script Reference.

A required parameter for using Solr with CoreMedia is the location of the *Solr Home* directory, which contains configuration files and additional libraries. See Section 3.2, "Solr Home and Core Directories" [17] for a description of that directory. The *Solr Home* directory needs to be specified at startup with the `-s` parameter of the "`bin/solr start`" script. Alternatively, you can set the environment variable `SOLR_HOME`, for example in `bin/solr.in.sh`.

After startup, the Solr administration page is available at `http://<host>:<port>/solr`.

You can stop a running Solr instance by invoking "`bin/solr stop`", or "`bin\solr.cmd stop`" in case of Windows.

Starting Solr for local development in Blueprint

For local development with *CoreMedia Blueprint*, you can simply start and stop a configured Solr instance from Maven as follows:

- Download the official Solr distribution and extract it into a directory of your choice.
- Set the environment variable `SOLR_SCRIPT` to point to the Solr start/stop script in the extracted directory. Choose "`bin/solr`" for Unix or "`bin\solr.cmd`" for a Windows shell.
- Go to directory "`apps/solr/modules/search/solr-config`".
- Execute "`mvn exec:exec@start-solr` to start Solr.
- Execute "`mvn exec:exec@stop-solr` to stop Solr.

After startup, the Solr administration page is available at `http://localhost:40080/solr`.

3.2 Solr Home and Core Directories

Solr uses a directory called *Solr Home* for configuration files and additional libraries. It is specified with parameter `-s` of the "`bin/solr start`" script or as environment property `SOLR_HOME`, for example, in `bin/solr.in.sh`. The directory has the following general structure:

```
<solr-home>/
    solr.xml
    configsets/
        <configset1>/
            conf/
                schema.xml
                solrconfig.xml
                ...
        <configset2>/
        ...
    lib/
        <additional jar files>
```

solr.xml

The file `solr.xml` is the central Solr configuration file. It contains just a few settings, which you do not need to change. Most of Solr's configuration is placed in other configuration files.

It specifies the `coreRootDirectory`, which is the directory where *Solr cores* and their data are stored. The default `solr.xml` uses the directory that is set with system property `coreRootDirectory`. If no such system property is set, Solr will store cores in the directory `<solr-home>/cores`. It's recommended to configure a different absolute path outside of *Solr Home*.

You can set the `coreRootDirectory` system property with the parameter "`-a -DcoreRootDirectory=<path>`" when invoking "`bin/solr start`". Alternatively, you can set the environment variable `SOLR_OPTS`, for example in `bin/solr.in.sh`:

```
SOLR_OPTS="$SOLR_OPTS -DcoreRootDirectory=/var/coremedia/solr-data"
```

You can find more information about the `solr.xml` file in Solr Reference Guide: Configuring solr.xml.

Config Sets

Index-specific configuration files are organized as named config sets, which are subdirectories of the `configsets` directory. A config set defines an index schema with index fields and types in `conf/schema.xml` and lots of configuration options for indexing, searching and additional features in `conf/solrconfig.xml`. The latter file for example contains search request handler definitions with default settings such as the default index field to search in.

The *CoreMedia Search Engine* comes with three config sets: "`content`" for *Content Feeder* indices, "`cae`" for *CAE Feeder* indices and "`elastic`" for *Elastic Social* indices. They configure different index fields and Solr features such as search request handlers as required. Projects may customize these files or create additional config sets according to their needs. Note that some index fields are required for operation. See the comments in the configuration files for details.

Lib directory

The directory `<solr-home>/lib` contains additional libraries that can be used by all Solr cores and are not available in the Solr distribution itself. This includes some required *CoreMedia* extensions.

Core Root Directory

The `coreRootDirectory` contains the actual Solr cores, which are the indices used by CoreMedia applications. The directory must be writable and should provide fast disk I/O for good performance. Solr automatically discovers cores by looking for `core.properties` files below that directory. Each directory with a `core.properties` file represents a Solr Core. CoreMedia Feeder applications create cores dynamically, so the directory can be empty at first start.

With the default configuration, *Content Feeder* and *CAE Feeders* will create these Solr cores when started the first time:

- `studio`: an index of *CoreMedia* contents used for searching in *Studio*, which gets its data from the *Content Feeder*.
- `preview`: an index of *CoreMedia* content beans used for searching in the *Content Application Engine* of the *Content Management Environment* (aka *preview*), which gets its data from the *CAE Preview Feeder*.
- `live`: an index of *CoreMedia* content beans used for searching in the *Content Application Engine* of the *Content Delivery Environment* (aka *live*), which gets its data from the *CAE Live Feeder*.

Further cores will be created by *Elastic Social* applications for users and comments for different tenants, for example:

- `blueprint_corporate-de-de_users`: an index of *Elastic Social* users for tenant `corporate-de-de` used for searching in the *Studio* **User Management**, which gets its data from an *Elastic Social Worker*.

- `blueprint_corporate-de-de_comments`: an index of *Elastic Social* comments for tenant `corporate-de-de` used for searching in the *Studio* **Moderation**, which gets its data from an *Elastic Social Worker*.

The `coreRootDirectory` has the following general structure:

```
<coreRootDirectory>/
    <core1>/
        core.properties
        data/
            index/
                <index files>
            tlog/
                <transaction log files>
    <core2>/
    ...
```

The file `core.properties` contains Solr core configuration properties, most importantly the name of the used config set with the `configSet` property. The core "`studio`" uses the "`content`" config set, the cores "`preview`" and "`live`" use the "`cae`" config set, and *Elastic Social* cores use the "`elastic`" config set.

Index Data

Each Solr core has its own `data` directory with index files and transaction log. The actual index files are written to the directory `data/index`. In addition to the index, Solr maintains a transaction log with latest and/or pending changes for the index files. The transaction log is stored in the directory `data/tlog`.

3.3 Leader/Follower Index Replication

For a production setup, it is recommended to use a SolrCloud cluster or Solr leader/follower index replication. With Solr leader/follower index replication one Solr node - the leader - handles index updates, while one or more other Solr nodes - the followers - handle high query load. Solr followers periodically replicate index changes from the Solr leader. Such a setup allows the distribution of high query load across multiple Solr follower nodes and also provides basic fault tolerance for the query side. For replication without latency and better fault tolerance consider SolrCloud, which is described in Section 3.4, "SolrCloud" [22].

You can find more information about leader/follower index replication in Solr Reference Guide: User-Managed Index Replication.

3.3.1 Connecting CoreMedia applications

CoreMedia applications are configured with property `solr.url` to connect to one or more Solr instances.

Content Feeder, *CAE Feeder* and *Elastic Social* worker applications must be configured to connect to the Solr leader, because all indexing requests are handled by the leader.

Studio should also be configured to query the Solr leader to use the most up-to-date index. Solr followers lag some seconds behind and editors would not be able to find newly created content immediately in *Studio*. The default replication poll interval is set to 20 seconds, and such a delay is not desirable in *Studio* search results.

The *Content Application Engine* can be configured to connect to multiple Solr followers. To this end, a comma-separated list of Solr URLs can be configured in property `solr.url`. The *CAE*s will then use a simple round robin load balancing with automatic failover when a server goes down.

3.3.2 Replication Handler Configuration

Replication is configured with the `ReplicationHandler` section in the Solr index configuration file `solrconfig.xml`. *CoreMedia Blueprint* defines the

`ReplicationHandler` for the config sets "`content`" and "`cae`" in module `apps/solr/modules/search/solr-config`.

Blueprint default configuration of the `ReplicationHandler` references some system properties that need to be set accordingly when starting a Solr instance that is part of a leader/follower setup.

- `solr.leader`: set to `true` for the Solr leader node, defaults to `false`
- `solr.follower`: set to `true` for Solr follower nodes, defaults to `false`
- `solr.leader.url`: set to the Solr URL of the Solr leader node, for Solr follower nodes

Note, that hostname and port of the leader node must also be set in the `solr.allowUrls` system property of Solr follower nodes. Alternatively, the corresponding checks can be disabled with `-Dsolr.disable.allowUrls=true`. See the Solr Reference Guide for details.

When developing with *CoreMedia Blueprint*, you can start Solr locally from Maven as described in Section 3.1, "Starting" [16]. You can also start a Solr follower node to test replication in the same way by invoking "`mvn exec:exec@start-solr-follower`". Under the hood, this will set the above system properties. See the configuration of the `exec-maven-plugin` in file `apps/solr/modules/search/solr-config/pom.xml` for details.

3.3.3 Solr Follower Index Creation

Content Feeder and *CAE Feeder* create their indices at the Solr leader when started the first time. To start replication, these indices must be created on Solr followers as well. To create the default indices "`studio`", "`preview`" and "`live`", you have to send the following HTTP requests to the followers. In the example, the Solr follower is running at port `40081`:

```
curl 'http://localhost:40081/solr/admin/cores?action=CREATE&name=studio&configSet=content&dataDir=data'
curl 'http://localhost:40081/solr/admin/cores?action=CREATE&name=preview&configSet=cae&dataDir=data'
curl 'http://localhost:40081/solr/admin/cores?action=CREATE&name=live&configSet=cae&dataDir=data'
```

The requests specify the name of the created index in the query attribute "`name`" and the name of the used config set in the attribute "`configSet`".

Solr followers will start replication after their indices have been created. You can check the state of replication on the Solr follower's admin UI on page Replication after selecting the corresponding Solr core.

3.4 SolrCloud

SolrCloud is Solr's capability to run as a cluster of Solr servers to achieve fault tolerance and high availability for both indexing and search functionality. For using SolrCloud, read the documentation in Solr Reference Guide: Getting Started with SolrCloud.

NOTE

Be aware, that according to ZooKeeper Ensemble Configuration you should not use Solr's embedded ZooKeeper, but an external ZooKeeper setup.

3.4.1 Connecting CoreMedia applications

SolrCloud uses ZooKeeper for cluster configuration and coordination. In a Solr-Cloud setup, CoreMedia applications are not configured with the URL(s) of one or multiple Solr servers, but with ZooKeeper address(es) instead. ZooKeeper maintains the list of currently active Solr servers that clients can use for search and indexing.

To configure a CoreMedia application to connect to SolrCloud, you simply set the property `solr.cloud=true` to enable SolrCloud mode, and property `solr.zookeeper.addresses` with the addresses of the ZooKeeper servers. For example:

```
solr.cloud=true
solr.zookeeper.addresses=zookeeper1:2181,zookeeper2:2181,zookeeper3:2181
```

3.4.2 SolrCloud Configuration

In SolrCloud, ZooKeeper maintains the index configuration files and ensures that the whole cluster uses the same configuration. To this end, the config sets from the Solr Home directory need to be uploaded to ZooKeeper initially. In the following example, the config sets for *Content Feeder*, *CAE Feeder* and *Elastic Social* indices are uploaded to ZooKeeper.

```
cd apps/solr/modules/search/solr-config
$SOLR_SCRIPT zk upconfig -z :40085 -d target/solr-config/configsets/content/conf -n content
```

```
$SOLR_SCRIPT zk upconfig -z :40085 -d target/solr-config/configsets/cae/conf     -n cae
$SOLR_SCRIPT zk upconfig -z :40085 -d target/solr-config/configsets/elastic/conf -n elastic
```

The shell variable `$SOLR_SCRIPT` is set to the path of the `bin/solr` script from the Solr installation. The `-z` option specifies the ZooKeeper address. In the example, ZooKeeper is running at port 40085. See also Solr Reference Guide: ZooKeeper File Management.

When developing with *CoreMedia Blueprint*, you can start Solr locally from Maven as described in Section 3.1, "Starting" [16]. You can also start a single node SolrCloud cluster with embedded ZooKeeper to test the configuration by invoking "`mvn exec:exec@start-solr-cloud`". You still need to upload configuration files manually as described above. See the description and configuration of the `exec-maven-plugin` in file `apps/solr/modules/search/solr-config/pom.xml` for details.

3.5 Reindexing

There are several reasons why you might want to reindex all index documents. This includes changes in the Solr configuration how text gets indexed (for example to activate certain features such as stemming) and changes in configuration or code so that different data is sent to Solr. In any case, reindexing a whole index is a very expensive operation and takes some time.

See also the chapter about reindexing in Solr Reference Guide: Reindexing.

3.5.1 Reindexing Elastic Social Indices

Elastic Social indices can be reindexed by invoking the JMX operation `reindex` of interface `com.coremedia.elastic.core.api.search.management.SearchServiceManager` of an *Elastic Social* application.

You can find the `SearchServiceManager` MBean of the `elastic-worker` application for tenant `media` under the object name `com.coremedia:application=elastic-worker,type=searchServiceManager,tenant=media`.

The operation takes the name of the index without prefix and tenant as parameter. For example, to reindex the Solr core `blueprint_media_users` the operation has to be invoked with the parameter `users`. It then clears the index and reindexes every index document afterwards.

3.5.2 Partial Reindexing of Content Feeder Indices

You can make a *Content Feeder* reindex selected contents by invoking JMX operations of MBean `com.coremedia:type=AdminBackgroundFeed,application=content-feeder`, or by using the `reindex` Spring Boot actuator endpoint. Reindexing happens in a background thread, and will not block indexing of repository changes.

If custom code or configuration was changed, and contents of a certain type need to be indexed differently, you can trigger reindexing for all content items of a specific type. To this end, the "`reindexByType`" JMX operation can be used. Alternatively, you can send an HTTP POST request to the actuator endpoint at `http://host:port/actuator/reindex` with an `application/json` body like `{"contentType": "CMArticle"}`.

The "`reindexByQuery`" JMX operation is more generic and takes a *Unified API* query as documented in interface `com.coremedia.cap.content.query.QueryService`. All contents that match the query (and are not excluded from feeding) will be reindexed. Alternatively, a JSON body like `{"query": "BELOW PATH '/Sites'"}` can be sent to the `reindex` actuator endpoint. Make sure to escape quotes correctly, if you call the actuator endpoint with a tool like 'curl' from the command-line.

Both operations can take an optional comma-separated list of `com.coremedia.cap.feeder.FeedableAspect` IDs. If specified, the Feeder will not reindex whole index documents but uses partial updates for these aspects only. See Section 4.1.2, "Partial Updates" [45] for details on partial updates. For example, specify the aspect "`issues`" to reindex content issues only. For the actuator endpoint, aspects can be specified as additional JSON attribute, for example `{"contentType": "CMArticle", "aspects": "issues"}`

The "`reindexAll`" JMX operation triggers reindexing of all contents. You can also restrict it to certain partial update aspects. A POST request with empty JSON object to the `reindex` actuator endpoint can be used alternatively.

If the *Content Feeder* is stopped during reindexing, it will continue with the next content after restart. The reindexing progress is persisted in a special document in the index itself.

There's another JMX operation "`cancel`" at the same MBean to cancel a running reindexing operation. The reindexing progress is available with MBean attribute "`NumberOfPendingContents`".

3.5.3 Partial Reindexing of CAE Feeder Indices

You can make a *CAE Feeder* reindex selected contents by invoking JMX operation "`reindexContent`", "`reindexByQuery`", or "`reindexByType`" of MBean `com.coremedia:type=CaeFeeder`.

Reindexing can be expensive, if many contents are affected. It may block indexing of other repository changes. It can be used for example to reindex all content of a specific type after indexing rules for that type have been changed. To this end, the "`reindexByType`" JMX operation can be used.

The "`reindexByQuery`" operation is more generic and takes a *Unified API* query as documented in interface `com.coremedia.cap.content.query.QueryService`. All contents that match the query (and are not excluded from feeding) will be reindexed.

An alternative way to trigger a partial reindexing without directly performing JMX operations, is by using the `reindex` Spring Boot actuator endpoint at `http://host:port/actuator/reindex`. The provided handler accepts `HTTP POST` requests with an `application/json` body containing the following `JSON` valid fields: "`ids`", "`contentTypes`" and "`query`". The listed elements can be used in isolation or combined together to select the content needed to be re-indexed. For performance reasons the information sent are always merged together to form one single query execution. Others not recognized data are ignored by default.

The endpoint accepts `JSON` objects with at least one of the properties:

- `ids`: single numerical content id or a comma-separated list of ids `{"ids": 1234}` or `{"ids": "1234,5678"}`
- `contentTypes`: comma-separated list of doc-types `{"contentTypes": "CMArticle,CMMedia"}`
- `query`: string containing a query in a format accepted by `com.coremedia.cap.content.query.QueryService`. When the query value is used in conjunction with `contentTypes`, the two are combined together and the type selection should be omitted from the value. The `JSON` `{"query": "TYPE CMArticle : BELOW PATH '/Sites/Calista'"}` and `{"contentTypes": "CMArticle", "query": "BELOW PATH '/Sites/Calista'"}` are retrieving the same contents.

When the fields are used together, `contentTypes` and `query` are joined by logical AND, while `ids` will be added by logical OR to the overall contents to re-index. The following example `{"ids": "1234,5678", "contentTypes": "CMArticle,CMMedia", "query": "BELOW PATH '/Sites/Calista'"}` will generate a single query like `"id=1234 OR id=5678 OR TYPE CMArticle,CMMedia: BELOW PATH '/Sites/Calista'"`. In case data are invalid or the request will result in a malformed query, a `HTTP` status code `500` will be returned to the client.

After re-indexing was triggered, the *CAE Feeder* will mark affected content items as invalid in the database, before the actual re-indexing starts. If a request for re-indexing affects many content items and the *CAE Feeder* is restarted while content items are still marked as invalid, then some content items may not be re-indexed. Content items that have already been marked as invalid will be re-indexed, even if the *CAE Feeder* was restarted.

3.5.4 Reindexing Content Feeder and CAE Feeder Indices from Scratch

The most simple approach for *Content Feeder* and *CAE Feeder* indices is to clear the existing index and restart the Feeder. The Feeder will then reindex everything from scratch. In most cases this is not what you want, because search will be unavailable (or only return partial results) until reindexing has completed. See Section 4.6.4, “Clear Search Engine index” [78] and Section 5.3.2, “Resetting” [87] for instructions how to clear an existing index for *Content Feeder* and *CAE Feeder*, respectively.

A better solution is to feed a new index from scratch but keep using the old one for search until the new index is up to date. Applications can use the new index when reindexing is complete. When everything is fine, the old index can be deleted afterwards. This approach does not only have the advantage of avoiding search downtime but makes it also possible to test changes before enabling the index for all search applications.

Reindexing in Existing Solr

This approach is appropriate if the current Solr version is to be kept and just data needs to be reindexed.

To prepare a new index, you need to set up an additional Feeder and configure it to feed the new index. The new Feeder instance will eventually replace the existing Feeder instance.

The following steps describe the procedure for a standalone Solr server with optional leader/follower replication. For a SolrCloud cluster, different steps have to be taken. See Solr Reference Guide: Reindexing - Index to Another Collection for reindexing into another SolrCloud collection.

1. Add a new Solr core for the new index. The Solr Admin UI supports adding Solr cores in general but currently still lacks support for named config sets (SOLR-6728), so you have to create the new core with a HTTP request. To this end, you just need to send a request to the following URL with correct parameters, for example by opening it in your browser.

   ```
   http://<hostname>:<port>/solr/admin/cores?action=CRE
   ATE&name=<name>&configSet=<configSet>&dataDir=data
   ```

   a. Replace `<hostname>` and `<port>` with host name and port of the Apache Solr leader.

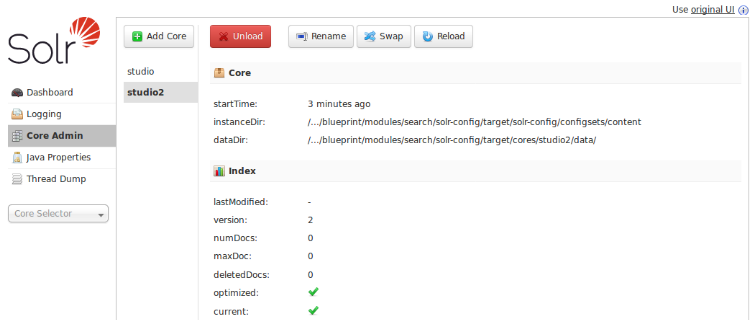
b. Replace `<name>` with the name of the new core. You can choose any name you like as long as no such core and no such directory below the configured `coreRootDirectory` exists yet. If you are using *Elastic Social* you should also avoid names that start with the configured `elastic.solr.index-prefix` followed by an underscore (for example, `blueprint_`) to avoid name collisions with automatically created Solr cores.

c. Replace `<configSet>` with the name of the config set of the new core. This should be "`content`" for *Content Feeder* indices and "`cae`" for *CAE Feeder* indices. Alternatively you can set it to the name of a custom config set, if you are using differently named config sets in your project.

2. Check that the new core was successfully created in the `coreRootDirectory`. There should be a new subdirectory with the name of the newly created core which contains a `core.properties` file. For example, if a core `studio2` with config set `content` was created, then `<coreRootDirectory>/studio2/core.properties` should contain something like:

```
#Written by CorePropertiesLocator
#Mon Feb 27 14:45:44 UTC 2017
name=studio2
dataDir=data
configSet=content
```

You can also open the Solr Admin UI at `http://<hostname>:<port>/solr`, which shows the newly created core on the **Core Admin** page:



startTime: 3 minutes ago
instanceDir: /.../blueprint/modules/search/solr-config/target/solr-config/configsets/content
dataDir: /.../blueprint/modules/search/solr-config/target/cores/studio2/data/

lastModified: -
version: 2
numDocs: 0
maxDoc: 0
deletedDocs: 0
optimized: ✓
current: ✓

Figure 3.1. New Solr Core

3. Set up a new Feeder instance and configure it to feed into the new Solr core. In the *Content Feeder*, the name of the new core must be configured with property `solr.content.collection`. In the *CAE Feeder*, the name of the new core must be configured with property `solr.cae.collection`.

For example, to configure a newly set up *Content Feeder* to feed into the new core with name `studio2`, set in `application.properties`:

```
solr.content.collection=studio2
```

In case of a *CAE Feeder*, you must also configure it with a separate empty database schema.

4. Start the new Feeder and wait until the new index is up-to-date, for example by checking the log files or searching for a recent change in the new index. Depending on the size of the content repository this may take some time.
5. Stop the Feeders for both the old and new Solr core.
6. To activate the new index, it's now time to swap the cores so that the new core replaces the existing one. You can swap cores with the **[Swap]** button on the **Core Admin** page of the Solr Admin UI. Afterwards, all search applications automatically use the new core, which is now available under the original core name.

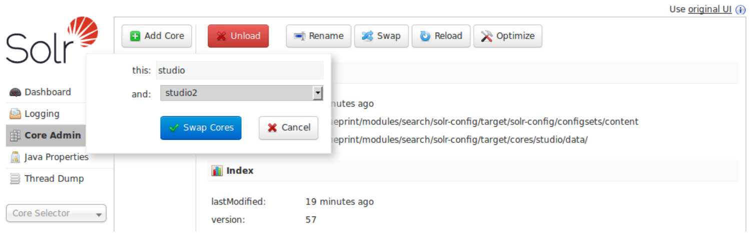


Figure 3.2. Swap Solr Cores

It's important to understand that this operation does not change the directory structure in `<coreRootDirectory>` but just the `name` property in the respective `core.properties` files. For the example of swapping cores `studio` and `studio2`, you now have a newly indexed Solr core named `studio` in directory `<coreRootDirectory>/studio2`. You can verify this by looking into its `core.properties` file:

```
#Written by CorePropertiesLocator
#Mon Feb 27 15:06:27 UTC 2017
name=studio
dataDir=data
configSet=content
```

7. Reconfigure the new Feeder instance to use the new core under the original name. To this end, the value of property `solr.content.collection` for the *Content Feeder* or property `solr.cae.collection` for the *CAE Feeder* needs to be changed accordingly. Start the new Feeder instance.

   For example, to configure the *Content Feeder* to feed into the new core which is now available under name `studio`, set in `application.properties`:

```
solr.content.collection=studio
```

8. If you're using Solr replication, the new index will be replicated automatically to the Solr followers after a commit was made on the Solr leader for the new core. The restart of the Feeder in the previous step caused a Solr commit so that replication should have started automatically. If not, a Solr commit can also be triggered with a request to the following URL, for example in your browser with `http://localhost:40080/solr/studio/update?commit=true` for the Solr core named `studio` on the Solr leader running on localhost and port 40080.

   Note that depending on the index size, replication of the new core may take some seconds up to a few minutes during which the old index is still used when searching from Solr followers. You can see the progress of replication on the Solr follower's Admin UI on page **Replication** after selecting the corresponding core.

9. To clean things up, you can now unload the old Solr core from the Solr leader with the **[Unload]** button on the **Core Admin** page of the Solr Admin UI. In the example, this would be the core named `studio2`.

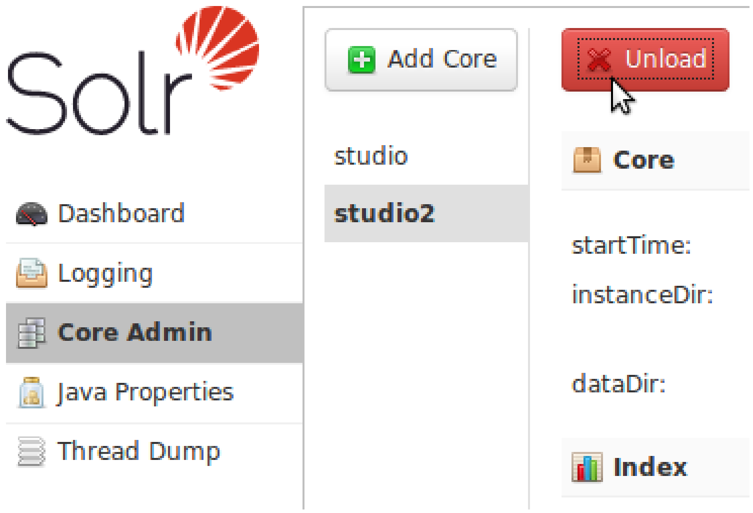


Figure 3.3. Unload old Solr Core

If you like, you can now also delete the old Feeder installation and the directory of the old Solr core with its index. In this example that would be `<coreRoot Directory>/studio`

NOTE

You can use HTTP requests to perform the **[Swap]** and **[Unload]** actions instead of using the Solr Admin UI as described above. For details, see Solr Reference Guide: CoreAdmin API.

Reindexing in New Solr

This approach is appropriate, if the Solr version is to be updated (e.g., in the course of an AEP update) and data needs to be reindexed in a dedicated instance of this new Solr version.

To prepare the new Solr index, you need to set up additional Feeders and an instance of the new Solr version. The additional Feeders must be configured to feed the new index. The new Feeder instances and Solr will eventually replace the existing Feeder instances and Solr.

The following steps outline the procedure.

1. Provide instances of updated Feeders and Solr from the CoreMedia release with the updated Solr version. Do this on dedicated new hosts to avoid port clashes with existing Feeders and Solr. Also provide dedicated database schemas for new Feeders and space for new Solr indexes. Configure Feeders to attach to Content Servers of the existing installation while sending index data to the new Solr.

   There should now be a logical setup as in the following diagram (excerpt from full CMS). Light-gray boxes represent components from the existing CMS, light-green boxes represent components with updated versions.

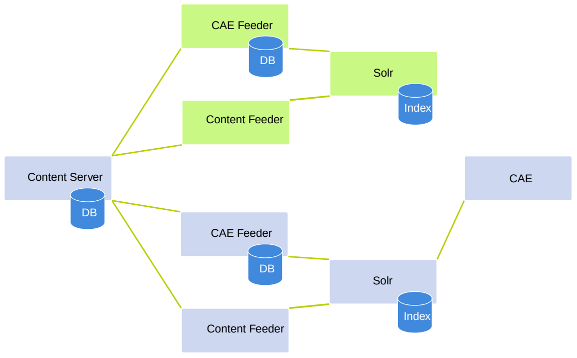


Figure 3.4. Setup for Reindexing in New Solr

NOTE

Although mixed operation of Feeders and Content Servers in different versions is generally not supported, the Feeders will typically connect successfully to Content Servers from several releases back. Actual success of mixed operation needs to be tested for any concrete setup.

2. Start new Feeders and check that data is indexed in new Solr. For that purpose, go to the new Solr's Admin UI and wait until all cores have caught up with the cores of the old Solr installation in terms of number of indexed documents. A small difference may be neglected as the new Feeders will continue to catch up when the CMS is fully updated.
3. When the new Solr has indexed all (or the majority of) documents, proceed with updating the CMS as usual. You may leave the running new Solr installation untouched. Feeders should be shut down temporarily, though, to avoid unnecessary errors in logs.

   Reconfigure Solr clients to attach to the new Solr installation.

   Do not restart old Feeder and Solr installations with the updated CMS. They may be removed at a later point.

There should now be a logical setup as in the following diagram (excerpt from full CMS). Light-gray boxes represent components from the old CMS (now shut down), light-green boxes represent components with updated versions.

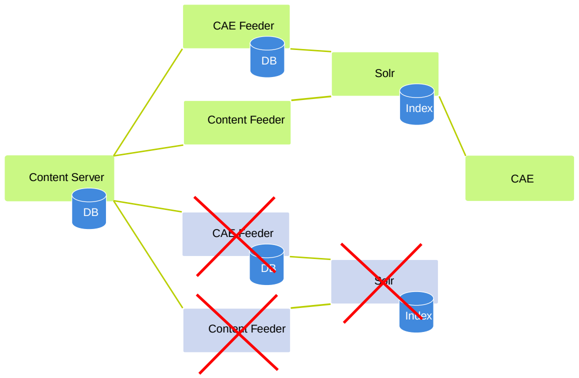


Figure 3.5. Setup after Reindexing in New Solr and Updating CMS

4. After successful update, the old Feeders and Solr, together with their databases and indexes, may be deleted.

3.6 Creating Backups

In order to create a backup of the *CoreMedia Search Engine* you have to do two things in the following order:

1. Back up the state of the Feeders
2. Back up the Solr index

If you plan to back up the *Content Server* database at the same time, make sure to take the backup of the *Content Server* after backing up Feeder state and Solr index. This is important for restoring backups: The restored *Content Server* database must not be older because *CAE Feeder* database and *Content Feeder* Solr index store timestamps from the *Content Server*. These timestamps must exist in the *Content Server* to successfully start a Feeder after restoring a backup.

3.6.1 Back up the state of the Feeders

For the *Content Feeder* this step can be skipped, as it stores its state in the Solr index.

The *CAE Feeder* stores its state in a dedicated SQL database. This database has to be backed up and it is important to do so *before* taking the backup of the Solr index.

The reason for this is that if the restored Solr index is newer than the restored *CAE Feeder* database, the *CAE Feeder* might feed some index documents once again which is okay, but if the Solr index were older than the *CAE Feeder* database, index changes between the time of the Solr backup and *CAE Feeder* backup could be lost.

If your database / tools provide the feature of hot backup, you do not have to stop the *CAE Feeder* for taking backups.

3.6.2 Back up the Solr index

See Solr Reference Guide: Backup and Restore for the documentation how to take backups of the Solr index.

3.7 Restoring Backups

In order to restore from a backup of the *CoreMedia Search Engine* (see Section 3.6, “Creating Backups” [34]) you have to do two things in the following order:

1. Restore the backup of the *CAE Feeder*
2. Restore the backup of the Solr index

   For details, see Solr Reference Guide: Backup and Restore.

NOTE

In case you also performed a backup of a *Content Server* database, you have to restore this database before restoring the *CAE Feeder* and the Solr Index.

3.8 Searching in Different Languages

Processing steps for multi-language use

The *CoreMedia Search Engine* enables you to search in content of many languages. This requires some preliminary processing steps:

- Detecting the used language
- Splitting the text into searchable words
- Indexing the words into language dependent fields
- Searching in language dependent fields

These steps are highly customizable. The default configuration works well for most languages, so you do not necessarily need to change the configuration. However, Solr provides advanced support for many languages, that can be enabled to further improve search functionality.

3.8.1 Details of Language Processing Steps

The following paragraphs describe some details of the language processing steps.

Language Detection

Language detection

The Solr config sets `content` and `cae` for *Content Feeder* and *CAE Feeder* indices define the field `language` in their index schema in `schema.xml`. This field holds the language of the index document, if available.

It's recommended to let feeder applications set the language of index documents, if a language is available at that point. The *Content Feeder* and *CAE Feeder* applications of the *CoreMedia Blueprint* automatically set the `language` field for `CMLocalized` documents and content beans. See Section 4.2.2, “Content Configuration” [54] and Section 5.4.3, “Customizing Feedables” [90] to learn how to set index fields such as the `language` field in the *Content Feeder* and *CAE Feeder*.

If the `language` field is not already set by the feeder, then the search engine will try to detect the language of the index document by its content and set the field accordingly. To this end, the file `solrconfig.xml` configures the Solr `LangDetectLanguageIdentifierUpdateProcessorFactory` to detect the language of incoming index documents. It is described in detail in

Solr Reference Guide: Language Detection. See Section 6.6, “Supported Languages in Solr Language Detection” [138] in the reference of this manual for a list of supported languages. The language code from that list is stored as value in `language` field.

NOTE

Language detection may not always return the correct language, especially for very short texts. The language should be set by the feeder, if it is known in advance.

Knowing the language of an index document is a prerequisite to index text in a language-specific way. The search engine can put the text in a field that is specially configured for that language, for example with correct rules to break the text into single words.

Tokenization

Tokenization

To provide search functionality, the search engine needs to split text into searchable words. This process is commonly referred to as tokenization or word segmentation. Most languages use whitespace to separate words, which means that text can be tokenized by splitting it at whitespaces. Chinese, Japanese and Korean texts cannot be tokenized this way. Chinese and Japanese don't use whitespaces at all and Korean does not use whitespaces consistently. Apache Solr supports tokenization and analysis for many languages, for details refer to Solr Reference Guide: Document Analysis in Solr.

Indexing into language dependent fields

Indexing into language dependent fields

Text must be indexed into a separate language dependent field to tokenize or preprocess it according to its language. This is the basis for efficient language dependent search. Depending on your requirements you can configure different tokenization strategies or add some language-specific analysis steps such as stemming. In both cases you need to configure language dependent fields.

Example

A customized `schema.xml` defines the index fields `name` and `name_ja`. If the feeder feeds an index document with Japanese text in its name, then the text will be indexed in the field `name_ja`. The index field `name` will be empty for that document. Another document contains German text in its name that will be indexed in the field `name`, because `schema.xml` does not define a field `name_de`.

Search in language-dependent fields

Search in language-dependent fields

When searching in *Studio*, Blueprint *CAE* or with the *Unified API*'s `SearchService`, searching is automatically performed across multiple fields including language-dependent fields. To this end, the *Search Engine* contains a CoreMedia-specific Solr query parser named `cmdismax`. This parser is a variant of Solr's standard dismax query parser (see Solr Reference Guide: DisMax Query Parser for more details). The improvements of the `cmdismax` parser are support for wildcard searches (for example, core*) and searching across all language-dependent fields.

The default Solr config sets for *Content Feeder* and *CAE Feeder* indices configure search request handlers to use the `cmdismax` parser in `solrconfig.xml`: the handler `/editor` for editorial search in the `content` config set and the handler `/cmdismax` for website search in the `cae` config set.

If you want to use a different query parser such as the default Lucene query parser or the Solr Extended DisMax (edismax) query parser, you must explicitly search in all required language-dependent fields. For the edismax query parser this would mean enumerating all required language-dependent fields in the *`qf`* (query fields) parameter.

3.8.2 Configuring Multi-Language Search

Configuring multi-language search

The process of multi-language search configuration consists of the following steps, that are described in the next paragraphs:

1. Defining text tokenization and filtering in different field types
2. Defining index fields for different languages
3. Defining the fields from which the language is determined
4. Defining where the detected language is stored.
5. Configuring language dependent field handling
6. Configuring the search request handler

NOTE

It's not necessary to adapt the feeder configuration for multi-language support. Feeders just feed text into some fields (for example `name` and `textbody`) and the search engine puts the text into the correct language-dependent fields.

Configuring different field types

Configuring different field types

Text tokenization and filtering in Apache Solr can be configured in the file `conf/schema.xml` of a Solr config set. For example in `<solr-home>/configsets/content/conf/schema.xml` for the `content` config set.

For each field, a field type is defined. That is, which kind of data is written to this field. In the default `content` config set, for example, the field `textbody` is of type `text_general`. The field type is connected with a certain analyzer which is used to tokenize and filter the text. The default configuration contains some field types with different analyzers, for example:

- `text_general`, configured with Solr StandardTokenizer with reasonable cross-language defaults
- `text_zh`, configured for tokenization of Simplified and Traditional Chinese (outcommented by default)

Apache Solr provides special field types for lots of languages in its example configuration, for example `text_ja` for Japanese and `text_ko` for Korean. Most of these field types are not defined in the default configuration of the *CoreMedia Search Engine* to keep the configuration files simple and avoid unnecessary overhead. If required, add field types from the Solr example configuration to your configuration. You can find these additional field types in the configuration file `server/solr/configsets/_default/conf/managed-schema` after downloading and unpacking the Apache Solr distribution. You can download Solr from http://solr.apache.org.

Example

If you index text of one language only and want to use a special field type, you can simply change field definitions from type `text_general` to the chosen field type in `schema.xml`, for example to `text_de` for German text.

```
<fields>
  ...
  <field name="textbody" type="text_de" ... />
</fields>
```

Configuring multi-language index fields

Configuring multi-language index fields

You need to define language-dependent fields for all languages that need a special analyzer. To do so, simply add a new field element with the name followed by the language code. Section 6.6, “Supported Languages in Solr Language Detection” [138] in the reference shows the list of supported languages.

NOTE

Note, that language-dependent fields must be indexed. A field declaration with attribute `indexed="false"` cannot be used as language-dependent field.

Fields in the `content` config set must also be declared with attribute `stored="true"` or `docValues="true"` to make it possible to use partial updates in the *Content Feeder*.

The following example shows fields and additional types in `<solr-home>/configsets/content/conf/schema.xml` for using dedicated field types for Simplified Chinese, Japanese, Korean while using the field type `text_general` for other languages. The example shows the fields `name` and `textbody` of the `content` config set. To enable sorting on field `name`, it uses Solr field types based on `SortableTextField`.

```
<field name="name"                   type="text_gen_sort"
                                     indexed="true" stored="true"/>
<field name="name_ja"                type="text_ja_sort"
                                     indexed="true" stored="true"/>
<field name="name_zh-cn"             type="text_zh_sort"
                                     indexed="true" stored="true"/>
<field name="name_ko"                type="text_ko_sort"
                                     indexed="true" stored="true"/>
...
<field name="textbody"               type="text_general"
                                     indexed="true" stored="false"
                                     multiValued="true"/>
<field name="textbody_ja"            type="text_ja"
                                     indexed="true" stored="false"
                                     multiValued="true"/>
<field name="textbody_zh-cn"         type="text_zh"
                                     indexed="true" stored="false"
                                     multiValued="true"/>
<field name="textbody_ko"            type="text_ko"
                                     indexed="true" stored="false"
                                     multiValued="true"/>

<!-- field types "text_general", "text_gen_sort" and "text_zh" are
     already defined in the default configuration, the latter
     needs to be enabled, because it's outcommented by default -->

<!-- field types "text_ja" and "text_ko" can be
     copied from the Apache Solr example configuration -->

<!-- field types "text_ja_sort", "text_zh_sort" and
     "text_ko_sort" can be copied from the field types without
     "_sort" suffix, adapting the name and replacing
     "solr.TextField" with "solr.SortableTextField" -->
...
```

In the above example, Japanese text goes into `name_ja` and `textbody_ja`, Simplified Chinese text goes into `name_zh-cn` and `textbody_zh-cn`, Korean text goes into `name_ko` and `textbody_ko` and text from all other languages is indexed in the fields `name` and `textbody`.

Besides Simplified Chinese you can also configure Traditional Chinese text with the fields `name_zh-tw` and `textbody_zh-tw`. The language code `zh` from previous CoreMedia releases is not generated anymore, but existing fields `name_zh` and `textbody_zh` are still used as fallback when indexing and searching.

Configuring language detection

Configuring language detection

By default, the *Search Engine* detects the language of the index fields `name` and `textbody` for *Content Feeder* indices (config set `content`) and of index field `textbody` for *CAE Feeder* indices (config set `cae`). Both use the field `language` to store the detected language. Language detection is skipped if the field `language` has been set by the feeder. You can change these settings in the config set's file `conf/solrconfig.xml` below the element `<updateRequestProcessorChain>` with class `LangDetectLanguageIdentifierUpdateProcessorFactory`:

```
<processor class="org.apache.solr.update.processor.
    LangDetectLanguageIdentifierUpdateProcessorFactory">
  <str name="langid.fl">textbody,name</str>
  <str name="langid.langField">language</str>
  <str name="langid.fallback">en</str>
</processor>
```

The parameter `langid.langField` defines the index field that will be filled with the language code of the document. Section 6.6, “Supported Languages in Solr Language Detection” [138] in the reference shows the list of supported languages. The value in parameter `langid.fl` is a comma-separated list of index fields that are used for language detection. The parameter `langid.fallback` configures English as fallback if the language can not be detected from the text.

For more details about the Solr `LangDetectLanguageIdentifierUpdateProcessorFactory`, see Solr Reference Guide: Language Detection.

Configuring language-dependent field handling

Configuring index feeding

In order to be flexible, the *Search Engine* separates language detection and the handling of language-dependent fields. Therefore, field handling is configured in a separate class.

You can change these language-dependent field handling settings in the config set's file `conf/solrconfig.xml` below the element `<updateRequestProcessorChain>` with class `LanguageDependentFieldsProcessorFactory`.

```
<processor class="com.coremedia.solr.update.processor.
           LanguageDependentFieldsProcessorFactory">
  <str name="languageField">language</str>
  <str name="textFields">textbody,name</str>
</processor>
```

The parameter `languageField` defines the index field that contains the language code of the document. This must be the same value as configured for language detection above.

The value in the parameter `textFields` is a comma-separated list of fields whose content should be put into language-dependent fields if such fields exist for the language. Normally, this is the same value as configured for language detection except if you want to exclude some text fields from language detection.

Configuring the search request handler

Configuring the search request handler

By default, the search request handlers for *Content Feeder* and *CAE Feeder* indices are configured in `solrconfig.xml` to search across multiple index fields. For example, the config set `content` configures the `/editor` search request handler with the `qf` parameter to search in fields `textbody`, `name` and `numericid`. Matches in the field `name` are scored higher than matches in `textbody` because of the configured `^2` boost. Note that the language-dependent fields `name_*` and `textbody_*` are not configured here but will be picked up automatically.

```
<requestHandler name="/editor" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="defType">cmdismax</str>
    <str name="echoParams">none</str>
    <float name="tie">0.1</float>
    <str name="qf">textbody name^2 numericid^10</str>
    <str name="pf">textbody name^2</str>
    <str name="mm">100%</str>
    <str name="q.alt">*:*</str>

    <str name="suggest.spellcheck.dictionary">textbody</str>
  </lst>
  <arr name="last-components">
    <str>suggest</str>
    <str>spellcheck</str>
  </arr>
</requestHandler>
```

Adapt the configuration of the request handler's `qf` and `pf` parameters if you want to use other default search fields.

The predefined request handlers can also be used in custom search applications. They can be selected in SolrJ, for example, by calling `org.apache.solr.client.solrj.SolrRequest.setPath("/cmdismax")`. If you prefer Solr's standard search handler you will have to explicitly search across language-dependent fields, by constructing "OR" queries in a Lucene query syntax or by configuring all fields for standard Solr dismax or edismax query parsers, for instance.

4. Searching for Content

This chapter describes how to configure and operate content search for editorial applications such as *CoreMedia Studio* or custom editor applications. While you may use this search service also for website search, in most cases for website search it makes more sense to search for content beans as described in Chapter 5, *Searching for CAE Content Beans* [80].

There are the following building blocks to search for content:

- The *Content Feeder* to feed the *Search Engine* with content
- The *Search Engine* itself, which indexes the content and makes it searchable
- The search service in the *Content Server*, which provides the search functionality of the *Search Engine* to custom applications that use the Unified API Search Service, and to the `cm search` command-line tool.
- Search applications such as the *Studio* or custom ones, which connect to the *Search Engine* directly

The *Search Engine* itself is covered in Chapter 3, *Search Engine* [15]. This chapter describes the operation and configuration of the *Content Feeder*, *Studio* the *Content Server's* search service and the configuration of the *Search Engine* for content search in custom applications.

The next sections describe

- Concepts of content search in Section 4.1, “Concepts” [44]
- Configuration of the *Content Feeder* in Section 4.2, “Configure the Content Feeder” [52]
- Configuration of the search service of the *Content Server* in Section 4.3, “Configure Search for the Content Server” [68]
- Configuration of the *Search Engine* for *Studio* in Section 4.4, “Configure Search for Studio” [70]
- Modification of the *Search Engine* index schema for custom search applications in Section 4.5, “Modify the Search Index” [75]
- Operation of the *Content Feeder* in Section 4.6, “Operation of the Content Feeder” [76]
- Hints for implementing a custom search application in Section 4.7, “Implementing Custom Search” [79]

4.1 Concepts

The *Content Feeder* sends properties and metadata of CoreMedia content to the *CoreMedia Search Engine*. The *Search Engine* indexes that data, and provides the possibility to search for the contents. The *Content Feeder* is an application that connects to the *Content Server* and to the *Search Engine*.

The *CoreMedia Content Server* provides a search service which hides the functionality of the *CoreMedia Search Engine* from clients. The server contacts the *CoreMedia Search Engine* to serve client search requests. Custom clients that use the Unified API `SearchService` get the search results directly from the *CoreMedia Content Server*.

It is also possible to send search requests from custom clients directly to the *CoreMedia Search Engine* using the native API of the underlying search engine. This is recommended in most cases because the search service of the *Content Server* does not support all search features of Apache Solr and adds some performance overhead compared to a direct connection. The *Studio* back-end is an example for a search client that sends search requests directly to the *Search Engine*.

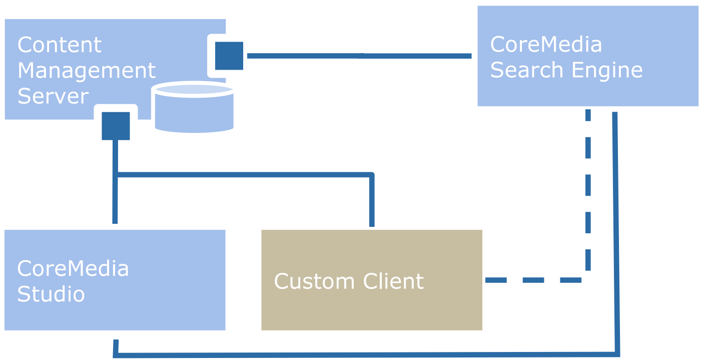


Figure 4.1. Search Engine Integration

The *CoreMedia Content Feeder* feeds an index which is needed for the full-text search feature in *CoreMedia Studio*. Multiple *Content Feeder*s can use the same *CoreMedia Search Engine* but require separate indices.

To provide full-text search for contents in the *Content Delivery Environment*, a separate *Content Feeder* can be set up that connects to the *CoreMedia Master Live Server* and feeds another index.

4.1.1 Feeding the Search Engine

When the *Content Feeder* starts for the first time, it iterates over the contents in the repository and sends them to the *Search Engine* for indexing. After this initialization phase, the *Content Feeder* sends contents to the *Search Engine* after they have changed or when they are newly created.

When the *Content Feeder* restarts, it automatically continues its work with the next content that needs to be indexed. This content is determined from a timestamp stored by the *Content Feeder* in the same index of the *Search Engine*. During restart the *Content Feeder* retrieves the timestamp from the *Search Engine* to continue feeding.

The *CoreMedia Search Engine* indexes textual data from content properties and a number of metadata attributes such as the path of the content, the name of its creator and the last time the content was published. In the configuration of the *Content Feeder* you can restrict the indexed contents by their type and the indexed properties by their name and type. Note, that the *CoreMedia Search Engine* only indexes the latest or working version of CoreMedia documents.

4.1.2 Partial Updates

The *Content Feeder* can use partial updates if only content metadata has changed. This means, it does not need to send the whole content to the search engine but just a small set of changed metadata, for example a changed path after contents have been moved to another place in the repository. This can greatly improve performance, especially if lots of contents are affected and expensive operations such as parsing text from PDF can be avoided.

The *Content Feeder* can use partial updates, if the connected search engine supports it. Apache Solr supports partial updates if index fields are configured with `stored="true"` or `docValues="true"` as in the default configuration. See the description of the configuration properties `feeder.solr.partial-updates.enabled`, `feeder.solr.partial-updates.skip-index-check` and `feeder.content.partial-update-aspects` in Section 3.10, "Content Feeder Properties" in *Deployment Manual* for more details.

4.1.3 Content Issues

The *Content Feeder* can index content issues that are reported by content validators. For details about content validators, see Section 9.23.1, "Validators" in *Studio Developer Manual*. Validators need to be registered as Spring beans in the application context of the *Content Feeder* to make their reported issues available in the search index. If content issues are indexed, it will become possible to find content items with errors or warnings in the search view of the *Studio* library.

In the Solr index, issues are represented as nested index documents of their corresponding content. These nested issue documents contain the value `NESTED` in the index field `feederstate`, and data about the issue in several index fields like `issueCode` and `issueSeverity`. For details about index fields, have a look at the field definitions in the file `schema.xml` of the *Content Feeder* index.

In the default configuration, issue indexing is enabled. It can be disabled by setting the *Content Feeder* configuration property `feeder.content.issues.index` to `false`. If enabled, the Solr schema must contain the index fields `_root_` and `_nest_path_` in the Solr configuration file `schema.xml` for the *Content Feeder* index. The file from the Blueprint already contains these fields, but they were not always present in previous releases like 2107 and before. When adding or removing these fields, you must recreate the Solr index from scratch, and let the *Content Feeder* index all content items. It would not be sufficient to trigger reindexing of content items in an existing index.

If enabled, issue computation and indexing causes additional work for the *Content Feeder*, and can reduce its throughput. With enabled issue feeding, content issues are still not computed during initial feeding of an empty index, so that initial feeding is not delayed. The *Content Feeder* will index issues for all content items immediately after the index has been initialized. This happens with lower priority and does not block feeding of editorial changes.

Note, that indexed issues are not always up-to-date. Issues are recomputed and reindexed immediately, when the properties of the corresponding content have changed. Issues are not updated immediately, if other content items have changed or, for example, if a content was just renamed without a change to its properties. To eventually still have correct issues in the search result, the *Content Feeder* periodically recomputes and reindexes issues of all content items with a configurable delay. For details, see the configuration properties starting with `feeder.content.issues` in Section 3.10, “Content Feeder Properties” in *Deployment Manual*. Periodic issue reindexing happens with lower priority in the background and does not block feeding of editorial changes.

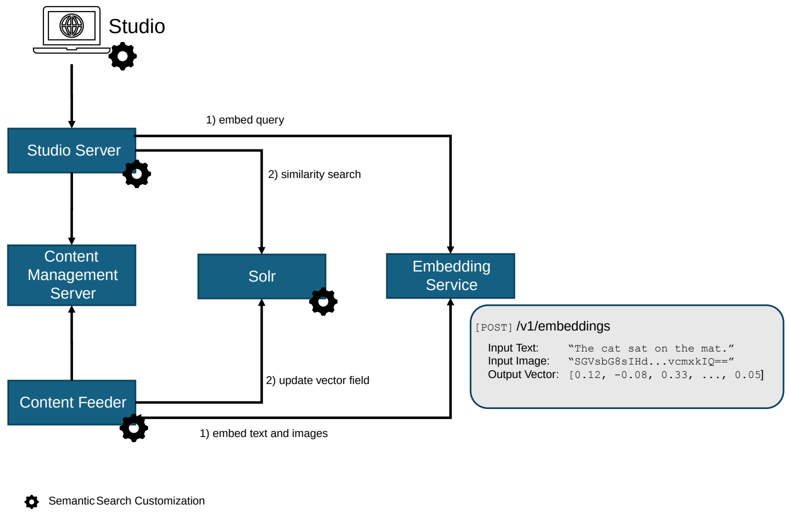
Section 6.2, “Content Feeder Metrics” [111] describes some metrics that may be helpful to understand *Content Feeder* performance in general and the impact of issue feeding. Furthermore, you may query Solr directly to check how up-to-date indexed issues really are: The Solr field `issuesUpdated` of an indexed content contains the date when indexed issues were last computed for that content. The Solr Stats Component can be used in a Solr query to check the maximum age of issues in the index. For example, a native Solr query could be extended with `stats=true&stats.field=issuesUpdated` to get the minimum and maximum date values, or with `stats=true&stats.field={!func}ms(NOW,issuesUpdated)` to get the minimum and maximum age in milliseconds. The Solr Stats Component is described in the Solr Reference Guide, section: Stats Component.

4.1.4 Semantic Search

The Semantic Search Feature enables the editor to search for content items by their meaning. Instead of matching keywords, the search finds content items that are semantically similar to the search query. For this the content is represented as vectors in a high-dimensional space, called embeddings. These embeddings are generated using machine learning models that capture the meaning of the content. The *Content Feeder* and the *Studio Server* connect to a managed *Embedding Service* provided by Amazon Bedrock in the default configuration.

Please note that you need to license the Semantic Search Feature to use it in your environment. Please contact the CoreMedia support for more information.

CAUTION
Please note that Amazon Bedrock may incur additional costs depending on your usage.



```
[POST] /v1/embeddings
  Input Text:     "The cat sat on the mat."
  Input Image:    "SGVsbG8sIHd...vcmxkIQ=="
  Output Vector:  [0.12, -0.08, 0.33, ..., 0.05]
```

Figure 4.2. Semantic Search Architecture

- **Solr:** Stores index data for fulltext search and embeddings for semantic search.
- **Embedding Service:** Provides vector representations for content and queries.
- **Content Feeder:** Extracts content, requests embeddings, and sends them to the Solr.
- **Studio Server:** Uses embeddings to perform semantic search and return relevant results.

The *Content Feeder* with semantic search enabled sends text, images and textual download data to the *Embedding Service* to retrieve embedding vectors for each content item. These embeddings are stored in the search index alongside the other indexed content data.

The *Studio Server* receives the search request from *Studio*. When a user submits a query, the *Studio Server* sends the query term to the *Embedding Service* which now generates an embedding for the query. This embedding is used to perform a vector similarity search against the indexed content embeddings to find the most relevant results. This search is performed by Solr.

This architecture enables semantic search capabilities, allowing users to find content based on meaning and context rather than just keyword matching.

The blueprint workspace includes example configurations for both embedding service integrations:

- `global/deployment/docker/compose/development-semantic-search-aws-nova.yml`
- `global/deployment/docker/compose/development-semantic-search-aws-titan.yml`

4.1.4.1 Embedding Service Configuration

To enable the Semantic Search feature, ensure that your CoreMedia license includes the required feature flag. Please contact CoreMedia support for licensing details.

The *Embedding Service* is configured via Spring Boot properties in both the *Content Feeder* and the *Studio Server*. These properties specify which embedding model to use and how to connect to the service.

Multiple embedding service implementations are supported:

- **Amazon Bedrock Nova (default)**: Offers multimodal embeddings, supporting both text and images.
- **Amazon Bedrock Titan**: Provides text-only embeddings, fully integrated with Spring AI.

Shared AWS Connection Properties: The embedding service is integrated via Spring AI. The following AWS connection properties are required for both models:

- `spring.ai.bedrock.aws.region`
- `spring.ai.bedrock.aws.access-key`
- `spring.ai.bedrock.aws.secret-key`

Amazon Bedrock Nova: To enable Amazon Bedrock Nova use the following properties:

- `ai.model.embedding=bedrock-nova`
- All properties starting with `ai.bedrock-nova.*`

For a complete list of Nova configuration properties, see the Content Feeder in *Deployment Manual* and Studio Server in *Deployment Manual* property references in the Deployment Manual.

Amazon Bedrock Titan: This model is fully integrated via Spring AI. Use the following properties:

- `spring.ai.model.embedding=bedrock-titan`
- All properties starting with `spring.ai.bedrock.titan.*`

For details, refer to the Spring AI Bedrock Titan documentation.

For further configuration details, see the following sections:

- **Content Feeder:** Section 4.2.3.8, “Configuring Semantic Search for Content Feeder” [67]
- **Studio Server:** Section 4.4.4, “Configuring Semantic Search for Studio Server” [74]

4.1.4.2 Solr Storage Requirements for Vectors

Storing vector embeddings for semantic search requires significantly more space in the search index than traditional metadata. The following table provides example sizes for float32 vectors (1024 dimensions) per document:

Documents	Estimated Size
1	~4KB
1,000	~4–7MB
1,000,000	~4–7GB

Table 4.1. Estimated Storage Requirements for Vectors

With multimodal embedding, both media and text vectors are stored for images. This means two vector fields per image, effectively doubling the storage required compared to plain text documents.

Example: For 2,000 images (each with 2 vectors) and 1,000 articles (each with 1 vector), the total is 5,000 vectors. Using a planning estimate of 8MB per 1,000 vectors, this results in approximately 40MB of storage required for the vectors.

These are rough estimates; actual storage requirements may vary depending on your Solr and Embedding Service configuration.

4.1.5 Batches

For better performance the *Content Feeder* sends batches to the *Search Engine*. A batch contains changes of multiple contents. A batch that was sent to the *Search Engine* is called an *open batch* until all contained changes have been written to the *Search Engine*'s index persistently.

4.1.6 Error conditions

If the *Content Feeder* or the *Search Engine* is unable to process a certain content, an error index document is indexed instead. It serves as placeholder for the original content in the index of the *Search Engine*.

When a content contains binary data of an unsupported format, no error index document is written. Instead, such contents are indexed without the binary data and the content can still be found based on its other fields.

Error index documents contain the value `ERROR` in the index field `feederstate` and are not returned as search result by the *Content Server* or *Studio*. You can search for error index documents using the administration page of the *Content Feeder*. An error index document is replaced with the correct content when the content changes in the *CoreMedia Content Server* and the cause of the error has been removed.

Communication problems to the *CoreMedia Search Engine* lead to search errors in clients. The *Content Feeder* retries feeding until the *Search Engine* responds successfully. Search requests from clients succeed as soon as the communication problems have been resolved.

4.1.7 Restrictions

The *CoreMedia Search Engine* provides a fast and efficient full-text search for indexed contents. However, because of the asynchronous nature of the indexing process, search results do not always reflect the current state of the repository. A content may need a couple of seconds after it was sent to the *Search Engine*, and before it appears in the search results. If you need always up-to-date results and can accept slower query execution, then take a look at the built-in query feature of the *CoreMedia Content Server* that is described in Section 5.8, “Query Service” in *Unified API Developer Manual*.

Indexed content issues can be outdated for an even longer time. Issues for a content are updated in the index after the properties of that content have changed. Other changes, like editing a linked content, or moving a content to another folder, do not lead to an immediate update of a content's issues.

The *CoreMedia Search Engine* supports search for the latest document version or working version only. If you want to search for older versions you have to use the query feature of the *CoreMedia Content Server* or use the *CoreMedia CAE Feeder* to index the required data as part of content beans.

4.2 Configure the Content Feeder

Configure the *Content Feeder* to provide full-text search for contents of the *Content Management Environment*, for example in *CoreMedia Studio*.

Configuration of the *Content Feeder* is described in the following sections:

- Section 4.2.1, “Required Configuration” [52]

  In this section you can read how to configure the essential Feeder settings. These are the connection settings with the Search Engine and the Content Server.

- Section 4.2.2, “Content Configuration” [54]

  This section explains which information for which content types and properties you want to index into which fields. This configuration is not required, because by default all relevant content types and properties are indexed for search.

- Section 4.2.3, “Advanced Configuration” [62]

  Here, you can read how to optimize your *Content Feeder* in order to improve speed and error handling.

For custom search applications, you may also want to set up a *Content Feeder* connected to the *CoreMedia Master Live Server* to provide full-text search for contents in the *Content Delivery Environment*. Note that for website search you typically search for content beans that were fed by a *CAE Feeder*, see Chapter 5, *Searching for CAE Content Beans* [80] for details.

4.2.1 Required Configuration

For connection of the *Content Feeder* to a *Content Server*, refer to Section 3.12.1, “Unified API Spring Boot Client Properties” in *Deployment Manual*. The following sections cover configuration specific to the *Content Feeder*.

The *Content Feeder* requires a user account to access the contents of the *Content Server*. During the initialization of the *Content Server* a dedicated user is created with the name and password `feeder`. For security reasons, change the password afterwards. The account requires at least read rights on the content to be indexed. A license of the service `feeder` is consumed by a running *Content Feeder*.

4.2.1.1 Configuring the Search Engine Location

The *Content Feeder* needs to connect to the search engine. Configure the URL of Apache Solr in property `solr.url` as in the following example:

```
solr.url=http://localhost:40080/solr
```

For SolrCloud, do not configure property `solr.url` but set `solr.cloud=true` and the ZooKeeper address(es) instead as in the following example:

```
solr.cloud=true
  solr.zookeeper.addresses=zookeeper1:2181,zookeeper2:2181,zookeeper3:2181
```

If Apache Solr has been secured and needs HTTP Basic authentication, you must also configure the required user name and password in the properties `solr.username` and `solr.password`.

4.2.1.2 Configuring the Search Engine Collection

Configure the property `solr.content.collection` with the name of the *CoreMedia Search Engine* collection or Solr Core.

The Solr core is the index used by the *Content Feeder*. See Section 3.2, “Solr Home and Core Directories” [17] for a description of Solr cores and their configuration in Apache Solr.

Example

```
solr.content.collection=studio
```

If the collection does not exist in Solr yet, the *Content Feeder* will create it when started. It will create the collection based on the Solr config set "`content`". If necessary, a different config set name can be configured with *Content Feeder* property `solr.content.config-set`.

4.2.2 Content Configuration

4.2.2.1 Configuring Content Types

You can restrict the indexed contents by their type with the following two properties:

```
feeder.content.type.includes=Content_
feeder.content.type.excludes=\
  EditorPreferences,Preferences,Dictionary,Query
```

NOTE

Configuration not mandatory: The default configuration includes all content types except *EditorPreferences, Preferences, Dictionary and Query*.

The property `feeder.content.type.includes` contains a comma-separated list of content types to be included. Contrary the property `feeder.content.type.excludes` contains a comma-separated list of content types to be excluded. With a specified type all subtypes are included and excluded, respectively. It is an error to specify the same content type in both properties. Rules for more specific types override rules for less specific types.

CAUTION

Note, that the *Content Feeder* does not update already processed contents after changing the content types to index. A configuration change only affects newly processed contents. You must reindex as described in Section 3.5, "Reindexing" [24], if you want to update all contents or contents of a certain type.

4.2.2.2 Configuring Properties for Indexing

You can restrict the indexed properties of a content by their name and type. You can further restrict the indexed XML properties by their grammar and the indexed blob properties by their MIME type and size.

If you want to restrict the content fields, you can specify a map entry with included or excluded fields for some or all content types. A map entry for a super type is valid for all subtypes, if not overridden with an entry for a subtype. If no entry is specified for a content type or its ancestors, all content properties are included. The wildcard `*` stands for all properties and can be used to include or exclude all properties of a type. Note however that you can either configure a list of included or excluded properties for a certain type but not both, and property lists from different entries will not be merged.

NOTE

Configuration not mandatory in Blueprint: The default configuration includes all String and CoreMedia RichText XML properties. In the Blueprint, the default configuration also limits blob properties to the MIME types `text/*`, `application/pdf`, `application/msword` and `application/vnd.openxmlformats-officedocument.wordprocessingml.document` (docx files) that are not larger than 5 MB.

You can configure indexed content properties by their name by customizing the Spring beans `feederContentPropertyIncludes` and `feederContentPropertyExcludes` in the file `applicationContext.xml`.

The following example configures the *Content Feeder* to index only the properties 'Author' and 'Text' of content type Article and all properties of content type Picture except the property 'Copyright'. Only the listed properties will be indexed for content type Article and only the not listed properties for content type Picture will be indexed. Content types not listed here will by default be indexed with all properties if not configured otherwise via excluded or included properties.

```
<customize:append id="feederContentPropertyIncludesCustomizer"
bean="feederContentPropertyIncludes">
  <map>
    <entry key="Article" value="Author,Text"/>
  </map>
</customize:append>

<customize:append id="feederContentPropertyExcludesCustomizer"
bean="feederContentPropertyExcludes">
  <map>
    <entry key="Picture" value="Copyright"/>
  </map>
</customize:append>
```

Note that it is an error to specify both included and excluded properties for the same type.

See the description of the beans in file `applicationContext.xml` for more details.

NOTE

The CoreMedia Feeder applications use *Apache Tika* for text extraction from binary formats. You can find the list of formats supported by Tika at https://tika.apache.org/3.2.3/formats.html. Note however, that the Blueprint Feeder applications do not include all transitive Tika libraries to reduce the total number of dependencies and avoid potential version conflicts. Libraries for less common formats such as NetCDF scientific files and many more have been excluded. Have a look at the classpath of the Feeder applications and extend it if needed. Libraries for common formats such as Microsoft Office or PDF are supported by default.

You can also change the indexed content properties by their type. The following example shows the Blueprint default configuration for property types:

```
# indexed property types
feeder.content.property-type.string=true
feeder.content.property-type.integer=false
feeder.content.property-type.date=false
feeder.content.property-type.link-list=false
feeder.content.property-type.struct=false
```

```
# Indexed xml properties, configured by xml grammar
# comma separated grammar names (as used in the content
# type definition, attribute Name of element XmlGrammar)
feeder.content.property-type.xml-grammars=coremedia-richtext-1.0
```

```
# The MIME-types of indexed blobs
feeder.blob.enabled[text/*]=true
feeder.blob.enabled[application/pdf]=true
feeder.blob.enabled[application/msword]=true
feeder.blob.enabled[application/vnd.openxmlformats-officedocument.wordprocessingml.document]=true

# The maximum size of blobs.
# Larger values are ignored and will not be sent to the Search Engine.
feeder.blob.max-size[*/*]=5MB
```

CAUTION

Note, that the *Content Feeder* does not update already processed contents after changing the properties. A configuration change only affects newly processed contents. You must reindex as described in Section 3.5, “Reindexing” [24], if you want to update all contents or contents of a certain type.

4.2.2.3 Configuring Fields to Index in

The *Content Feeder* can be configured to index content properties into special index fields. You can search for content in these fields if your *Search Engine* indexes these fields. To this end, the fields must be added to the file `schema.xml` in the Apache Solr config set for the *Content Feeder* in directory `<solr-home>/configsets/content/conf`. Please refer to the Apache Solr documentation for more information.

NOTE

Configuration not mandatory: By default, all content properties are indexed in the index field `textbody`. They are also indexed in fields whose name starts with `cm` and ends with the lowercase name of the property – if such fields exist in the index. For example, a property `Headline` is indexed in the field `cmheadline`. This configuration allows you to use different index field names.

The *Content Feeder* supports two types of field configuration, the `PropertyField` and the `FeedablePopulator`. A `PropertyField` maps a content property to an index field and whether the property value should also be indexed in the field `textbody`. The more flexible `FeedablePopulator` interface allows you to populate a `Feedable` object from a given content.

If you configure a new field in the Solr `schema.xml`, you can search for text in that specific field. Note, that searching in specific fields is not possible in *CoreMedia Studio* but only in custom search applications using *CoreMedia* APIs or native Search Engine APIs.

The following example adds a field with the name `myfield` to the *Apache Solr* `schema.xml`. Fields must be configured with the attributes `indexed="true"` to enable support for searching, and `stored="true"` (or at least `docValues="true"`) to support partial updates. For a more information, see the *Apache Solr* documentation.

```
<fields>
  ...
  <field name="myfield" type="text_general"
                        stored="true" indexed="true"/>
</fields>
```

Configuring PropertyField Beans

Beans of type `PropertyField` are configured in a `customize:append` element in file `applicationContext.xml`. A `PropertyField` bean requires the attributes `name`, `doctype` and `property`. Attribute `name` specifies the index field name as configured in the Solr `schema.xml`. Attribute `doctype` specifies the name of the content type and attribute `property` specifies the name of the content property, which is mapped to the index field. Furthermore, it's possible to configure whether the property's value should also be indexed in the field `textbody`. By default, it will be indexed in `textbody` but you can disable this by setting the attribute `textBody="false"`. Another optional attribute `ignoreIfEmpty` configures whether a missing or empty property value should be indexed. The default value is `false` meaning an empty value is indexed.

Note that excluded content types will not be indexed even if a matching `PropertyField` is configured. The following example configures indexing of the property *headline* of content type *Article* into the index field `myfield`. It is not indexed in field `textbody` and empty values are ignored:

```
<customize:append id="addFeedableProperties"
bean="contentConfiguration" property="propertyFields">
  <list>
    <bean class="com.coremedia.cms.feeder.content.PropertyField">
      <property name="name" value="myfield"/>
      <property name="doctype" value="Article"/>
      <property name="property" value="headline"/>
      <property name="textBody" value="false"/>
      <property name="ignoreIfEmpty" value="true"/>
  </list>
</bean>
</customize:append>
```

Configuring FeedablePopulator Beans

`FeedablePopulator` Spring beans are configured in the list property `feedablePopulators` and/or in the list property `partialUpdateFeedablePopulators` of Spring bean `index` using a `customize:append` element, for example in file `applicationContext.xml`. There are some existing `FeedablePopulator` public API classes that you may use. For example:

- `PropertyPathFeedablePopulator`: Index specific values from a struct content property.
- `XPathFeedablePopulator`: Extracts a text fragment from an XML content property.
- `ImageDimensionFeedablePopulator`: Set image attributes like image orientation, dimension, and size category.
- `ContentStatusFeedablePopulator`: Set the content status (approved, deleted, etc).

Your own populator classes just need to implement the `FeedablePopulator` interface and can then be configured the same way. The method `FeedablePopulator#populate` will be called with a `com.coremedia.cap.con tent.Content object, that is the type parameter T of FeedablePopulator implementations must be Content or a super type of Content.

Populators registered at property feedablePopulators of Spring bean index are called when a content gets added or updated and the whole content data is sent to the search engine. Populators registered at property partialUpdateFeedablePopulators are called for partial updates, when only content metadata is sent to the search engine. You can also register a custom FeedablePopulator at both list properties and use method isPartialUpdate of the passed in Feedable to detect whether a partial update is being processed. Method getUpdatedAspects returns which aspects of the index document are changed with a partial update.

CAUTION

When you configure a FeedablePopulator for a Solr index field, you must make sure that the type of the index field matches the possible values. For example, you should never configure a PropertyPathFeedablePopulator or an XPathFeedablePopulator to set a numeric or date index field. Even if a nested struct property at the configured path is typically used for dates, some content may contain a text value and cause indexing errors. In such a case, you should use a custom FeedablePopulator implementation and check the value type instead.

PropertyPathFeedablePopulator

The PropertyPathFeedablePopulator is configured with a dot-separated property path to index a specific property value from a struct content property. The first name in the property path denotes the struct property itself while the following names specify nested properties of the struct. The constructor argument type selects the type of the content. The argument element maps to the field name in the index. Furthermore, it's possible to configure whether the value should also be indexed in the field textbody using the property textBody. By default, it will not be indexed in the textbody field but you can enable this by setting the property textBody to true.

The following example configures a populator to feed the index field author from a localSettings.metadata.author struct property path of Article contents.

```
<customize:append id="addAuthorFeedablePopulator"
 bean="index" property="feedablePopulators">
  <list>
    <ref bean="authorFeedablePopulator"/>
  </list>
</customize:append>

<bean class=
"com.coremedia.cap.feeder.populate.PropertyPathFeedablePopulator">
```

```
  <constructor-arg index="0" name="type" value="Article"/>
  <constructor-arg index="1" name="propertyPath"
                   value="localSettings.metadata.author"/>
  <constructor-arg index="2" name="element" value="author"/>
</bean>
```

XPathFeedablePopulator

`XPathFeedablePopulators` extract text of a fragment from an XML property. The fragment is specified with an XPath expression in the property `XPath`. The required property `element` maps to the field name in the index. The property `contentType` selects the type of the content and the property `property` selects the content property. Furthermore, it's possible to configure whether the property's value should also be indexed in the field `textbody`. By default, it will be indexed in `textbody` but you can disable this by setting the property `textBody` to `false`. The namespaces property defines namespaces which can be used in the XPath expression.

The following example configures a populator to feed the index field `tabletext` from `Text` properties in `Article` contents.

```
<customize:append id="addFeedablePopulators"
 bean="index" property="feedablePopulators">
  <list>
    <bean
     class="com.coremedia.cap.feeder.populate. \
      XPathFeedablePopulator">
      <property name="element" value="tabletext"/>
      <property name="contentType" value="Article"/>
      <property name="property" value="Text"/>
      <property name="textBody" value="false"/>
      <property name="XPath" value="/r:div/r:table"/>
      <property name="namespaces">
        <map>
 <entry key="r"
  value="http://www.coremedia.com/2003/richtext-1.0"/>
        </map>
      </property>
    </bean>
  </list>
</customize:append>
```

ImageDimensionFeedablePopulator

The `ImageDimensionFeedablePopulator` is used to detect the orientation (portrait, square, landscape), dimension (width, height) and size category (small, medium, large) of an image. After detection the following index fields are set:

- **`imageOrientation`**: portrait (value=0), square (value=1) and landscape (value=2) mode.
- **`imageSizeCategory`**: small (value=0), medium (value=1) and large (value=2) mode.
- **`imageWidth`**: image width in pixel.
- **`imageHeight`**: image height in pixel.

- **`imageMaxLength`**: maximum of `imageWidth` and `imageHeight`

An image has portrait(landscape) mode if its height(width) is larger than its width(height). If width and height are equal, it has square mode. An image is categorized as large(as medium) if its width is larger than or equal to the configured `largeWidth` (`mediumWidth`) property and its height is also larger than or equal to the configured `largeHeight` (`mediumHeight`) property. The image is small, if its width is smaller than `mediumWidth` or its height is smaller than `mediumHeight`.

To categorize image orientation (portrait, square, landscape) and image size (small, medium, large), some filter properties must be configured:

- **`docType:`** the type of the content to be indexed, including subtypes
- **`widthPropertyName`**: the property name of the content which holds the width value
- **`heightPropertyName:`** the property name of the content which holds the height value
- **`dataPropertyName:`** the property name of the content which holds the image data. The value of this object must be of type `com.coremedia.cap.common.Blob`.

You must set either `widthPropertyName` and `heightPropertyName` or `dataPropertyName` or both. If the two dimension properties do not exist, the blob data is read to determine the dimension.

- **`largeWidth:`** lower bound width of large images
- **`largeHeight:`** lower bound height of large images
- **`mediumWidth:`** lower bound width of medium images
- **`mediumHeight:`** lower bound height of medium images

The following example shows an `ImageDimensionFeedablePopulator` configuration.

```
<customize:append id="addFeedablePopulators"
 bean="index" property="feedablePopulators">
  <list>
    <bean
     class=
"com.coremedia.cap.feeder.populate.ImageDimensionFeedablePopulator">
      <property name="largeWidth"
       value="${feeder.populator.imageDimension.largeWidth}"/>
      <property name="largeHeight"
       value="${feeder.populator.imageDimension.largeHeight}"/>
      <property name="mediumWidth"
       value="${feeder.populator.imageDimension.mediumWidth}"/>
      <property name="mediumHeight"
       value="${feeder.populator.imageDimension.mediumHeight}"/>
      <property name="docType"
       value="${feeder.populator.imageDimension.docType}"/>
      <property name="widthPropertyName"
       value="${feeder.populator.imageDimension.widthPropertyName}"/>
      <property name="heightPropertyName"
       value="${feeder.populator.imageDimension.heightPropertyName}"/>
      <property name="dataPropertyName"
       value="${feeder.populator.imageDimension.dataPropertyName}"/>
    </bean>
  </list>
</customize:append>
```

The property values of the populator bean are filtered from a property file.

ContentStatusFeedablePopulator

The `ContentStatusFeedablePopulator` classifies a content in one of four status categories:

- **`0:`** in production (not approved and not deleted)
- **`1:`** approved (place and content)
- **`2:`** published (place and content)
- **`3:`** deleted

After classification, the status value of the content is stored in the index field `status`. The following example shows a `ContentStatusFeedablePopulator` configuration:

```
<customize:append id="addFeedablePopulators"
bean="index" property="feedablePopulators">
  <list>
    <bean class="com.coremedia.cap.feeder. \
   populate.ContentStatusFeedablePopulator"/>
  </list>
</customize:append>
```

CAUTION

Note, that the *Content Feeder* does not update already processed contents after changing the fields to index. A configuration change only affects newly processed contents. You must reindex as described in Section 3.5, “Reindexing” [24], if you want to update all contents.

4.2.3 Advanced Configuration

4.2.3.1 Configuring Batch Handling

The *Content Feeder* sends content changes to the *CoreMedia Search Engine* in batches. You can configure the number of index documents in a batch and when to send a batch. Batch sizes and sending rate influence the indexing speed.

NOTE

Configuration not mandatory: Normally you do not need to change the default settings.

The *Content Feeder* sends a batch when one of the following conditions is fulfilled:

- The maximum number of index documents in a batch has been reached.
- The batch size in bytes would exceed the configured maximum if more index documents were added.
- Maximum time delays are reached.

Use these properties to configure batch settings:

- `feeder.batch.max-size`: The maximum number of index documents in a batch. A smaller batch may be sent if the maximum byte size is reached before.
- `feeder.batch.max-bytes`: The maximum number of bytes allowed in a batch. A smaller batch may be sent if the maximum batch size is reached before.
- `feeder.batch.send-idle-delay`: The maximum time in milliseconds to wait before sending a new batch if the *Content Feeder* is idle. This value should be small to update the index quickly and have up-to-date search results after some content was changed by an editor.
- `feeder.batch.send-max-delay`: The maximum time in milliseconds to wait sending a new batch if the batch is not yet full. This value normally is higher to avoid sending small batches, for example when large amounts of content are created by an import process.

CAUTION

Note, that open batches are kept in main memory. You have to reserve `2*maxBatchByteSize` bytes for the batches.

4.2.3.2 Configuring Error Handling

The *Content Feeder* automatically retries operation after some communication problems with the *CoreMedia Search Engine*. The following properties configure the retry behavior:

- `feeder.batch.retry-send-idle-delay`: The maximum time in milliseconds to wait sending a failed batch again, if the *Content Feeder* is idle.
- `feeder.batch.retry-send-max-delay`: The maximum time in milliseconds to wait sending a failed batch again, if the batch is not yet full.
- `feeder.solr.send-retry-delay`: The delay in milliseconds between a failed batch sending and the next try. The default value is 30000.
- `feeder.content.retry-connect-to-index-delay`: The delay between retries to connect to the *Search Engine* on startup. The default value is 10s.
- `solr.connection-timeout`: The connection timeout set on the SolrJ `SolrClient`. It determines how long the client waits to establish a connection without any response from the server. The default value is 0. That means it will wait forever. You can configure the timeout as `java.time.Duration`.
- `solr.socket-timeout`: The socket timeout set on the SolrJ `SolrClient`. It determines how long the client waits for a response from the server after the connection was established and the request was already sent. The default value is set to 10 minutes.

4.2.3.3 Configuring Tika

The Feeder uses Apache Tika to extract text and metadata from blob properties for indexing.

Tika Caching

Extracted text and metadata is cached in heap memory to avoid repeated potentially expensive processing of the same blob. Caching can improve feeding performance, if a content was modified but its blob property was not changed, or if the same blob value is used in multiple content items. Use configuration property `cache.capacities.feeder.tika.heap` to configure the cache capacity in estimated bytes. The Blueprint configures a default capacity of 10 MB as follows:

Example

```
cache.capacities.feeder.tika.heap=10485760
```

Tika Parsers

Tika provides parsers for various formats, which can be customized in a special Apache Tika XML configuration file. The default configuration covers typical formats so that a custom configuration is rarely needed. If you need to fine-tune the configuration of Apache Tika, please have a look at the documentation of Apache Tika for the format of the Tika Config XML file. The location of this file can be configured with the Spring configuration property `feeder.tika.config`. The value of this property is a Spring Resource location. The following example configures an Apache Tika Config file from the local file system:

Example

```
feeder.tika.config=file:/opt/path/tika-config.xml
```

4.2.3.4 Configuring Tika Zip Bomb Prevention

Apache Tika uses a heuristic to detect 'Zip Bombs', that is files that expand to a huge amount of text when parsed. Parsing such files can lead to severe memory and/or performance issues in the Feeder. To prevent denial of service attacks or problems caused by malfunctioning parsers, the prevention is enabled by default. If Tika detects a blob to be a 'Zip Bomb', no text will be extracted from that blob and a warning will be logged instead. Note that 'Zip Bomb' attacks are not limited to ZIP files but can also occur for example with PDF files.

Normally, there's no need to change the configuration but if you encounter false positives, you may want to tweak the settings for Tika's heuristic or even turn off the prevention. You can disable 'Zip Bomb' detection with property `feeder.tika.zip-bomb-prevention.enabled=false` and tweak the heuristic with various properties starting with `feeder.tika.zip-bomb-prevention`. For details, see Section 3.10, “Content Feeder Properties” in *Deployment Manual*.

4.2.3.5 Configuring Tika metadata extraction

In addition to extracting body text, Tika can extract metadata for some binary formats such as the creator of a Microsoft Word file. You can use the configuration properties `feeder.tika.append-metadata` and `feeder.tika.copy-metadata` to extract and index metadata from binary formats.

The property `feeder.tika.append-metadata` takes a comma-separated list of metadata identifiers. The *Content Feeder* simply appends the matching metadata values to the indexed body text when Apache Tika extracts such a value.

The property `feeder.tika.copy-metadata` takes a comma-separated list where each entry consists of a metadata identifier followed by an equal sign (`=`) and the name of the index field the metadata should be copied to. When a matching metadata value is found, it will be stored in the configured index field. Note that with Apache Solr target index fields must be defined as `multiValued="true"` to avoid indexing errors if there are multiple metadata values with the same identifier. See also Section 4.5, “Modify the Search Index” [75].

Example

```
feeder.tika.copy-metadata=dc:creator=author
```

The above example configures the *Content Feeder* to store the `dc:creator` metadata value in the index field `author`. Note that the index field must be declared in the Solr schema for this to work.

Metadata identifiers are specific to Apache Tika. You can find some of them in the API documentation of Apache Tika class `org.apache.tika.metadata.TikaCoreProperties`.

4.2.3.6 Configuring Tika ParseContext

Tika uses an instance of `org.apache.tika.parser.ParseContext` to pass advanced configuration to its parsers. If required, you can customize the ParseContext in the Spring context by adding entries to the map bean `tika ParseContext`. The map uses `java.lang.Class` objects as keys and values must be instances of their keys. The following example configures a custom Tika `org.apache.tika.extractor.DocumentSelector` to decide whether text gets extracted from embedded documents such as attachments in a PDF.

Example

```
<customize:append id="tikaConfigCustomizer" bean="tikaParseContext">
  <map key-type="java.lang.Class" value-type="java.lang.Object">
    <entry key="org.apache.tika.extractor.DocumentSelector">
      <bean class="com.example.CustomTikaDocumentSelector"/>
    </entry>
  </map>
</customize:append>
```

4.2.3.7 Configuring Updates of Rights Rule Changes

The *Content Feeder* indexes the groups with potential read rights to a content in the index field `groups`. The set of groups is then used to narrow a user's search down to the contents where he could have read rights to. This is an optimization to reduce the number of search results on which the client must check read rights and for more accurate search suggestion numbers. The downside of this optimization is a slightly increased feeding load, because the index field must be updated for all contents below a folder whose rights rules have changed. You can disable this optimization by setting the property `feeder.con` `tent.index-groups` to `false`. If you've set that property to `false`, then you must also configure *Studio* and *CoreMedia Content Server* to not add a query condition for the indexed groups. To this end, set the *Studio* property `studio.rest.searchService.useGroupsFilterQuery` and the *CoreMedia Content Server* property `solr.useGroupsFilterQuery` to `false`. In general, it's recommended to keep property `feeder.content.index-groups` at its default value `true`.

Because rights changes may lead to lots of reindexing, the *Content Feeder* treats these changes differently than normal editorial changes. It updates index documents after rights changes in the background when it is idle. Rights changes are processed with lower priority than editorial changes. Feeding of rights changes does not block feeding of editorial changes.

4.2.3.8 Configuring Semantic Search for Content Feeder

The *Content Feeder* connects to the *Embedding Service* to extract semantic embeddings during content feeding. These embeddings are then sent to the search engine for indexing.

For details on the supported embedding services and their configuration, refer to Section 4.1.4.1, “Embedding Service Configuration” [49].

Content Feeder Specific Configuration: All configuration properties specific to the *Content Feeder* semantic search feature start with `feeder.content.semantic-search.*`. For a full list, see Section 3.10, “Content Feeder Properties” in *Deployment Manual*.

4.3 Configure Search for the Content Server

To search for documents in custom applications that use the Unified API SearchService, you need to configure the connection to Apache Solr at the *Content Server*. The *CoreMedia Content Server* connects to the Apache Solr to handle search requests for its clients.

4.3.1 Enable or Disable Search

Search functionality is disabled by default. You can enable it by setting property `cap.server.search.enable` to `true`. It is typically enabled in the *Content Management Server* and disabled in the *Master Live Server* and *Replication Live Server*. If disabled in the *Content Management Server*, no search functionality will be available for custom clients that use the *Unified API* `SearchService` and the `cm search` command-line tool.

If search functionality is enabled, the connection to Apache Solr must be configured at the *CoreMedia Content Server* as follows:

4.3.2 Configuring the Search Engine Location

Configure the URL to connect to Apache Solr in property `solr.url`, for example:

```
solr.url=http://localhost:40080/solr
```

You can also configure multiple comma-separated URLs in this property if you want to use multiple Solr follower nodes for failover and simple load balancing.

For SolrCloud, do not configure property `solr.url` but set `solr.cloud=true` and the ZooKeeper address(es) instead as in the following example:

```
solr.cloud=true
solr.zookeeper.addresses=zookeeper1:2181,zookeeper2:2181,zookeeper3:2181
```

If Apache Solr has been secured and needs HTTP Basic authentication, you must also configure the required user name and password in the properties `solr.username` and `solr.password`.

4.3.3 Configuring the Search Engine Collection

Configure the property `solr.content.collection` with the name of the collection, for example:

```
solr.content.collection=studio
```

4.4 Configure Search for Studio

To search for contents in *CoreMedia Studio*, you need to configure it to connect to Apache Solr. Solr also provides search suggestions for the *Studio* library, which can be fine-tuned in the Solr configuration file `solrconfig.xml`.

4.4.1 Configuring the Search Engine Location

Configure the URL to connect to Apache Solr in property `solr.url`, for example:

```
solr.url=http://localhost:40080/solr
```

For up-to-date search results this should be the URL to the Solr leader if you are using a Solr leader/follower setup with index replication.

For SolrCloud, do not configure property `solr.url` but set `solr.cloud=true` and the ZooKeeper address(es) instead as in the following example:

```
solr.cloud=true
solr.zookeeper.addresses=zookeeper1:2181,zookeeper2:2181,zookeeper3:2181
```

If Apache Solr has been secured and needs HTTP Basic authentication, you must also configure the required user name and password in the properties `solr.username` and `solr.password`.

4.4.2 Configuring the Search Engine Collection

Configure the property `solr.content.collection` with the name of the collection, for example:

```
solr.content.collection=studio
```

4.4.3 Configure Studio Search Suggestions

NOTE

Configuration not mandatory: Search suggestions in *Studio* work with the default configuration. This section describes how you can configure the index fields used for suggestions and how you can tune the performance of suggestions.

CoreMedia Studio shows autocomplete search suggestions when a user starts typing search queries in the library window. These suggestions are based on the indexed content and computed by a special search component in Apache Solr, which can be configured in the Solr configuration file `<solr-home>/configsets/content/conf/solrconfig.xml`.

The configuration consists of:

- **Request handler parameters**

  Studio uses the Solr request handler `/editor` for searching and getting search suggestions. Suggestions are configured with parameter `suggest.spellcheck.dictionary` as in the following example (the other parameters may vary in your configuration):

  ```
  <requestHandler name="/editor" class="solr.SearchHandler">
   <lst name="defaults">
    <str name="defType">cmdismax</str>
    <str name="echoParams">none</str>
    <float name="tie">0.1</float>
    <str name="qf">textbody name^2 numericid^10</str>
    <str name="pf">textbody name^2</str>
    <str name="mm">100%</str>
    <str name="q.alt">*:*</str>
    <str name="suggest.spellcheck.dictionary">textbody</str>
   </lst>
   ...
  ```

  The parameter `suggest.spellcheck.dictionary` references a Suggester dictionary to compute suggestions from. This dictionary must be configured in `solrconfig.xml` as well as described further below. In the default configuration it is named after the index field `textbody` but you can use different dictionary names as you like. You can also use multiple dictionaries to compute suggestions from the content of multiple index document fields. To this end, you just need to repeat the element `<str name="suggest.spellcheck.dictionary">` multiple times with different values. Note that you must also configure multiple dictionaries if you want to suggest words from language dependent fields. For example, if you've defined the fields `textbody`, `textbody_en` and `textbody_de` in the index schema as described in Section 3.8, "Searching in Different Languages" [36], then you need to add three dictionaries to get suggestions from all of these fields.

- **Request handler components**

  The same request handler `/editor` is configured to use the necessary search components for suggestions as shown below. These referenced components are configured as `<searchComponent ...>` elements in `solrconfig.xml` as well.

```
<requestHandler name="/editor" class="solr.SearchHandler">
  <lst name="defaults">
    ...
  </lst>
  <arr name="last-components">
    <str>suggest</str>
    <str>spellcheck</str>
  </arr>
</requestHandler>
```

- **SpellCheckComponent and dictionary configuration**

  The above configuration references the search component named `spellcheck` with a dictionary `textbody`. Now it's time to look at the configuration of that component. The relevant part for suggestions looks as follows:

```
<searchComponent name="spellcheck"
                 class="solr.SpellCheckComponent">

  <str name="queryAnalyzerFieldType">text_general</str>

  <lst name="spellchecker">
    <str name="name">textbody</str>
    <str name="classname">
      org.apache.solr.spelling.suggest.Suggester
    </str>
    <str name="lookupImpl">
      org.apache.solr.spelling.suggest.fst.WFSTLookupFactory
    </str>
    <str name="field">textbody</str>
    <float name="threshold">0.0005</float>
  </lst>

</searchComponent>
```

  If you choose different names for spell check component or dictionary, make sure that you use the correct names in the configuration of the `/editor` request handler.

  The element `<lst name="spellchecker">` configures a dictionary for suggestions based on the content of the index field `textbody`. The parameter `threshold` configures the dictionary to just consider words that occur in at least the given percentage of index documents. It can take a value between `0` and `1`. A value of `0.01` would mean that a word must appear in at least 1% of the documents in that field. More rare words will be ignored and not returned as suggestions. While you can set this value to `0` to include all words, this would increase the size of the in-memory data structure and the time needed to build it. You can use the parameter to tune the suggestions: higher values lead to smaller memory usage and better performance while smaller values provide more detailed suggestions.

To define dictionaries for multiple index fields, you just need to repeat the `<lst name="spellchecker">` section but use a different name for the dictionary in `<str name="name">` and set the name of the index field in `<str name="field">`.

- **Dictionary rebuilding configuration**

  Suggester dictionaries are in-memory data structures that must be rebuilt after index changes to make new words appear in the suggestions. The search component `DictionaryRebuilder`, which is also configured in file `solrconfig.xml`, rebuilds all configured dictionaries after index updates. Its configuration takes the name of the spell check component with parameter `spellCheckComponent` and the names of the dictionaries with parameter `dictionary`. For multiple dictionaries you just need to repeat the `<str name="dictionary">` element with different values.

```
<searchComponent name="dictionaryRebuilder"
      class="com.coremedia.solr.suggest.DictionaryRebuilder">
  <str name="spellCheckComponent">spellcheck</str>
  <str name="dictionary">textbody</str>
  <long name="minimumIntervalSeconds">60</long>
</searchComponent>
```

  With the default configuration in parameter `minimumIntervalSeconds`, the dictionary will be rebuilt at most once per minute if the index is constantly changed.

  Note that Solr already provides a different method to rebuild dictionaries after commits, which can be enabled with parameter `<str name="buildOnCommit">true</str>` in the `<lst name="spellchecker">` dictionary configuration. However, while it rebuilds the dictionary similarly to the `DictionaryRebuilder`, it will do this after every Solr commit even if commits come in very fast. It will also delay the visibility of the committed index changes in the search results as long as the dictionary is built. Depending on the size of the dictionary (affected by index size and the configured `threshold` parameter) it may take some seconds to rebuild a suggestion dictionary. Use the `DictionaryRebuilder` and not `buildOnCommit` to avoid such delays.

4.4.4 Configuring Semantic Search for Studio Server

The *Studio Server* connects to an *Embedding Service* to generate semantic embeddings for user queries. These embeddings are used to perform vector similarity searches against the indexed content embeddings in the search engine.

For details on the supported embedding models and their configuration, refer to Section 4.1.4.1, “Embedding Service Configuration” [49].

Studio Server Specific Configuration: All configuration properties specific to the *Studio Server* semantic search feature start with `studio.rest.search-service.semantic.*`. For a full list, see Section 3.4, “Studio Properties” in *Deployment Manual*.

Studio pulls the semantic search configuration from the *Studio Server* to enable semantic search for users. No additional configuration is required in *Studio*.

4.5 Modify the Search Index

NOTE

Configuration not mandatory: Change the *Apache Solr* configuration file `schema.xml` in `<solr-home>/configsets/content/conf` if you want to add a custom index field.

By default, search is performed in index fields `textbody`, `name`, `numericid` and their language-dependent variants `textbody_*` and `name_*` when using the `/editor` request handler configured in file `<solr-home>/configsets/content/conf/solrconfig.xml`. This request handler is used when you perform a search in *Studio*. The values from content properties are fed into the `textbody` index field. This default request handler configuration is useful for most situations.

Only if you want to search in an additional field but not in the `textbody` field, you can add the additional index field in the file `schema.xml.` Then you can feed the field with a `PropertyField` or `FeedablePopulator` as described in Section 4.2, “Configure the Content Feeder” [52].

You can search in a specific field with the method `SearchService#searchNative` from the Unified API (for details see Section 5.9, “Search Service of the Unified API” in *Unified API Developer Manual*). Another possibility is to use the Apache Solr API directly.

4.6 Operation of the Content Feeder

This section describes the operation of the *Content Feeder*.

4.6.1 Re-Indexing

Section 3.5, "Reindexing" [24] describes how to re-index search indices in general. You can re-index everything from scratch as described in Section 3.5.4, "Reindexing Content Feeder and CAE Feeder Indices from Scratch" [27], or only parts of the index as described in Section 3.5.2, "Partial Reindexing of Content Feeder Indices" [24]. The latter section also describes how to re-index only some aspects of contents, for example content issues.

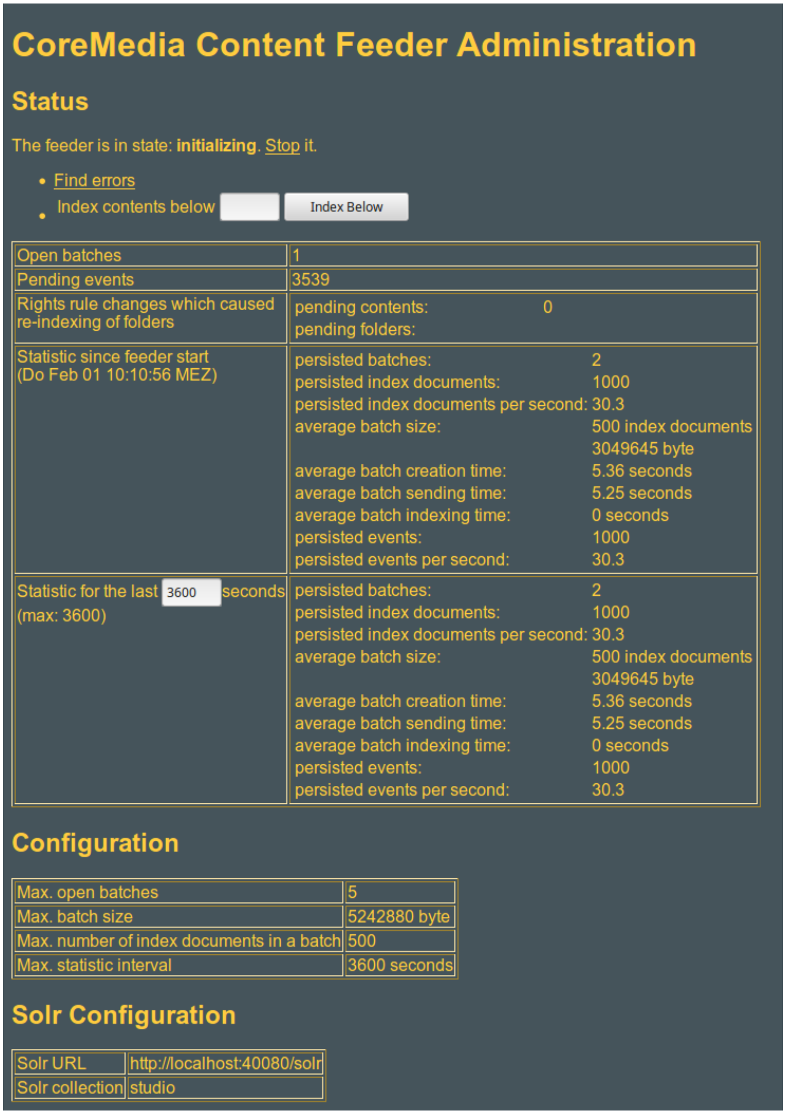
4.6.2 Administration Page

The *Content Feeder* provides a site for administration. The URL to the administration site: `http://<FEEDER_HOST>:<FEEDER_PORT>/admin`

The administration page requires HTTP authentication. The user and password are configured in the following properties:

```
feeder.content.management.user=feeder
feeder.content.management.password=feeder
```

It is recommended to change the password in productive environments.



CoreMedia Content Feeder Administration

Status

The feeder is in state: **initializing**. Stop it.

- Find errors
- Index contents below [ ] Index Below

Open batches	1
Pending events	3539
Rights rule changes which caused re-indexing of folders	pending contents: 0 pending folders:
Statistic since feeder start (Do Feb 01 10:10:56 MEZ)	persisted batches: 2 persisted index documents: 1000 persisted index documents per second: 30.3 average batch size: 500 index documents 3049645 byte average batch creation time: 5.36 seconds average batch sending time: 5.25 seconds average batch indexing time: 0 seconds persisted events: 1000 persisted events per second: 30.3
Statistic for the last 3600 seconds (max: 3600)	persisted batches: 2 persisted index documents: 1000 persisted index documents per second: 30.3 average batch size: 500 index documents 3049645 byte average batch creation time: 5.36 seconds average batch sending time: 5.25 seconds average batch indexing time: 0 seconds persisted events: 1000 persisted events per second: 30.3

Configuration

Max. open batches	5
Max. batch size	5242880 byte
Max. number of index documents in a batch	500
Max. statistic interval	3600 seconds

Solr Configuration

Solr URL	http://localhost:40080/solr
Solr collection	studio

Figure 4.3. Content Feeder Administration

The administration page shows the current status, statistic information and configuration of the *Content Feeder*. At the top of the page is a link to stop the *Content Feeder*.

Furthermore, there is a link to show errors for contents that were not processed successfully by the *Content Feeder* or the *CoreMedia Search Engine*. The page contains links to manually retry indexing of contents with errors. If not used, the *Content Feeder* retries indexing with the next change of the content.

Errors can also be found with a search engine query for all index documents with the value `ERROR` in the index field `feederstate`. The field `feederinfo` contains an error description.

Index contents below

This option enables the user to reindex all contents below a particular folder. Reindexing contents below a folder is achieved by entering the folder ID of the targeted folder in the "*index contents below*" input field and clicking on "Index Below" button.

4.6.3 Start and Stop the Content Feeder

The *Content Feeder* is started and stopped like any other application. You can also manually stop the *Content Feeder* with the stop link on the administration page. Note that the *Content Feeder* can only be restarted by restarting the application.

4.6.4 Clear Search Engine index

You can clear the *Search Engine* index of the *Content Server* by clicking on a corresponding link at the *Content Feeder* admin page. The *Content Feeder* must be stopped using the stop link on the administration page before the collection can be cleared. When stopped, a link *"Clear the Search Engine index"* shows up on the *Content Feeder* admin page.

This will remove all content of the *Content Server* from the *Search Engine* index. All contents will be reindexed when the *Content Feeder* is restarted.

Alternatively, you can use the JMX operation `clearCollection()` of the Feeder MBean. See the reference of the *Content Server Manual* for a description of all available JMX attributes and operations.

4.7 Implementing Custom Search

Custom search applications can use the full power of *Apache Solr* through Solr's Java API SolrJ. Please see the documentation of Apache Solr and its SolrJ API for details.

There are just a few things to keep in mind when implement search for content:

- Feeder applications such as the *CAE Feeder* and the *Content Feeder* require separate *Apache Solr* collections. When searching you must always specify the collection name, for example as parameter of the SolrJ method `org.apache.solr.client.solrj.SolrClient#query`.
- Successfully indexed documents carry the value `SUCCESS` in the index field `feederstate`. To avoid finding index documents that are used to store errors or internal state, you should always add a `feederstate:SUCCESS` filter query to your queries.

You can restrict the number of returned fields in a search result by setting the Solr `fl` (field list) parameter. Generally you just need the content id, which is stored in its numeric form in the index field `id`. You can use IDs of the search results to get the Content objects back from the Unified API. See the Unified API Developer Manual for details.

5. Searching for CAE Content Beans

This chapter describes concepts and structure of the *CoreMedia CAE Feeder* and contains information on how to make content beans of the *CoreMedia CAE* searchable with the *CoreMedia Search Engine*. It also describes configuration and operation of the *CAE Feeder*.

- Section 5.1, "Architectural Overview" [81] gives an overview over the architecture of the CAE Feeder
- Section 5.2, "Configuring the CAE Feeder" [82] describes the configuration of the *CAE Feeder* environment
- Section 5.3, "Operations of the CAE Feeder" [87] describes the operation of the *CAE Feeder*
- Section 5.4, "Indexing Content Beans" [89] describes how to configure and customize the *CAE Feeder* to make the content beans of your application searchable
- Section 5.5, "Integrating a Different Search Engine" [104] describes how to use the *CAE Feeder* with a different search engine or external system
- Section 5.6, "Implementing Custom Search" [107] provides some hints for implementing search in a *CAE* application

NOTE

You can find a helpful tool for the work with the *CAE Feeder* in the CoreMedia contributions repository at https://github.com/coremedia-contributions/cae-feeder-tools. Select the appropriate branch for your CoreMedia version.

5.1 Architectural Overview

The *CAE Feeder* is an application, which enables search functionality not only for single *CoreMedia* contents, as the *Content Feeder* does, but for content beans, where data may be computed from multiple source contents. To do so, the *CAE Feeder* sends the content bean's data to the *Search Engine*, which adds it to the index.

Feedable

The process of sending data to the *Search Engine* is called feeding the *Search Engine*. A piece of data used to add a new or update an existing index document is called a feedable. For efficiency reasons, the *CAE Feeder* sends batches of multiple feedables to add or update index documents and batches of multiple identifiers to remove index documents.

The *CAE Feeder* can share the content bean code with an existing *CAE* application. The *CAE Feeder* proactively sends data to the *Search Engine* after new content beans were added, changed or removed. It keeps the index up-to-date after changes in the data of the underlying content beans. Furthermore, it keeps track of the current feeding state to continue seamlessly after restarts of the application. To this end, it stores its state in a database.

The following figure shows the overall architecture:

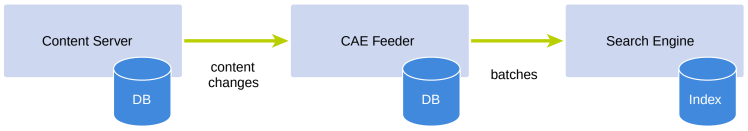


Figure 5.1. CAE Feeder architecture

5.2 Configuring the CAE Feeder

This section describes common configuration tasks.

For connection of the *CAE Feeder* to a *Content Server*, refer to Section 3.12.1, "Unified API Spring Boot Client Properties" in *Deployment Manual*.

The *CAE Feeder* requires a user account to access the contents of the *Content Server*. During the initialization of the *Content Server* a dedicated user is created with the name and password `feeder`. For security reasons, change the password afterwards. The account requires at least read rights on the content to be indexed. A license of the service `feeder` is consumed by a running *CAE Feeder*.

See Section 3.11, "CAE Feeder Properties" in *Deployment Manual* for a detailed description of configuration settings specific to the *CAE Feeder*. All properties can be configured in the file `application.properties` of the *CAE Feeder* application.

5.2.1 Configuring the Database

The *CAE Feeder* persists its feeding state in a relational database. Configure the connection to the database with properties `caefeeder.datasource.url`, `caefeeder.datasource.username`, and `caefeeder.datasource.password`. For details, see Section 3.11, "CAE Feeder Properties" in *Deployment Manual*

CAUTION

Do not run multiple *CAE Feeder* applications on the same database schema.

5.2.2 Configuring the Search Engine

The configuration of the *CoreMedia Search Engine* includes the location of Apache Solr and the name of the target Solr collection. This is done by setting the properties `solr.url` or `solr.zookeeper.addresses`, and `solr.cae.collection`. Each feeding application needs a different collection. Do not use the same collection for multiple instances of the *CAE Feeder* or the *Content Feeder*. For example:

```
solr.url=http://localhost:40080/solr
solr.cae.collection=preview
```

For SolrCloud, do not configure property `solr.url` but set `solr.cloud=true` and the ZooKeeper address(es) instead as in the following example:

```
solr.cloud=true
solr.zookeeper.addresses=zookeeper1:2181,zookeeper2:2181,zookeeper3:2181,
solr.cae.collection=preview
```

If the collection does not exist in Solr yet, the *CAE Feeder* will create it when started. It will create the collection based on the Solr config set "`cae`". If necessary, a different config set name can be configured with *CAE Feeder* property `solr.cae.config-set`.

If Apache Solr has been secured and needs HTTP basic authentication, you must also configure the required user name and password in the properties `solr.username` and `solr.password`.

5.2.3 Configuring Tika

The Feeder uses Apache Tika to extract text and metadata from blob properties for indexing.

Tika Caching

Extracted text and metadata is cached in heap memory to avoid repeated potentially expensive processing of the same blob. Caching can improve feeding performance, if a content was modified but its blob property was not changed, or if the same blob value is used in multiple content items. Use configuration property `cache.capacities.feeder.tika.heap` to configure the cache capacity in estimated bytes. The Blueprint configures a default capacity of 10 MB as follows:

Example

```
cache.capacities.feeder.tika.heap=10485760
```

Tika Parsers

Tika provides parsers for various formats, which can be customized in a special Apache Tika XML configuration file. The default configuration covers typical formats so that a custom configuration is rarely needed. If you need to fine-tune the configuration of Apache Tika, please have a look at the documentation of Apache Tika for the format of the Tika Config XML file. The location of this file can be configured with the Spring configuration property `feeder.tika.config`. The value of this property is a Spring Resource location. The following example configures an Apache Tika Config file from the local file system:

Example

```
feeder.tika.config=file:/opt/path/tika-config.xml
```

5.2.4 Configuring Tika Zip Bomb Prevention

Apache Tika uses a heuristic to detect 'Zip Bombs', that is files that expand to a huge amount of text when parsed. Parsing such files can lead to severe memory and/or performance issues in the Feeder. To prevent denial of service attacks or problems caused by malfunctioning parsers, the prevention is enabled by default. If Tika detects a blob to be a 'Zip Bomb', no text will be extracted from that blob and a warning will be logged instead. Note that 'Zip Bomb' attacks are not limited to ZIP files but can also occur for example with PDF files.

Normally, there's no need to change the configuration but if you encounter false positives, you may want to tweak the settings for Tika's heuristic or even turn off the prevention. You can disable 'Zip Bomb' detection with property `feeder.tika.zip-bomb-prevention.enabled=false` and tweak the heuristic with various properties starting with `feeder.tika.zip-bomb-prevention`. For details, see Section 3.11, “CAE Feeder Properties” in *Deployment Manual*.

5.2.5 Configuring Tika metadata extraction

In addition to extracting body text, Tika can extract metadata for some binary formats such as the creator of a Microsoft Word file. You can use the following properties to extract and index metadata from binary formats:

- `feeder.tika.append-metadata`
- `feeder.tika.copy-metadata`

The property `feeder.tika.append-metadata` takes a comma-separated list of metadata identifiers. The *CAE Feeder* simply appends the matching metadata values to the indexed body text when Apache Tika extracts such a value.

The property `feeder.tika.copy-metadata` takes a comma-separated list where each entry consists of a metadata identifier followed by an equal sign (`=`) and the name of the index field the metadata should be copied to. When a matching metadata value is found, it will be stored in the configured index field. Note that with Apache Solr target index fields must be defined as `multiVal` `ued="true"` to avoid indexing errors if there are multiple metadata values with the same identifier. See also Section 5.4.4, “Modifying the Search Index” [94].

Example

```
feeder.tika.copy-metadata=dc:creator=author
```

The above example configures the *CAE Feeder* to store the `dc:creator` metadata value in the index field `author`. Note that the index field must be declared in the Solr schema for this to work.

Metadata identifiers are specific to Apache Tika. You can find some of them in the API documentation of Apache Tika class `org.apache.tika.metadata.TikaCoreProperties`.

5.2.6 Configuring Tika ParseContext

Tika uses an instance of `org.apache.tika.parser.ParseContext` to pass advanced configuration to its parsers. If required, you can customize the ParseContext in the Spring context by adding entries to the map bean `tika ParseContext`. The map uses `java.lang.Class` objects as keys and values must be instances of their keys. The following example configures a custom Tika `org.apache.tika.extractor.DocumentSelector` to decide whether text gets extracted from embedded documents such as attachments in a PDF.

Example

```
  <customize:append id="tikaConfigCustomizer" bean="tikaParseContext">
    <map key-type="java.lang.Class" value-type="java.lang.Object">
      <entry key="org.apache.tika.extractor.DocumentSelector">
        <bean class="com.example.CustomTikaDocumentSelector"/>
      </entry>
    </map>
  </customize:append>
```

5.2.7 Configuring Error Handling

The *CAE Feeder* automatically retries operation after some communication problems with the Solr Search Server. The following properties configure the retry behavior:

- `feeder.solr.send-retry-delay`
- `solr.connection-timeout`

- `solr.socket-timeout`

Details for these properties can be found in Section 3.11, “CAE Feeder Properties” in *Deployment Manual*.

5.3 Operations of the CAE Feeder

This section describes administration and operation of the *CoreMedia CAE Feeder*. The *CAE Feeder* provides full-text search capabilities for custom content applications by sending the data of content beans to the *CoreMedia Search Engine*. Custom applications can use the *Search Engine* to find the content beans afterwards.

5.3.1 Starting and Stopping

During application start, the *CAE Feeder* will wait for the *Content Management Server* and for *Apache Solr* to become available.

5.3.2 Resetting

To reset the *CAE Feeder* and feed all contents again, both the *CAE Feeder* database and the used *Search Engine* index must be cleared. You can trigger clearing the database and Solr index with the `cm resetcaefeeder` command-line tool. The tool sets a reset flag for the *CAE Feeder* in the database and the *CAE Feeder* drops its database and index when it is restarted.

The `cm resetcaefeeder` tool is available in the *Blueprint* module `caefeeder-tools-application` and can be used as follows:

cm resetcaefeeder reset	Trigger a reset of the *CAE Feeder* for the next restart
cm resetcaefeeder cancel	Cancel a triggered reset
cm resetcaefeeder status	Show whether a reset was triggered or not

Note that the *CAE Feeder* must be able to connect to both the database and to Solr when restarted after calling `cm resetcaefeeder reset`. Do not stop the *CAE Feeder* when it is clearing database and search index. However, if it was stopped between clearing database and search index, then you must call `cm resetcaefeeder reset` once more and restart the *CAE Feeder*.

See also Section 3.5, “Reindexing” [24] to learn how to reindex without search downtime.

5.3.3 Disabling Invalidations

The *CAE Feeder* refeeds content beans when dependencies of these beans are invalidated. In some cases, this behavior might be cumbersome. For example, for initial indexing, you may want to first index the whole set of content beans, before processing invalidations for already indexed ones. This can be achieved by pausing invalidations for some time. Note, that invalidations are never skipped, and all changes will be handled as soon as invalidation handling is turned on again.

To temporarily disable invalidations, set the property `caefeeder.invalidation.paused=true` and restart the *CAE Feeder*.

You can also disable invalidations by setting the JMX attributes `InvalidationStopped` of MBean `com.coremedia:type=ContentDependencyInvalidator,application=caefeeder` and of MBean `com.coremedia:type=TimedDependencyInvalidator,application=caefeeder` to `true`. Changes made with JMX are reset when the *CAE Feeder* is restarted.

After all content beans have been indexed initially, set the property or JMX attributes back to "false", otherwise no invalidations would reach the *CAE Feeder*.

5.4 Indexing Content Beans

Indexing of content beans requires the following steps, which are described in the subsections of this section:

1. Specify by type and location the content beans you want to index
2. Provide content bean classes
3. Customize feedables to define which and how properties of content beans are indexed
4. Adapt the Solr index schema, if necessary

5.4.1 Specifying the Set of Indexed Content Beans

Each content bean in the *CAE* represents a content object from the *CoreMedia Content Server*.

In order to specify the indexed content beans, you have to define the set of source contents using a content selector.

Definition of content selector

The Spring bean `contentSelector` of interface `ContentSelector` is responsible for selecting source contents for feeding. The default implementation `PathAndTypeContentSelector` selects contents by type and path and can be configured with the following properties:

`caefeeder.content.path`	Specifies the content repository folder paths below which content gets indexed. Folder paths must start with a slash and must not overlap. The value of this property is a Map that maps folder paths to 'true' for included paths. Note, that paths mapped to 'false' are not excluded, but simply ignored.
`caefeeder.content.type`	Specifies the content types for which content gets indexed. The value of this property is a Map that maps content types to 'true' for included types. Note, that types mapped to 'false' are not excluded, but simply ignored.
`caefeeder.content.include-subtypes`	Specifies whether subtypes of the configured content types are selected as well. The default is true.

See Section 3.11, “CAE Feeder Properties” in *Deployment Manual* for a detailed description of configuration properties.

NOTE

Changing the `caefeeder.content.*` properties will not trigger any re-indexing of already indexed content. See Section 5.3.2, “Resetting” [87] for details on re-indexing.

Example

Example 5.1, “ContentSelector example” [90] selects all contents of type `CMMedia`, `CMArticle`, `CMDownload` and `CMCollection` (including sub types) which are located below the paths `/Sites` or `/Settings/Taxonomies`:

```
caefeeder.content.path[/Sites]=true
caefeeder.content.path[/Settings/Taxonomies]=true
caefeeder.content.type.CMArticle=true
caefeeder.content.type.CMCollection=true
caefeeder.content.type.CMDownload=true
caefeeder.content.type.CMMedia=true
caefeeder.content.include-subtypes=true
```

Example 5.1. ContentSelector example

5.4.2 Configuring Content Bean Classes

The *CAE Feeder* needs a definition of used content bean classes in its Spring context and the implementation of the content beans in its classpath similar to the configuration of the *CAE*. So you can reuse your *CAE* content beans configuration.

Configure the content bean classes in the Spring application context as described in the Content Application Developer Manual.

Make sure, that the configured classes are available in the classpath of the *CAE Feeder*.

5.4.3 Customizing Feedables

A Feedable

A feedable is an object which is generated from the data of a content bean and which the *CAE Feeder* sends to the *Search Engine* for indexing. Customizing feedables means that you define which values from a content bean are mapped to fields of the feedable and are therefore added to the index if a corresponding Solr index field exists. The following paragraphs describe the involved classes.

Create an identifier for index documents

An implementation of `com.coremedia.cap.feeder.persistentcache.KeyTransformer` is used to create identifiers for *Search Engine* documents in the index. The default KeyTransformer implementation creates identifiers of the same format as the IdProvider of the *CoreMedia CAE*.

Example: a content bean for the content with the numerical id `42` is represented by an *Apache Solr* document with the value `contentbean:42` in the field `id`. Search applications can use the `IdProvider` to get a content bean for the identifier again.

Filling the Feedable with a FeedablePopulator

The *CAE Feeder* uses implementations of the `com.coremedia.cap.feeder.populate.FeedablePopulator` interface to set elements of the feedable with content bean data. By default, a `BeanMappingFeedablePopulator` is used, which can be configured to map data from `ContentBean` instances to elements of the created feedable.

For more flexibility, you can configure additional custom `FeedablePopulator` implementations and add them to the Spring bean `feedablePopulators`, which is a list of `FeedablePopulator<T>` beans. Each configured populator can add different parts of data to the same feedable. The type parameter `<T>` of a configured `FeedablePopulator` bean must be `ContentBean`, `Content` or a super type of these. You can find some existing `FeedablePopulator` implementations in package `com.coremedia.cap.feeder.populate`. For example, you may configure an additional `PropertyPathFeedablePopulator` to index certain nested values of struct properties.

Error handling

FeedablePopulator implementations should avoid throwing exceptions, but if they do throw an unexpected exception, then the *CAE Feeder* will index a so-called error document as placeholder. Error documents can be recognized by the value `ERROR` in the index field `feederstate`. The stack trace of the exception is stored in the index field `feederinfo`. Do not forget to always add a `feederstate:SUCCESS` filter query to Solr queries to find successfully indexed documents. Feeding will be retried automatically after 10 minutes by default, or when a dependency is invalidated that was accessed before the exception was thrown. See configuration property `caefeeder.evaluation.error-retry-delay` in Section 3.11, “CAE Feeder Properties” in *Deployment Manual* if you want to change the retry delay.

5.4.3.1 Configuring the BeanMappingFeedablePopulator

The predefined `BeanMappingFeedablePopulator` can be configured to map content bean data to feedable elements. Its property `beanMappings` takes a list of mappings where each mapping applies to one bean class. A mapping for a single bean class is represented by class `com.coremedia.cap.feeder.bean.BeanFeedableMapping`, which can be configured to map data from a given content bean instance to a feedable element. The list of mappings is available as Spring bean `caeFeederBeanMappings`, to which you can add custom mappings in the Spring configuration as shown in the following example.

Example

```
@Bean(autowireCandidate = false)
@Customize("caeFeederBeanMappings")
public BeanFeedableMapping<ContentBean> contentBeanFeedableMapping() {
  BeanFeedableMapping<ContentBean> mapping = new
BeanFeedableMapping<>(ContentBean.class);
  mapping.addElement("documenttype", bean ->
bean.getContent().getType().getName());
  mapping.addElement("freshness", bean ->
bean.getContent().getModificationDate());
  return mapping;
}

@Bean(autowireCandidate = false)
@Customize("caeFeederBeanMappings")
public BeanFeedableMapping<CMLinkable> cmLinkableBeanFeedableMapping() {
  BeanFeedableMapping<CMLinkable> mapping = new
BeanFeedableMapping<>(CMLinkable.class);
  mapping.addElement("keywords", CMLinkable::getKeywords, true);
  mapping.addElement("segment", CMLinkable::getSegment);
  return mapping;
}
```

Example 5.2. Example Content Bean to Feedable Mapping

The example defines two mappings, one for the superclass of all content beans `com.coremedia.objectserver.beans.ContentBean`, and one for `CMLinkable` beans. Similar more lengthy mappings exist in the Blueprint configuration in Spring configuration class `CaeFeederBlueprintAutoConfiguration`.

The first mapping for class `ContentBean` defines two elements `documenttype` and `freshness` with functions to compute their values from a given content bean. Values will be indexed in Solr fields with the same name, if such fields exist in the index. This mapping will be used for all content beans.

The second mapping for class `CMLinkable` configures feeding of bean properties `keywords` and `segments` into Solr index fields of the same name, if such fields exist in the Solr index. Keywords are also indexed in the Solr field `textBody`, as specified by the third parameter `true` of method `addElement`. This mapping will be used for all content beans that implement `CMLinkable`.

Of course, functions passed to `addElement` methods can use more complex logic than just calling content bean methods. They can convert values from content bean properties to a suitable format for indexing, provide default values, or use other custom logic as needed.

NOTE

A content bean class can inherit from or extend other content beans classes, and multiple `BeanFeedableMapping` configurations can match for the class of a content bean. If so, all matching mappings will be used. However, it is an error to configure multiple matching mappings with the same element name. In such a case, a warning will be logged, and the configuration from the first `BeanFeedableMapping` in the list of mappings will be used. There is no mechanism to override the configuration for a Feedable element in a mapping for a content bean subclass.

5.4.3.2 Mapping Values to Element Types

The *CAE Feeder* supports String, Number, Date, XML and binary element types. The following table describes the default mapping from value classes to element types:

value class	element type
`com.coremedia.cap.common.Blob`	Binary
`java.util.Date`, `java.util.Calendar`, and `java.time.Instant`	Date
`com.coremedia.xml.Markup`	XML
`java.lang.Number` and primitive number types	Number
`java.lang.String`	String
`java.lang.Collection` with elements of above types	depends on collection's element type

Table 5.1. Feedable Element Types for Value Classes

Values of other classes map to String elements with the value of their `toString` method. Collections must contain elements of one type, otherwise the value of the elements' `toString` method will be used.

Blob values are only used, if their MIME-type is configured for indexing in configuration property `feeder.blob.enabled`, and if its size does not exceed the maximum size configured in configuration property `feeder.blob.max-size`. See Section 3.11, "CAE Feeder Properties" in *Deployment Manual* for a description of these configuration properties. The default configuration in the Blueprint Maven module `caefeeder-blueprint-component` in file `caefeeder-blueprint.properties` sets a maximum size of 5MB and includes the following MIME-types:

- `text/*`
- `application/pdf`
- `application/msword`
- `application/vnd.openxmlformats-officedocument.wordprocessingml.document`

Collection elements can be used to feed multi-value fields in Apache Solr.

5.4.4 Modifying the Search Index

NOTE

Configuration not mandatory

Change the Apache Solr `schema.xml` in `<solr-home>/configsets/cae/conf` if you want to add index fields.

By default, search is performed in the index field `textbody` and language-dependent variants `textbody_*` when using the `/cmdismax` request handler configured in file `<solr-home>/configsets/cae/conf/solrconfig.xml`.

If you want to search in a different field, or want to use a special field for sorting, faceting or anything like that, then you must add that field to the Solr configuration file `schema.xml`.

The *CAE Feeder* sets the additional field when an indexed feedable contains an element whose name matches the field's name. See Section 5.4.3, "Customizing Feedables" [90] for details on feedables and their construction.

5.4.5 Using Revalidating Fragments

Recorded dependencies

When computing the data for a feedable, dependencies on accessed objects are tracked and recorded by the *CAE Feeder*. Modifications of recorded dependencies will lead to the invalidation of the feedable. The *CAE Feeder* will then construct a new feedable with recomputed data and send it to the search engine. For example, a content bean will be reindexed after changing some content that was used to compute the feedable for that content bean.

In some cases, however, the invalidation of a dependency does not necessarily lead to a different value for feeding and the overhead of reindexing could be avoided for better performance.

Unnecessary invalidation

For example, an indexed bean property gets its data from a content with global settings. Such a content may contain lots of different settings in different properties or in a single struct property. Imagine, that a single setting `S1` from this content is accessed during the construction of each indexed feedable. Because of this, each indexed bean will depend on the properties of the settings content. Now, if somebody changes the content, for example by changing setting `S2`, all indexed beans will be invalidated and reindexed. This can take some time. And the data did not even change.

Skipping reindexing with fragment keys

Of course, you want to avoid such situations. One possibility is to disable such expensive dependencies by wrapping the code that creates them with the methods `disableDependencies()` and `enableDependencies()` of the class `com.coremedia.cache.Cache`. But often this is not possible, because sometimes an invalid dependency really indicates changed data and the index must be updated. To solve this problem, the *CAE Feeder* supports fragment keys, which can be used to revalidate an unchanged result of a computation after some of its dependencies became invalid. Revalidation means that the *CAE Feeder* recognizes that an invalidation of a dependency does not change the result so that expensive reindexing can be skipped.

CAUTION

Revalidating fragment keys should be used when it's possible to encapsulate a fragment that is used for the computation of many feedables, and if dependencies get invalidated without changing the feedable's data.

You should not use fragment keys, if each fragment is used in just one feedable instance. The overhead of maintaining a lot of fragment keys in the *CAE Feeder* can be much higher than reindexing a few content beans. The number of fragment keys should be lower than the number of indexed content beans, for which the fragment keys are used.

This section continues with an example how to use revalidating fragments to avoid unnecessary reindexing.

5.4.5.1 Example: Using Revalidating Fragments for the Repository Path

In the following example, users should be able to search for articles below a given repository path. Therefore, the *CAE Feeder* is configured to feed the repository path into the field `folderpath`. The path is indexed as path of numeric IDs. For example for a content that resides in folder `/foo/bar` the value `/1/41/43/` will be indexed if foo's ID is 41 and bar's ID is 43. `/1` represents the root folder here. The advantage of this approach is that folders can be renamed without the need to reindex contents. To find all articles below the folder `/foo`, the search application can simply use foo's ID in a query.

The *CAE Feeder* is configured to index the folder path for content beans of type Article by setting the following property:

```
feeder.content.type.Article=true
```

and customizing bean mappings:

```
@Bean(autowireCandidate = false)
@Customize("caeFeederBeanMappings")
public BeanFeedableMapping<Article> articleFolderPathFeedableMapping() {
  BeanFeedableMapping<ContentBean> mapping = new
BeanFeedableMapping<>(Article.class);
  mapping.addElement("folderpath", Article::getFolderPath);
  return mapping;
}
```

Without fragment keys the implementation of the Article's bean property might look like:

```
public String getFolderPath() {
  Content content =  getContent().getParent();
  StringBuilder sb = new StringBuilder();
  while (content != null) {
    sb.insert(0, "/" + IdHelper.parseContentId(content.getId()));
    content = content.getParent();
  }
  return sb.toString();
}
```

`Content#getParent` creates a dependency on the place of the content, which is invalidated if either the name or the parent of the content changes. If the name of a parent folder changes, the article will be reindexed, even though the indexed value has not changed. You can avoid this by using revalidating fragments. Using revalidating fragments in this example consists of the following steps:

1. Implement a fragment key that encapsulates the part of the computation that can be revalidated when collecting data for the feedable.
2. Implement a fragment key factory that returns a fragment key from a serialized version of the key.
3. Register your factory in the Spring context.
4. Inject the factory into the content bean and use the factory to get the fragment key's value.
5. Configure the capacity of the internally used cache.

Implementing a Fragment Key

First, implement a fragment key class that extends `RevalidatingFragment-PersistentCacheKey`. This key encapsulates the computation of the repository path in its `evaluate()` method. The computed path constitutes a fragment of the overall computation of the feedable's data. The implementation uses the *Persistent Cache*, which is an internal component of the *CAE Feeder*, to recursively get the fragment value for the parent folder.

```
package com.customer.example;
import com.coremedia.cap.content.*;
import com.coremedia.cap.common.IdHelper;
import com.coremedia.cap.persistentcache.*;
import java.io.UnsupportedEncodingException;

public class IdPathKey
       extends RevalidatingFragmentPersistentCacheKey<String> {

  static final String PREFIX = "idpath:";
  private final PersistentCache persistentCache;
  private final ContentRepository contentRepository;
  private final String contentId;

  public IdPathKey(PersistentCache persistentCache,
                   ContentRepository contentRepository,
                   String contentId) {
    this.persistentCache = persistentCache;
    this.contentRepository = contentRepository;
    this.contentId = contentId;
  }

  @Override
  public String getSerialized() {
    return PREFIX + contentId;
  }

  @Override
  public String evaluate() throws Exception {
    Content content = contentRepository.getContent(contentId);
    if (content==null) {
      String s = getSerialized();
      throw new InvalidPersistentCacheKeyException(s);
    }
    return getPath(content.getParent()) + '/' +
IdHelper.parseContentId(contentId);
  }

  private String getPath(Content content) {
    if (content == null) {
      return "";
    }
    IdPathKey key = new IdPathKey(persistentCache, contentRepository,
content.getId());
    return (String)persistentCache.getCached(key);
  }

  @Override
  public byte[] getBytesForHashing(String value) {
    try {
      return String.valueOf(value).getBytes("UTF-8");
    } catch (UnsupportedEncodingException e) {
      throw new RuntimeException("UTF-8 not supported", e);
    }
}
```

Example 5.3. Example of a fragment key implementation

To implement a fragment key, the methods `getSerialized()`, `evaluate()` and `getBytesForHashing(String)` are implemented. In the following, the methods are described in general.

evaluate()

Method `evaluate()` computes the fragment value. It does not take any parameters that specify the source data for the computation. Such parameters are part of the key's identity and are passed to its constructor. In the example, the `contentId` is such a key parameter.

Method calls on `com.coremedia.cap.content.Content` objects in the implementation of `evaluate()` implicitly trigger all relevant dependencies. These content dependencies are automatically invalidated after corresponding content changes.

There may be situations where you want to avoid content dependencies. To this end, you can use the following pattern to disable dependency tracking for a code block by calling static methods of class `com.coremedia.cache.Cache`:

```
Cache.disableDependencies();
try {
  // dependencies are disabled for this code block
  ...
} finally {
  Cache.enableDependencies();
}
```

Additional dependencies may be triggered explicitly by calling the following static methods from inside the `evaluate()` method:

- `com.coremedia.cache.Cache#cacheFor(long millis)`: Triggers a relative time dependency making the value become invalid when the time is reached.

- `com.coremedia.cache.Cache#cacheUntil(Date date)`: Triggers an absolute time dependency again making the value become invalid when the time is reached.
- `com.coremedia.cache.Cache#dependencyOn(Object dependent)`: Triggers an explicit dependency on a certain object. The *CAE Feeder* only supports dependencies on `java.lang.String` values. Dependencies of other types are ignored.

  Custom dependencies on `java.lang.String` values can be invalidated programmatically by invoking method `invalidate(Object)` of class `com.coremedia.cap.persistentcache.dependencycache.PersistentDependencyCacheManagement` on the Spring bean `persistentDependencyCacheManager`. Alternatively, you can invalidate a String dependency with the JMX operation `invalidateSerialized(String)` of the `PersistentDependencyCache` MBean. The parameter of this JMX operation is the String dependency itself, prefixed with `"string:"` (that is, `"string:" + value`).

getSerialized()

Method `getSerialized()` returns the key's serialized form as `java.lang.String` as it is stored in the database of the *CAE Feeder*. The returned string contains all parameters that are needed to reconstruct the fragment key instance. It is good practice to use different prefixes for different types of fragment keys. In the example, the prefix `"idpath:"` and the Content ID are used to create serialized keys such as `idpath:coremedia:///cap/content/41`.

Keep in mind, that the serialized key is stored in the database when making the dependencies persistent. Thus, using short keys will result in less disk space usage.

getBytesForHashing(String value)

Method `getBytesForHashing(String)` returns a byte representation for a computed value. The *CAE Feeder* computes a hash from these bytes and stores it in its database. The hash is used to detect if a fragment value has changed after it was recomputed. The *CAE Feeder* avoids reindexing if nothing has changed.

Implementing a Factory for Fragment Keys

Next, you need a `PersistentCacheKeyFactory`, which is used to create fragment key instances based on the keys' serialized representations. Its method `createKey(String)` is the inverse function for the fragment key's method `getSerializedKey()`.

In an environment where several types of fragment keys and therefore several `PersistentCacheKeyFactory` instances are used, a mechanism for selecting the right factory needs to be provided. As a convention, a `PersistentCacheKeyFactory` may answer `null` to signal that it is not responsible for a given serialized key. The *CAE Feeder* sequentially asks all known `PersistentCacheKeyFactories` until a factory returns a non null result.

In case that the `PersistentCacheKeyFactory` is asked to reconstruct a key whose resources are no longer available, it nevertheless must return a fragment key. This returned key should throw an `com.coremedia.cap.persistentcache.InvalidPersistentCacheKeyException` when its `evaluate()` method is called. You may use the static method `InvalidPersistentCacheKeyException.wrap(String serializedKey)` for creating such an instance.

In the example, the `PersistentCacheKeyFactory` just creates an instance of `IdPathKey` with the Content ID extracted from the serialized key. It returns `null` if the serialized key does not start with the correct prefix:

```
package com.customer.example;

import com.coremedia.cap.common.CapObjectDestroyedException;
import com.coremedia.cap.content.*;
import com.coremedia.cap.persistentcache.*;
import com.google.common.base.Throwables;

public class IdPathKeyFactory
      implements PersistentCacheKeyFactory {
  private PersistentCache persistentCache;
  private ContentRepository contentRepository;

  public void setPersistentCache(PersistentCache pc) {
    this.persistentCache = pc;
  }

  public void setContentRepository(ContentRepository cr) {
    this.contentRepository = cr;
  }

  public PersistentCacheKey createKey(String serializedKey) {
    if (serializedKey.startsWith(IdPathKey.PREFIX)) {
      int l = IdPathKey.PREFIX.length();
      String contentId = serializedKey.substring(l);
      return keyForContent(contentId);
    }
    return null;
  }

  private PersistentCacheKey keyForContent(String contentId) {
    return new IdPathKey(persistentCache, contentRepository,
                         contentId);
  }

  public String get(Content content) {
    String contentId = content.getId();
    PersistentCacheKey key = keyForContent(contentId);
    try {
      return (String) persistentCache.getCached(key);
    } catch (EvaluationException e) {
      if (Throwables.getCausalChain(e).stream().anyMatch(
          t -> t instanceof CapObjectDestroyedException
```

```
            || t instanceof InvalidPersistentCacheKeyException)) {
          return "";
        }
        Throwables.throwIfUnchecked(e.getCause());
        throw e;
      }
    }
}
```

Example 5.4. Example of a PersistenCacheKeyFactory implementation

The `PersistentCacheKeyFactory` for creating fragment keys must be defined in the Spring application context and registered as a fragment key factory. Note, that the key factory is initialized with the `persistentDependencyCache` bean for the `persistentCache` property. It's important to always use the `persistentDependencyCache` bean to get fragment keys.

```
<bean id="idPathKeyFactory"
      class="com.coremedia.amaro.feeder.beans.IdPathKeyFactory">
  <property name="persistentCache"
            ref="persistentDependencyCache"/>
  <property name="contentRepository"
            ref="contentRepository"/>
</bean>

<customize:append id="idPathKeyFactoryCustomizer"
                  bean="fragmentPersistentCacheKeyFactory"
                  property="keyFactories">
  <list>
    <ref local="idPathKeyFactory"/>
  </list>
</customize:append>
```

Example 5.5. Define and register the factory in the Spring context

Using the Fragment Key Value in a Content Bean

The `IdPathKeyFactory` example class contains the convenience method `get(Content)`, which can be used in the content bean implementation to get the path for a Content. The example implementation of method `get` ignores exceptions that were triggered by invalid keys or destroyed content.

```
package com.customer.example.beans;

public class ArticleImpl extends ArticleBase implements Article {
  private IdPathKeyFactory factory;

  public void setIdPathKeyFactory(IdPathKeyFactory factory) {
    this.factory = factory;
  }

  public String getFolderPath() {
    Content parent = getContent().getParent();
    if (parent == null) {
      return "";
    }
    return factory.get(parent);
  }
}
```

Example 5.6. Using the fragment key in the content bean

The content bean definition for the article bean must be configured with the key factory:

```
<bean name="contentBeanFactory:Article"
      class="com.customer.example.beans.ArticleImpl"
      scope="prototype" parent="abstractContentBean">
  <property name="idPathKeyFactory" ref="idPathKeyFactory"/>
</bean>
```

Example 5.7. Configure content bean with factory

This example's content bean implementation depends directly on the `PersistentCacheKeyFactory` and can only be used in the *CAE Feeder*. If you want to use the same implementation in the *CAE* application, you should extract the logic to compute the path into a strategy interface.

Getting the Fragment Key Value from the Persistent Cache

`IdPathKeyFactory#get(Content)` and `IdPathKey#getPath(Content)` use method `getCached` of `com.coremedia.cap.persistentcache.PersistentCache` to retrieve a fragment value. This method uses in-memory `CacheKey`s to cache fragment values. Cached lookup improves performance if lots of keys access the fragment's value. It does not only avoid the repeated computation of the fragment but it also avoids database queries to check whether newly computed values have changed since the last computation.

Configure the cache

In-memory cache keys created by the method `getCached` have the default cache class `com.coremedia.cap.persistentcache` and a default cache weight equal to one. The capacity for the cache class defaults to `10000` (can be changed with configuration property `cache.capacities.com.coremedia.cap.persistentcache`). If you want to use a different cache class or weight, you can still create an in-memory `CacheKey` yourself which then calls `PersistentCache#get(PersistentCacheKey)` in its `evaluate` method.

Do not introduce cycles

Be careful to not introduce cycles when calling methods `get` or `getCached` of the `PersistentCache` interface from another fragment key's `evaluate` method. Simple cycles on the same thread will result in an `IllegalStateException`, for example if `key:1` gets `key:2` which in turn gets `key:1` again. But code might still hang if multiple threads are involved, for example if one thread gets `key:1` which gets `key:2` while another thread gets `key:2` which gets `key:1`.

5.5 Integrating a Different Search Engine

This section describes the necessary steps to make the *CAE Feeder* feed content bean data to a different search engine or another external system. The default integration uses *Apache Solr* but the *CAE Feeder* provides an `Indexer` interface that can be implemented to feed other external systems such as a search engine that is integrated in your company's IT infrastructure.

The following simple example explains how you can replace the standard *Apache Solr* indexer with a custom indexer that just writes messages to the log file.

1. Create a new Maven module, for example `caefeeder-custom-component` with the following `pom.xml`:

```
      <?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    ...
  </parent>

  <modelVersion>4.0.0</modelVersion>
  <artifactId>caefeeder-custom-component</artifactId>

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

    <dependency>
      <groupId>com.coremedia.cms</groupId>
      <artifactId>cap-feeder-api</artifactId>
    </dependency>

    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
    </dependency>

  </dependencies>
</project>
```

2. Create a new source folder `src/main/java` in the module.
3. Create the java class `LogIndexer` for the new indexer in package `com/customer:`

```
      package com.customer;

import com.coremedia.cap.feeder.Feedable;
import com.coremedia.cap.feeder.FeedableElement;
import com.coremedia.cap.feeder.index.IndexException;
import com.coremedia.cap.feeder.index.IndexerResult;
import com.coremedia.cap.feeder.index.direct.DirectIndexerBase;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

public class LogIndexer extends DirectIndexerBase {
  private static final Logger LOG
    = LoggerFactory.getLogger(LogIndexer.class);

  public IndexerResult index(
      Collection<? extends Feedable> feedables,
      Collection<String> removeIds) throws IndexException {

    if (LOG.isInfoEnabled()) {
      for (Feedable feedable: feedables) {
        Collection<FeedableElement> elements
          = feedable.getElements();
        Map<String, Object> values
          = new HashMap<>(elements.size());
        for (FeedableElement element: elements) {
          values.put(element.getName(), element.getValue());
        }
        LOG.info("Updating {} with {}",
          feedable.getId(), values);
      }
      if (!removeIds.isEmpty()) {
        LOG.info("Removing {}", removeIds);
      }
    }
    return IndexerResult.persisted();
  }

  public String getDocumentInfo(String s) throws IndexException {
    return null;
  }
}
```

4. Create a new source folder `src/main/resources/META-INF/coremedia` in the module.

5. Create a Spring configuration file for the component named `component-caefeeder-custom.xml` in this folder

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
">
```

```
  <bean id="feederIndexer" class="com.customer.LogIndexer"/>
</beans>
```

6. In the file `pom.xml` of the *CAE Feeder* web application replace the dependency on `caefeeder-solr-component` with a dependency to your new component: `caefeeder-custom-component`.

7. Add a corresponding logger to the logback configuration of the *CAE Feeder* application.

```
<logger name="com.customer" additivity="false" level="debug">
<appender-ref ref="file"/>
</logger>
```

5.6 Implementing Custom Search

Custom search applications can use the full power of *Apache Solr* through Solr's Java API SolrJ. Please see the documentation of Apache Solr and its SolrJ API for details.

There are just a few things to keep in mind when implement search for content beans:

- Feeder applications such as the *CAE Feeder* and the *Content Feeder* require separate *Apache Solr* collections. When searching you must always specify the collection name, for example as parameter of the SolrJ method `org.apache.solr.client.solrj.SolrClient#query`.
- Successfully indexed documents carry the value `SUCCESS` in the index field `feederstate`. To avoid finding placeholder index documents for feeding errors or internal index documents, you should always add a `feederstate:SUCCESS` filter query to your queries.

You can restrict the number of returned fields in a search result by setting the Solr `fl` (field list) parameter. In a *CAE* application you generally just need the content bean id, which is stored in field `id`. You can use IDs of the search results to get the Content Bean objects back from the CAE using an `IdScheme` or `IdProvider`. See the Content Application Developer Manual for details on Content Beans and their IDs.

6. Reference

6.1 Configuration Property Reference

6.1.1 Content Feeder Properties

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

- Section 3.12.1, "Unified API Spring Boot Client Properties" in *Deployment Manual* contains properties for the configuration of the connection to the *Content Server*.
- Table 3.47, "Content Feeder Configuration Properties" in *Deployment Manual* contains properties for the configuration of the *Content Feeder*.
- Table 3.48, "Content Feeder Solr Configuration Properties" in *Deployment Manual* contains properties for the configuration of the Apache Solr search engine used by the *Content Feeder*.
- Table 3.49, "Feeder Batch Configuration Properties" in *Deployment Manual* contains properties for the configuration of batch processing.
- Table 3.50, "Feeder Tika Configuration Properties" in *Deployment Manual* contains properties for the configuration of Apache Tika used by the *Content Feeder* for text extraction.
- Table 3.51, "Feeder Core Configuration Properties" in *Deployment Manual* contains properties for the configuration of the executor queue capacity and the executor retry delay.

6.1.2 CAE Feeder Properties

Different aspects of the *CAE Feeder* 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 reference the Spring application context properties for the *CAE Feeder*.

- Section 3.12.1, "Unified API Spring Boot Client Properties" in *Deployment Manual* contains properties for the configuration of the connection to the *Content Server*.

- section “General Properties” in *Deployment Manual* contains properties for the general configuration of the *CAE Feeder*.
- section “Database Properties” in *Deployment Manual* contains properties for the database configuration of the *CAE Feeder*.
- section “Apache Tika Properties” in *Deployment Manual* contains properties for the configuration of Apache Tika used by the *CAE Feeder* for text extraction.
- section “Apache Solr Client Properties” in *Deployment Manual* contains properties for the configuration of the Apache Solr search engine used by the *CAE Feeder*.

6.2 Content Feeder Metrics

Metrics about the operation of a running *Content Feeder* are mostly available as attributes of JMX Managed Beans, that are described in Section 6.3, “Content Feeder JMX Managed Beans” [113]. This section lists some additional metrics that are available at the Spring Boot Actuator Metrics Endpoint.

feeder.index

The `feeder.index` metric is a counter that measures the number of triggered index updates. It includes both full and partial updates.

The metric supports the following optional tag to select more specific measurements:

Tags of the "feeder.index" Metric

trigger	The type of trigger that caused the update. Typical types are "initialize" for initial feeding and "event" for changes caused by editorial changes. A name that starts with "background." indicates changes that were triggered by low priority background feeding, for example "background.admin" for externally triggered reindexing, or "background.issues" for periodic reindexing of content issues.

feeder.populator

The `feeder.populator` metric is a timer that measures the invocation count and time spent in `com.coremedia.cap.feeder.populate.FeedablePopulator` calls.

The metric supports the following optional tags to select more specific measurements:

Tags of the "feeder.populator" Metric

class	The class name of the `FeedablePopulator` implementation. Note, that names of non-public API classes may change without notice in future releases.
partialupdate	If `true`, only partial updates are measured. If `false`, partial updates are not measured. See Section 4.1.2, “Partial Updates” [45] for a description of partial updates.
active	If `true`, only invocations that actually modify the Feedable are measured. If unset, `FeedablePopulator` invocations

are also counted if they don't do anything, for example, if they return immediately if a content item is not of a specific type.

6.3 Content Feeder JMX Managed Beans

The *Content Feeder* exports attributes and operations with the following MBeans, whose attributes and operations are described in more detail in the tables of this section:

- Feeder MBean: `com.coremedia:type=Feeder,application=content-feeder`
- UpdateGroupsBackgroundFeed MBean: `com.coremedia:type=UpdateGroupsBackgroundFeed,application=content-feeder`. This MBean shows the status of updating the index after changes to rights rules in the repository. See also "Configuring updates of rights rule changes" in Section 4.2.3, “Advanced Configuration” [62].
- AdminBackgroundFeed MBean: `com.coremedia:type=AdminBackgroundFeed,application=content-feeder`. This MBean is related to the reindexing functionality described in Section 3.5, “Reindexing” [24].
- SolrIndexer MBean: `com.coremedia:type=SolrIndexer,application=content-feeder`, which is described in Section 6.5, “Solr Indexer JMX Managed Beans” [136].

Depending on active Blueprint features, there can be more available MBeans, that are not listed here.

Feeder MBean Attributes

The following table shows the attributes of MBean `com.coremedia:type=Feeder,application=content-feeder`:

Attribute	Type	Description
`IndexAverageBatchCreationTime`	Read-only	Average batch creation time in the statistics interval.
`IndexAverageBatchIndexingTime`	Read-only	Average batch indexing time in the statistics interval. If Apache Solr is used, this property is 0 because contents are indexed immediately when they are sent to the search engine. Indexing time is then part of `IndexAverageBatchSendingTime`.

Attribute	Type	Description
`IndexAverageBatchSendingTime`	Read-only	Average batch sending time in the statistics interval.
`IndexBatches`	Read-only	Number of indexed batches in the statistics interval.
`IndexBytes`	Read-only	Number of indexed bytes in the statistics interval.
`IndexDocuments`	Read-only	Number of indexed documents in the statistics interval.
`IndexDocumentsPerSecond`	Read-only	Number of documents indexed per second in the statistics interval.
`IndexMaxBatchBytes`	Read-only	The maximum batch size in bytes.
`IndexMaxBatchSize`	Read-only	The maximum number of index documents in a batch.
`IndexAverageLagTime`	Read-only	The average delay in seconds of the index documents that represent content and that were indexed in the last <*n*> seconds, where <*n*> is the value of the attribute `IndexStatisticInterval`. If <*n*> is `0` or greater than the value of attribute `IndexMaxStatisticInterval`, this attribute will contain the value since the start of the *Content Feeder*. The difference of the time when a *batch* was successfully sent and the feedable field *freshness* are used for each feedable object where *feederstate* is `SUCCESS`. The set of index documents used to compute this value can be restricted by introducing a `java.util.function.Predicate`. This predicate can be injected into the Spring bean `index`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides whether to include the index document into the computation of this value.

Attribute	Type	Description
		To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the index bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="index" custom-ref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexContentDocuments`	Read-only	The number of index documents that represent content and that were indexed in the last <*n*> seconds, where <*n*> is the value of the attribute `BatchStatisticsIntervalSeconds`. If <*n*> is 0, this attribute will contain the value since the start of the *Content Feeder*. The set of index documents used to compute this value can be restricted by introducing a `java.util.function.Predicate`. This predicate can be injected into the Spring bean `index`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides whether to include the index document into the computation of this value. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the feeder bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="index" custom-ref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexMaxLagTime`	Read-only	The maximum delay in seconds of the index documents that represent content and that were indexed in the last <*n*> seconds, where <*n*> is the value of the attribute `IndexStatisticInterval`. If <*n*> is `0` or greater than the value of attribute `IndexMaxStatisticInterval`, this attribute will contain the value since the start of the *Content Feeder*. The difference of the time

Attribute	Type	Description
		when a *batch* was successfully sent and the feedable field *freshness* are used for each feedable object where *feederstate* is `SUCCESS`. The set of index documents used to compute this value can be restricted by introducing a `java.util.function.Predicate`. This predicate can be injected into the Spring bean `index`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides whether to include the index document into the computation of this value. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the index bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="index" custom-ref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexMinLagTime`	Read-only	The minimum delay in seconds of the index documents that represent content and that were indexed in the last <*n*> seconds, where <*n*> is the value of the attribute `IndexStatisticInterval`. If <*n*> is `0` or greater than the value of attribute `IndexMaxStatisticInterval`, this attribute will contain the value since the start of the *Content Feeder*. The difference of the time when a *batch* was successfully sent and the feedable field *freshness* are used for each feedable object where *feederstate* is `SUCCESS`. The set of index documents used to compute this value can be restricted by introducing a `java.util.function.Predicate`. This predicate can be injected into the Spring bean `index`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides whether to include the index document into the computation of this value.

Attribute	Type	Description
		To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the index bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="index" custom-ref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexMaxStatisticInterval`	Read-only	Maximum interval in seconds for the computation of statistics.
`IndexOpenBatches`	Read-only	Number of open batches.
`IndexStatisticInterval`	Read/Write	Time interval in seconds for which the statistics are calculated.
`LastFailure`	Read-only	Last failure that led to a stop of the *Content Feeder*.
`LatestIndexing`	Read-only	The time when last indexing happened for the last <*n*> seconds, where <*n*> is the value of the attribute `IndexStatisticInterval`. The set of index documents used to compute this value can be restricted by introducing a `java.util.function.Predicate`. This predicate can be injected into the Spring bean `index`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides whether to include the index document into the computation of this value. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the index bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="index" custom-ref="myPredicate" property="batchStatisticsFeedablePredicate" />`

Attribute	Type	Description
`PendingEvents`	Read-only	The number of events the *Content Feeder* is behind the most recent event. It is computed as the difference between the sequence number of the Content Server's current timestamp and the sequence number of the timestamp of the last event whose changes have been persisted in the index. Unified API subsequence numbers are not taken into account, that is two Unified API events with the same sequence number (but different subsequence numbers) are counted as single event. Each content is counted as one additional event when the *Content Feeder* is still initializing. The value of this attribute increases with changes to content, users or groups in the *Content Server*. It is decreased after the *Content Feeder* has processed these changes. Note that the value of this attribute may stay at a non-zero value for a short time after starting the *Content Feeder* and before the next change happens in the *Content Server*. This only happens if the latest events in the *Content Server* are user or group changes. This exceptional case does not indicate a lagging *Content Feeder*.
`PersistedEvents`	Read-only	The number of persisted events for the last <*n*> seconds, where <*n*> is the value of the attribute `IndexStatisticInterval`. If <*n*> is zero or greater than the value of attribute `IndexMaxStatisticInterval`, this attribute contains the total number of persisted events since starting the *Content Feeder*. Persisted events are computed as difference between sequence numbers of timestamps for which all changes have been persisted in the index. Unified API subsequence numbers are not taken into account, that is, two Unified API events with the same sequence number (but different subsequence numbers) are counted as single event.

Attribute	Type	Description
		This attribute contains the number of persisted contents as long as the *Content Feeder* is still initializing.
`PersistedEventsPerSecond`	Read-only	The number of persisted events per second for the last <*n*> seconds, where <*n*> is the value of the attribute `IndexStatisticInterval`. If <*n*> is zero or greater than the value of attribute `IndexMaxStatisticInterval`, this attribute contains the persisted events per second since starting the *Content Feeder*. Persisted events are computed as difference between sequence numbers of timestamps for which all changes have been persisted in the index. Unified API subsequence numbers are not taken into account, that is, two Unified API events with the same sequence number (but different subsequence numbers) are counted as single event. This attribute contains the persisted contents per second as long as the *Content Feeder* is still initializing.
`RetryConnectToIndexDelay`	Read-only	The time in seconds between retries to connect to the *Search Engine* on startup
`State`	Read-only	State of the *Content Feeder* (stopped, starting, initializing, running, failed).
`StateNumeric`	Read-only	State of the *Content Feeder* (0=stopped, 1=starting, 2=initializing, 3=running, 4=failed).
`Uptime`	Read-only	Uptime of the *Content Feeder* in milliseconds.

Table 6.1. JMX attributes of the Feeder MBean

Feeder MBean Operations

The following table shows the operations of MBean `com.coremedia:type=Feeder,application=content-feeder`:

Operation	Parameter	Description
stop		Stop the *Content Feeder*
clearCollection		Clears the Search Engine index. The *Content Feeder* must have been stopped with the stop operation before. All contents will be reindexed when the *Content Feeder* is restarted.

Table 6.2. JMX operations of the Feeder MBean

UpdateGroupsBackgroundFeed MBean Attributes

The following table shows the attributes of MBean `com.coremedia:type=UpdateGroupsBackgroundFeed,application=content-feeder`.

Attribute	Type	Description
`CurrentPendingContents`	Read-only	The number of contents in the currently processed folder still to be reindexed after rights rule changes.
`PendingFolders`	Read-only	The IDs of all pending folders which are not yet reindexed completely due to rights rule changes. The Content Feeder may already have started indexing contents from the first returned folder.

Table 6.3. JMX attributes of the UpdateGroupsBackgroundFeed MBean

UpdateGroupsBackgroundFeed MBean Operations

The following table shows the operations of MBean `com.coremedia:type=UpdateGroupsBackgroundFeed,application=content-feeder`:

Operation	Parameter	Description
estimatePendingContents		Returns the total number of contents still to be reindexed after rights rule changes, that is, the

Operation	Parameter	Description
		number of contents in the folders returned by JMX attribute `PendingFolders`. This is an expensive operation.

Table 6.4. JMX operations of the UpdateGroupsBackgroundFeed MBean

AdminBackgroundFeed MBean Attributes

The following tables show the attributes of MBean `com.coremedia:type=AdminBackgroundFeed,application=content-feeder`.

Attribute	Type	Description
`NumberOfPendingContents`	Read-only	The number of contents left for triggered reindexing.
`State`	Read-only	A string that describes the internal state of the background feed.

Table 6.5. JMX attributes of the AdminBackgroundFeed MBean

AdminBackgroundFeed MBean Operations

The following table shows the operations of MBean `com.coremedia:type=AdminBackgroundFeed,application=content-feeder`:

Operation	Parameter	Description
reindexAll	• optional: comma-separated list of aspect IDs for partial update	Triggers reindexing of all contents. If no aspects are specified, the whole contents get reindexed. If aspects are specified, partial updates are used.
reindexByQuery	• *Unified API* query string • optional: comma-separated list of as-	Triggers reindexing of all contents that fulfill the given UAPI query. If no aspects are specified, the whole contents get reindexed. If aspects are specified, partial updates are used.

Operation	Parameter	Description
	pect IDs for partial update	
reindexByType	• content type name • optional: comma-separated list of aspect IDs for partial update	Triggers reindexing of all contents of the given type and subtypes. If no aspects are specified, the whole contents get reindexed. If aspects are specified, partial updates are used.
cancel		Cancels reindexing triggered by this interface.

Table 6.6. JMX operations of the AdminBackgroundFeed MBean

6.4 CAE Feeder JMX Managed Beans

The *CAE Feeder* exports multiple JMX MBeans. The following overview describes attributes and operations of the MBeans `CaeFeeder`, `Feeder`, and `ProactiveEngine`. The MBean `SolrIndexer` is described in Section 6.5, "Solr Indexer JMX Managed Beans" [136]. The *CAE Feeder* exports more MBeans and attributes, which aren't documented in detail here.

CaeFeeder MBean

Operation	Parameter	Description
reindexContent	• Content ID	Triggers reindexing of the content with the given ID. The ID can be the numeric content ID or in a format like `coremedia:///cap/content/42`.
reindexByQuery	• *Unified API* query string	Triggers reindexing of all contents that fulfill the given UAPI query, and the configuration of base folders and content types for the *CAE Feeder*. Warning: This can be a very expensive operation, if many contents are reindexed.
reindexByType	• content type name	Triggers reindexing of all contents of the given type and subtypes, if configured for the *CAE Feeder*. Warning: This can be a very expensive operation, if many contents are reindexed.

Table 6.7. JMX operations of the CaeFeeder MBean

Feeder MBean

Attribute	Type	Unit	Description
`BatchAverageCreationTime`	read-only	milliseconds	The average creation time of persisted batches for the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the average time since the start of the Feeder.

Attribute	Type	Unit	Description
			The creation time is the time span between the time the first entry was put into a batch and the time the batch was ready for sending to the *CoreMedia Search Engine*.
`BatchAverageSendingTime`	read-only	milliseconds	The average sending time of persisted batches for the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the average time since the start of the Feeder. The sending time indicates how long it took to actually send the batch to the *CoreMedia Search Engine*, that is, the time it took to invoke the `index` method on the `AsyncIndexer` or `DirectIndexer` interfaces.
`BatchAverageProcessingTime`	read-only	milliseconds	The average processing time of persisted batches for the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the average time since the start of the Feeder. The processing time is the time span between the time a batch was successfully sent to the *CoreMedia Search Engine* and the time when the Feeder received a callback from the *Search Engine* which indicates that the batch has been processed. Callbacks are only used with custom `AsyncIndexer` implementations. For Apache Solr, this attribute is always `0`.
`BatchAveragePersistingTime`	read-only	milliseconds	The average persisting time of batches for the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is

Attribute	Type	Unit	Description
			`0`, this attribute will contain the average time since the start of the Feeder. The persisting time is the time span between the time a batch was processed by the *CoreMedia Search Engine* and the time when the Feeder received a callback from the *Search Engine* which indicates that the batch has been persisted. Callbacks are only used with custom `AsyncIndexer` implementations. For Apache Solr, this attribute is always `0`.
`BatchBytes`	read-only	byte	The sum of the byte size of persisted batches for the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the value since the start of the Feeder. Note that byte computation is a rough estimate only.
`BatchCount`	read-only	batches	The number of persisted batches for the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the value since the start of the Feeder.
`BatchEntriesPerSecond`	read-only	batch entries / second	The number of persisted batch entries per second in the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the value since the start of the Feeder. Batch entries are basically creations, updates or removals of index documents. Note that this value decreases if the Feeder is idle.

Attribute	Type	Unit	Description
`BatchEntryCount`	read-only	batch entries	The number of persisted batch entries for the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the value since the start of the Feeder. Batch entries are basically creations, updates or removals of index documents.
`BatchStatisticsIntervalSeconds`	read/write	seconds	The time in seconds used to compute statistic values for other attributes. If the value is `0` or greater than `BatchStatisticsMaxIntervalSeconds`, the time since the start of the Feeder is used.
`BatchStatisticsMaxIntervalSeconds`	read/write	seconds	The maximum value that can be used for `BatchStatisticsIntervalSeconds`. It defines how long statistic data will be kept by the Feeder. You cannot recover statistics for the past by increasing the value.
`BatchStatisticsLogIntervalSeconds`	read/write	seconds	The time interval in seconds in which the Feeder writes statistics to its log file (log level INFO).
`CallbackQueueSize`	read-only	callback objects	The number of pending `com.coremedia.cap.feeder.FeederCallback` objects in the internal callback queue.
`DeferredEntryCount`	read-only	batch entries	The number of batch entries that are currently deferred. New batch entries will be deferred as long as a batch with an entry that affects the same index document is currently being sent to the *Search Engine* or was not yet persisted by the *Search Engine*.

Attribute	Type	Unit	Description
			Batch entries are basically creations, updates or removals of index documents.
`ExecutorQueueCapacity`	read/write	objects	The number of `java.lang.Runnable` objects that fit into the internal executor queue. This is an internal setting and does not need to be changed.
`ExecutorQueueSize`	read-only	objects	The number of pending `java.lang.Runnable` objects in the internal executor queue.
`ExecutorRetryDelay`	read/write	milliseconds	The time to wait before the *CAE Feeder* retries to access the source data after errors. This is used if custom code calls method `execute` of `com.coremedia.cap.feeder.Feeder`.
`IndexAverageLagTime`	read-only	seconds	The average delay in seconds of the index documents that represent content beans and that were indexed in the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the value since the start of the Feeder. The difference of the time when a *batch* was successfully sent and the feedable field *freshness* are used for each feedable object where *feederstate* is `SUCCESS`. The set of index documents used to compute this value can be restricted by introducing a `java.util.function.Predicate`. This predicate can be injected into the Spring bean `feeder`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides whether to include the index document into the computation of this value.

Attribute	Type	Unit	Description
			To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the `feeder` bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="feeder" custom-ref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexContentDocuments`	read-only	documents	The number of index documents that represent content beans and that were indexed in the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the value since the start of the Feeder. The set of index documents used to compute this value can be restricted by introducing a `java.util.function.Predicate`. This predicate can be injected into the Spring bean `feeder`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides whether to include the index document into the computation of this value. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the `feeder` bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="feeder" custom-ref="myPredicate" prop`

Attribute	Type	Unit	Description
			`erty="batchStatisticsFeedablePredicate" />`
`IndexMaxLagTime`	read-only	seconds	The maximum delay in seconds of the index documents that represent content beans and that were indexed in the last *<n>* seconds, where *<n>* is the value of the attribute `BatchStatisticsIntervalSeconds`. If *<n>* is `0`, this attribute will contain the value since the start of the Feeder. The difference of the time when a *batch* was successfully sent and the feedable field *freshness* are used for each feedable object where *feederstate* is `SUCCESS`. The set of index documents used to compute this value can be restricted by introducing a `java.util.function.Predicate`. This predicate can be injected into the Spring bean `feeder`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides whether to include the index document into the computation of this value. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the `feeder` bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="feeder" custom-ref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexMinLagTime`	read-only	seconds	The minimum delay in seconds of the index documents that represent content beans and that were indexed in the last

Attribute	Type	Unit	Description
			<n> seconds, where *<n>* is the value of the attribute BatchStatisticsIntervalSeconds. If *<n>* is 0, this attribute will contain the value since the start of the Feeder. The difference of the time when a *batch* was successfully sent and the feedable field *freshness* are used for each feedable object where *feederstate* is SUCCESS. The set of index documents used to compute this value can be restricted by introducing a java.util.function.Predicate. This predicate can be injected into the Spring bean feeder. The include method accepts an object of type com.coremedia.cap.feeder.Feedable. The custom implementation decides whether to include the index document into the computation of this value. To inject a custom predicate use the bean customizer and replace the BatchStatisticsFeedablePredicate of the feeder bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="feeder" custom-ref="myPredicate" property="batchStatisticsFeedablePredicate" />`
LatestIndexing	read-only	date and time	The time when last indexing happened for the last *<n>* seconds, where *<n>* is the value of the attribute BatchStatisticsIntervalSeconds. The set of index documents used to compute this value can be restricted by introducing a java.util.function.Predicate. This predicate can be injected into the Spring bean feed

Attribute	Type	Unit	Description
			`er`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides whether to include the index document into the computation of this value. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the `feeder` bean: `<customize:replace id="batchStatisticsFeedablePredicate" bean="feeder" custom-ref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`MaxBatchSize`	read/write	batch entries	The maximum number of entries in a batch. It is sent to the *Search Engine* when the maximum number is reached. It defaults to the configured property `feeder.batch.max-size`.
`MaxBatchBytes`	read/write	byte	The maximum size of a batch in bytes. The *CAE Feeder* sends a batch to the *Search Engine* if its maximum size would be exceeded when adding more entries. It defaults to the configured property `feeder.batch.max-bytes`. Note that byte computation is a rough estimate only.
`MaxOpenBatches`	read/write	batches	The maximum number of batches indexed in parallel. This setting is not used with the default integration of Apache Solr but only with custom implementations of the `com.coremedia.cap.feeder.index.async.AsyncIndexer` interface. The *CAE Feeder* does not call the

Attribute	Type	Unit	Description
			index method of the AsyncIndexer interface to index another batch if the maximum number of parallel batches has been reached. The method will not be called until a callback about the persistence of one of these batches has been received. It defaults to the configured property `feeder.batch.max-open`.
`MaxProcessedBatches`	read/write	batches	The maximum number of batches processed by the Indexer in parallel. This setting is not used with the default integration of Apache Solr but only with custom implementations of the `com.coremedia.cap.feeder.index.async.AsyncIndexer` interface. The *CAE Feeder* does not call the index method of the AsyncIndexer interface to index another batch if the configured number of currently processed batches has been reached. The method will not be called until a callback about completed processing or persistence of one of these batches has been received. It defaults to the configured property `feeder.batch.max-open`.
`OpenBatches`	read-only	batches	The number of currently open batches which have been passed to a custom implementation of the `com.coremedia.cap.feeder.index.async.AsyncIndexer` interface but for which the *CAE Feeder* has not received a persisted callback yet.
`ProcessedBatches`	read-only	batches	The number of currently processed batches which have been passed to a custom implementation of the `com.coremedia.cap.feeder.index.async.AsyncIndexer` inter-

Attribute	Type	Unit	Description
			face but for which the *CAE Feeder* has not received a processed callback yet.
`RetrySendIdleDelay`	read/write	milliseconds	The *CAE Feeder* sends a batch which only contains retried entries and is not full with regard to the `MaxBatchSize` attribute after the *CAE Feeder* was idle for the time configured in this property. A retried entry is an entry which was sent to the *Search Engine* before but could not be indexed successfully. If the batch contains entries which are not retried, the value of attribute `SendIdleDelay` is used instead. It defaults to the configured property `feeder.batch.retry-send-idle-delay`.
`RetrySendMaxDelay`	read/write	milliseconds	The maximum time in milliseconds between the time the *CAE Feeder* received an error from the *Search Engine* and the time, the *CAE Feeder* tries to send the failed entry as part of a batch to the *Search Engine* again. The time is exceeded if `MaxOpenBatches` or `MaxProcessedBatches` are reached or an error occurs while contacting the *Search Engine*. If the batch contains entries which are not retried, the value of attribute `SendMaxDelay` is used instead. It defaults to the configured property `feeder.batch.retry-send-max-delay`.
`SendIdleDelay`	read/write	milliseconds	The *CAE Feeder* sends a batch which is not full with regard to the `MaxBatchBytes` attribute after the *CAE Feeder* was idle for the configured time in milliseconds. A *CAE Feeder* is idle when it is not processing a request from clients such as the *Proactive Engine*.

Attribute	Type	Unit	Description
			It defaults to the configured property `feeder.batch.send-idle-delay`.
`SendMaxDelay`	read/write	milliseconds	The maximum time in milliseconds between the points in time where the *CAE Feeder* receives a request from a client and sends this request as part of a batch to the *Search Engine*. The time is exceeded if `MaxOpenBatches` or `MaxProcessedBatches` are reached or an error occurs while contacting the *Search Engine*. It defaults to the configured property `feeder.batch.send-max-delay`.
`StartTime`	read-only	date and time	The time when the *CAE Feeder* was started.

Table 6.8. Attributes of the Feeder MBean

ProactiveEngine MBean

Attribute	Type	Unit	Description
`KeysCount`	read-only	number	The total number of "keys" that need to be kept up-to-date by the *CAE Feeder*. This is the sum of the number of Content Beans selected for feeding (that is, beans that have been sent or need to be sent to the search engine) plus the number of used fragment keys as described in Section 5.4.5, "Using Revalidating Fragments" [95]. The value is initialized when the *CAE Feeder* is started. It increases if new content is created that needs to be indexed.
`ValuesCount`	read-only	number	The number of "keys" whose latest evaluation is still up-to-date. This is a subset

Attribute	Type	Unit	Description
			of the total number of keys returned by attribute `KeysCount`. The value decreases after content has changed and when the *CAE Feeder* needs to recompute data that is then sent to the search engine. The difference of `KeysCount` and `ValuesCount` is a good indicator for the remaining work until the *CAE Feeder* has processed changes or completed initial feeding. When the *CAE Feeder* is idle, then `ValuesCount` is equal to `KeysCount`.

Table 6.9. Attributes of the ProactiveEngine MBean

6.5 Solr Indexer JMX Managed Beans

This managed bean is exported by the *CAE Feeder* and the *Content Feeder*.

SolrIndexer MBean

Attribute	Type	Unit	Description
`SolrCloud`	read-only	Boolean	Returns whether the Feeder is configured to connect to SolrCloud with configuration property `solr.cloud`.
`Url`	read-only	string	The URL of Apache Solr for feeding as configured in property `solr.url`.
`ZookeeperAddresses`	read-only	string	The ZooKeeper addresses as configured in property `solr.zookeeper.addresses`.
`Collection`	read-only	string	The Apache Solr collection.
`SendRetryDelay`	read/write	milliseconds	The time to wait before sending a batch to the *Search Engine* again after sending failed with an error in the *Search Engine*. It defaults to the configured property `feeder.solr.send-retry-delay`.
`NoRetryDocumentIdsCsv`	read/write	comma-separated string values	Index document IDs for which indexing must not be retried after errors. The SolrIndexer automatically triggers a retry when an index document cannot be sent to Solr because of temporary errors such as connection problems to Solr. Permanent errors that are caused by the content (for example, if it was destroyed in the meantime) are not retried. In rare cases, the SolrIndexer may treat an error that cannot be resolved quickly as temporary one and indexing is retried forever. In such a case, an administrat-

Attribute	Type	Unit	Description
			or can add the index document ID to the value of this JMX attribute to make the Sol-rIndexer skip errors for the index document. IDs must conform to the value of the Solr `id` field, for example `42` for a content indexed with the *Content Feeder* and `content bean:42` for a content bean indexed with the *CAE Feeder*. The value is empty by default after starting the Feeder. It is not persisted.

Table 6.10. Properties of SolrIndexer MBean

6.6 Supported Languages in Solr Language Detection

The Solr language detection implementation is based on the Google Code language detection project https://github.com/shuyo/language-detection which supports the following 53 languages and has some advanced CJK support.

Language Code	Language
`af`	Afrikaans
`ar`	Arabic
`bg`	Bulgarian
`bn`	Bengali
`cs`	Czech
`da`	Danish
`de`	German
`el`	Greek
`en`	English
`es`	Spanish
`et`	Estonian
`fa`	Persian
`fi`	Finnish
`fr`	French
`gu`	Gujarati

Language Code	Language
`he`	Hebrew
`hi`	Hindi
`hr`	Croatian
`hu`	Hungarian
`id`	Indonesian
`it`	Italian
`ja`	Japanese
`kn`	Kannada
`ko`	Korean
`lt`	Lithuanian
`lv`	Latvian
`mk`	Macedonian
`ml`	Malayalam
`mr`	Marathi
`ne`	Nepali
`nl`	Dutch
`no`	Norwegian
`pa`	Punjabi
`pl`	Polish

Language Code	Language
pt	Portuguese
ro	Romanian
ru	Russian
sk	Slovak
sl	Slovene
so	Somali
sq	Albanian
sv	Swedish
sw	Swahili
ta	Tamil
te	Telugu
th	Thai
tl	Tagalog
tr	Turkish
uk	Ukrainian
ur	Urdu
vi	Vietnamese
zh-cn	Simplified Chinese

Language Code	Language
zh-tw	Traditional Chinese

Table 6.11. Supported Languages

Glossary

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

Content Feeder	The *Content Feeder* is a separate web application that feeds content items of the CoreMedia repository into the *CoreMedia Search Engine*. Editors can use the *Search Engine* to make a full text search for these fed items.
Content item	In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.
Content Management Environment	The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules: • *CoreMedia Content Management Server* • *CoreMedia Workflow Server* • *CoreMedia Studio* • *CoreMedia Search Engine* • *CoreMedia Native Personalization* • *CoreMedia Preview CAE*
Content Management Server	Server on which the content is edited. Edited content is published to the Master Live Server.
Content Repository	*CoreMedia CMS* manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.
Content Server	*Content Server* is the umbrella term for all servers that directly access the CoreMedia repository: *Content Servers* are web applications running in a servlet container. • *Content Management Server* • *Master Live Server* • *Replication Live Server*
Content type	A content type describes the properties of a certain type of content. Such properties are for example title, text content, author, ...
Contributions	Contributions are tools or extensions that can be used to improve the work with *CoreMedia CMS*. They are written by CoreMedia developers - be it clients, partners or CoreMedia employees. CoreMedia contributions are hosted on Github at https://github.com/coremedia-contributions.
Control Room	*Control Room* is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
CORBA (Common Object Request Broker Architecture)	The term *CORBA* refers to a language- and platform-independent distributed object standard which enables interoperation between heterogenous applications over a network. It was created and is currently controlled by

	the Object Management Group (OMG), a standards consortium for distributed object-oriented systems. CORBA programs communicate using the standard IIOP protocol.
CoreMedia Studio	*CoreMedia Studio* is the working environment for business specialists. Its functionality covers all the stages in a web-based editing process, from content creation and management to preview, test and publication. As a modern web application, *CoreMedia Studio* is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.
Dead Link	A link, whose target does not exist.
Derived Site	A derived site is a site, which receives localizations from its master site. A derived site might itself take the role of a master site for other derived sites.
DTD	A Document Type Definition is a formal context-free grammar for describing the structure of XML entities. The particular DTD of a given Entity can be deduced by looking at the document prolog: `<!DOCTYPE coremedia SYSTEM "http://www.coremedia.com/dtd/coremedia.dtd"` There're two ways to indicate the DTD: Either by Public or by System Identifier. The System Identifier is just that: a URL to the DTD. The Public Identifier is an SGML Legacy Concept.
Elastic Social	*CoreMedia Elastic Social* is a component of *CoreMedia CMS* that lets users engage with your website. It supports features like comments, rating, likings on your website. *Elastic Social* is integrated into *CoreMedia Studio* so editors can moderate user generated content from their common workplace. *Elastic Social* bases on NoSQL technology and offers nearly unlimited scalability.
EXML	EXML is an XML dialect used in former CoreMedia Studio version for the declarative development of complex Ext JS components. EXML is Jangaroo 2's equivalent to Apache Flex (formerly Adobe Flex) MXML and compiles down to ActionScript. Starting with release 1701 / Jangaroo 4, standard MXML syntax is used instead of EXML.
Folder	A folder is a resource in the CoreMedia system which can contain other resources. Conceptually, a folder corresponds to a directory in a file system.
FTL	FTL (FreeMarker Template Language) is a Java-based template technology for generating dynamic HTML pages.

gRPC	gRPC is an open source high performance Remote Procedure Call (RPC) framework.
Headless Server	CoreMedia Headless Server is a CoreMedia component introduced with CoreMedia Content Cloud which allows access to CoreMedia content as JSON through a GraphQL endpoint. The generic API allows customers to use CoreMedia CMS for headless use cases, for example delivery of pure content to Native Mobile Applications, Smartwatches/Wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of *Best current practice* (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese, or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.
IOR (Interoperable Object Reference)	A CORBA term, *Interoperable Object Reference* refers to the name with which a CORBA object can be referenced.
Jangaroo	*Jangaroo* is a JavaScript framework developed by CoreMedia that supports TypeScript (formerly MXML/ActionScript) as an input language which is compiled down to JavaScript compatible with Ext JS. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net. Jangaroo 4 is the ActionScript/MXML/Maven based version for CMCC 10. Since CMCC 11 (2110), Jangaroo uses TypeScript and is implemented as a *Node.js* and *npm* based set of tools.
Java Management Extensions (JMX)	The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
Locale	Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server	The *Master Live Server* is the heart of the *Content Delivery Environment*. It receives the published content from the *Content Management Server* and makes it available to the *CAE*. If you are using the *CoreMedia Multi-Master Management Extension* you may use multiple *Master Live Server* in a CoreMedia system.

Master Site	A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites.
MIME	With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multimedia emails and of web documents is standardised.
MXML	MXML is an XML dialect used by Apache Flex (formerly Adobe Flex) for the declarative specification of UI components and other objects. Up to CMCC 10 (2107), CoreMedia Studio used the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 7. Starting with CMCC 11 (2110), a new, Node.js and npm based version of Jangaroo is used that supports standard TypeScript syntax instead of MXML/ActionScript, still compiling to Ext JS 7 JavaScript.
OCI (Open Container Initiative)	The Open Container Initiative (OCI) is a Linux Foundation project that defines open industry standards for container formats and runtimes. OCI specifications ensure compatibility and interoperability between container tools, engines, and orchestration platforms like Docker and Kubernetes.
ORAS (OCI Registry As Storage)	ORAS (OCI Registry As Storage) is a tool and specification that extends OCI registries to store and distribute OCI artifacts beyond container images. It provides a standardized way for developers to push and pull arbitrary content types to and from container registries, enabling these registries to function as general artifact stores.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	With projects you can group content and manage and edit it collaboratively, setting due dates and defining to-dos. Projects are created in the Control Room and managed in project tabs.
Property	In relation to CoreMedia, properties have two different meanings: In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type. In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Replication Live Server	The aim of the *Replication Live Server* is to distribute load on different servers and to improve the robustness of the *Content Delivery Environment*. The *Replication Live Server* is a complete Content Server installation. Its content is an replicated image of the content of a *Master Live Server*. The *Replication Live Server* updates its database due to change events from the *Master Live Server*. You can connect an arbitrary number of *Replication Live Servers* to the *Master Live Server*.
Resource	A folder or a content item in the CoreMedia system.

ResourceURI	A ResourceUri uniquely identifies a page which has been or will be created by the *Active Delivery Server*. The ResourceUri consists of five components: Resource ID, Template ID, Version number, Property names and a number of key/value pairs as additional parameters.
Responsive Design	Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.
Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site. A typical site also has a master site it is derived from.
Site Folder	All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.
Site Indicator	A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.
Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Template	In CoreMedia, FreeMarker templates used for displaying content are known as Templates. OR In *Blueprint* a template is a predeveloped content structure for pages. Defined by typically an administrative user a content editor can use this template to quickly create a complete new page including, for example, navigation, predefined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes Application	The *User Changes Application* is a *Content Repository* listener, which collects all content, modified by *Studio* users. This content can then be managed in the *Control Room*, as a part of projects and workflows.
Variants	The set of all content items in a multi-site hierarchy related to each other via master references. This includes the top-level master content items themselves.
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.

Weak Links	In general *CoreMedia CMS* always guarantees link consistency. But links can be declared with the *weak* attribute, so that they are not checked during publication or withdrawal. Caution! Weak links may cause dead links in the live environment.
Workflow	A workflow is the defined series of tasks within an organization to produce a final outcome. Sophisticated applications allow you to define different workflows for different types of jobs. So, for example, in a publishing setting, a document might be automatically routed from writer to editor to proofreader to production. At each stage in the workflow, one individual or group is responsible for a specific task. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process.
Workflow Server	The *CoreMedia Workflow Server* is part of the Content Management Environment. It comes with predefined workflows for publication but also executes freely definable workflows.
XLIFF	XLIFF is an XML-based format, standardized by OASIS for the exchange of localizable data. An XLIFF file contains not only the text to be translated but also metadata about the text. For example, the source and target language. *CoreMedia Studio* allows you to export content items in the XLIFF format and to import the files again after translation.

Index