Elastic Social Manual

CoreMedia Content Cloud – v13

List of Figures

List of Tables

List of Examples

1. Preface

This manual describes the usage of *CoreMedia Elastic Social*.

- Section 2.1, "Architectural Overview" [14] gives an architectural overview of *CoreMedia Elastic Social*.
- Chapter 3, *Administration and Operation* [18] gives an overview over the administration and operation of *CoreMedia Elastic Social*.
- Chapter 4, *Development* [35] describes how to develop with *CoreMedia Elastic Social*.

Functionality only for Self-Managed Installation

Elastic Social is only available for a self-managed installation of CoreMedia Content Cloud. It is not availabe for the hosted CoreMedia Content Cloud Service solution.

ⓘ

1.1 Audience

This manual is intended for developers who integrate *CoreMedia Elastic Social* into their projects.

1.2 Typographic Conventions

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

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

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

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

Table 1.2. Pictographs

1.3 CoreMedia Services

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

> **NOTE**
>
> **CoreMedia User Orientation for CoreMedia Developers and Partners**
>
> Find the latest overview of all CoreMedia services and further references at:
>
> http://documentation.coremedia.com/new-user-orientation

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

1.3.1 Registration

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

1.3.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

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

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

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

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

Maven artifacts

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

https://repository.coremedia.com

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

npm packages

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

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

The .npmrc is configured to be able to utilize the registry (see Section 3.1, "Prerequisites" in *Blueprint Developer Manual*).

License files

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

1.3.3 Documentation

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

https://documentation.coremedia.com

The manuals have the following content and use cases:

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

Manual	Audience	Content
		type model which lies at the heart of a CoreMedia system, about user and rights management, database configuration, and more.
Deployment Manual	Developers, architects, administrators	This manual describes the concepts and usage of the CoreMedia deployment artifacts. That is the deployment archive and the Docker setup. You will also find an overview of the properties required to configure the deployed system.
Elastic Social Manual	Developers, architects, administrators	This manual describes the concepts and administration of the *Elastic Social* module and how you can integrate it into your websites.
Frontend Developer Manual	Frontend Developers	This manual describes the concepts and usage of the Frontend Workspace. You will learn about the structure of this workspace, the CoreMedia themes and bricks concept, the CoreMedia Freemarker facade API, how to develop your own themes and how to upload your themes to the CoreMedia system.
Headless Server Developer Manual	Frontend Developers, administrators	This manual describes the concepts and usage of the *Headless Server*. You will learn how to deploy the Headless Server and how to use its endpoints for your sites.
Multi-Site Manual	Developers, Multi-Site Administrators, Editors	This manual describes different otions to desgin your site hierarchy with several languages. It also gives guidance to avoid common pitfalls during your work with the multi-site feature.
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.

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

Table 1.3. CoreMedia manuals

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

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

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

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

https://www.coremedia.com/training

Contact the training department at the following email address:

Email: training@coremedia.com

1.3.5 CoreMedia Support

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

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

https://support.coremedia.com/

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

Email: support@coremedia.com

Create a support request

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

Support request

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

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

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

To put it in a nutshell, CoreMedia needs: *Support checklist*

1. a person in charge (ideally, the CoreMedia system administrator)

2. extensive and sufficient system specifications

3. detailed error description

4. log files for the affected component(s)

5. if required, system files

An essential feature for the CoreMedia system administration is the output log   *Log files*
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 break-
down, you should be logging at debug level. See Section 4.7, "Logging" in *Opera-
tions Basics* for details.

Which Log File?

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

Where do I Find the Log Files?

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

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

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

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

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

1.4 Changelog

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

Section	Version	Description

Table 1.4. Changes

2. Overview

This chapter gives an overview about the architecture of *CoreMedia Elastic Social* and the data privacy aspects that have to be considered.

2.1 Architectural Overview

Elastic Social combines four major components:

- Elastic Core is the foundation of Elastic Social and provides several services for building horizontally scalable applications
  - `ModelService`, for schema-free persistence
  - `StagingService`, staging of changes on models
  - `CounterService`, `AverageCounterService`, atomic counters
  - `HistogramCounterService`, counters with a histogram
  - `BlobService`, storage of large binary objects
  - `TaskQueueService`, asynchronous parallel execution of background tasks
  - `SearchService`, full-text search
  - `UserService`, for users
  - `TemplateService`, for template rendering
  - `TenantService`, for tenant management
- Elastic Social services for social use cases:
  - `CommunityUserService`, for community users
  - `CommentService`, for commenting
  - `ReviewService`, for reviews
  - `BlacklistService`, for blacklists
  - `RatingService`, for rating
  - `LikeService`, for likes
  - `RegistrationService`, for user registration
  - `MailService`, for sending mails
  - `MailTemplateService`, for creating mails from localized templates
- A Plugin for CoreMedia Studio

  The plugin allows the premoderation and post-moderation of users, reviews and comments which can include pictures, processing complaints, managing users and searching for comments and using them for curated content.

- A reference implementation based on the development workspace that is showing the integration of social software use cases into *CoreMedia Blueprint*.

  The reference implementation shows registration, login, password loss, user self service, commenting, citing, reviews, premoderation and post-moderation of comments, reviews and users, ignoring users, handling of anonymous users, automatic rejection of comments, automatic blocking of users, display of top reviewed, most reviewed and most commented content.

Elastic Social and Elastic Core are supplied as a set of Java libraries that can easily be integrated into any Java application, see Section 4.7, "Integration" [55].

2.1.1 Logical Components

The rational behind *Elastic Core* is to provide services that allow the agile, cost-effective and riskless development of horizontally scalable, high available, elastic, cloud-based applications. The following diagram depicts the logical components that are required for this approach:
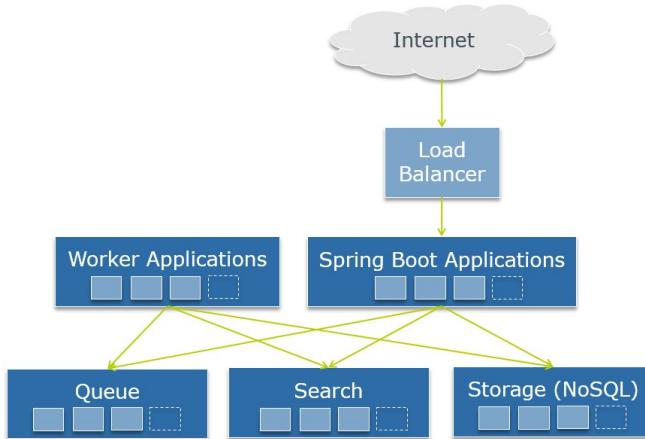
Figure 2.1. Logical components of Elastic Social

2.1.2 Software Stack

Reference implementation, *Elastic Social* and *Elastic Core* can be seen as a software stack that offers APIs for flexibility and extensibility on each level. The following image depicts how a sample application uses the Elastic Social, Elastic Core and Unified API to enrich a website with social use cases. Everything is running within a *Content Application Engine* as a container:
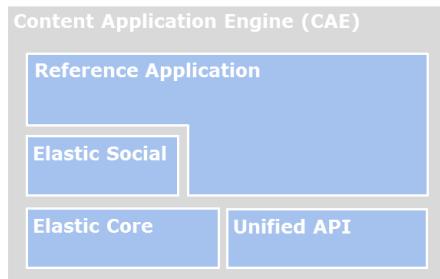
Figure 2.2. Software Stack of Elastic Social

2.2 Data Privacy Considerations

CoreMedia delivers building blocks as part of the *CoreMedia Elastic Social* add-on module and the respective Blueprint Extensions that enable you to build communities and social features. CoreMedia provides tooling to facilitate compliance with legal privacy regulations including requests for information, change and deletion of personal data – however establishing compliance remains the responsibility of the customer implementing and operating the product. Depending on whether or where technically you choose to persist personal data of your end users, you may need to seek and document consent from your users and/or establish other legal grounds for use of personal data based on your applicable legal regulations. Any recommendations provided by CoreMedia are not to be established as legal advice or consultation, please contact your legal counsel.

3. Administration and Operation

This chapter describes the administration and operation of *Elastic Social*.

3.1 Installation Guide

In this chapter you find help to set up components necessary to run *Elastic Social*. It is also possible and recommended to use suitable MongoDB installation packages in your project depending on your operating system. This chapter only helps you to quickly setup a development environment.

Install

- Install the supported versions of Java and Maven

- Download and extract the latest supported version of MongoDB:

    http://www.mongodb.org/downloads/

    For details how to set up MongoDB, consult the MongoDB Manuals.

- Download and extract the latest *CoreMedia Blueprint*

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

    See the Blueprint Developer Manual for further instructions on how to set up and use *CoreMedia Blueprint*.

- Enable the Elastic Social Extension: Section 4.2.1.2, "Enabling the Elastic Social Extension" in *Blueprint Developer Manual* .

3.2 Deployment

This section describes the deployment of *CoreMedia Elastic Social* within the context of a *CoreMedia CAE* application based on *CoreMedia CMS*.

3.2.1 Setup

The basic setup is the same as for a *CoreMedia CAE* application. Additionally, a MongoDB installation is required for deploying an Elastic Social enabled application. See the https://releases.coremedia.com/cmcc-13/artifacts/CMCC 13 – Supported Environments.pdf document for the supported versions.

Please refer to the MongoDB documentation to install and administrate MongoDB. CoreMedia highly recommends to use Replica Sets for automated failover and distribution of read load. In order to scale write load, CoreMedia suggests to use Sharding. While Replica Sets should be used in any deployment scenario, sharding is optional and can be enabled when load increases.
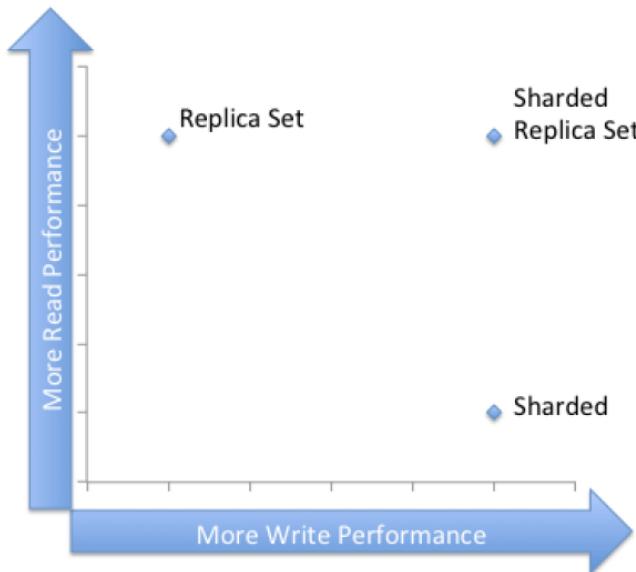
Figure 3.1. Use of sharding and replication sets

3.2.2 Single Data Center Deployment

The deployment of *CoreMedia Elastic Social* and *CoreMedia CMS* offers a lot of flexibility. The following diagram depicts a typical single data center deployment showing the well known *CoreMedia CMS* components and the *CoreMedia Elastic Social* extensions:
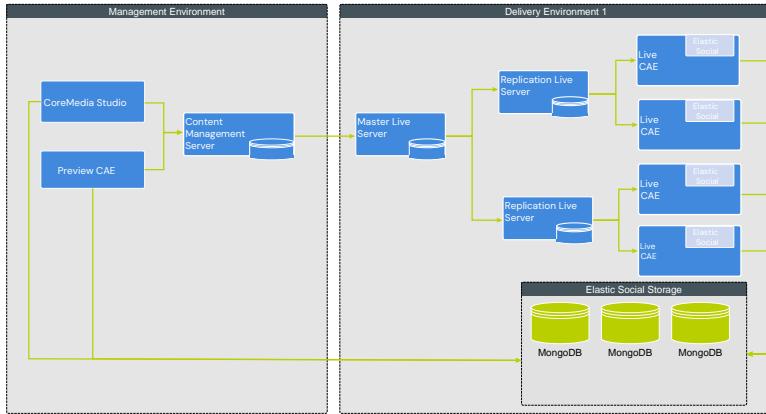
Figure 3.2. Single data center deployment

The deployment options for a single data center deployment range from small to large:

Small 'S'

The 'S' deployment abandons high availability for cost efficiency and runs MongoDB on a single node. When equipped with 8 GB of RAM for each node it may serve a working set of 100000 users and 100000 comments, likes or ratings.

Medium 'M'

The 'M' deployment consists of three nodes running MongoDB as one Replica Set. This setup offers high availability and hot failover with three MongoDB nodes and can survive the failure of one node if configured appropriately. When equipped with 16 GB of RAM for each node it may serve a working set of 1 million users and 1 million comments, likes or ratings.

Large 'L'

The 'L' deployment matches the 'M' deployment and uses vertical scaling and better I/O throughput to boost read and write performance. When equipped with 64 GB of RAM and fast HDDs or SSDs for each node it may serve a working set of 5 million users and 5 million comments, likes or ratings.

3.2.3 Multiple Data Center Deployment

A multiple data center deployment of *CoreMedia* with *Elastic Social* can either be set up with one MongoDB Replica Set or multiple sharded Replica Sets. In both setups, the Replica Sets need to be distributed over the data centers to ensure data integrity in case of datacenter failure.

For more information have a look at the MongoDB documentation **ht‐tps://docs.mongodb.com/manual/**.

Possible deployment options for a multiple data center deployment in extra large and XXL:

Extra Large 'XL'

The 'XL' deployment consists of six nodes running MongoDB configured as two sharded Replica Sets distributed over the data centers. This setup offers sharding, high availability and hot failover with six MongoDB nodes and can survive the failure of one data center if configured appropriately. When equipped with 256 GB of RAM for each node it may serve a working set of 10 million users and 30 million comments, likes or ratings.

Extra Extra Large 'XXL'

The 'XXL' deployment matches the 'XL' deployment and uses vertical scaling and better I/O throughput to boost read and write performance. Please contact CoreMedia for serious recommendations.

3.2.4 Cloud deployment

Due to technical limitations there is no dedicated Cloud deployment option yet. A Cloud deployment of CoreMedia CMS components and CoreMedia Elastic Social extensions is actually a multiple data center deployment where one or more data centers are based on Cloud infrastructure.

Please refer to the MongoDB on AWS Whitepaper to install and administrate MongoDB on AWS.

3.2.5 Performance

When sizing the deployment of an Elastic Social enabled application, you should take into account that adding user generated content to pages increases the page delivery time depending on the caching strategy. When using a HTTP proxy like Varnish that caches all pages for a fixed time (one minute, for instance) or when using a timed dependency CAE cache key any extra costs can be eliminated. Delivering user generated content directly from the database roughly doubles the amount of CAEs required. Using a mixed strategy for dynamically serving all requests with a session and statically caching everything else allows you to reduce the amount of extra CAEs required. With 10% dynamic requests, 20% more CAEs are required; with 20% dynamic requests, it's 40% and so on. However, the response time remains constant regardless of the number of users and the amount of the user generated content they create.

The statements above have been verified in a test deployment on Amazon EC2. EC2 was used to run the tests on a comparable and reproducible environment. The setup consisted (among other servers) of 3 m1.xlarge instances running the CoreMedia CAE Live web application in Apache Tomcat 7, one load balancer and 3 m1.xlarge instances running MongoDB in a Replica Set. Up to 10 million users and 10 million comments have been imported into the Elastic Social database. The load balancer has been configured to distribute load evenly between the CAE instances. An article page has been used to measure response time and throughput. Two scenarios have been tested, one with user feedback disabled and one with 10 comments on the article page.

Adding user generated content to pages increases the page delivery time depending on the caching strategy:

- static: a HTTP proxy that caches all pages for one minute or a timed dependency CAE cache key eliminates any extra costs

- dynamic: delivering directly from the store roughly doubles the amount of CAEs required

- mixed: use the dynamic strategy for all requests with a session and the static strategy for everything else allows you to reduce the amount of extra CAEs: with 10% dynamic requests, 20% more CAEs are required; with 20% dynamic requests, it's 40%

During various tests the following best practices have been showing up:

- The amount of RAM dedicated to a single MongoDB process (mongod) should exceed the working set size of the data.
- The usage of fast HDDs or SSDs is mandatory if writing becomes a bottleneck.
- When using sharding, the MongoDB Routing processes (mongos) should be deployed on the same machine as the CoreMedia CAE thus eliminating one network hop and reducing latency for database queries.
- The MongoDB routing processes (mongos) and configuration servers (mongod) consume only very few resources.
- For MongoDB and Apache Solr the CPU is typically not limiting but Memory and I/O.

The numbers have been measured on a developer machine and can be used as a conservative lower limit to estimate performance and space requirements:

Category	MongoDB RAM [Bytes]	MongoDB disk space [Bytes]	MongoDB Throughput [1/h]
Users	2500	2500	1800000
Comments	4000	4000	900000
Ratings	2500	2500	1800000
Likes	3500	3500	1200000

Table 3.1. Measured performance

3.2.6 Availability

MongoDB replicates and balances data transparently between the available nodes, checks node's health, detects new nodes and waits for old nodes to join again. Typical clustering services like failover, replication, data and request distribution is handled transparently to Elastic Social and Elastic Core based applications.

During various tests the following best practices have been showing up:

- One million users, ratings or likes require less than 10 GB of hard disk space per node. User profile pictures are not included in this upper limit estimation. See the Mongo DB documentation for details.

3.2.7 Logging

CoreMedia Elastic Social controls and processes personal data. Thus it is important to deal carefully with data logged by applications having *Elastic Social* enabled. In general it is advisable to turn off any debug logging and below as debug logging events might contain further personal data.

SLF4j Logging Markers

Logging events containing personal data or which might contain personal data are marked with so called SLF4j Logging Markers. There are two markers in `BaseMarker` dedicated to mark personal data logging events:

`PERSONAL_DATA ("personalData")`	Marks any logging event revealing personal data
`UNCLASSIFIED_PERSONAL_DATA ("unclassifiedPersonalData")`	Marks any logging event possibly revealing personal data. One example are logged exception stack traces which are raised by third-party libraries where you have no control if any of your personal data you handed over to the library will be mentioned in the exception message. You should expect many false-positives in unclassified personal data logging events.

Logback Marker Filters

The SLF4j Logging Markers can be used to configure Logback, so that logging events containing personal data can either be ignored or redirected to dedicated files which for example are better secured. To do so, configure Logback Filters.

```xml
<appender
    name="personalData"
    class="ch.qos.logback.core.rolling.RollingFileAppender"
    additivity="false">
 <filter
     class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator
        class="ch.qos.logback.classic.boolex.OnMarkerEvaluator">
      <marker>personalData</marker>
    </evaluator>
    <OnMismatch>DENY</OnMismatch>
    <OnMatch>ACCEPT</OnMatch>
 </filter>
 <file>personalData.log</file>
 [...]
```

```
</appender>
```

Example 3.1. Logback Filtering using OnMarkerEvaluator

Example 3.1, "Logback Filtering using OnMarkerEvaluator" [25] shows an example
which will redirect any personal data logging events to an extra file and remove
it from other files. This includes events which contain personal data and those
which might contain personal data (unclassified).

```java
package com.acme;

import ch.qos.logback.classic.Level;
import ch.qos.logback.classic.boolex.MarkerList;
import ch.qos.logback.classic.spi.ILoggingEvent;
import ch.qos.logback.classic.spi.IThrowableProxy;
import ch.qos.logback.classic.spi.LoggerContextVO;
import ch.qos.logback.classic.spi.ThrowableProxy;
import ch.qos.logback.core.boolex.EvaluationException;
import ch.qos.logback.core.boolex.EventEvaluatorBase;
import ch.qos.logback.core.boolex.Matcher;
import org.slf4j.Marker;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

public class CustomExpressionEvaluator extends
EventEvaluatorBase<ILoggingEvent> {

    public boolean evaluate(ILoggingEvent event) {
        return event.getMarker() != null
                && event.getMarker().contains("personalData")
                && !event.getMarker().contains("unclassifiedPersonalData");

    }
}
```

Example 3.2. Logback Filtering using custom evaluator

and its appender configuration:

```xml
<appender
    name="personalData"
    class="ch.qos.logback.core.rolling.RollingFileAppender"
    additivity="false">
  <filter
      class="ch.qos.logback.core.filter.EvaluatorFilter">
    <evaluator class="com.acme.CustomExpressionEvaluator"/>
    <OnMismatch>DENY</OnMismatch>
    <OnMatch>ACCEPT</OnMatch>
  </filter>
  <file>personalData.log</file>
  [...]
</appender>
```

Example 3.3. Logback Filtering using custom evaluator

The Logback default evaluator provides more sophisticated control right within the logging configuration without providing a custom evaluator. Example 3.2, "Logback Filtering using custom evaluator" [26] shows an example how to only filter those events which really contain personal data and ignore those which might contain false positives.

Identifying *Elastic Social* Applications

In order to adjust the logging configuration for *Elastic Social* it is important to know which applications have *Elastic Social* enabled. To identify these applications you can search for transitive dependencies on any of the *Elastic Social* modules with Maven `groupId` com.coremedia.elastic.social. Example 3.4, "Elastic Social Applications Search" [27] shows how you might find such usages based on GNU Grep and xargs.

```
$ grep --recursive --files-with-matches --ignore-case \
    --include "pom.xml" "<packaging>war</packaging>" | \
    xargs --replace \
        mvn --file {} dependency:tree \
            -Dincludes="com.coremedia.elastic.social*::jar"
            -DoutputFile=$TMP/elastic-social-applications.txt \
            -DappendOutput=true
```

Example 3.4. Elastic Social Applications Search

In default *CoreMedia Blueprint* the following applications use *Elastic Social*:

- `cae`
- `es-worker-component`
- `studio-client`
- `studio-server`

For details on application logging configuration see:

- Section 4.7, "Logging" in *Operations Basics*

3.2.8 Backup

Even with replica sets and journaling, it is still a good idea to regularly back up your data. You can find an overview about the topic and possible strategies here.

Passive MongoDB node

One approach is to run a passive MongoDB node for all backups and filesystem snapshots to take the actual backup. If journaling is enabled, it's possible to take hot snapshots of a MongoDB data directory. Without journaling it's recommended to fsync and lock the passive node and then take the snapshot from there. See the code below for an example:

```
from pymongo import Connection
def do_backup():
    <insert your snapshot and backup code here>
def lock_and_backup():
    conn = Connection(slave_okay=True)
    try:
        conn.admin.command("fsync", lock=True)
        do_backup()
    finally:
        conn.admin["$cmd.sys.unlock"].find_one()
```

Example 3.5. Snapshot from a passive node

A more detailed example how this pattern can be used with Amazon S3 can be found here.

Backup Tools

MongoDB provides tools to dump and restore the current content of the databases. `mongodump` and `mongorestore` allow you to create exact copies of your current database. You can find a detailed description here.

Incremental backup

Incremental backup is only useful in rare cases. Usually you want to restore data, if your primary is down. But if your primary is down, you will want to restore your data as quick as possible. Restoring an old state and slowly adding your incremental backup parts will take lots of time that you usually do not have in these moments. Incremental backups make restoring your data more complicated and slow them down. All you gain is mildly less disk usage. Look here for a more detailed discussion on incremental backups.

Sharding

MongoDB sharding can be used when one MongoDB replication set becomes too small to handle the application load. Sharding does not need to be configured in advance, servers can be added during normal operation and the configuration

can be updated to enable sharding. Make sure to read the MongoDB sharding documentation for a deeper insight.

For an efficient sharding configuration you need to know which databases and collections are used by *Elastic Social*.

Four databases are created for each tenant. The database names are generated from the `mongodb.prefix` setting, the tenant name and the service name separated by underscores. The service name is one of blobs, counters, models and tasks. When `mongodb.prefix` is "blueprint" and the tenant name is "media" then four databases named "blueprint_media_blobs", "blueprint_media_counters", "blueprint_media_models" and "blueprint_media_tasks" will be created.

The `BlobService` uses MongoDB GridFS for storing blobs and metadata. Please refer to the MongoDB documentation on how to configure sharding for GridFS. Example for configuring sharding for GridFS:

```
db.runCommand({   shardcollection   :   "blueprint_me
dia_blobs.fs.chunks", key : { files_id : 1 }});
```

The counter services create six collections with the counters database. The highest_average_counters and highest_histogram_counters can not be sharded. They contain aggregated counter values so these collections are rather small and this imposes no limitation. The other collections in the counters database can be sharded with the name attribute as shard key. An example is given below:

```
db.runCommand( { shardcollection : "blueprint_media_counters.average_counters"
,
key : { name : 1 } } );
db.runCommand( { shardcollection :
"blueprint_media_counters.average_histogram_counters" ,
key : { name : 1 } } );
db.runCommand( { shardcollection : "blueprint_media_counters.counters" ,
key : { name : 1 } } );
db.runCommand( { shardcollection :
"blueprint_media_counters.histogram_counters" ,
key : { name : 1 } } );
```

Example 3.6. Shard other collections

The models database contains one collection per model collection. Sharding of the blacklist and complaints collections is not recommended because they are comparatively small. For the other model collections the following shard keys are recommended:

Collection	Shard Key
comments	target : 1

Collection	Shard Key
likes	target : 1
ratings	target : 1
shares	target : 1
users	name : 1 or email: 1
notes	user : 1

Table 3.2. Recommended shard keys

An example is given below:

```
db.runCommand( { shardcollection : "blueprint_media_models.comments",
key : { target : 1 } } );
db.runCommand( { shardcollection : "blueprint_media_models.likes",
key : { target : 1 } } );
db.runCommand( { shardcollection : "blueprint_media_models.ratings",
key : { target : 1 } } );
db.runCommand( { shardcollection : "blueprint_media_models.users",
key : { name : 1 } } );
```

Example 3.7. Creating shard keys

The tasks database contains one collection per task queue. Configuring sharding for the task collections is not recommended because the tasks are removed after successful executions thus making the collections small.

If you are running a multi-tenant application you should consider spreading the databases of each tenant across the cluster so that the load is distributed evenly.

3.3 Administration

This section describes the configuration and administration of *CoreMedia Elastic Social*.

3.3.1 Block Users automatically

If the number of complaints for a user exceeds a defined quantity (`elastic.social.users.auto-block-limit`, see configuration), the user is blocked automatically.

The `AutoBlockUsersTask` is executed in a configured time interval (`users.autoBlock.interval`, see configuration).

With the default configuration no user is blocked automatically as `elastic.social.users.auto-block-limit` defaults to 0.

3.3.2 Reject Comments automatically

If the number of complaints for a comment exceeds a defined quantity (`elastic.social.comments.auto-reject-limit`, see configuration), the comment is rejected automatically.

The `AutoRejectCommentsTask` is executed in a configured time interval (`elastic.social.comments.auto-reject-interval-ms`, see configuration).

With the default configuration no comment is rejected automatically as `elastic.social.comments.auto-reject-limit` defaults to 0.

3.3.3 Reindex

Elastic Social uses JMX for all management operations. This requires that you enable JMX remoting when accessing remote hosts. To reindex the search index for users or comments execute the JConsole with JMX remoting enabled on Windows OS like this:

```
"%JAVA_HOME%\bin\jconsole" -J-classpath ^
-J"%JAVA_HOME%\lib\jconsole.jar;%USERPROFILE%\
```

```
.m2\repository\javax\management\jmxremote_optional\1.0.1_03\
jmxremote_optional-1.0.1_03.jar"
```

Example 3.8. Start JConsole on Windows OS

or on Unix based OS like this:

```
$JAVA_HOME/bin/jconsole -J-classpath \
-J$JAVA_HOME/lib/jconsole.jar:$HOME/ \
.m2/repository/javax/management/jmxremote_optional/ \
1.0.1_03/jmxremote_optional-1.0.1_03.jar
```

Example 3.9. Start JConsole alternatively on UNIX based OS

Open a new connection to the JMX port of a CAE or Studio host. For a remotely running preview CAE the default is:

```
service:jmx:rmi:///jndi/rmi://servername:40099/jmxrmi
```

Then navigate to the node `com.coremedia/SearchServiceMan ager/blueprint/media/Operations` (where media is the tenant name and blueprint the application name) and execute

```
reindex( users )
```

to reindex the search service index with the name "users". Use "comments" to reindex all comments.

3.3.4 Refresh counters

Counters are calculated automatically in defined aggregation time intervals (see configuration).

To refresh the average and histogram counters manually for the tenant media, start the JConsole as described above, navigate to the node `core media.com/AverageCounterServiceManager/blueprint/me dia/operations` where media is the tenant name and blueprint the application name and execute

```
refreshCounters( <interval\> )
```

to refresh the counters for the given interval where *LAST_DAY*, *LAST_WEEK*, *LAST_MONTH*, *LAST_YEAR* and *INFINITY* are valid values. Basically the same procedure applies for the `HistogramCounterServiceManager`, but *IN-FINITY* is not a valid value here, because it is calculated differently internally.

3.3.5 Managing Stored Personal Data

CoreMedia provides tools in *CoreMedia Studio* for accessing, changing, deleting and administration of *Elastic Social* users and their contributions. Please refer to the Chapter 8, *Working with User Generated Content* in *Studio User Manual* for more information.

Export of Stored Personal Data

CoreMedia Elastic Social stores personal data of registered users in the MongoDB database including user profile data, comments, reviews, counters and much more. Personal data needs to be secured and can be subject to regulations such as the European Union's General Data Protection Regulation (GDPR).

One part of the GDPR grants a user the right to access his stored personal data ("Right of access by the data subject"). To support the implementation of a process for such user requests, the *Blueprint* provides an example script that outputs personal data for a specific Elastic Social user.

Note that the script just outputs user data for features implemented in the product. If you've implemented custom extensions such as other contribution types or user-specific counters, additional personal data might be stored. The script serves as an example and its output must be carefully reviewed. You must still decide yourself which data is send to a user upon request.

Usage of dump-es-user-data.js script

The script is located in the *Blueprint* workspace in `global/examples/dump-es-user-data.js`. It is a script for the MongoDB Shell `mongosh` (https://docs.mongodb.com/mongodb-shell), which needs to be started with a connection to the *CoreMedia Elastic Social* models database. When authentication is enabled for MongoDB, the corresponding credentials must be passed as username (-u) and password (-p) together with the authenticationDatabase. The script is passed to the shell as parameter. The name of the user must be passed as variable `userName` with the `--eval` option. For example, to output data of user "paul" for the tenant "corporate" stored in a locally running MongoDB, invoke the script as follows:

```
mongosh localhost:27017/blueprint_corporate_models -u [mongodb_user] -p
[mongodb_password]
        --authenticationDatabase admin --quiet --eval "var userName='paul'"
 dump-es-user-data.js
```

Example 3.10. Dump data of user "paul"

If the given user exists, the script will output JSON for the user's profile, his contributions, complaints, internal notes about the user and user-specific counters. Binary attachments such as a user's profile image or comment attachments are mentioned at the end of the script with instructions how to dump the binary data with the *mongofiles* utility (https://docs.mongodb.com/manual/reference/program/mongofiles/).

4. Development

This chapter describes how you adapt *Elastic Social* to your own needs.

4.1 Security

SQL Injection

Elastic Social does not rely on SQL for database access so all Elastic Social components are immune to SQL injection attacks.

The MongoDB NoSQL database used in *Elastic Social* transfers BSON encoded data. To communicate with the MongoDB server Elastic Social uses the MongoDB Java Driver which takes care of the necessary encoding of BSON messages which prevents injection of unintended data. For information about SQL injection attacks please refer to the MongoDB documentation and forums.

4.2 Persistence Model

The *Elastic Core* persistence is based on instances of `Model`s to which the data that is stored in MongoDB is mapped at runtime. The idea is that not the Java classes determine how the MongoDB documents are structured but the MongoDB document is mapped to a given Java instance. Parts of the documents that do not fit the given Java instance are mapped into a generic data pool to make sure that no data is lost when the Java instance is persisted back into the MongoDB document just because the given Java instance does not understand them:
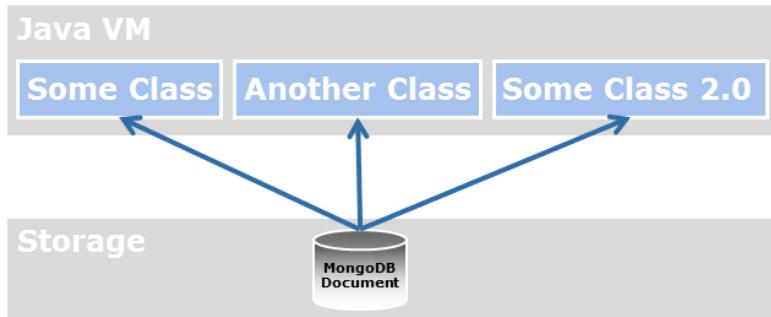
Figure 4.1. Mapping of Java classes and MongoDB documents

This mapping behavior offers a lot more flexibility to update Java classes without running into the hassles of schema evolution. For example, it allows for different Model classes accessing the same data at the same time. But it is different from typical mappers like Morphia, Spring Data for MongoDB or Hibernate that take a Java class as the source how to structure the data in the storage underneath.

Mapping properties

The mapping algorithm uses Java Bean properties as entities to load and store data. That means if some `Model` class is used to load data via for example the `ModelService` get(…) methods, the `Query` or the `SearchService`, the mapping algorithm first creates an instance of the given `Model` class and then calls the setters of the instance to transfer data from the MongoDB document to the instance. If a `Java Bean` property is defined in the `Model` instance, its setter method is called by the mapping algorithm and its value is accessible via the getter method. If no `Java Bean` property is defined the data is stored in the generic data pool of the instance, which is accessible via `Model#getProp erty()`.

If an instance of a `Model` class is stored with `Model#save()` or `ModelService#save()`, the mapping algorithm calls the getters of the given instance and joins them with the generic data pool to map these properties into a MongoDB document. The key for storing of data is the same combination of ID and Collection that was used to lookup the data.

In all implementations of this interface all setter methods for non-primitive types must support null values, even if a default value is used during initialization. Code or data migration might still cause the setter to be called with a null value.

Mapping atomic values

The following table describes the mapping of BSON values to the corresponding Java types:

BSON	Java
Boolean false/true	Boolean
Floating point	double
32-bit Integer	int
64-bit Integer	long
Boolean false/true	`java.lang.Boolean`
UTC date time	`java.util.Date`
Floating point	`java.lang.Double`
32-bit Integer	`java.lang.Integer`
64-bit Integer	`java.lang.Long`
UTF-8 string	`java.lang.String`
Object ID	`org.bson.types.ObjectId`

Table 4.1. Mapping of BSON values to Java types

Mapping collection values

The following table describes the mapping of BSON collection values to the corresponding Java types:

BSON	Java
Array	`java.util.List`
Embedded document	`java.util.Map`

Table 4.2. Mapping of BSON collection values to Java types

Please note that the mapping is defined from BSON values to Java types which means that you are limited to `java.util.List` and `java.util.Map` and cannot use the full expressiveness of the Java collection framework.

Mapping references

References to other `Model`s or user defined classes are supported via `TypeConverter`s.

To make the implementation of custom `TypeConverter`s easier, the helper class `AbstractTypeConverter` is there to provide a basic implementation for user defined types. For `Model`s there is a specialized `AbstractModel Converter` that provides a basic implementation for user defined `Model`s.

The following table describes which Maven module contains support for the given types:

Module	Mapped Class
`core-impl`	`com.coremedia.elastic.core.api.blobs.Blob`
	`com.coremedia.elastic.core.api.models.Model`
	`com.coremedia.elastic.core.api.users.User`
	`java.lang.Enum`
	`java.lang.Locale`

Module	Mapped Class
`social-impl`	`com.coremedia.elastic.social.api.comments.Comment`
	`com.coremedia.elastic.social.api.reviews.Review`
	`com.coremedia.elastic.social.api.users.CommunityUser`
`core-cms`	`com.coremedia.cap.content.Content`
	`com.coremedia.objectserver.beans.ContentBean`
	`com.coremedia.xml.Markup`

Table 4.3. Which module contains support for which type

MongoDB Collections and IDs

MongoDB documents are stored in collections which can be seen as named groupings of documents which share roughly the same structure or purpose. Indexes and queries are defined per MongoDB collection. The key for the lookup of data in the MongoDB is the combination of ID and Collection. It is accessible via `Model#getId()` and `Model#getCollection()`.

Extending models, users and comments

The basic idea to extend `Model`s is to keep it simple for the API user, but hide and reuse the implementation. You should never extend internal subclasses. Extending public interfaces is possible and supported but not necessary. If you want to extend the API interfaces, create an interface and an implementation for that aspect you are missing like this:

```
public interface FooUser extends User {
   String getFoo();
   void setFoo(String foo);
}

public abstract class FooUserImpl implements FooUser {
   private String foo;

   public String getFoo() {
```

```
    return foo;
  }

  public void setFoo(String foo) {
    this.foo = foo;
  }
}
```

Example 4.1. Extending the API interfaces

Instances of the class above are enhanced with the internal implementation of `Model` and `User` when calling `UserService#createUser()`. Beware that this call does not persist the returned instance to give the caller a possibility to modify the returned instance before saving it with `Model#save()`.

```
FooUser fooUser = userService.createUser("foos-id", FooUserImpl.class);
fooUser.setFoo("foo");
fooUser.save();
```

Example 4.2. Modifying returned instance

When you already have a `User`, just use `UserService#createFrom()` to turn it into `FooUser` with a copy of the data that the User had. All data from User is still readable and writable through the methods for the generic data pool:

```
User user = userService.getUserById("4711");
FooUser fooUser = userService.createFrom(user, FooUserImpl.class);
fooUser.setFoo("bar");
fooUser.setProperty("name", "Foobar");
fooUser.save();
```

Example 4.3. Create user from existing user

> **NOTE**
>
> `user` and `fooUser` are different instances. Any changes to user are not visible at the `fooUser` instance. Saving a modified `user` and then a modified `fooUser` in the scenario above will overwrite the changes applied to user.

Changing the class of an instance

`ModelService#createFrom` may be used to change the class for a given `Model` instance without reloading the data from the underlying MongoDB document.

4.3 Indexing

Model indexing

Typically, the access to `Models` is very cheap for the id property and calls to `ModelService#get(id,collection)` and very expensive for all other properties. A `ModelIndex` helps to speed up the access to other properties.

To create a `ModelIndex` for the collection myobjects and the x property of all MongoDB documents inside the collection, define a `ModelIndexConfiguration` like this:

```
@Named
public class MyObjectsModelIndexes implements ModelIndexConfiguration {
  @Inject
  private ModelIndexConfigurationBuilder builder;

  public Collection<ModelIndex> getModelIndexes() {
    return builder.
            configure("myobjects", "x").
            build();
  }
}
```

Example 4.4. Creating a ModelIndex

This speeds up the executions of `Query`s to the property `x` to the same level as those for the property `id` when called like this:

```
MyObject myObject = modelService.query("myobjects").
     filter("x", EQUAL, "1234").get(MyObject.class);
```

Example 4.5. Create a query

> **NOTE**
>
> The creation of indexes is not enabled by default to speed up faster initial bulk loading. To enable the creation of indexes, set `mongodb.models.create-indexes` to true as described in the Configuration properties.

> **NOTE**
>
> Keep the number of indexes to an absolute minimum because they consume precious heap memory in the MongoDB.

Model collection configuration

With a `ModelCollectionConfiguration` an automatic removal of `Model`s after a defined time span can be configured.

The `ModelCollectionConfiguration` is configured for a collection name, a `Date` property of the `Model`, a time to live time span in seconds.

The configured `ModelCollectionConfiguration` adds an index to a specified Date field of a collection with the time to live interval and removes the models automatically, when the time span has expired.

If a sparse option is required for the collection property, a separate ModelIndex has to be configured. On index creation the index configuration will be merged resulting in one sparse TTL index for that field.

To create a `ModelCollectionConfiguration` for the collection myobjects, the date property creationDate and the time to live period of 180 days, define a `ModelCollectionConfiguration` like this:

```
@Named
public class MyObjectsModelCollectionConfigurations implements
ModelCollectionConfiguration {

  private static final int EXPIRE_AFTER_SECONDS = 180*24*60*60; //180 days

  @Inject
  private ModelCollectionConfigurationBuilder builder;

  public Collection<CollectionConfiguration> getCollectionConfigurations()
{
    return builder.
            configureTTL(
              "myobjects",
              "creationDate",
              EXPIRE_AFTER_SECONDS).
            build();
  }
}
```

Example 4.6. Creating a ModelCollectionConfiguration

> **NOTE** ⓘ
>
> The creation of a TTL index can be prevented by setting the time to live time span to 0. This will not drop an existing index.

> **NOTE** ⓘ
>
> A TTL index cannot be created, if a single field index already exists for that field. To create the TTL index, the existing index must be dropped first.

Search indexing

For the full text retrieval and suggestions for `Model`s the `SearchService` is used.

To create a `SearchIndex` with the name myindex for models of the collection mycollection, the reindex property `creationDate` and their title and text property, define a `SearchIndexConfiguration` like this:

```
@Named
public class MyObjectsSearchIndexes implements SearchIndexConfiguration {
  @Inject
  private SearchIndexConfigurationBuilder builder;

  public Collection<SearchIndex> getSearchIndexes() {
    return builder.
          configure("myindex", "mycollection", "creationDate", null, "title",
 "text").
            build();
  }
}
```

Example 4.7. Create a SearchIndexConfiguration

You can define `SearchIndexCustomizer`s to customize how a `Model` will actually be indexed, for example, if you need to index references to other models or lists. An example `SearchIndexCustomizer` that adds an author's name and email to the comment search index looks like this:

```
@Named
@Order(value=100)
public class CommentAuthorSearchIndexCustomizer implements
SearchIndexCustomizer {
  @Inject
  private CommentService commentService;

  public void customize(String indexName, Model model, Map<String, Object>
serializedObject) {
```

```
    if ("comments".equals(model.getCollection())) {
      Comment comment = commentService.createFrom(model);
      if (comment != null) {
        CommunityUser user = comment.getAuthor();
        if (!user.isAnonymous()) {
          serializedObject.put("authorName", user.getName() + " " +
user.getEmail());
        }
      }
    }
  }
```

You can use the Spring Framework `@Order` annotation or the `Ordered` interface to define a priority for a customizer. A higher priority means that you can overwrite values defined by customizers with a lower or no priority. The `SearchIndexCustomizer`s defined in the product have no priority defined, so they can easily be overwritten.

> **NOTE**
>
> When you work with `SearchIndexCustomizer`s to add information about referenced models, changes to the referenced models will only be indexed when the referring model itself is changed or the whole index is rebuilt.

> **NOTE**
>
> The indexing of models as described above is implemented via the `TaskQueueService`. To enable it, set `taskqueues.worker-node` to true as described in the Configuration properties and configure the location of the Apache Solr server with `elastic.solr.url` (or `elastic.solr.cloud=true` and `elastic.solr.zookeeper.addresses` for SolrCloud).

Caching

Differing from the CoreMedia CMS *Content Server* and its Unified API the latencies and throughput of the MongoDB are more similar to memcached. This means, caching should only be introduced if performance tests show up bottlenecks.

To avoid bottlenecks, minimize the amount of requests to the MongoDB by minimizing the amount of calls to the Elastic Core and Elastic Social API. Do not refetch `Model`s but keep them during one request.

Referential Integrity

The `ModelService` does not ensure referential integrity between `Model`s or from Models to content beans. When accessing model properties of these types, the implementation will return proxy objects regardless of whether the targeted Model or `ContentBean` exists. When trying to access the proxy objects, the references will be resolved and in case that the referenced object does not exist, an `UnresolvableReferenceException` will be thrown. The application developer needs to deal with this case by surrounding access to referenced objects by try/catch blocks (or #attempt blocks in FTLs). Examples are given below.

```
for (Comment comment : commentService.getNextUnapprovedComments(true, 10))
{
  try {
    if (!comment.getAuthor().isActivated()) {
      ...
    }
  } catch (UnresolvableReferenceException e) {
    LOG.warn("...", e);
  }
}

<#if comments?has_content>
  <#list comments as comment>
    <#attempt>
      ...
      <div class="comment-author">
        ${comment.author.name}
      </div>
      ...
    <#recover>
      <#-- ignore -->
    </#attempt>
</#if>
```

Example 4.8. Example try catch

4.4 Listening to Model Changes

Differing from the *CoreMedia CMS Content Server* and its *Unified API* the `ModelServiceListener` is a local listener at `ModelService` that is only notified before and after `Model#save()` and `Model#remove()` calls from models that were created from that `ModelService`.

To register a `ModelServiceListener` at the `ModelService` it has to be in the application context. This can be achieved by annotating the `ModelServiceListener` implementation with `javax.inject.Named` and using component scanning.

For a fault-tolerant processing of `ModelServiceListener` events, it is recommended to immediately queue the work to be done with the `TaskQueueService`. A listener following this pattern looks like this:

```
@Named
public class MyObjectsModelServiceListener extends ModelServiceListenerBase
{
  @Inject
  private TaskQueueService taskQueueService;

  private MyTask defer() {
    return taskQueueService.queue("mytasks", MyTasks.class);
  }

  public void afterSave(Collection<? extends Model> models) {
    defer().processSave(models);
  }

  public void afterRemove(Collection<? extends Model> models) {
    defer().processRemove(models);
  }
}
```

Example 4.9. Listener

4.5 Message Queue Model

The *Elastic Core* message queue is based on the idea that method calls (called tasks) may be deferred (that is, queued) to a later point of time where they can be processed concurrently by a pool of worker applications. It is ensured that a task is executed at least once. On errors the task is automatically retried by another worker until an error count limit is reached.

The `TaskQueueService` persists its information in the same MongoDB as the `ModelService` and uses the same mapping algorithm to store the arguments of the method calls.

A typical method call sequence when using the `TaskQueueService` looks like this:
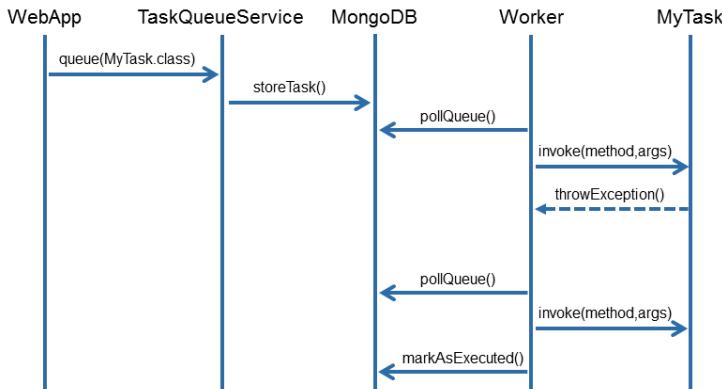

Figure 4.2. Method call sequence using the TaskQueueService

Creating task queues

To create a `TaskQueue` with the name mytasks, define a `TaskQueueConfiguration` like this:

```
@Named
public class MyTaskQueues implements TaskQueueConfiguration {
  @Inject
  private TaskQueueConfigurationBuilder builder;

  public Iterable<TaskQueue> getTaskQueues() {
    return builder.
           configure("mytasks").
           build();
  }
```

```
}
```

Example 4.10. TaskQueueConfiguration

Executing tasks

Tasks are simple classes that contain methods which can have parameters that are handled by the mapping algorithm:

```
@Named
public class MyTask {
  @Inject
  private ModelService modelService;

  public void doSomething(int id, String name, Object value) {
    Model model = modelService.get(id);
    model.setProperty(name, value);
    model.save();
  }
}
```

Example 4.11. A task class

Execute such a task (called mytasks) via the `TaskQueue` as follows:

```
@Inject
private TaskQueueService taskQueueService;

public void executeInTaskQueue() {
  taskQueueService.queue("mytasks", MyTask.class).doSomething(4711, "hello",
  "world");
}
```

Example 4.12. Execute a task

4.6 Counters

This section describes the configuration and usage of Counters in *CoreMedia Elastic Social*.

The following `CounterServices` are available in Elastic Social:

- `CounterService`: for simple counters with a given name and value which can increment or decrement a value.
- `HistogramCounterService`: for counters which also contain a date. This is necessary if you want to determine a counter value for a certain time period, for instance the most commented articles in the last week.
- `AverageCounterService`: for counters which can increment and decrement two values, the total sum and the number of samples to calculate an arithmetic mean, for instance if you want to calculate the average rating. It handles counters with and without a date.

Counters are stored in the database [prefix]_[tenant]_counters. The following collections contain counter values:

Name	Description
counters	Counters with aggregated value
histogram_counters	Histogram counters with date and sum
average_counters	Average counters with aggregated sum and quantity
average_histogram_counters	Average counters with date, sum and quantity

Table 4.4. Counter collections

Each `counter` is stored aggregated with a value in the counters collection.

Each `histogram counter` is stored separately with sum and date in the `histogram_counters` collection and aggregated with value in the `counters` collection.

Each `average counter` is stored separately with sum, quantity and date in the `average_histogram_counters` collection and aggregated with sum and quantity in the `average_counters` collection.

A sorted list for highest values for simple counters without a date can easily be calculated using a simple query. Lists which need to consider an average value or a certain time interval need to be aggregated using map and reduce jobs.

The following collections contain these aggregated sorted lists of counter values, for instance the most commented targets in a given time interval:

Name	Description
highest_average_coun-ters	The highest average counters without time limitation (infinity)
highest_average_coun-ters_[INTERVAL]	The highest average counters for the given time interval for instance the last week ("LAST_WEEK")
highest_histo-gram_counters_[INTER-VAL]	The highest histogram counters for the given time interval for instance the last week ("LAST_WEEK")

Table 4.5. Aggregated counter collections

All aggregated counter lists are updated in given time intervals that are configurable (`counters.aggregation-interval-milliseconds[.interval]`, see Table 3.43, "Counters Properties" in *Deployment Manual*).

Counters can also be refreshed manually using JMX, see Section 3.3.4, "Refresh counters" [32].

The following tables list the predefined counters in *Elastic Social* which you can access via the counter services.

The following `counters` are implemented in *CoreMedia Elastic Social*:

Name	Description
`user:number_of_logins`	The number of logins of the user
`comments:approvedComments`	Number of approved comments
`comments:rejectedComments`	Number of rejected comments
`reviews:approvedReviews`	Number of approved reviews

Name	Description
`reviews:rejectedRe views`	Number of rejected reviews
`complaints:comments`	Number of complaints for a comment
`complaints:users`	Number of complaints for a user

Table 4.6. Counters used in CoreMedia Elastic Social

The following `histogram counters` are implemented in *CoreMedia Elastic Social*:

Name	Description
`comments:mostCommen ted[:category]`	Most commented target [per category]
`re views:mostReviewed[:cat egory]`	Most reviewed target [per category]
`share[:category]`	Number of shares for a target [per category]
`like[:category]`	The number of likes for a target [per category]
`author:num ber_of_likes`	Number of likes from the author
`author:number_of_rat ings`	Number of ratings from the author
`author:number_of_re views`	Number of reviews from the author

Table 4.7. Histogram counters

The following `average counters` are implemented in *CoreMedia Elastic Social*:

Name	Description
rating[:category]	The number of ratings for a target [per category]

Table 4.8. Average counters

4.7 Integration

This section describes the integration of *CoreMedia Elastic Social* into a Spring Boot application.

4.7.1 Apache Maven

CoreMedia provides BOM POMs for simple dependency management with Apache Maven. To use *Elastic Social* artifacts, your POM needs to import the BOM POMs. The BOM POMs ensure that you use artifacts of compatible versions and also manage the scope of all *Elastic Social* dependencies. API modules have compile scope, test utility modules have test scope and all other modules have runtime scope.

When using *Elastic Social*, you need to define dependencies to the API modules and to the implementation modules you are going to use. A typical usage of Elastic Social dependencies is shown below. Besides the API dependencies, the Elastic Core implementations for MongoDB, Apache Solr and Spring Security are included as well as the Elastic Social implementation module. For testing a dependency to the Elastic Core test utility module is declared.

```
<?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">
  ...
  <dependencies>
    ...
    <!-- allowed Elastic Core and Elastic Social dependencies:
         core-api, social-api: compile
         others: runtime
     -->
    <dependency>
      <groupId>com.coremedia.elastic.core</groupId>
      <artifactId>core-api</artifactId>
    </dependency>
    <dependency>
      <groupId>com.coremedia.elastic.social</groupId>
      <artifactId>social-api</artifactId>
    </dependency>
    <dependency>
      <groupId>com.coremedia.elastic.core</groupId>
      <artifactId>core-solr</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>com.coremedia.elastic.core</groupId>
      <artifactId>core-mongodb</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>com.coremedia.elastic.social</groupId>
      <artifactId>social-spring-security</artifactId>
```

```
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>com.coremedia.elastic.social</groupId>
      <artifactId>social-impl</artifactId>
      <scope>runtime</scope>
    </dependency>
    ...
  </dependencies>
...
</project>
```

Example 4.13. Typical Elastic Social dependencies

Application context setup

To configure *Elastic Social* you need to enable Spring classpath scanning for the package `com.coremedia.elastic`. Configuration properties will be accessed through the Spring framework `Environment` which collects all property sources. Two additional beans need to be configured. A bean of type `org.springframework.mail.javamail.JavaMailSender` needs to be defined for the `MailService` and an implementation of a `MailTemplateService` needs to be provided. An example for a Spring configuration is shown below. If you use the `InMemoryMailTemplateService`, you need to have a dependency on the Elastic Social `social-base` module.

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

<context:component-scan base-package="com.coremedia.elastic"/>

<bean class="org.springframework.mail.javamail. \
      JavaMailSenderImpl">
   <property name="host" value="mail.example.com"/>
   <property name="port" value="25"/>
</bean>

<bean class="com.coremedia.elastic.social. \
      base.mail.InMemoryMailTemplateService">
   <property name="mailTemplates">
     <set>
       <bean class="com.coremedia.elastic.social. \
             base.mail.InMemoryMailTemplate">
         <property name="name">
           <util:constant static-field="com.coremedia.elastic. \
               social.api.MailTemplates.COMMENT_REJECTED"/>
         </property>
         <property name="locale" value="ROOT"/>
         <property name="from" value="reject-contribution@example.com"/>
```

```
          <property name="subject" value="Rejected contribution \
              at example.com"/>
          <property name="text">
            <value><![CDATA[Hello ${name},

your comment below from ${commentDate} has not been published:

"${commentText}"

Please comply to our community policy when writing contributions.

Kind regards,
the editors
]]></value>
          </property>
        </bean>
      </set>
    </property>
  </bean>
</beans>
```

Example 4.14. Application context Spring example configuration

If you have a CoreMedia CAE application, just name the property file `/WEB-INF/component-elastic.properties` and its properties will be automatically be loaded without the need to configure a `PropertyPlaceholder Configurer`.

Note that default values cannot be configured using a standard Spring `Proper tiesSourcesPlaceholderConfigurer` as shown in Example 4.15, "Invalid configuration setup" [57].

```
<context:property-placeholder
location="classpath:/com/acme/es-defaults.properties"/>
```

Example 4.15. Invalid configuration setup

You must use a custom configuration class and Spring annotations `org.springframework.context.annotation.Configuration` and `org.springframework.context.annotation.PropertySource` instead, as shown in Example 4.16, "Default configuration setup example" [57].

```
@Configuration(proxyBeanMethods = false)
@PropertySource(name = "es-defaults", value =
{"classpath:/com/acme/es-defaults.properties"})
public class MyElasticSocialConfiguration {
  ...
}
```

Example 4.16. Default configuration setup example

An example of a `/com/acme/es-defaults.properties` file used by the Spring configuration above is shown below:

```
mongodb.prefix=example-project-prefix
mongodb.client-uri=mongodb://mongo1.example.com:27017, \
    mongo2.example.com:27017,mongo3.example.com:27017

mongodb.models.create-indexes=true
taskqueues.worker-node=true

elastic.solr.indexPrefix=example-project-prefix
elastic.solr.url=http://solr.example.com:40080/solr
```

Example 4.17. Example of the /com/acme/es-defaults.properties file

4.7.2 Multi-Tenancy

Elastic Core supports multi-tenancy. A tenant can have many sites, but each site belongs to exactly one tenant. In a multi-tenancy environment a `Tenant ForSiteStrategy` is used to determine the tenant for a given site. *CoreMedia Blueprint* contains a solution based on settings. For each call to the Elastic Core API a tenant has to be defined or an exception will be raised. If only one tenant is required, you can define a default tenant using the property `tenant.de fault`. Tenants have to be registered at the `TenantService` and may then be set and cleared for each thread. It is recommended to set the tenant as early in a request cycle as possible. Elastic Core includes a servlet filter that uses a `TenantLookupStrategy` to determine the tenant for a request. A `Ten antLookupStrategy` is only required in a multi-tenancy setup. Elastic Social comes with an implementation for Studio REST calls and *Blueprint* defines a strategy for CAE applications as well. If you have your own project application, you need to define the Servlet Filter that comes with Elastic Social and implement your own `TenantLookupStrategy`.

The default tenant can only be statically configured and is used at runtime for every thread that otherwise has no tenant. The default tenant cannot be dere-gistered but its tenant scope is destroyed when the application context is closed so that destruction callbacks are invoked.

The `TenantFilter` needs to be configured as `FilterRegistrationBean`, see `ESCaeFilters` for details.

```
@Configuration(proxyBeanMethods = false)
public class EsCaeFilters {
  @Bean
  public FilterRegistrationBean tenantFilterRegistration(Filter tenantFilter)
  {
    return RegistrationBeanBuilder
            .forFilter(tenantFilter)
            .urlPatterns("/servlet/*")
            .order(120)
            .build();
  }
```

```
    }
```

Example 4.18. Configure a tenant filter and its mapping in your own application context

4.7.3 Using Elastic Social Services

Elastic Core uses dependency injection for configuration of components, specifically JSR–330: Dependency Injection for Java and JSR 250: Common Annotations for the Java Platform. These standards are supported by Spring 3.0 and later versions.

Use the @Inject annotation to get *Elastic Core* and *Elastic Social* services injected into any Spring Bean. The following example shows a Spring controller which uses the UserService.

```java
import com.coremedia.elastic.core.api.user.User;
import com.coremedia.elastic.core.api.user.UserService;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import javax.inject.Inject;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ExampleController implements Controller {
  @Inject
  private UserService userService;

  public ModelAndView handleRequest(HttpServletRequest request,
      HttpServletResponse response) throws Exception {
    User user = userService.getUserById(
        request.getParameter("userId"));
    response.setContentType("text/plain");
    response.getWriter().format("Hello %s!", user == null ?
        "World" : user.getName());
    return null;
  }
}
```

Example 4.19. Spring controller with UserService

4.7.4 Authentication and Authorization

Elastic Social is designed to be as flexible and modular as possible when it comes to identity and access management. It comes preintegrated with Spring Security and its own user database provided by the CommunityUserService to cover identity and access management out of the box but every component may be replaced.

The following picture depicts the components involved in identity and access management:
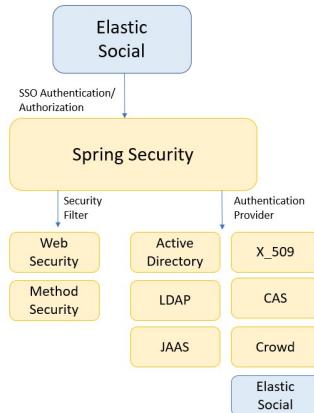

Figure 4.3. Components in identity and access management

4.7.4.1 Elastic Social Authentication

This section covers only the configuration of the Elastic Social extensions for Spring Security. Please refer to Section 4.3.8, "Spring Security" in *Content Application Developer Manual* and the Spring Security Reference Documentation for details about customizing the Spring Security configuration for the CAE.

Elastic Social provides a `social-spring-security` module which contains Spring Security auto configurations and further classes (like `UserAuthenticationProvider`) that are used for authentication against the user database provided by the `CommunityUserService`. For customizations extend the `SocialHttpSecurityConfigurer`, override its `configure` methods and provide it as a bean. For more detailed information see the `API documentation for package com.coremedia.elastic.social.springsecurity`.

4.7.4.2 LDAP Authentication

When using an LDAP server for user authentication the user database provided by the `CommunityUserService` can be used as a proxy so that the LDAP

server will only be used for authentication and the user details will be copied to and queried from the Elastic Social user database.

In this case a different Spring Security configuration has to be used and a Maven dependency to `org.springframework.security:spring-security-ldap` has to be added. Please refer to the Spring Security LDAP documentation for details. Instead of the `SocialWebSecurityAutoConfiguration.authenticationProvider`, an `LdapAuthenticationProvider` must be configured. To get access to extended user information, an `InetOrgPersonContextMapper` is used. And to copy the user details to the *Elastic Social* user database after successful authentication, an `ApplicationListener` must be implemented.

```
package com.example.es.security.ldap;

import com.coremedia.elastic.core.api.users.UserService;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.ldap.core.support.BaseLdapPathContextSource;
import org.springframework.ldap.core.support.LdapContextSource;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.BeanIds;
import
org.springframework.security.config.ldap.LdapBindAuthenticationManagerFactory;
import
org.springframework.security.ldap.userdetails.InetOrgPersonContextMapper;

@Configuration(proxyBeanMethods = false)
public class LdapAuthenticationConfiguration {

  @Bean(BeanIds.AUTHENTICATION_MANAGER)
  @ConditionalOnMissingBean
  AuthenticationManager authenticationManager(BaseLdapPathContextSource
contextSource) {
    LdapBindAuthenticationManagerFactory factory =
            new LdapBindAuthenticationManagerFactory(contextSource);
    factory.setUserDnPatterns("uid={0},ou=people");
    factory.setUserDetailsContextMapper(new InetOrgPersonContextMapper());
    return factory.createAuthenticationManager();
  }

  @Bean
  LdapContextSource contextSource() {
    LdapContextSource source = new LdapContextSource();
    source.setUrl("ldap://ldap.example.com:389/dc=example,dc=com");
    return source;
  }

  @Bean
  ExampleAuthenticationSuccessEventListener
authenticationSuccessEventListener(UserService userService) {
    return new ExampleAuthenticationSuccessEventListener(userService);
  }
}
```

Example 4.20. Configuring LDAP Authentication

```
package com.example.es.security.ldap;

import com.coremedia.elastic.core.api.users.User;
```

```
import com.coremedia.elastic.core.api.users.UserService;
import org.springframework.context.ApplicationListener;
import
org.springframework.security.authentication.event.AuthenticationSuccessEvent;
import org.springframework.security.ldap.userdetails.InetOrgPerson;

public class ExampleAuthenticationSuccessEventListener
        implements ApplicationListener<AuthenticationSuccessEvent> {

  private final UserService userService;

  public ExampleAuthenticationSuccessEventListener(UserService userService)
  {
    this.userService = userService;
  }

  @Override
  public void onApplicationEvent(AuthenticationSuccessEvent event) {
    InetOrgPerson principal = (InetOrgPerson)
event.getAuthentication().getPrincipal();
    User user = userService.getUserByName(principal.getUsername());
    if (user == null) {
      user = userService.createUser(principal.getUsername(),
principal.getMail());
      user.save();
    } else if (!user.getEmail().equals(principal.getMail())) {
      user.setEmail(principal.getMail());
      user.save();
    }
  }
}
```

Example 4.21. Implementing an ApplicationListener

```
<?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">
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>com.coremedia.elastic.social</groupId>
      <artifactId>social-spring-security</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-config</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-ldap</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.ldap</groupId>
      <artifactId>spring-ldap-core</artifactId>
    </dependency>
  </dependencies>
  ...
</project>
```

Example 4.22. Spring LDAP dependencies

4.7.5 Emails

CAE

Emails can be sent to a user for specific user actions or events. For the following events corresponding listeners are triggered and can be customized:

- Event: State change of a `CommunityUser`,

  Listener: `CommunityUserServiceListener#onStateChanged`

- Event: Registration requested

  Listener: `RegistrationServiceListener#onRegistrationRe quested` or `RegistrationServiceListenerBase#onRegistra tionRequested`

- Event: A `CommunityUser` requested to reset his password

  Listener: `RegistrationServiceListener#onPasswordResetRe quested` or `RegistrationServiceListenerBase#onPasswordRe setRequested`

- Event: State change of a `Comment` or of a `Review`

  Listener: `CommentServiceListener#onStateChanged`

Studio

For the following events, an email is sent automatically. The corresponding `MailTemplates` must be provided.

- User Blocked: The `CommunityUser#State` changes to `Community User.State.BLOCKED`.
- User Restored: The `CommunityUser` has a changed profile and the moderator resets the profile to the last values. The email is only sent for a user who has not the state `CommunityUser.State.ANONYMIZED`, `Community User.State.IGNORED` or `CommunityUser.State.BLOCKED`.
- User Deleted: The `CommunityUser` is deleted.
- Comment rejected: A comment of the `CommunityUser` is rejected. The email is only sent for a user who has not the state `Community User.State.ANONYMIZED`, `CommunityUser.State.IGNORED` or `CommunityUser.State.BLOCKED`.

- User Profile Changed: A property of the `CommunityUser` changed. The email is only sent for a user who has not the state `CommunityUser.State.ANONYMIZED`, `CommunityUser.State.IGNORED` or `CommunityUser.State.BLOCKED`.

For the following event, an email is sent, if the corresponding listener is implemented and the mail template is provided:

- Resend Registration Confirmation: The moderator clicks on the "resend registration confirmation" link in the user details section. The email is only sent for a user who has the state `CommunityUser.State.REGISTRATION_REQUESTED` and if the listener `RegistrationServiceListener#onRegistrationRequested` is implemented.
- User Activated: The email is sent when using premoderation and when a newly registered and activated user is actually approved. The listener `CommunityUserServiceListener#onStateChanged` must be implemented.

4.7.6 BBCode

BBCode is supported for comment formatting. Supported BBCode tags are shown in Example 4.23, "Supported BBCode" [64]. Use `Comment#getTextAsHtml()` to retrieve the comment text with BBCode tags converted to HTML.

```
[b]bold[/b]
[i]italic[/i]
[quote]Block Quote[/quote]
[url]www.coremedia.com[/url]
[url=www.coremedia.com]Coremedia[/url]
[url="https://www.coremedia.com/"]Coremedia[/url]
```

Example 4.23. Supported BBCode

The configuration of the BBCode text processor KefirBB is customizable. A user defined configuration file is looked up first in `classpath*:kefirbb.xml`. If no user defined configuration is found, the *Elastic Social* configuration is used.

NOTE

The *Elastic Social* configuration of KefirBB converts line endings to `<br/>`

4.8 Known Limitations

This page describes known limitations of *CoreMedia Elastic Social.*

Using `Query#skip` for MongoDB Queries can be very costly

The MongoDB has the following text to this issue:

> Unfortunately skip can be (very) costly and requires the server to walk from the beginning of the collection, or index, to get to the offset/skip position before it can start returning the page of data (limit). As the page number increases skip will become slower and more CPU intensive, and possibly IO bound, with larger collections. Range based paging provides better use of indexes but does not allow you to easily jump to a specific page.

Queries for content with interfaces which do not extend Model

In some cases you want to persist your objects, but you do not want to expose in your interface how you do it. For instance, a rating is persisted internally as a Model, but the interface does not extend the Model interface. Your interface and implementation for a Custom object would look like this:

```
public interface Custom {
}
public class CustomModelImpl implements Custom, Model {
}
```

Example 4.24. Custom interface

If you query for those Custom objects, you need to use implementation class which extends Model:

```
List<CustomModelImpl> impls = modelService.query("customModels",
CustomModelImpl.class).fetch();
```

Example 4.25. Custom implementation

If you want to have a query result list you need to manually copy all query results to a new list:

```
public List<Custom> getCustoms() {
    List<CustomModelImpl> impls = modelService.query("customModels",
        CustomModelImpl.class).fetch();
    List<Custom> result = new ArrayList<Custom>(impls.size());
    for (Custom impl : impls) {
        ratings.add(impl);
    }
    return result;
}
```

Example 4.26. Get query result list

Non public properties

You might want to have properties which are part of the implementation, but
not of the interface definition. For example, your interface and implementation
might look like this:

```
public interface CustomModel extends Model {
}

public class CustomModelImpl implements CustomModel {
  private int level;

  public int getLevel() {
    return level;
  }

  public void setLevel(int level) {
    this.level = level;
  }
}
```

Example 4.27. Interface and implementation

If you have a service using this model, you want the service to define methods
for the interface, not the implementation.

```
public class CustomModelService {
  public void doSomething(CustomModel model);
  }
}
```

Example 4.28. Model method definition

You cannot easily cast the model to its implementation class because the type
is actually generated at runtime:

```
((CustomModelImpl) model).setLevel(5);
  // ClassCastException because the type is actually generated at runtime
```

Example 4.29. Casting of models

The best workaround for this is to use the `setProperty` method of the model using constants, which you should define in your implementation class `Custom ModelImpl`:

```
model.setProperty(LEVEL_PROPERTY, 5)
```

Example 4.30. Set model properties

Overloaded Service methods

Every Service that offers a method which returns a Model or a bunch of Models has to offer this method in three variants to ensure a maximum of extensibility. This leads to a lot of code that may be hardly reused when implementing the method.

```
public interface CustomModel extends Model {
}
```

Example 4.31. Customize models

A typical implementation for the three method variants has to follow this pattern:

```
public class CustomModelServiceImpl implements CustomModelService {
  public List<CustomModel> getSomeModels() {
    Query<CustomModel> query = createQuery();
    return query.fetch();
  }

  public <T extends CustomModel> List<T> getSomeModels(
      Class<? extends T> type) {
    return getSomeModels(type, ModelService.NO_SUPER_TYPES);
  }

  public <T extends CustomModel> List<T> getSomeModels(
      Class<? extends T> type,
      List<Class<? extends Model>> superTypes) {
    Query<CustomModel> query = createQuery();
    return query.fetch(type, superTypes);
  }
}
```

Example 4.32. Custom model services

Configuration Property Reference

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

- Table 3.42, "MongoDb Properties" in *Deployment Manual* contains properties for the configuration of MongoDB used by *CoreMedia Elastic Social* to store user data.
- Table 3.43, "Counters Properties" in *Deployment Manual* contains properties for the configuration of counters for Elastic Social data.
- Table 3.44, "Task-Queues Properties" in *Deployment Manual* contains properties for the configuration of the remote service of *Headless Server*.
- Section 3.9.5, "Elastic Social Link Building Properties" in *Deployment Manual* contains properties for the configuration for the link building of *CoreMedia Elastic Social*.
- Table 3.46, "Elastic Solr Properties" in *Deployment Manual* contains properties for the configuration of the Solr search engine for *CoreMedia Elastic Social*.

Index