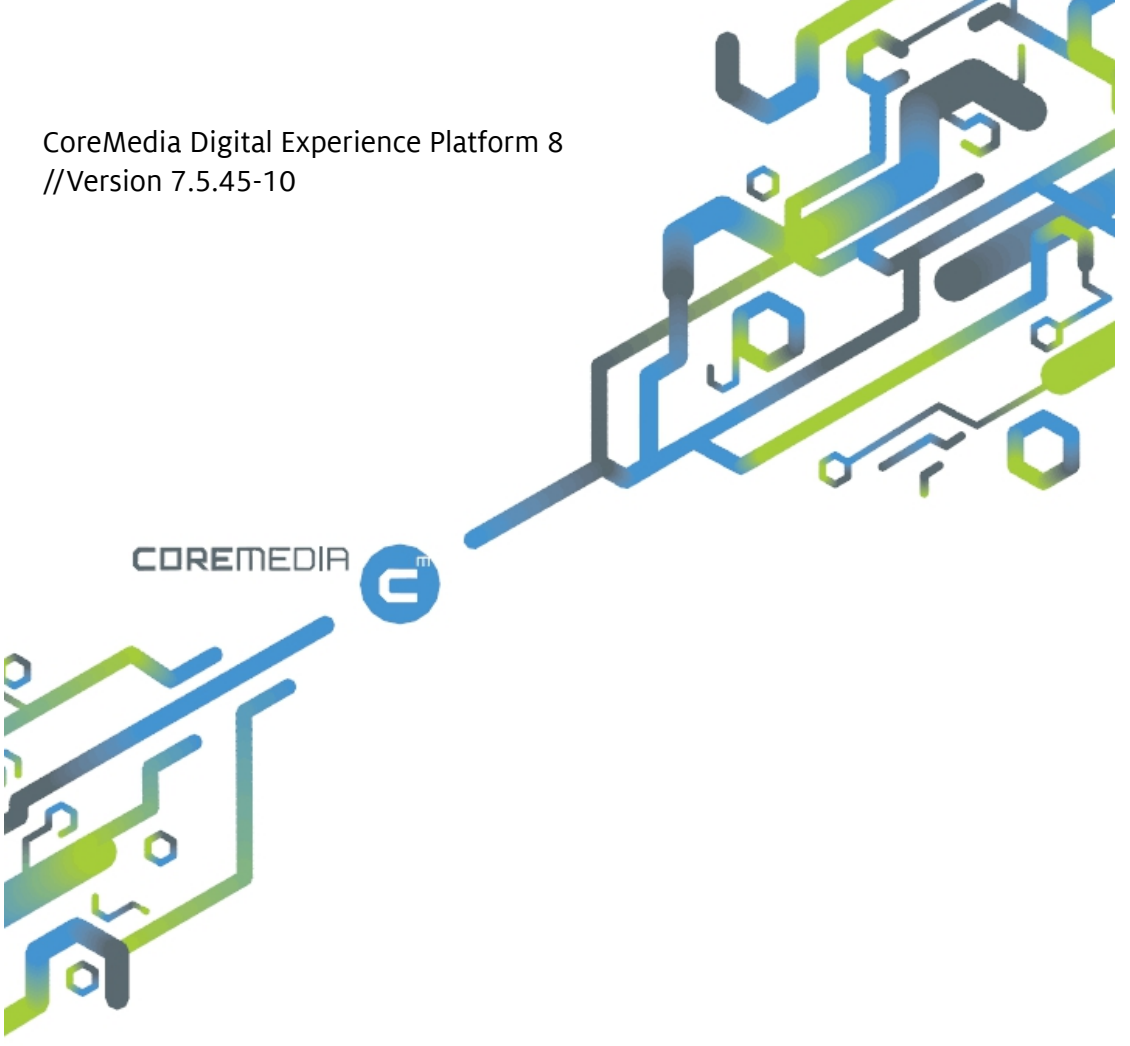
CoreMedia Digital Experience Platform 8
//Version 7.5.45-10



CoreMedia Search Manual

CoreMedia Search Manual

CoreMedia AG

Ludwig-Erhard-Straße 18

20459 Hamburg

07.Mar 2017

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:

Table 1.1. Typographic conventions

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

In addition, these symbols can mark single paragraphs:

Table 1.2. Pictographs

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

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

1.3 CoreMedia Services

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

CoreMedia User Orientation for CoreMedia Developers and Partners

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

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

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

1.3.1 Registration

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

1.3.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

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

http://releases.coremedia.com/dxp8

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

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

Maven artifacts

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

https://repository.coremedia.com

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

License files

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

1.3.3 Documentation

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

http://documentation.coremedia.com/dxp8

The manuals have the following content and use cases:

Table 1.3. CoreMedia manuals

Manual	Audience	Content
CoreMedia Utilized Open-Source Software	Developers, architects, administrators	This manual lists the third-party software used by CoreMedia and lists, when required, the licence texts.
Supported Environments	Developers, architects, administrators	This document lists the third-party environments with which you can use the CoreMedia system, Java versions or operation systems for example.
Studio User Manual, English	Editors	This manual describes the usage of *CoreMedia Studio* for editorial and administrative work. It also describes the usage of the *Adaptive Personalization* and *Elastic Social* GUI that are integrated into *Studio*.

Manual	Audience	Content
LiveContext for IBM WebSphere Manual	Developers, architects, administrators	This manual gives an overview over the structure and features of CoreMedia LiveContext. It describes the integration with the IBM WebSphere Commerce system, the content type model, the *Studio* extensions, folder and user rights concept and many more details. It also describes administrative tasks for the features. It also describes the concepts and usage of the project workspace in which you develop your CoreMedia extensions. You will find a description of the Maven structure, the virtualization concept, learn how to perform a release and many more.
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application or the usage of the watchdog component.
Adaptive Personalization Manual	Developers, architects, administrators	This manual describes the configuration of and development with *Adaptive Personalization*, the CoreMedia module for personalized websites. You will learn how to configure the GUI used in *CoreMedia Studio*, how to use predefined contexts and how to develop your own extensions.
Analytics Connectors Manual	Developers, architects, administrators	This manual describes how you can connect your CoreMedia website with external analytic services, such as Google Analytics.
Content Application Developer Manual	Developers, architects	This manual describes concepts and development of the *Content Application Engine (CAE)*. You will learn how to write JSP or Freemarker templates that access the other CoreMedia modules and use the sophisticated caching mechanisms of the CAE.
Content Server Manual	Developers, architects, administrators	This manual describes the concepts and administration of the main CoreMedia component, the *Content Server*. You will learn about the content type model which lies at the heart of a CoreMedia system, about user and rights management, database configuration, and more.

Manual	Audience	Content
Elastic Social Manual	Developers, architects, administrators	This manual describes the concepts and administration of the *Elastic Social* module and how you can integrate it into your websites.
Importer Manual	Developers, architects	This manual describes the structure of the internal CoreMedia XML format used for storing data, how you set up an *Importer* application and how you define the transformations that convert your content into CoreMedia content.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.
Site Manager Developer Manual	Developers, architects, administrators	This manual describes the configuration and customization of *Site Manager*, the Java based stand-alone application for administrative tasks. You will learn how to configure the *Site Manager* with property files and XML files and how to develop your own extensions using the *Site Manager API*.
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of *CoreMedia Studio*. You will learn about the underlying concepts, how to use the development environment and how to customize *Studio* to your needs.
Unified API Developer Manual	Developers, architects	This manual describes the concepts and usage of the *CoreMedia Unified API*, which is the recommended API for most applications. This includes access to the content repository, the workflow repository and the user repository.
Workflow Manual	Developers, architects, administrators	This manual describes the *Workflow Server*. This includes the administration of the server, the development of workflows using the XML language and the development of extensions.

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

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

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

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

http://www.coremedia.com/training

Contact the Training department at the following email address:

Email: training@coremedia.com

1.3.5 CoreMedia Support

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

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

http://support.coremedia.com/

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

Email: support@coremedia.com

Create a support request

Support request

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

- Which CoreMedia component(s) did the problem occur with (include the release number)?
- Which database is in use (version, drivers)?
- Which operating system(s) is/are in use?
- Which Java environment is in use?
- Which customizations have been implemented?
- A full description of the problem (as detailed as possible)
- Can the error be reproduced? If yes, give a description please.
- How are the security settings (firewall)?

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

Support checklist

To put it in a nutshell, CoreMedia needs:

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

Log files

An essential feature for the CoreMedia system administration is the output log of Java processes and CoreMedia components. They're often the only source of information for error tracking and solving. All protocolling services should run at the highest log level that is possible in the system context. For a fast breakdown, you should be logging at debug level. The location where component log output is written is specified in its `<appName>-logback.xml` file.

Which Log File?

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

Where do I Find the Log Files?

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

Table 1.4. Log files check list

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

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

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

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

Component	Problem	Log files
		`workflowserver.log` `capclient.properties`

	import	`importer.log`
		`coremedia.log`
		`capclient.properties`

	workflow	`editor.log`
		`workflow.log`
		`coremedia.log`
		`capclient.properties`

	spell check	`editor.log`
		`MS Office version details`
		`coremedia.log`

	licenses	`coremedia.log` (*Content Management Server*) `coremedia.log` (Master Live Server)

Server and client	communication errors	`editor.log`
		`coremedia.log`
		(*Content Management Server*)
		`coremedia.log`
		(Master Live Server)
		`*.jpif files`

	preview not running	`coremedia.log` (content server) `preview.log`

	website not running	`coremedia.log`
		(*Content Management Server*)
		`coremedia.log`
		(Master Live Server)
		coremedia.log
		(*Replication Live Server*)
		`Blueprint.log`
		`capclient.properties`
		`license.zip`

Server	not starting	`coremedia.log`
		(*Content Management Server*)
		`coremedia.log`
		(Master Live Server)
		`coremedia.log`
		(*Replication Live Server*)
		`capclient.properties`
		`license.zip`

1.4 Change Chapter

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

Table 1.5. Changes

Section	Version	Description
Section 5.4, “Indexing Content Beans” [67]	7.5.41	Added chapter about revalidating fragment keys.
Section 4.1, “Concepts” [34] and Section 6.1, “Content Feeder Configuration” [88]	7.1.9	Added a small section about *Content Feeder* partial update functionality and configuration properties `solr.partialUpdates`, `solr.partialUpdatesSkipIndexCheck` and `feeder.partialUpdate.aspects` in the appendix.

2. Overview

The *CoreMedia Search Engine* adds full-text search capabilities to the *CoreMedia CMS*. You can use it to quickly find documents 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.

Document search is available in the *Site Manager* and in *Studio*. You can 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 documents to the *Search Engine* for indexing. This makes it possible to search for documents in the *Site Manager*, *Studio* and custom content applications.

  Chapter 4, *Searching for Content* [33] describes concepts, configuration and operation of the required components in detail.

- Content applications often require search functionality not only for single documents 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* [60] 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.workerNode=true`.

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

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

Apache Solr is open source and you can find the Solr reference guide at https://cwiki.apache.org/confluence/display/solr/Apache+Solr+Reference+Guide. Some links in this manual point to HTML pages of the Solr reference guide, which may describe newer versions of Solr. If in doubt, please read the reference guide for the correct Solr version as used in *CoreMedia Digital Experience Platform 8*. You can get the Solr reference guide as a PDF for the correct version at http://archive.apache.org/dist/lucene/solr/ref-guide/.

More useful information is available on the *Apache Solr* website at http://lucene.apache.org/solr/.

3.1 Starting

Apache Solr is a web application. So, in order to start the *CoreMedia Search Engine* you simply have to start the web container into which the Search Engine is deployed. The Solr administration page is available at `http://<host>:<port>/solr`, for example at `http://localhost:44080/solr` when started locally in the *CoreMedia Blueprint*.

3.2 Solr Home Directory

In addition to the actual web application, Solr uses a special directory called *Solr Home* for configuration files, additional libraries and index files. It is configured either via JVM system property `solr.solr.home` or via JNDI lookup of `java:comp/env/solr/home` and needs to be writable by the Solr process. It has the following general structure:

```
<solr-home>/
    solr.xml
    configsets/
        <configset1>/
            conf/
                schema.xml
                solrconfig.xml
                ...
        <configset2>/
        ...
    cores/
        <core1>/
            core.properties
            data/
                index/
                    <index files>
                tlog/
                    <transaction log files>
        <core2>/
        ...
    lib/
        <additional jar files>
```

The Solr server manages multiple indices with possibly different configurations. Each of these indices is stored as a *Lucene index* on disk. An index managed by a Solr server is called a *Solr Core* (or shortly a core) in Solr terminology.

solr.xml

The file `solr.xml` is the central Solr configuration file. It contains only few settings, which you do not need to change. Most of Solr's configuration is placed in other configuration files. It however enables *core discovery mode* for Solr, which means that available Solr Cores are automatically discovered. In earlier versions of Solr available cores were listed explicitly in the file `solr.xml`. This *legacy mode* is not used in the *CoreMedia CMS*.

You can find more information about the `solr.xml` file in the Solr Reference Guide at https://cwiki.apache.org/confluence/display/solr/Format+of+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.

Cores

The `cores` directory contains the actual Solr Cores, which are the indices used by your applications. Solr automatically discovers cores by looking for `core.properties` files below the Solr Home directory. Each directory with a `core.properties` file represents a Solr Core. The *CoreMedia Search Engine* comes with three predefined cores:

- `studio`: an index of *CoreMedia* documents used for searching in *Studio* and *Site Manager*, 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*.

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

Elastic Social applications create Solr Cores for users and comments automatically when they are started the first time. With *CoreMedia Blueprint* and tenant `media`, you will see additional directories `blueprint_media_comments` and `blueprint_media_users` for these cores below `<solr-home>/cores`. These Solr cores use the `elastic` config set, if not configured otherwise with *Elastic Social* configuration property `elastic.solr.indexConfig`.

Earlier version of *CoreMedia CMS* used a single shared index for *Content Feeder* and *CAE Feeder* applications. Using separate Solr cores has a number of advantages:

- It becomes possible to use Solr's runtime administration capabilities such as reloading existing cores after configuration changes, adding new cores and even replacing existing cores.
- Separate cores provide better performance and use less memory. Solr caches work more efficiently because they only need to store data for the searched index and not for a larger shared index. Also, caches won't be invalidated after changing documents of other indices.
- It avoids problems with relevancy scoring. Index statistics such as the term frequency are used to compute the relevancy of search results. In a shared index, unrelated documents may change the scoring unintentionally.
- It becomes possible to back up and restore indexes independently from one another.
- It becomes possible to move a single index to another Solr installation.
- Different indices can use different configurations and index schema.

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`.

Lib directory

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

3.3 Reindexing

There are several reasons why you might want to reindex all index documents. This includes changes in the *Search Engine* 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 the *Search Engine*. In any case, reindexing a whole index is a very expensive operation and takes some time.

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` web 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.

Reindexing Content Feeder and CAE Feeder indices

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 almost all cases this is not what you want because search will be unavailable (or only return partial results) until reindexing has completed. See section "Clear Search Engine index" [58] and Section 5.3.2, "Resetting" [65] 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.

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.

You can follow these steps to reindex from scratch:

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=CREATE&name=<name>&instanceDir=cores/<name>&configSet=<configSet>&dataDir=data
```

a. Replace `<hostname>` and `<port>` with host name and port of the servlet container that runs the Apache Solr master.

b. Replace `<name>` with the name of the new core. Mind that it appears twice in the above URL. You can choose any name you like as long as no such core and no such directory below `<solr-home>/cores` exists yet. If you are using *Elastic Social* you should also avoid names that start with the configured `elastic.solr.indexPrefix` 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 directory `<solr-home>/cores`. 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 `<solr-home>/cores/studio2/core.properties` should contain something like:

```
#Written by CorePropertiesLocator
#Thu Dec 11 17:16:47 CET 2014
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:

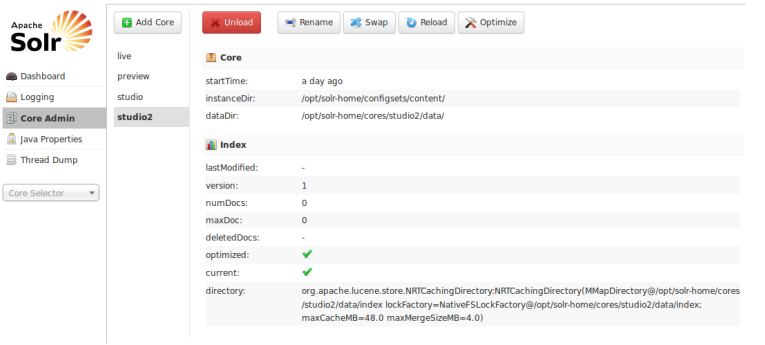


Figure 3.1. New Solr Core

3. Set up a new Feeder instance and configure it to feed into the new Solr core by setting the property `feeder.solr.url` accordingly. Do not change the property `feeder.solr.collection`.

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

   ```
   feeder.solr.url=http://localhost:44080/solr/studio2
   feeder.solr.collection=studio
   ```

   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 document 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.

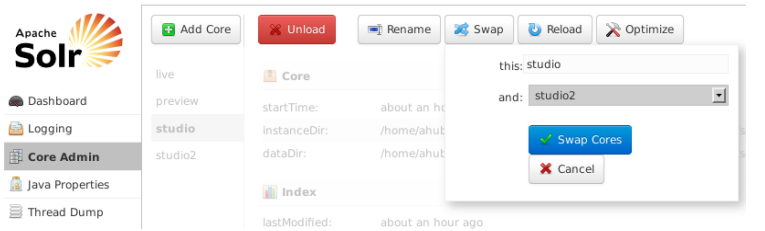


Figure 3.2. Swap Solr Cores

   It's important to understand that this operation does not change the directory structure in `<solr-home>/cores` 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 `<solr-home>/cores/studio2`. You can verify this by looking into its `core.properties` file:

   ```
   #Written by CorePropertiesLocator
   #Thu Dec 11 17:26:33 CET 2014
   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 `feeder.solr.url` 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 `WEB-INF/application.properties`:

```
feeder.solr.url=http://localhost:44080/solr/studio
feeder.solr.collection=studio
```

8. If you're using Solr replication, the new index will be replicated automatically to the Solr slaves after a commit was made on the Solr master 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:44080/solr/studio/update?commit=true` for the Solr core named `studio` on the Solr master running on localhost and port 44080.

   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 slaves. You can see the progress of replication on the Solr slave'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 master with the **[Unload]** button on the **Core Admin** page of the Solr Admin UI. In the example, this would be the core named `studio2`.

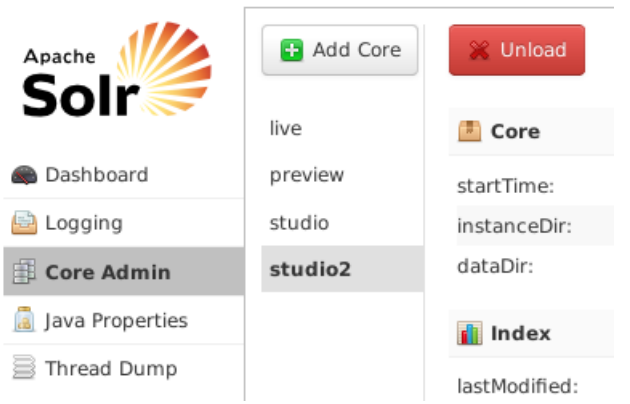


Figure 3.3. Unload old Solr Core

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 `<solr-home>/cores/studio`

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 the Solr Reference Guide at https://cwiki.apache.org/confluence/display/solr/CoreAdmin+API.

3.4 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

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* in contrast 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 Solr index is fresher than the *CAE Feeder* database, the *CAE Feeder* will possibly redundantly refeed some documents which is OK, but if the Solr index is older than the *CAE Feeder* database the commits between the time of the *CAE Feeder* backup and the Solr backup would be lost.

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

Back up of the Solr index

To take a hot back up of the Solr index you can use Solr's `ReplicationHandler`. Once configured, a backup can be taken with the following HTTP request to the Solr Master server. Replace `<core>` with the name of the Solr core you want to back up.

```
http://<host>:<port>/solr/<core>/replication?command=backup
```

You can find the snapshot files under the Solr core's `data` directory afterwards. For details see https://cwiki.apache.org/confluence/display/solr/Index+Replication.

3.5 Searching in Different Languages

Processing steps for multi-language use

The *CoreMedia Search Engine* enables you to search in documents 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. For standard western languages, such as English, German, French, you do not necessarily need to change the configuration, because the standard configuration already handles these languages quite well. If you use Asian languages, such as Chinese, Japanese or Korean (known as CJK languages) you have to do some configuration because these languages must be treated differently to extract searchable words.

3.5.1 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” [39] and Section 5.4.3, “Customizing Feedables” [68] 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 a Solr `LangDetectLanguageIdentifierUpdateProcessorFactory` to detect the language of incoming index documents. It is described in detail in the Solr Reference Guide at https://cwiki.apache.org/confluence/display/solr/Detecting+Languages+During+Indexing. See Section 6.6, “Supported Languages in Solr Language Detection” [125] in the appendix of this manual for a list of supported languages. The language code from that list is stored as value in `language` field.

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.

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 correct tokenization for CJK languages or add some language-specific analysis steps such as stemming for western languages. In both cases you need to configure language dependent fields.

Example

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

Search in language-dependent fields

Search in language-dependent fields

When searching in *Studio*, *Site Manager* or with *Unified API*'s SearchService methods, searches are 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 https://cwiki.apache.org/confluence/display/solr/The+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.

3.5.2 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

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 for tokenization of non-CJK languages with reasonable cross-language defaults
- `text_zh`, configured for tokenization of Chinese (Simplified and Traditional)

Apache Solr provides special field types for lots of languages in its example configuration, for example `text_ja` for Japanese and `text_cjk` which can be used 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 file `example/solr/collection1/conf/schema.xml` after downloading and unpacking the Apache Solr distribution. You can download Solr from http://lucene.apache.org/solr/.

Example

If you index Chinese text only, you can simply change field definitions from type `text_general` to type `text_zh` in `schema.xml`:

```
<fields>
  ...
  <field name="textbody" type="text_zh" ... />
</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” [125] in the appendix shows the list of supported languages.

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"` to make it possible to use partial document updates in the *Content Feeder*.

The following example shows necessary fields and additional types in `<solr-home>/configsets/content/conf/schema.xml` for supporting Simplified Chinese, Japanese, Korean and non-CJK languages in the predefined fields `name_tokenized` and `textbody` of the `content` config set.

```
<field name="name_tokenized"       type="text_general"
                                   indexed="true" stored="true"/>
<field name="name_tokenized_ja"    type="text_ja"
                                   indexed="true" stored="true"/>
<field name="name_tokenized_zh-cn" type="text_zh"
                                   indexed="true" stored="true"/>
<field name="name_tokenized_ko"    type="text_cjk"
                                   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_cjk"
                                    indexed="true" stored="false"
                                    multiValued="true"/>

<!-- field types "text_general" and "text_zh" are
     already defined in default configuration -->

<!-- field types "text_cjk" and "text_ja" are
     copied from the Apache Solr example configuration -->
...
```

In the above example, Japanese text goes into `name_tokenized_ja` and `textbody_ja`, Simplified Chinese text goes into `name_tokenized_zh-cn` and `textbody_zh-cn`, Korean text goes into `name_tokenized_ko` and `textbody_ko` and text from all other languages is indexed in the fields `name_tokenized` and `textbody`.

Besides Simplified Chinese you can also configure Traditional Chinese text with the fields `name_tokenized_zh-tw` and `textbody_zh-tw`. The language code `zh` from previous CoreMedia releases is not generated anymore, but existing fields `name_tokenized_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_tokenized` 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_tokenized</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” [125] in the appendix 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 the Solr reference guide at https://cwiki.apache.org/confluence/display/solr/Detecting+Languages+During+Indexing.

Configuring index feeding

Configuring language-dependent field handling

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_tokenized</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_tokenized` and `numericid`. Matches in the field `name_tokenized` are scored higher than matches in `textbody` because of the configured `^2` boost. Note that the language-dependent fields `name_tokenized_*` 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_tokenized^2 numericid^10</str>
    <str name="pf">textbody name_tokenized^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 by calling `SolrQuery.setParam(CommonParams.QT, "/cmdismax");` or by appending `/cmdismax` to the URL used to connect to Solr. 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 the *Site Manager*, *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* [60].

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 its clients such as the *Site Manager*
- and 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*, the *Content Server's* search service and the configuration of the *Search Engine* for content search in custom applications and in *Studio*.

The next sections describe

- the concepts of content search in Section 4.1, “Concepts” [34]
- the configuration of the *Content Feeder* in Section 4.2, “Configure the Content Feeder” [37]
- the configuration of the search service of the *Content Server* in Section 4.3, “Configure Search for the Content Server” [50]
- the configuration of the *Search Engine* for search suggestions in the *Studio* in Section 4.4, “Configure Search Suggestions for Studio” [52]
- the modification of the *Search Engine* index schema for custom search applications in Section 4.5, “Modify the Search Index” [55]
- the operation of the *Content Feeder* in Section 4.6, “Operation of the Content Feeder” [56]
- Section 4.7, “Implementing Custom Search” [59] provides some hints for implementing a custom search application

4.1 Concepts

The *Content Feeder* sends content and metadata of documents to the *CoreMedia Search Engine*. The *Search Engine* extracts the textual data of the documents, indexes them and provides the possibility to search for these documents. The *Content Feeder* is a web 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. The *Site Manager* and 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*.

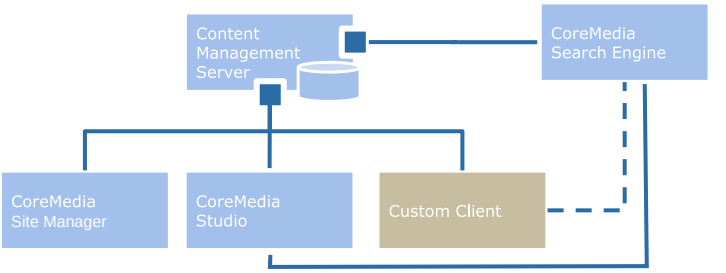


Figure 4.1. Search Engine Integration

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

To provide full-text search for documents 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.

Feeding the Search Engine

When the *Content Feeder* starts for the first time, it iterates over the documents in the repository and sends them to the *Search Engine* for indexing. After this initialization phase, the *Content Feeder* sends documents 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 document that needs to be indexed. This document 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 document properties and a number of metadata attributes such as the path of the document, the name of its creator and the last time the document was published. In the configuration of the *Content Feeder* you can restrict the indexed documents by their type and its indexed properties by their name and type. Note, that the *CoreMedia Search Engine* only indexes the latest document version.

Partial Updates

The *Content Feeder* can use partial updates if only document metadata has changed. This means, it does not need to send the whole document data to the search engine but just a small set of changed metadata, for example a changed path after documents have been moved to another place in the repository. This can greatly improve performance, especially if lots of documents 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 as stored as in the default configuration. See the description of the configuration properties `solr.partialUpdates`, `solr.partialUpdatesSkipIndexCheck` and `feeder.partialUpdate.aspects` in Section 6.1, “Content Feeder Configuration” [88] for more details.

Batches

For better performance the *Content Feeder* sends batches to the *Search Engine*. A batch contains changes of multiple documents. 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.

Error conditions

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

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

Error documents contain the value `ERROR` in the index field `feederstate` and are not returned as search result by the *Content Server*. You can search for error

documents using the administration page of the *Content Feeder*. An error document is replaced with the correct document when the document 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.

Restrictions

The *CoreMedia Search Engine* provides a fast and efficient full-text search for the indexed documents. However, because of the asynchronous nature of the indexing process, search results do not always reflect the current state of the repository. A document may need a couple of seconds after it was sent to the *Search Engine*, before it appears in the search results. Sometimes you can query for changes faster if you use the more powerful but in general slower built-in query feature of the *CoreMedia Content Server*.

The *CoreMedia Search Engine* supports search in the content of the latest document version. If you want to search for older versions or for folders 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 documents of the *Content Management Environment*, for example in the *Site Manager*.

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

→ Section 4.2.1, “Required Configuration” [37]

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” [39]

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

→ Section 4.2.3, “Advanced Configuration” [47]

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 documents 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* [60] for details.

Configuration of the Content Feeder

Like most *CoreMedia* web applications the *Content Feeder* web application uses the Application architecture. Therefore, configuration of properties can be done in `WEB-INF/application.properties`, via JNDI or JVM system properties or in an additional property files. Bean configuration can be done in `WEB-INF/application.xml`. For details please consult the [CoreMedia DXP 8 Manual].

4.2.1 Required Configuration

Configuring the Content Server URL

The property `repository.url` has to be set to the IOR URL of the *Content Server*.

Example

```
repository.url=http://localhost:44441/coremedia/ior
```

Configuring the Search Engine Location

The *Content Feeder* needs the URL of the search engine. Configure the URL of Apache Solr in property `feeder.solr.url`. The URL has the following format

```
http://<host>:<port>/<solr-webapp>/<solr-core>
```

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

Example

```
feeder.solr.url=http://localhost:8082/solr/studio
```

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

Configuring the Search Engine Collection

Configure the property `feeder.solr.collection` with the name of the *CoreMedia Search Engine* collection. The *Content Feeder* writes the collection name to the field *collection* in the Solr index.

Example

```
feeder.solr.collection=studio
```

Configuring the user account

The *Content Feeder* requires a user account to access the documents 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*.

If you migrated from a release prior to CMS 2005, the *Content Feeder* fails to start when the *CoreMedia Content Server* starts, the first time because the user account does not exist. In that case, create the user account manually. Afterwards you can use the administration page to start the *Content Feeder*.

→ Configure the user account for the *Content Feeder* with the properties `repository.user` and `repository.password`.

For example:

```
repository.user=feeder
repository.password=secret
```

4.2.2 Content Configuration

Configuring Document Types

You can restrict the indexed documents by their type in the file `feeder.properties`. The document types are configured with the following two properties:

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

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

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

Note, that the *Content Feeder* does not update already processed documents after changing the document types to index. A configuration change only affects newly processed documents. If you want to update all documents, restart the *Content Feeder* with an empty index.

Configuring Properties for Indexing

You can restrict the indexed properties of a document 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.

Configuration not mandatory: The default configuration includes all String and CoreMedia RichText XML properties. It also includes blob properties of 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 document 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 document type Article and all properties of document type Picture except the property 'Copyright'.

```
<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.

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/1.13/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, Java class 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 document properties by their type in the file `feeder.properties`. The following example shows the default configuration for property types:

```
# indexed property types
feeder.content.propertyType.string=true
feeder.content.propertyType.integer=false
feeder.content.propertyType.date=false
feeder.content.propertyType.linkList=false
feeder.content.propertyType.struct=false
```

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

```
# Indexed blob properties, configured by comma-separated MIME-types
# If you don't configure any MIME-types in the includes property,
# no blob properties will be indexed.
# You can exclude a more specific type (for example, text/xml) while
# including the corresponding primary type (for example, text/*)
feeder.content.propertyType.blobMimeType.includes=text/*, \
application/pdf,application/msword,application/ \
vnd.openxmlformats-officedocument.wordprocessingml.document
feeder.content.propertyType.blobMimeType.excludes=

# The maximum size in byte for included blob properties;
# larger blobs will be skipped.
# This configuration can be overridden in a Spring XML configuration
# file where you can configure the maximum size per MIME-type by
# customizing the bean 'feederContentBlobMaxSizePerMimeType'.
# See applicationContext.xml for an example.
feeder.content.propertyType.blobMaxSize=5242880
```

Note, that the *Content Feeder* does not update already processed documents after changing the properties. A configuration change only affects newly processed documents. If you want to update all documents, restart the *Content Feeder* with an empty index.

Configuring Fields to Index in

The *Content Feeder* can be configured to index document 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.

Configuration not mandatory: By default, all document 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 document 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 document.

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 the *Site Manager* and *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 `stored="true"` and `indexed="true"`. 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 document type and attribute `property` specifies the name of the document 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 document types will not be indexed even if a matching `PropertyField` is configured. The following example configures indexing of the property *headline* of document 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`. The following `FeedablePopulator` classes already exist:

- `PropertyPathFeedablePopulator`: Index specific values from a struct document property.

- `XPathFeedablePopulator`: Extracts a text fragment from an XML document property.
- `ImageDimensionFeedablePopulator`: Set image attributes like image orientation, dimension, and size category.
- `ContentStatusFeedablePopulator`: Set the document 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.content.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 document gets added or updated and the whole document data is sent to the search engine. Populators registered at property `partialUpdateFeedablePopulators` are called for partial updates, when only document 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` of the extended interface Feedable2 returns which aspects of the index document are changed with a partial update.

PropertyPathFeedablePopulator

The PropertyPathFeedablePopulator is configured with a dot-separated property path to index a specific property value from a struct document property. The first name in the property path denotes the struct document property itself while the following names specify nested properties of the struct. The constructor argument `type` selects the type of the documents. 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` documents.

```
<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 document and the property `property` selects the document 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` documents.

```
<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 document 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 document 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 document 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>
```

Note, that the *Content Feeder* does not update already processed documents after changing the fields to index. A configuration change only affects newly processed documents. If you want to update all documents, restart the *Content Feeder* with an empty index.

4.2.3 Advanced Configuration

Configuring Batch Handling

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

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

The file `feeder.properties` contains properties to configure batch sending.

- `feeder.maxBatchSize`: The maximum number of index documents in a batch. A smaller batch may be sent if the maximum byte size is reached before.
- `feeder.maxBatchByteSize`: The maximum number of bytes allowed in a batch. A smaller batch may be sent if the maximum batch size is reached before.
- `feeder.sendIdleDelay`: The maximum seconds to wait sending a new batch if the *Content Feeder* is idle. This value normally is small to feed a document quickly for low latency, such as when a document was changed by an editor.
- `feeder.sendMaxDelay`: The maximum seconds 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 documents are imported with an importer.

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

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.retrySendIdleDelay`: The maximum seconds to wait sending a failed batch again, if the *Content Feeder* is idle.
- `feeder.retrySendMaxDelay`: The maximum seconds to wait sending a failed batch again, if the batch is not yet full.
- `feeder.solr.sendRetryDelay`: The delay in seconds between a failed batch sending and the next try. The default value is 30 seconds.
- `feeder.retryConnectToIndexDelay.seconds`: The delay in seconds between retries to connect to the *Search Engine* on startup. The default value is 10 seconds.
- `feeder.solr.connection.timeout`: The connection timeout set on the SolrJ `SolrServer`. 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 in milliseconds.
- `feeder.solr.socket.timeout`: The socket timeout set on the SolrJ `SolrServer`. 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 600000 milliseconds. That means it will wait for 10 minutes.

Configuring Tika

Apache Tika is used to extract text from blob properties for indexing. It 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
```

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.appendMetadata` and `feeder.tika.copyMetadata` to extract and index metadata from binary formats.

The property `feeder.tika.appendMetadata` 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.copyMetadata` 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” [55].

Example

```
feeder.tika.copyMetadata=creator=author
```

The above example configures the *Content Feeder* to store the creator as extracted from the metadata 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`.

Configuring updates of rights rule changes

The *Content Feeder* indexes the groups with potential read rights to a document in the index field `groups`. The set of groups is then used to narrow a user's search down to the documents 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 an increased feeding load, because documents must be reindexed after changing rights rules on any parent folder up to the root folder. You can disable this optimization by setting the property `feeder.indexGroups` to `false` in the file `feeder.properties`. If you've set that property to `false`, then you should also configure the *Studio* application to not add a superfluous query condition for the indexed groups by setting its property `studio.rest.searchService.useGroupsFilterQuery` to `false`.

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.

4.3 Configure Search for the Content Server

To search for documents in the *Site Manager*, *Studio* or custom client applications, you need to configure the *CoreMedia Search Engine* with the *CoreMedia Content Server*. The *CoreMedia Content Server* connects to the *CoreMedia Search Engine* to handle search requests for its clients.

Configure in the following files below `WEB-INF`:

- `properties/corem/contentserver.properties`
- `config/contentserver/spring/search/search-solr.properties`
- `config/contentserver/spring/search/applicationContext.xml`

Configuring the Search Engine Location

In file `search-solr.properties` configure the property `search.solr.urls` with the URL of the Apache Solr core. For example `http://localhost:8081/solr/studio`. If you change this setting, you have to restart the server. You can also configure multiple comma-separated URLs in this property if you want to use multiple Solr servers for failover and simple load balancing, but note that Studio always uses only the first configured URL.

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

Configuring the Search Engine Collection

In file `search-solr.properties` configure the property `search.solr.collection` with the name of the *CoreMedia Search Engine* collection.

```
search.solr.collection=studio
```

The *Content Feeder* stores the collection name in the field `collection` of Solr index documents. A search result only contains documents belonging to the same collection. To achieve this, the content server automatically adds the collection name to a user query. If you change this setting, you have to restart the server.

Enable or Disable Search

Search functionality is enabled by default. You can disable it by setting property `cap.server.search.enable` to `false` in the file `contentserver.proper`

`ties`. If disabled in the *Content Management Server*, no search dialog will be available in the *Site Manager*. Note that *Studio* requires search to be enabled in the *Content Management Server*.

Configuring the CoreMedia Search Engine Timeout

In file `search-solr.properties` configure the property `search.solr.connection.timeout` with a timeout value in milliseconds used in HTTP requests to the search engine. The default value is 0, which means no timeout is applied.

```
search.solr.connection.timeout=0
```

4.4 Configure Search Suggestions for Studio

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 documents 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_tokenized^2 numericid^10</str>
  <str name="pf">textbody name_tokenized^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 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 3.5, “Searching in Different Languages” [26], 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">0.0001</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 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.

4.5 Modify the Search Index

Configuration not mandatory: Change the *Apache Solr* `schema.xml` in `<solr-home>/configsets/content/conf` if you want to add an index field used for search with *CoreMedia* or native search engine APIs.

By default, search is performed in index fields `textbody`, `name_tokenized`, `numericid` and their language-dependent variants `textbody_*` and `name_tokenized_*` 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* or in the *Site Manager*. The content from the document properties is 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” [37].

You can search in a specific field with the method `SearchService#searchNative` from the Unified API (for details see *CoreMedia Unified Developer Manual*, Section "Search Service" in chapter "The Content Repository"). Another possibility is to use the search engine native API directly.

4.6 Operation of the Content Feeder

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

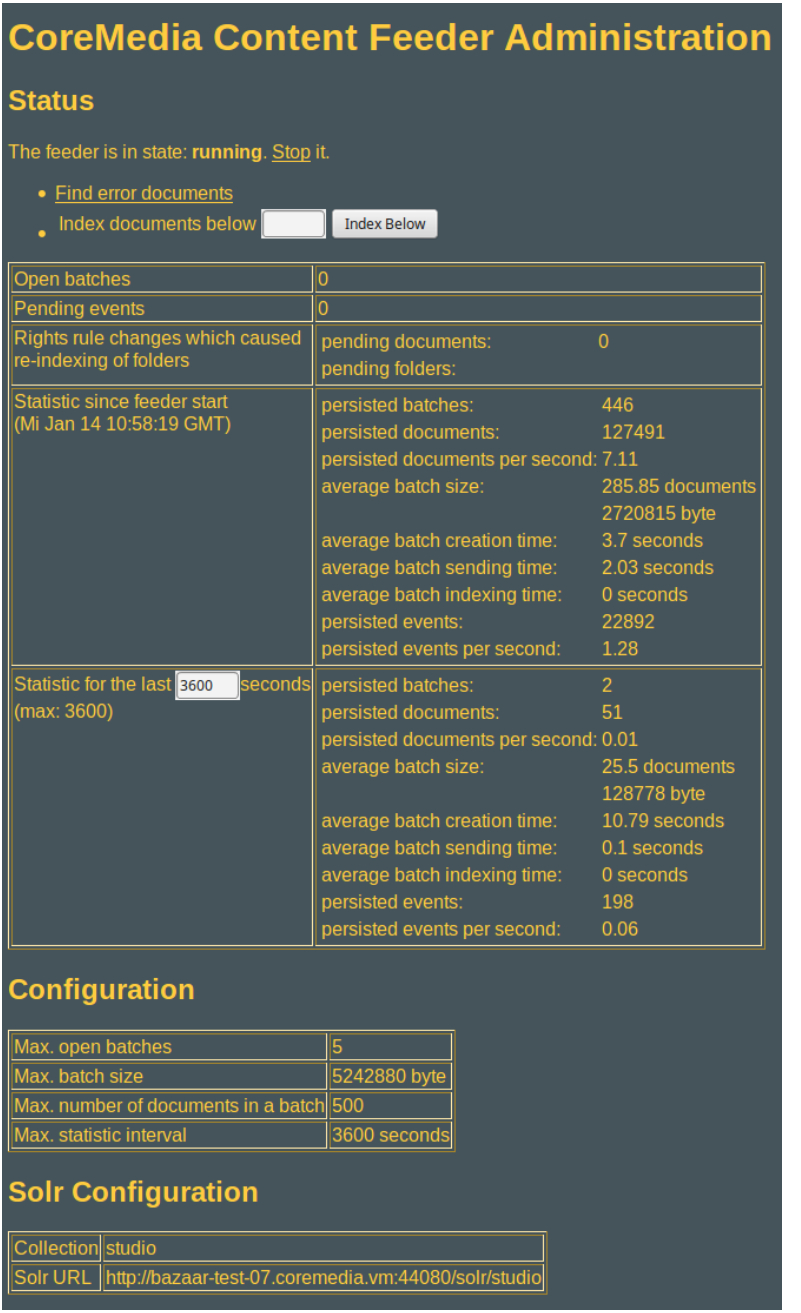
Administration Page

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

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

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

It is recommended to change the password in productive environments.



CoreMedia Content Feeder Administration

Status

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

- Find error documents
- Index documents below [ ] Index Below

Open batches	0	
Pending events	0	
Rights rule changes which caused re-indexing of folders	pending documents: pending folders:	0
Statistic since feeder start (Mi Jan 14 10:58:19 GMT)	persisted batches: persisted documents: persisted documents per second: average batch size: average batch creation time: average batch sending time: average batch indexing time: persisted events: persisted events per second:	446 127491 7.11 285.85 documents 2720815 byte 3.7 seconds 2.03 seconds 0 seconds 22892 1.28
Statistic for the last [3600] seconds (max: 3600)	persisted batches: persisted documents: persisted documents per second: average batch size: average batch creation time: average batch sending time: average batch indexing time: persisted events: persisted events per second:	2 51 0.01 25.5 documents 128778 byte 10.79 seconds 0.1 seconds 0 seconds 198 0.06

Configuration

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

Solr Configuration

Collection	studio
Solr URL	http://bazaar-test-07.coremedia.vm:44080/solr/studio

Figure 4.2. 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 error documents. They represent documents that were not processed successfully by the *Content Feeder* or the *CoreMedia Search Engine*. The page contains links to manually retry indexing of error documents. If not used, the *Content Feeder* retries indexing of error documents the next time the document changes.

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

Index documents below

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

Start and Stop the Content Feeder

The *Content Feeder* is started and stopped like any other web 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 web application.

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 documents of the *Content Server* from the *Search Engine* index. All documents will be reindexed when the *Content Feeder* is restarted.

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

See also Section 3.3, “Reindexing” [20] to learn how to reindex without search downtime.

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 beans:

- Feeder applications such as the *CAE Feeder* and the *Content Feeder* require separate *Apache Solr* cores. When searching you must specify a core in the Apache Solr URL to get results for the specific application only.
- 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. Generally you just need the content id, which is stored in field `id`. You can use IDs of the search results to get the Content objects back from the Unified API. See the *CoreMedia 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” [61] gives an overview over the architecture of the CAE Feeder
- Section 5.2, “Configuring the CAE Feeder” [62] describes the configuration of the *CAE Feeder* environment
- Section 5.3, “Operations of the CAE Feeder” [65] describes the operation of the *CAE Feeder*
- Section 5.4, “Indexing Content Beans” [67] 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” [81] describes how to use the *CAE Feeder* with a different search engine or external system
- Section 5.6, “CAE Feeder for API Use” [84] describes how to set up a *CAE Feeder* to develop custom applications using its public API
- Section 5.7, “Implementing Custom Search” [86] provides some hints for implementing search in a *CAE* application

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 a web application, which enables search functionality not only for single *CoreMedia* documents, as the *Content Feeder* does, but for content beans, where data may be computed from multiple source documents. 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* web 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:

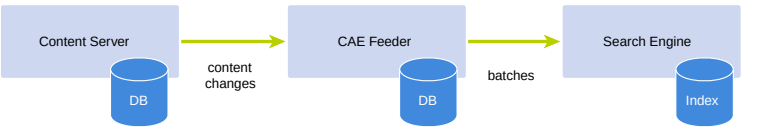


Figure 5.1. CAE Feeder architecture

Create your own application

If you do not want to have updates automatically send to the search engine after content changes, but control yourself when data is sent to the search engine, then you can use the API of the *CAE Feeder* and develop a custom application as described in Section 5.6, “CAE Feeder for API Use” [84].

5.2 Configuring the CAE Feeder

This section describes common configuration tasks. See Section 6.3, "CAE Feeder Configuration" [107] for a detailed description of configuration settings. All properties can be configured in the file `WEB-INF/application.properties` of the *CAE Feeder* web application.

Configuring the Content Server

The *CAE Feeder* can be used to index content beans for content from the *Content Management Server* or a *Live Server*. Configure the *Content Server* for the *CAE Feeder* as in the following example:

Example 5.1. Configure the Content Server

```
repository.url=http://localhost:44441/coremedia/ior
repository.user=webserver
repository.password=webserver
repository.domain=
```

The property `repository.url` specifies the URL of the *Content Server*. The properties `repository.user`, `repository.password` and `repository.domain` define the account of the user used by the *CAE Feeder* to log in to the *Content Server*.

Configuring the Database

The *CAE Feeder* persists the feeding state in a database. Configure the connection to the database with the following properties:

`jdbc.driver`	Specifies the class of the database driver
`jdbc.url`	Contains the URL of the database
`jdbc.user`	Specifies the account name of the database user
`jdbc.password`	Specifies the account password of the database user

For example:

Example 5.2. Configure the database

```
jdbc.driver=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:oracle
jdbc.user=username
jdbc.password=password
```

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

Configuring the Search Engine

The configuration of the connection to the *CoreMedia Search Engine* includes setting host name and port of the installed search engine and the name of the target Solr core. This is done by setting the properties `feeder.solr.url` and `feeder.solr.collection`. Each feeding application needs a different index. Do not use the same index for multiple instances of the *CAE Feeder* or the *Content Feeder*.

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

Example 5.3. Configure the Search Engine for Apache Solr

```
feeder.solr.url=http://localhost:8001/solr/preview
feeder.solr.username=
feeder.solr.password=
feeder.solr.collection=preview
```

Configuring Tika

Extracting metadata

Apache Tika is used to extract text from blob properties for indexing. It 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
```

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.appendMetadata`
- `feeder.tika.copyMetadata`

The property `feeder.tika.appendMetadata` 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.copyMetadata` 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 5.4.4, “Modifying the Search Index” [73].

Example

```
feeder.tika.copyMetadata=creator=author
```

The above example configures the *CAE Feeder* to store the creator as extracted from the metadata in the index field `author`. You have to declare the index field 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`.

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:

Table 5.1. Properties for retry on Solr server

Property	Value	Default	Description
`feeder.solr.sendRetryDelay`	time in seconds	30	The delay between a failed batch sending and the next try.
`feeder.solr.connection.timeout`	time in milliseconds	0	The connection timeout set on the SolrJ `SolrServer`. It determines how long the client waits to establish a connection without any response from the server. The default value 0 means, that it will wait forever.
`feeder.solr.socket.timeout`	time in milliseconds	600000 (10 minutes)	The socket timeout set on the SolrJ `SolrServer`. It determines how long the client waits for a response from the server after the connection was established and the request was already sent.

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.

The *CAE Feeder* is available as a web application that can be deployed into a supported servlet container. The *resetcaefeeder* command-line tool of the *CAE Feeder* is available as a separate stand-alone application.

5.3.1 Starting and Stopping

You can start and stop the *CAE Feeder* with the servlet container.

The *CAE Feeder* will wait for the *Content Management Server* and for *Apache Solr* to become available if necessary.

5.3.2 Resetting

To reset the *CAE Feeder* and feed all documents 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 3.3, “Reindexing” [20] 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. If you have content beans with a lot of dependencies, for example, and you want to use the *CAE Feeder* to feed these beans into the *Search Engine* you might face problems when contents change during the initial feeding process. Because in this case, even few changes of the content beans might lead to a lot of invalidations of already fed beans.

To prevent this, you can temporarily disable invalidations of already fed beans.

To do so, set the property `contentDependencyInvalidator.invalidationStopped=true` and restart the *CAE Feeder*.

After initially feeding the content beans, set the property back to "false" otherwise no invalidations will 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.

Configuring the Content Selector

Definition of content selector

The file `caefeeder-triggers.xml` located in classpath `/framework/spring/caefeeder/` contains the Spring Framework bean definition of the content selector. The default implementation `PathAndTypeContentSelector` selects contents by type and path. You can configure it with the following properties:

`feeder.contentSelector.basePath`	Specifies a comma-separated list of content repository folder paths.
`feeder.contentSelector.contentTypes`	Contains a comma-separated list of content types.
`feeder.contentSelector.includeSubTypes`	Specifies whether subtypes of the configured content types are selected as well. The default is true.

Example

Example 5.4, “ContentSelector example” [67] selects all contents of type `CMMedia`, `CMArticle`, `CMDownload` and `CMCollection` (including sub types) which are located below the path `/Sites`:

Example 5.4. ContentSelector example

```
feeder.contentSelector.basePath=/Sites

feeder.contentSelector.contentTypes=CMMedia,CMArticle,CMDownload,CMCollection
feeder.contentSelector.includeSubTypes=true
```

Customizing the content types list

You can extend the set of indexed content beans by customizing a property of the content selector called `contentTypeNames`. This is useful when you use extensions (see the [CoreMedia DXP 8 Manual] for details), because an extension can not extend a property file but it can extend Spring configuration.

The following example defines a simple configuration which customizes the bean `contentTypeNames`, defined in file `caefeeder-triggers.xml`, by adding a `CMPicture` to the set of content types defined in `feeder.contentSelector.contentTypes`:

```
<customize:append id="contentTypeNamesCustomizer"
bean="contentTypeNames">
  <list>
    <value>CMPicture</value>
  </list>
</customize:append>
```

5.4.2 Configuring Content Bean Classes

The *CAE Feeder* needs a definition of the 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* configuration.

Configure the content bean classes in the Spring application context as described in the [CoreMedia 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 content of a content bean is mapped to fields of the feedable and is therefore added to the index if a corresponding Solr index field exists. The following paragraphs describe the involved classes.

The `FeedableContentBeanEvaluator` creates feedables from ContentBean objects. You can find the configuration in the file `caefeeder-triggers.xml`, which is located in the classpath `/framework/spring/caefeeder`.

Example 5.5. Definition of FeedableContentBeanEvaluator

```
<bean name="contentEvaluator" class=
"com.coremedia.amaro.cae.feeder.FeedableContentBeanEvaluator">
  <property name="contentBeanFactory" ref="contentBeanFactory"/>
  <property name="keyTransformer" ref="feederKeyTransformer"/>
  <property name="feedableFactory" ref="feedableFactory"/>
  <property name="feedablePopulator"
```

```
            ref="errorHandlingFeedablePopulator"/>
</bean>
```

Create an identifier for search 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 `FeedableContentBeanEvaluator` uses an implementation of com.coremedia.cap.feeder.populate.FeedablePopulator to fill the elements of the feedable with the values of a content bean. By default, a BeanMappingFeedablePopulator is used which maps Java bean properties of ContentBean objects to elements of the created feedable as configured.

If required, you can configure additional `FeedablePopulator` implementations in the property `populators` of the bean `compositeFeedablePopulator`. The property takes a list of `FeedablePopulator<T>` beans, which makes it possible to combine data from different implementations into 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

If a bean property's get method throws an exception, the *CAE Feeder* will index a so-called error document in the index 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` clause to your queries to find successfully indexed documents. Bean feeding will by default automatically be retried after 10 minutes or if a dependency is invalidated that was accessed before the exception was thrown. Errors are handled by an instance of class com.coremedia.cap.feeder.populate.ErrorHandlingFeedablePopulator which wraps all `FeedablePopulator` instances. It is available in the Spring Context as bean `errorHandlingFeedablePopulator` and can be customized as described in its API documentation.

Defining the Properties for Indexing

The BeanMappingFeedablePopulator class has two properties that you can use for customizing the mapping between content bean properties and Feedable.

- `beanPropertiesByClass`
- `beanMappings`

`beanMappings` offers more powerful options. You can, for example, add a property converter implementation that maps to a specific type.

Using `beanPropertiesByClass`

This configuration provides a simple way for bean properties which are mapped to feedable elements with the same name. The values of these bean properties are written to an index field with the same name, if it exists. Furthermore, the bean property values will always be appended to the `textbody` index field.

In more detail, the property `beanPropertiesByClass` of the BeanMappingFeedablePopulator takes a `java.util.Map` object, which maps bean classes to comma-separated strings of their indexed bean properties. This map is available in the Spring application context under the name `caeFeederBeanPropertiesByClass` and can be customized.

The following example defines the mapping for content beans of classes `com.coremedia.example.contentbeans.Text` and `com.coremedia.example.contentbeans.Download`. For content beans of class `Text` and subclasses, the Java bean properties `headline` and `text` map to elements of the feedable. When constructing a feedable the BeanMappingFeedablePopulator calls the property methods `getHeadline` and `getText` of class `Text` to retrieve the values for these elements.

```
<customize:append id="caeFeederBeanPropertiesByClassCustomizer"
                  bean="caeFeederBeanPropertiesByClass">
  <map>
    <entry key="com.coremedia.example.contentbeans.Text"
           value="headline,text"/>
    <entry key="com.coremedia.example.contentbeans.Download"
           value="data"/>
  </map>
</customize:append>
```

Using `beanMappings`

A more powerful configuration is available with the property `beanMappings` of the BeanMappingFeedablePopulator. The new options are:

- Define to which search field a content bean property is mapped
- Define that a content bean property should not be mapped to the `textBody` field of Solr
- Define your own property converter

- Define a default value when a property returns null
- Adding parameters to a feedable

The property `beanMappings` takes a list of mappings where each mapping applies to one bean class. You can customize this list of mappings as shown below. A mapping for a single bean class is represented by a com.coremedia.cap.feeder.bean.BeanFeedableMapping. Each BeanFeedableMapping contains a list of mappings for Java bean properties of the bean class in the property `beanPropertyMappings`. A mapping for a single Java bean property to an element of the Feedable is represented by a com.coremedia.cap.feeder.bean.BeanPropertyFeedableElementMapping. See Example 5.6, “Example Content Bean to Feedable Mapping” [71] for an example.

A content bean can inherit from or extend other content beans. In this case, you might have different BeanFeedableMapping elements that match for an instance of a content bean. If so, the order of the `BeanFeedableMapping` elements in the list of mappings is important: The first mapping of a property that matches overwrites all following mappings that match.

Example mapping using `beanMappings`

Example 5.6, “Example Content Bean to Feedable Mapping” [71] defines a mapping for the superclass of all content beans com.coremedia.objectserver.beans.ContentBean. The bean property `content.modificationDate` maps to the feedable element named `freshness`. The default Solr index schema defines an index field with that name, to which the bean property's value is written. The bean property uses the syntax of Spring framework's bean wrapper for nested properties. When constructing a feedable the BeanMappingFeedablePopulator calls the property methods `getContent().getModificationDate()` of class ContentBean to retrieve the value for the element. Furthermore, the value is not added to the `textbody` index field.

Overwritten mappings

Keep in mind, that if you define a mapping for `freshness` for any other content bean class and add it behind this example mapping to the list of mappings, it would be overwritten by our example definition and you would get a warning in the log file. So, avoid this.

Example 5.6. Example Content Bean to Feedable Mapping

```
<customize:append id="caeFeederBeanMappingsCustomizer"
                  bean="caeFeederBeanMappings">
  <list>
    <ref local="exampleBeanFeedableMapping"/>
  </list>
</customize:append>

<bean id="exampleBeanFeedableMapping"
      class="com.coremedia.cap.feeder.bean.BeanFeedableMapping">
  <property name="beanClass"
            value="com.coremedia.objectserver.beans.ContentBean"/>
  <property name="beanPropertyMappings">
```

```
    <list>
      <bean class="com.coremedia.cap.feeder.bean.
                   BeanPropertyFeedableElementMapping">
        <property name="beanProperty"
                  value="content.modificationDate"/>
        <property name="feedableElement" value="freshness"/>
        <property name="textBody" value="false"/>
      </bean>
    </list>
  </property>
</bean>
```

See the API documentation for a description of all properties of the classes BeanMappingFeedablePopulator, BeanFeedableMapping and BeanPropertyFeedableElementMapping in package `com.coremedia.cap.feeder.bean`.

Mapping of Property Types

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

Table 5.2. Feedable Element Types for Java Bean Properties

property value class	element type
com.coremedia.cap.common.Blob	Binary
`java.util.Date` and `java.util.Calendar`	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

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.

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

Configuring your own Property Converter

You can configure a property converter to convert the value to one of the supported types. A property converter implements the interface com.coremedia.cap.feeder.bean.PropertyConverter and can be configured with the `propertyConverter` property of the `BeanPropertyFeedableElementMapping`. Property converters are for example useful when indexing collection properties. The property converter implementations com.coremedia.cap.feeder.bean.CollectionPropertyConverter and com.coremedia.cap.feeder.bean.CollectionToStringPropertyConverter can be used for this purpose. Please see the Javadoc for details.

Default value for null results

Furthermore, it is possible to configure a default value which should be indexed if a bean property is `null` or a configured PropertyConverter returns `null`. A default value can be configured with the `defaultValue` property of the BeanPropertyFeedableElementMapping. Again, please see the Javadoc for details.

5.4.4 Modifying the Search Index

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” [68] 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 document with global settings. Such a document may contain lots of different settings in different properties or in a single struct property. Imagine, that a single setting `S1` from this document is accessed during the construction of each indexed feedable. Because of this, each indexed bean will depend on the properties of the settings document. Now, if somebody changes the document, 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 re-indexing 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.

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.

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 document 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 documents. 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.contentSelector.contentTypes=Article
```

and customizing the bean `caeFeederBeanPropertiesByClass`:

```
<customize:append id="caeFeederBeanPropertiesByClassCustomizer"
                  bean="caeFeederBeanPropertiesByClass">
  <map>
    <entry key="com.customer.example.beans.Article"
           value="folderpath"/>
```

```
  </map>
</customize:append>
```

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 RevalidatingFragmentPersistentCacheKey. 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.

Example 5.7. Example of a fragment key implementation

```
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 PersistentCache2 persistentCache;
  private final ContentRepository contentRepository;
  private final String contentId;

  public IdPathKey(PersistentCache2 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);
    }
}
```

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:"` (i.e. `"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:

Example 5.8. Example of a PersistenCacheKeyFactory implementation

```
package com.customer.example;
import com.coremedia.cap.content.*;
import com.coremedia.cap.persistentcache.*;

public class IdPathKeyFactory
      implements PersistentCacheKeyFactory {
  private PersistentCache2 persistentCache;
  private ContentRepository contentRepository;

  public void setPersistentCache(PersistentCache2 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);
    return (String)persistentCache.getCached(key);
  }
}
```

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.

Example 5.9. Define and register the factory in the Spring context

```
<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>
```

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:

Example 5.10. Using the fragment key in the content bean

```
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);
  }
}
```

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

Example 5.11. Configure content bean with factory

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

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* web 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.PersistentCache2 to retrieve a fragment value. This method uses in-memory CacheKeys 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 `java.lang.Object` and a default cache weight equal to one. You must configure a reasonable cache capacity for that cache class, for example:

```
<bean id="objectClassCacheCapacityConfigurer"
      class="com.coremedia.cache.CacheCapacityConfigurer"
      init-method="init">
  <property name="cache" ref="cache"/>
  <property name="capacities">
    <map>
      <entry key="java.lang.Object" value="10000"/>
    </map>
  </property>
</bean>
```

If you forget to configure the cache capacity, the value is not cached and the cache will log warnings about an unreasonable cache size. 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 PersistentCache#get or PersistentCache2#getCached 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-search-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-cae feeder-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* web application.

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

5.6 CAE Feeder for API Use

If you need more control, you can set up a *CAE Feeder* which does not automatically send updates to the search engine. Instead, you can use the public API to do so. Such a setup does not require a database. It is based on the *CAE Feeder* but requires some manual configuration.

For wiring such a Feeder, use the following Spring bean definitions, for example, as file `config/feeder/spring/applicationContext.xml`.

Example 5.12. caefeeder.xml

```
<?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 class="org.springframework.beans.factory.config.
              PropertyPlaceholderConfigurer">
    <property name="ignoreUnresolvablePlaceholders"
              value="true"/>
    <property name="locations" value=
"file:config/feeder/spring/environment.properties"/>
  </bean>
  <import resource=
"classpath:/framework/spring/feeder/feeder-core.xml"/>
  <import resource=
"classpath:/framework/spring/feeder/solr/feeder-solr.xml"/>
  <import resource=
"classpath:/framework/spring/feeder/tika/feeder-tika.xml"/>
  <bean class="com.coremedia.springframework.context.
              LifecycleManager">
    <property name="startableBeans" ref="feederStartables"/>
  </bean>

</beans>
```

The first bean `PropertyPlaceholderConfigurer` makes the *CAE Feeder* use the settings from the configured property files. Create the file `environment.properties` with the search engine connection settings as follows:

```
feeder.solr.url=http://localhost:8082/solr/mycore
feeder.solr.collection=collection
```

Next, required beans are imported using Spring import statements. To this end, you need the following libraries on your classpath as runtime dependencies.

```
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>cap-search-impl</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>cap-search-solr</artifactId>
  <scope>runtime</scope>
</dependency>
```

```
<dependency>
  <groupId>com.coremedia.cms</groupId>
  <artifactId>cap-search-tika</artifactId>
  <scope>runtime</scope>
</dependency>
```

The `LifecycleManager` bean starts the feeder when the Spring application context is created and stops it when the application context is closed.

In your custom Java code, access the Feeder API as follows. The example has a compile dependency to the artifact `cap-search-api`.

Example 5.13. Create CAE Feeder

```
FileSystemXmlApplicationContext context =
     new FileSystemXmlApplicationContext(
         "config/feeder/spring/applicationContext.xml");
context.registerShutdownHook();
Feeder feeder = (Feeder)context.getBean("feeder");
FeedableFactory feedableFactory =
    (FeedableFactory)context.getBean("feedableFactory");
```

First the Spring application context is created. `FileSystemXmlApplicationContext` is part of the Spring Framework in the Java package `org.springframework.context.support`. The next statements retrieve Feeder and FeedableFactory implementations from the application context. You can use them to send data to the *Search Engine* as described in the API documentation of the Java package `com.coremedia.cap.feeder`.

5.7 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 beans:

- Feeder applications such as the *CAE Feeder* and the *Content Feeder* require separate *Apache Solr* cores. When searching you must specify a core in the Apache Solr URL to get results for the specific application only.
- 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 IDs.

6. Appendix

6.1 Content Feeder Configuration

The *Content Feeder* is configured in the files `WEB-INF/application.properties` and `WEB-INF/application.xml`.

Solr specific configuration properties

These properties are configured in file `application.properties` of the *Content Feeder*.

Table 6.1. Solr specific properties

Attribute	Value	Default	Description
`feeder.solr.url`	URL	http://localhost:8082/solr/coremedia	The URL where the *Content Feeder* can reach the *Search Engine*. The URL points to the Apache Solr core for the *Content Feeder*.
`feeder.solr.username`	user name or empty	(empty)	User name for HTTP Basic authentication when connecting to the Apache Solr web application. Leave empty for no authentication.
`feeder.solr.password`	user name or empty	(empty)	Password for HTTP Basic authentication when connecting to the Apache Solr web application.
`feeder.solr.collection`	String	coremedia	The collection that should be used by the *Content Feeder*.
`feeder.solr.sendRetryDelay`	time in seconds	10	Delay in seconds between trying to send a batch.
`solr.partialUpdates`	true or false	true	Specifies whether partial updates are supported for updating document metadata in Solr. This requires that all fields in the Solr index are configured as `stored="true"` except fields that are `<copyField/>` destinations, which must be configured as `stored="false"`. This is because partial updates are applied to the index document reconstructed from the existing stored field values. Note that configuration property `feeder.partialUpdate.aspects` may still restrict usage of partial updates to certain document aspects.

Attribute	Value	Default	Description
`solr.partialUpdatesSkipIndexCheck`	true or false	false	If `solr.partialUpdates` is true, the Solr index schema is analyzed whether fields are stored as required for partial updates. The Feeder will log a warning and not use partial update functionality if the index seems to not support it. You can set this property to true to skip the check.

General Feeder configuration properties

These properties are configured in file `application.properties` of the *Content Feeder*.

Login data

The following properties are used to define the login data for the *Content Server* and the administration page of the *Search Engine*.

Table 6.2. Properties for login

Attribute	Value	Default	Description
`feeder.management.user`	user name	feeder	The user name to be used in the HTTP authentication of the administration page of the *Content Feeder*. This is not an account from the user management of the *Content Server*.
`feeder.management.password`	password	feeder	The password to be used in the HTTP authentication of the administration page of the *Content Feeder*.
`repository.user`	user name	feeder	The user account the *Content Feeder* uses to read content.
`repository.password`	password	feeder	The password for the user account of the *Content Feeder*.

Partial update configuration

With this property you can configure the usage of partial updates, if supported by the connected Indexer - for example for Solr as configured with property `solr.partialUpdates`.

Attribute	Value	Default	Description
`feeder.partialUpdate.aspects`	comma-separated list of document aspects or *	multiSite	The aspects of index documents that can be updated with a partial update, provided that the connected Indexer supports partial updates (for example, `solr.partialUpdates=true` for Solr). Multiple values are separated by comma. Use the special value * to use partial updates for all aspects, if possible. An empty value means that partial updates are not used. See the API documentation of `FeedableisPartialUpdate`, `FeedableAspect` and `ContentFeedableAspect` in package `com.coremedia.cap.feeder` for more details.

Table 6.3. Partial update configuration

Batch configuration

With these properties you can configure the processing of batches.

Attribute	Value	Default	Description
`feeder.maxBatchSize`	number of documents	500	The maximum number of documents in a batch.
`feeder.maxBatchByteSize`	number of bytes	5242880 (5 MB)	The maximum batch size in byte.
`feeder.sendIdleDelay`	time in seconds	3	The time to wait between adding a document to a batch and sending that batch to the search engine if the *Content Feeder* is idle. If a document was changed and no further changes are made within `sendIdleDelay` seconds, the document will be sent after that time to the search engine. This setting leads to a low latency for changes to become visible in search as long as the system is not very busy.
`feeder.sendMaxDelay`	time in seconds	20	The maximum time to wait between adding a document to a batch and

Table 6.4. Properties for batch configuration

Attribute	Value	Default	Description
			sending that batch. This setting is typically larger than `sendIdleDelay` to allow batches to grow for better throughput.
`feeder.maxOpenBatches`	int	5	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 *Content Feeder* does not call the 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.
`feeder.maxProcessedBatches`	int	1	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 *Content 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.

What to feed

You can use the following properties to define which elements the *Content Feeder* should feed to the *Search Engine*.

Table 6.5. Properties to feed additional items

Attribute	Value	Default	Description
`feeder.indexDeleted`	`true` or `false`	`true`	`true` if documents in the trash should be indexed. If you do not need to find documents in the trash and want to keep your index smaller, you can change this to `false`.
`feeder.indexPath`	`true` or `false`	`false`	Indicate whether a document's folder path is indexed in field 'folder'. If set to `true` (not recommended), folder renames lead to refeeding of all documents below that folder. The alternative field 'folderpath' which contains the folder path as folder ids is the recommended way to refer to a folder path.
`feeder.indexReferrers`	`true` or `false`	`false`	`true` to reindex a document after its referrers have changed.
`feeder.indexNameInTextBody`	`true` or `false`	`true`	Configures whether the document name should be indexed in index field textbody. It can make sense to disable this if lots of document names contain unique identifiers (from third-party systems, for example) to avoid problems with too many unique terms in field textbody.
`feeder.indexGroups`	`true` or `false`	`true`	`true` to index the groups with potential read rights with the document in the index field `groups`. This set of groups is then used to narrow a user's search to the documents where he might have read rights to. This is an optimization to get smaller search results for some queries and content structures and to get more accurate search suggestion counts. The client has to check for read rights anyway. If set to `false`, then you should also configure the *Studio* application to not add a superfluous query condition for the indexed groups by setting its property `stu`

Attribute	Value	Default	Description
			`dio.rest.searchService.useGroupsFilterQuery` to `false`.
`feeder.updateGroups.immediately`	`true` or `false`	`false`	If `feeder.indexGroups` is `true`, configures whether the field `groups` is updated immediately after a change of a folder's right rule. It is recommended to keep this set to `false` and let the *Content Feeder* update the index field `groups` in the background with lower priority than updates for editorial changes. It is quite expensive to set this to `true` because all documents below the folder will be reindexed.

Document types to feed

You can restrict the indexed documents by their type using the `includes` and `excludes` properties.

Table 6.6. Properties to specify document types.

Attribute	Value	Default	Description
`feeder.content.type.includes`	document type name	Document_	The name of the abstract or concrete document type whose documents should be indexed. Regular expressions are not allowed.
`feeder.content.type.excludes`	document type name	Preferences, EditorPreferences, Dictionary, Query	The name of the abstract or concrete document type whose documents should **not** be indexed. Regular expressions are not allowed.

Properties to feed

The default configuration feeds all properties for all specified document types. For configuration of indexed properties by their name, see the section for XML configuration below.

Property types to feed

You can only select a document property from a document type if its property type is specified with the following rules.

Table 6.7. Include property types

Property	Value	Default	Description
`feeder.content.propertyType.string`	`true` or `false`	`true`	Set this property to `false` in order to exclude `String` properties from indexing.
`feeder.content.propertyType.integer`	`true` or `false`	`false`	Set this property to `true` in order to include `Integer` properties when indexing.
`feeder.content.propertyType.date`	`true` or `false`	`false`	Set this property to `true` in order to include `Date` properties when indexing.
`feeder.content.propertyType.linkList`	`true` or `false`	`false`	Set this property to `true` in order to include `LinkList` properties when indexing.
`feeder.content.propertyType.struct`	`true` or `false`	`false`	Set this property to `true` in order to include `Struct` properties when indexing.
`feeder.content.propertyType.xmlGrammars`	List of included grammar names separated by comma	`core media-richtext-1.0`	You can define which XML properties should be indexed by specifying their grammar. **Example** `feeder.content.propertyType.xmlGrammars=core media-richtext-1.0`
`feeder.content.propertyType.blobMimeType.includes`	List of included MIME types separated by comma	See file	You can define which blob properties are indexed, depending on the MIME type. **Example** `feeder.content.propertyType.blobMimeType.includes=text/*` All blobs of MIME type `text/*` are indexed.
`feeder.content.propertyType.blobMimeType.excludes`	List of excluded MIME types separated by comma	(empty)	Exclude some blobs from indexing depending on the MIME type. If you've included a primary MIME type such as `text/*` or even the catch all type `*/*`, you can exclude some concrete types with this property. **Example**

Property	Value	Default	Description
			`feeder.content.propertyType.blobMimeType.excludes=text/plain` Blobs of MIME type `text/plain` will not be indexed.
`feeder.content.propertyType.blobMaxSize`	size in bytes	5242880 (5 MB)	Configure the maximum size of indexed blob properties. Larger values will be skipped. This configuration can be overridden in a Spring XML configuration file where you can configure the maximum size per MIME type by customizing the bean `feederContentBlobMaxSizePerMimeType`. See XML configuration for an example.

Tika configuration

You can customize text extraction with Apache Tika using the following properties:

Table 6.8. Tika configuration

Property	Value	Default	Description
`feeder.tika.config`	location of Apache Tika Config XML	(empty)	The location of an optional custom Apache Tika Config XML file with custom Tika parsers. The value is a Spring Resource location, for example a value such as `file:/path/tika-config.xml` can be used to reference a local file. Use an empty value for the default configuration.
`feeder.tika.appendMetadata`	comma-separated list of metadata identifiers	(empty)	Comma-separated list of metadata identifiers extracted from blob properties by Apache Tika that are appended to the extracted body text. See Section 4.2.3, “Advanced Configuration” [47]
`feeder.tika.copyMetadata`	comma-separated list of entries for the format <metadata identifi-	(empty)	Comma-separated list of metadata identifiers extracted from blob properties by Apache Tika and index field names to copy the metadata to. See Section 4.2.3, “Advanced Configuration” [47]

Property	Value	Default	Description
	er>=<index field name>		
feeder.tika.timeout.milliseconds	milliseconds	120000 (2 minutes)	Set the maximum time after which text extraction from binary data with Apache Tika fails. If extraction fails, the binary data will be skipped for the index document. Lower values will avoid that the Feeder is blocked for a long time in text extraction.
feeder.tika.warn.milliseconds	milliseconds	15000 (15 seconds)	Set the time after which a warning is logged when text extraction from binary data with Apache Tika takes some time.

Configuration of ImageDimensionFeedablePopulator

The following properties configure the ImageDimensionFeedablePopulator bean.

Table 6.9. Properties to configure ImageDimensionFeedablePopulator.

Attribute	Value	Default	Description
feeder.populator.imageDimension.docType	document type name	none (required)	The document type of the content to be indexed, including subtypes.
feeder.populator.imageDimension.widthPropertyName	document property name	none	The property name of the content which holds the width value. If not set, feeder.populator.imageDimension.dataPropertyName must be set.
feeder.populator.imageDimension.heightPropertyName	document property name	none	The property name of the content which holds the height value. If not set, feeder.populator.imageDimension.dataPropertyName must be set.
feeder.populator.imageDimension.dataPropertyName	document property name	none	The name of the blob property which holds the image data. The value of this object must be of type com.coremedia.cap.common.Blob. If not set, feeder.populator.imageD

Attribute	Value	Default	Description
			`imension.widthProperty Name` and `feeder.populator.imageD imension.heightProperty Name` must be set.
`feeder.populator.imageDimension.largeWidth`	positive number	none (required)	Lower bound width of large images.
`feeder.populator.imageDimension.largeHeight`	positive number	none (required)	Lower bound height of large images.
`feeder.populator.imageDimension.mediumWidth`	positive number	none (required)	Lower bound width of medium images.
`feeder.populator.imageDimension.mediumHeight`	positive number	none (required)	Lower bound height of medium images.

Error behavior

You can use the following properties to customize the *Content Feeder* behavior in case of errors.

Attribute	Value	Default	Description
`feeder.retrySendIdleDelay`	time in seconds	60	The time to wait before retrying to send documents to the search engine after failures to do so. This delay is used if the *Content Feeder* is idle.
`feeder.retrySendMaxDelay`	time in seconds	600	The maximum time to wait before retrying to send documents to the search engine after failures.
`feeder.retryConnectToIndexDelay.seconds`	time in seconds	10	The time to wait between retries to connect to the search engine on startup.

Table 6.10. Properties for Content Feeder configuration

Attribute	Value	Default	Description
`feeder.executorRetryDelay`	time in milliseconds	60000	The delay to wait before the *Content Feeder* retries to access the source data after failures.
`feeder.solr.connection.timeout`	time in milliseconds	0	The connection timeout set on the SolrJ `SolrServer`. It determines how long the client waits to establish a connection without any response from the server. The default value of 0 means it will wait forever.
`feeder.solr.socket.timeout`	time in milliseconds	600000 (10 minutes)	The socket timeout set on the SolrJ `SolrServer`. 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 value of 0 means it will wait forever.

Configure Statistics

You can configure time intervals to show statistics on the *Content Feeder* admin page and in the content server log.

Attribute	Value	Default	Description
`statisticInterval`	time in milliseconds	3600000	Maximum time interval to show statistics on the administration page. With the default you can show overall statistics (since starting the *Content Feeder*) and statistics for the last n seconds, where n <= `statisticInterval`.
`statisticLogInterval`	time in milliseconds	600000	Interval to log statistic information of the *Content Feeder* in the log file of the *CoreMedia Content Server* (`coremedia.log`).

Table 6.11. Attributes for statistics time intervals

XML configuration

The Spring XML configuration file `application.xml` allows more advanced configuration and customization. This section just describes the possibility to configure indexed document properties by name.

Properties to feed

If you want to restrict the document fields, you can specify a map entry with included or excluded fields for some or all document 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 document type or its ancestors, all document 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.

Configure included properties

The following example configures a map from document type names (abstract or concrete) to indexed properties. The values of the map are comma-separated property names of the respective document type. Only the listed properties will be indexed. Document types not listed here will by default be indexed with all properties if not configured otherwise via excluded properties.

```
<customize:append id="feederContentPropertyIncludesCustomizer"
bean="feederContentPropertyIncludes">
  <map>
    <entry key="doctype1" value="prop1,prop2"/>
    <entry key="doctype2" value="prop3"/>
  </map>
</customize:append>
```

Configure excluded properties

The following example configures a map from document type names (abstract or concrete) to properties excluded from indexing. The values of the map are comma-separated property names of the respective document type. Only the properties not listed here will be indexed. Document types not listed here will by default be indexed with all properties if not configured otherwise via included properties.

```
<customize:append id="feederContentPropertyExcludesCustomizer"
bean="feederContentPropertyExcludes">
  <map>
    <entry key="doctype4" value="prop4,prop5"/>

    <!--
      exclude all properties of doctype5
      only meta-data gets indexed
     -->
    <entry key="doctype5" value="*"/>
  </map>
</customize:append>
```

6.2 Content Feeder JMX Managed Beans

The *Content Feeder* exports an additional managed bean named `SolrIndexer` (Section 6.5, “Solr Indexer JMX Managed Beans” [124]).

MBean Attributes

Table 6.12. JMX manageable attributes of the 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 documents are indexed immediately when they are sent to the search engine. Indexing time is then part of `IndexAverageBatchSendingTime`.
`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 documents in a batch.
`IndexAverageLagTime`	Read-only	The average delay in seconds of last indexed documents for 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`.

Attribute	Type	Description
		The set of feedables used to compute the delay can be restricted by introducing a `com.coremedia.common.util.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 to include this feedable into these statistics. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the index bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="index" customref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexContentDocuments`	Read-only	The number of last indexed documents 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 *Content Feeder*. The set of feedables used to compute the number of content documents can be restricted by introducing a `com.coremedia.common.util.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 to include this feedable into these statistics. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the feeder bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="index" customref="myPredicate" prop`

Attribute	Type	Description
		`erty="batchStatisticsFeedablePredicate" />`
`IndexMaxLagTime`	Read-only	The maximum delay in seconds of last indexed documents for 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 feedables used to compute the delay can be restricted by introducing a `com.coremedia.common.util.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 to include this feedable into these statistics. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the index bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="index" customref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexMinLagTime`	Read-only	The minimum delay in seconds of last indexed documents for 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`.

Attribute	Type	Description
		The set of feedables used to compute the delay can be restricted by introducing a `com.coremedia.common.util.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 to include this feedable into these statistics. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the index bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="index" customref="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 feedables used to compute the latest index time can be restricted by introducing a `com.coremedia.commmon.util.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 to include this feedable into these statistics.

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" customref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`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 document 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

Attribute	Type	Description
		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. This attribute contains the number of persisted documents 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 documents per second as long as the *Content Feeder* is still initializing.
`CurrentPendingDocuments`	Read-only	The number of documents in the currently fed folder to reindex 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 feeder may already have started indexing documents from the first folder in the result.
`RetryConnectToIndexDelay`	Read-only	The time in seconds between retries to connect to the *Search Engine* on startup

Attribute	Type	Description
`State`	Read-only	State of the *Content Feeder* (running or stopped).
`Uptime`	Read-only	Uptime of the *Content Feeder* in milliseconds.

MBean Operations

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 documents will be reindexed when the *Content Feeder* is restarted.
getPendingDocuments		Returns the total number of documents to reindex after rights rule changes, that is, the number of documents in the folders with ids returned by the JMX attribute `PendingFolders` above. This might be an expensive operation.

Table 6.13. JMX operations of the Content Feeder

6.3 CAE Feeder Configuration

In this reference chapter you will find a description of the *CAE Feeder* configuration properties.

Table 6.14. Configuration of general properties independent from the type of the search engine

Property	Value	Default	Description
`repository.user`	user name	none	The name of the user to connect to the *CoreMedia Content Server*.
`repository.password`	password	none	The password of the user to connect to the *CoreMedia Content Server*.
`repository.domain`	domain	none	The domain of the user to connect to the *CoreMedia Content Server*. Empty String for a built-in user.
`repository.url`	URL	none	The URL to the IOR of the *CoreMedia Content Server*.
`jdbc.driver`	driver class	none	The class of the database driver. For example: `oracle.jdbc.driver.OracleDriver`
`jdbc.url`	URL	none	The URL to connect to the database.
`jdbc.user`	user name	none	The name of the user to connect to the database.
`jdbc.password`	password	none	The password of the user to connect to the database.
`feeder.contentSelector.basePath`	String	`/Sites`	A comma-separated list of base folders for which content beans are indexed.
`feeder.contentSelector.contentTypes`	String	`Document_`	A comma-separated list of content types for which content beans are indexed.
`feeder.contentSelector.includeSubTypes`	Boolean	`true`	Specifies whether the sub types of the content types configured with property `feeder.contentSelector.contentTypes` are selected as well.
`feeder.executorQueueCapacity`	int	2000	Capacity of the *CAE Feeder*'s executor queue, which is internally used to transfer evaluated values
`feeder.executorRetryDelay`	milliseconds	60000	The delay in milliseconds to wait before the *CAE Feeder* retries to access the source data after failures to do so.

Property	Value	Default	Description
`feeder.maxBatchBytes`	bytes	20971520 (20 MB)	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. Note, that byte computation is a rough estimate only.
`feeder.maxBatchSize`	int	500	The maximum number of entries in a batch. If the maximum number is reached, the *CAE Feeder* sends the batch to the *Search Engine*.
`feeder.maxOpenBatches`	int	5	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 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.
`feeder.maxProcessedBatches`	int	1	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.
`feeder.retrySendIdleDelay`	milliseconds	60000	The *CAE Feeder* sends a batch which only contains retried entries and is not full with regard to the `feeder.maxBatchSize` and `feeder.maxBatchBytes` properties

Property	Value	Default	Description
			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 property `feeder.sendIdleDelay` is used instead.
`feeder.retrySendMaxDelay`	milliseconds	600000	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 an error occurs while contacting the *Search Engine*. If the batch contains entries which are not retried, the value of property `feeder.sendMaxDelay` is used instead.
`feeder.beanPropertyMaxBytes`	number of bytes	-1	The maximum size in bytes for the value of a bean property or -1 for no limitation. Larger values are ignored and will not be sent to the Search Engine.
`feeder.beanMapping.mimeType.includes`	comma-separated list of included MIME types	*/*	List of included MIME types for blob properties configured for indexing at the BeanMappingFeedablePopulator. For details, see the API documentation of method `setMimeTypeIncludes` of com.coremedia.cap.feeder.bean.BeanMappingFeedablePopulator **Example** `feeder.beanMapping.mimeType.includes=text/*` Only indexes blobs of MIME type `text/*`.
`feeder.beanMapping.mimeType.excludes`	comma-separated list of excluded MIME types		List of excluded MIME types for blob properties configured for indexing at the BeanMappingFeedablePopulator. For details, see the API documentation of method `setMimeTypeEx`

Property	Value	Default	Description
			`cludes` of com.coremedia.cap.feeder.bean.BeanMappingFeedablePopulator **Example** `feeder.beanMapping.mimeType.excludes=text/xml` Indexes all blobs except blobs of MIME type `text/xml`.
`feeder.sendIdleDelay`	milliseconds	10000	The *CAE Feeder* sends a batch which is not full with regard to the `feeder.maxBatchSize` and `feeder.maxBatchBytes` properties after the *CAE Feeder* was idle for the configured time in milliseconds.
`feeder.sendMaxDelay`	milliseconds	120000	The maximum time in milliseconds after which the *CAE Feeder* sends a batch which is not full with regard to the `feeder.maxBatchSize` and `feeder.maxBatchBytes` properties. The time may be exceeded if an error occurs while contacting the *Search Engine* or if the *CAE Feeder* is under high load.
`feeder.tika.config`	location of Apache Tika Config XML	(empty)	The location of an optional custom Apache Tika Config XML file with custom Tika parsers. The value is a Spring Resource location, for example a value such as `file:/path/tika-config.xml` can be used to reference a local file. Use an empty value for the default configuration.
`feeder.tika.appendMetadata`	comma-separated list of metadata identifiers	(empty)	Comma-separated list of metadata identifiers extracted from blob properties by Apache Tika that are appended to the extracted body text. See Section 5.2, “Configuring the CAE Feeder” [62]
`feeder.tika.copyMetadata`	comma-separated list of entries for the format	(empty)	Comma-separated list of metadata identifiers extracted from blob properties by Apache Tika and index field names to copy the metadata to. See

Property	Value	Default	Description
	<metadata identifier>=<index field name>		Section 5.2, “Configuring the CAE Feeder” [62]
`feeder.tika.timeout.milliseconds`	milliseconds	`120000` (2 minutes)	Set the maximum time after which text extraction from binary data with Apache Tika fails. If extraction fails, the binary data will be skipped for the index document. Lower values will avoid that the Feeder is blocked for a long time in text extraction.
`feeder.tika.warn.milliseconds`	milliseconds	`15000` (15 seconds)	Set the time after which a warning is logged when text extraction from binary data with Apache Tika takes some time.
`proactiveengine.senders.evaluators`	number of threads	50	Number of evaluator threads in the *CAE Feeder*. The number of threads influences performance not only because evaluations can execute concurrently but also because higher values increase the probability that the *CAE Feeder* writes the state of multiple evaluations to the database in one database transaction.
`proactiveengine.senders.delay`	milliseconds	0	Minimum delay in milliseconds between notifications of the Feeder by the internal *Proactive Engine* sub component. Higher values lead to reduced throughput.
`proactiveengine.senders.idledelay`	milliseconds	10000	Delay in milliseconds between notifications of the Feeder by the internal *Proactive Engine* sub component if the application is idle. Smaller values can be configured to reduce the latency of the *CAE Feeder* but may lead to increased load on the database.
`dependencyStore.maxTransactionWeight`	maximum number of changed keys per database transaction	2500	The maximum weight of a database transaction to change stored dependencies. The weight is interpreted as the number of changed keys, that is, a transaction with one deleted key has weight 1. Multiple transactions will be used to process an event that causes the invalidation of more keys.

The following properties are only used for a *CoreMedia Search Engine* based on Apache Solr:

Table 6.15. Configuration properties for Apache Solr

Property	Value	Default	Description
`feeder.solr.url`	URL	http://localhost:8082/solr/coremedia	The URL where the *CAE Feeder* can reach the *Search Engine*. The URL points to the Apache Solr core for the *CAE Feeder*.
`feeder.solr.collection`	collection name	coremedia	The collection that should be used by the *CAE Feeder*.
`feeder.solr.username`	user name or empty	(empty)	User name for HTTP Basic authentication when connecting to the Apache Solr web application. Leave empty for no authentication.
`feeder.solr.password`	user name or empty	(empty)	Password for HTTP Basic authentication when connecting to the Apache Solr web application.
`feeder.solr.sendRetryDelay`	milliseconds	30000	The delay in milliseconds to wait before sending a batch to the *Search Engine* again after sending failed with an error in the *Search Engine*.
`feeder.solr.connection.timeout`	time in milliseconds	0	The connection timeout set on the SolrJ `SolrServer`. It determines how long the client waits to establish a connection without any response from the server. The default value of 0 means it will wait forever.
`feeder.solr.socket.timeout`	time in milliseconds	600000 (10 minutes)	The socket timeout set on the SolrJ `SolrServer`. 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 value of 0 means it will wait forever.

6.4 CAE Feeder JMX Managed Beans

The *CAE Feeder* exports multiple JMX MBeans. The following overview describes attributes of the MBeans `Feeder` and `ProactiveEngine`. The MBean `SolrIndexer` is described in Section 6.5, “Solr Indexer JMX Managed Beans” [124]. The *CAE Feeder* exports more MBeans and attributes, which aren't documented in detail here.

Feeder MBean

Table 6.16. Attributes of the 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. 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.

Attribute	Type	Unit	Description
			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`.
`BatchAverage-PersistingTime`	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 `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.

Attribute	Type	Unit	Description
`BatchEntries-PerSecond`	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 documents. Note that this value decreases if the Feeder is idle.
`BatchEntry-Count`	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 documents.
`BatchStatistic-sInter-valSeconds`	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.
`BatchStatistic-sMaxInter-valSeconds`	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.
`BatchStatistic-sLogInter-valSeconds`	read/write	seconds	The time interval in seconds in which the Feeder writes statistics to its log file (log level INFO).
`CallbackQueueS-ize`	read-only	callback objects	The number of pending com.coremedia.cap.feeder.FeederCallback objects in the internal callback queue.
`DeferredEntry-Count`	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 document is currently being sent to

Attribute	Type	Unit	Description
			the *Search Engine* or was not yet persisted by the *Search Engine*. Batch entries are basically creations, updates or removals of 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 last indexed documents 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. 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 feedables used to compute the delay can be restricted by introducing a `com.coremedia.common.util.Predicate`. This predicate can be injected into Spring bean `feeder`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides to include this feedable into these statistics. To inject a custom predicate use the bean customizer and replace the

Attribute	Type	Unit	Description
			`BatchStatisticsFeedablePredicate` of the `feeder` bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="feeder" customref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexContentDocuments`	read-only	documents	The number of last indexed documents 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. The set of feedables used to compute the number of content documents can be restricted by introducing a `com.coremedia.common.util.Predicate`. This predicate can be injected into Spring bean `feeder`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides to include this feedable into these statistics. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the `feeder` bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="feeder" customref="myPredicate" property="batchStatisticsFeedablePredicate" />`
`IndexMaxLagTime`	read-only	seconds	The longest delay in seconds of last indexed documents for the last <*n*> seconds, where <*n*> is the value of the attribute `BatchStatisticsIn`

Attribute	Type	Unit	Description
			`tervalSeconds`. 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 feedables used to compute the delay can be restricted by introducing a `com.coremedia.common.util.Predicate`. This predicate can be injected into Spring bean `feeder`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides to include this feedable into these statistics. 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 shortest delay in seconds of last indexed documents 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. 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 feedables used to compute the delay can be restricted by introducing a `com.coremedia.common.util.Predicate`. This pre-

Attribute	Type	Unit	Description
			dicate can be injected into Spring bean `feeder`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides to include this feedable into these statistics. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the `feeder` bean: `<customize:replace id="batchStatisticsFeedablePredicateCustomizer" bean="feeder" customref="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 feedables used to compute the latest index time can be restricted by introducing a `com.coremedia.common.util.Predicate`. This predicate can be injected into Spring bean `feeder`. The `include` method accepts an object of type `com.coremedia.cap.feeder.Feedable`. The custom implementation decides to include this feedable into these statistics. To inject a custom predicate use the bean customizer and replace the `BatchStatisticsFeedablePredicate` of the `feeder` bean: `<customize:replace id="batchStatisticsFeedablePredicate" bean="feeder" customref="myPredicate" prop`

Attribute	Type	Unit	Description
			`erty="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.maxBatchSize`.
`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.maxBatchBytes`. 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 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.maxOpenBatches`.
`MaxProcessed-Batches`	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

Attribute	Type	Unit	Description
			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.maxOpenBatches`.
`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 interface 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.retrySendIdleDelay`.
`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

Attribute	Type	Unit	Description
			`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 `Send MaxDelay` is used instead. It defaults to the configured property `feeder.retrySendMaxDelay`.
`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*. It defaults to the configured property `feeder.sendIdleDelay`.
`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 `MaxOpen Batches` or `MaxProcessed Batches` are reached or an error occurs while contacting the *Search Engine*. It defaults to the configured property `feeder.sendMaxDelay`.
`StartTime`	read-only	date and time	The time when the *CAE Feeder* was started.

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

Table 6.17. Attributes of the ProactiveEngine MBean

Attribute	Type	Unit	Description
			(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" [73]. 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 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`.

6.5 Solr Indexer JMX Managed Beans

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

SolrIndexer MBean

Table 6.18. Properties of SolrIndexer MBean

Attribute	Type	Unit	Description
`Collection`	read-only		The name of an existing collection of the *Search Engine* to use as configured in property `feeder.solr.collection`.
`Url`	read-only		The URL of the Apache Solr web application for feeding as configured in property `feeder.solr.url`.
`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.sendRetryDelay`.
`NoRetryDocumentIdsCsv`	read/write	comma-separated string values	Document IDs for which indexing must not be retried after errors. The SolrIndexer automatically triggers a retry when a 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 administrator can add the document ID to the value of this JMX attribute to make the SolrIndexer skip errors for the document. IDs must conform to the value of the Solr `id` field, for example `core media:///cap/content/42` for a document indexed with the *Content Feeder* and `contentbean:42` for a content bean indexed with the *CAE Feeder*. The value is empty by default after starting the Feeder. It is not persisted.

6.6 Supported Languages in Solr Language Detection

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

Table 6.19. Supported Languages

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
he	Hebrew
hi	Hindi
hr	Croatian
hu	Hungarian
id	Indonesian
it	Italian
ja	Japanese
kn	Kannada
ko	Korean

Language Code	Language
`lt`	Lithuanian
`lv`	Latvian
`mk`	Macedonian
`ml`	Malayalam
`mr`	Marathi
`ne`	Nepali
`nl`	Dutch
`no`	Norwegian
`pa`	Punjabi
`pl`	Polish
`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
`zh-tw`	Traditional Chinese

Glossary

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

→ *CoreMedia Adaptive Personalization*

Content Feeder

The *Content Feeder* is a separate web application that feeds content items of the CoreMedia repository into the *CoreMedia Search Engine*. Editors can use the *Search Engine* to make a full text search for these fed items.

Content item

In *CoreMedia CMS*, content is stored as self-defined content items. Content items are specified by their properties or fields. Typical content properties are, for example, title, author, image and text content.

Content Management Environment

The *Content Management Environment* is the environment for editors. The content is not visible to the end user. It may consist of the following modules:

→ *CoreMedia Content Management Server*

→ *CoreMedia Workflow Server*

→ *CoreMedia Importer*

→ *CoreMedia Site Manager*

→ *CoreMedia Studio*

→ *CoreMedia Search Engine*

→ *CoreMedia Adaptive Personalization*

→ *CoreMedia CMS for SAP Netweaver ® Portal*

→ *CoreMedia Preview CAE*

Content Management Server

Server on which the content is edited. Edited content is published to the Master Live Server.

Content Repository

CoreMedia CMS manages content in the Content Repository. Using the Content Server or the UAPI you can access this content. Physically, the content is stored in a relational database.

Content Server

Content Server is the umbrella term for all servers that directly access the CoreMedia repository:

Content Servers are web applications running in a servlet container.

→ *Content Management Server*

→ *Master Live Server*

→ *Replication Live Server*

Content type	A content type describes the properties of a certain type of content. Such properties are for example title, text content, author, ...
Contributions	Contributions are tools or extensions that can be used to improve the work with *CoreMedia CMS*. They are written by CoreMedia developers - be it clients, partners or CoreMedia employees. CoreMedia contributions are hosted on Github at https://github.com/coremedia-contributions.
Controm Room	*Controm Room* is a *Studio* plugin, which enables users to manage projects, work with workflows, and collaborate by sharing content with other *Studio* users.
CORBA (Common Object Request Broker Architecture)	The term *CORBA* refers to a language- and platform-independent distributed object standard which enables interoperation between heterogenous applications over a network. It was created and is currently controlled by the Object Management Group (OMG), a standards consortium for distributed object-oriented systems. CORBA programs communicate using the standard IIOP protocol.
CoreMedia Studio	*CoreMedia Studio* is the working environment for business specialists. Its functionality covers all of the stages in a web-based editing process, from content creation and management to preview, test and publication. As a modern web application, *CoreMedia Studio* is based on the latest standards like Ajax and is therefore as easy to use as a normal desktop application.
Dead Link	A link, whose target does not exists.
DTD	A Document Type Definition is a formal context-free grammar for describing the structure of XML entities. The particular DTD of a given Entity can be deduced by looking at the document prolog: `<!DOCTYPE coremedia SYSTEM "http://www.coremedia.com/dtd/coremedia.dtd"` There're two ways to indicate the DTD: Either by Public or by System Identifier. The System Identifier is just that: a URL to the DTD. The Public Identifier is an SGML Legacy Concept.
Elastic Social	*CoreMedia Elastic Social* is a component of *CoreMedia CMS* that lets users engage with your website. It supports features like comments, rating, likings on your website. *Elastic Social* is integrated into *CoreMedia Studio* so editors can moderate user generated content from their common workplace. *Elastic Social* bases on NoSQL technology and offers nearly unlimited scalability.

EXML	EXML is an XML dialect supporting the declarative development of complex Ext JS components. EXML is Jangaroo's equivalent to Adobe Flex MXML and compiles down to Actions Script.
Folder	A folder is a resource in the CoreMedia system which can contain other resources. Conceptually, a folder corresponds to a directory in a file system.
Home Page	The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.
IETF BCP 47	Document series of *Best current practice* (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese, or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.
Importer	Component of the CoreMedia system for importing external content of varying format.
IOR (Interoperable Object Reference)	A CORBA term, *Interoperable Object Reference* refers to the name with which a CORBA object can be referenced.
Jangaroo	*Jangaroo* is a JavaScript framework developed by CoreMedia that supports ActionScript as an input language which is compiled down to JavaScript. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net.
Java Management Extensions (JMX)	The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
JSP	JSP (Java Server Pages) is a template technology based on Java for generating dynamic HTML pages. It consists of HTML code fragments in which Java code can be embedded.
Locale	Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server	The *Master Live Server* is the heart of the *Content Delivery Environment*. It receives the published content from the *Content Management Server* and makes it available to the *CAE*. If you are using the *CoreMedia Multi-Site Management Extension* you may use multiple *Master Live Server* in a CoreMedia system.

Master Site	A master site is a site other localized sites are derived from. A localized site might itself take the role of a master site for other derived sites.
MIME	With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multimedia emails and of web documents is standardised.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	A project is a collection of content items in CoreMedia CMS created by a specific user. A project can be managed as a unit, published or put in a workflow, for example.
Property	In relation to CoreMedia, properties have two different meanings: In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content items depends on the content type. In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Replication Live Server	The aim of the *Replication Live Server* is to distribute load on different servers and to improve the robustness of the *Content Delivery Environment*. The *Replication Live Server* is a complete Content Server installation. Its content is an replicated image of the content of a *Master Live Server*. The *Replication Live Server* updates its database due to change events from the *Master Live Server*. You can connect an arbitrary number of *Replication Live Servers* to the *Master Live Server*.
Resource	A folder or a content item in the CoreMedia system.
ResourceURI	A ResourceUri uniquely identifies a page which has been or will be created by the *Active Delivery Server*. The ResourceUri consists of five components: Resource ID, Template ID, Version number, Property names and a number of key/value pairs as additional parameters.
Responsive Design	Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.
Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site. A typical site also has a master site it is derived from.

Site Folder	All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.
Site Indicator	A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.
Site Manager	Swing component of CoreMedia for editing content items, managing users and workflows.
Site Manager Group	Members of a site manager group are typically responsible for one localized site. Responsible means that they take care of the contents of that site and that they accept translation tasks for that site.
Template	In CoreMedia, JSPs used for displaying content are known as Templates. OR In *Blueprint* a template is a predeveloped content structure for pages. Defined by typically an administrative user a content editor can use this template to quickly create a complete new page including, for example, navigation, pre-defined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes web application	The *User Changes* web application is a *Content Repository* listener, which collects all content, modified by *Studio* users. This content can then be managed in the *Control Room*, as a part of projects and workflows.
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.
Weak Links	In general *CoreMedia CMS* always guarantees link consistency. But links can be declared with the *weak* attribute, so that they are not checked during publication or withdrawal. Caution! Weak links may cause dead links in the live environment.
WebDAV	WebDAV stands for World Wide Web Distributed Authoring and Versioning Protocol. It is an extension of the Hypertext Transfer Protocol (HTTP), which offers a standardised method for the distributed work on different data via the internet. This adds the possibility to the CoreMedia system to easily access CoreMedia resources via external programs. A WebDAV enabled application like Microsoft Word is thus able to open Word documents stored in the CoreMedia system. For further information, see http://www.webdav.org.

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

Index