CoreMedia Content Cloud

Importer Manual

CoreMedia

CoreMedia GmbH

Altes Klöpperhaus, 5. OG

Rödingsmarkt 9

20459 Hamburg

December 11, 2023 (Release 2310)

List of Tables

List of Examples

1. Preface

This manual describes the concepts of the CoreMedia Importer. You will learn how to configure and start the existing XML Importer and how to create your own importers for arbitrary XML formats.

- In Chapter 3, *Administration And Operation* [15] you will learn how to administrate and operate the importer as a standalone and web application.
- In Chapter 4, *XML Importers* [25] you will find a detailed description of the CoreMedia XML format and of the way to transform arbitrary XML data into this format.

1.1 Audience

This manual is addressed to developers of CoreMedia projects who want to write importers. That is, components which import and transform content into the CoreMedia repository.

This *Importer Manual* contains the following chapters:

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-11

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://npm.coremedia.io

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

License files

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

1.3.3 Documentation

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

https://documentation.coremedia.com

The manuals have the following content and use cases:

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

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

Manual	Audience	Content
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*.
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*. The Site Manager is deprecated for editorial work.
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of *CoreMedia Studio*. You will learn about the underlying concepts, how to use the development environment and how to customize *Studio* to your needs.
Studio User Manual	Editors	This manual describes the usage of *CoreMedia Studio* for editorial and administrative work. It also describes the usage of the *Adaptive Personalization* and *Elastic Social* GUI that are integrated into *Studio*.
Studio Benutzerhandbuch	Editors	The Studio User Manual but in German.
Supported Environments	Developers, architects, administrators	This document lists the third-party environments with which you can use the CoreMedia system, Java versions or operation systems for example.
Unified API Developer Manual	Developers, architects	This manual describes the concepts and usage of the *CoreMedia Unified API*, which is the recommended API for most applications. This includes access to the content repository, the workflow repository and the user repository.

Manual	Audience	Content
Utilized Open Source Software & 3rd Party Licenses	Developers, architects, administrators	This manual lists the third-party software used by CoreMedia and lists, when required, the licence texts.
Workflow Manual	Developers, architects, administrators	This manual describes the *Workflow Server*. This includes the administration of the server, the development of workflows using the XML language and the development of extensions.

Table 1.3. CoreMedia manuals

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

Email: documentation@coremedia.com

1.3.4 CoreMedia Training

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

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

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. See Section 4.7, "Logging" in *Operations Basics* for details.

Which Log File?

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

Where do I Find the Log Files?

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

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

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

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

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

1.4 Changelog

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

Section	Version	Description

Table 1.4. Changes

2. Overview

When you have installed and configured *CoreMedia CMS*, the most important part is still missing; the content. You can create your own content items with the CoreMedia editors, but *CoreMedia CMS* also offers the possibility to automatically import external content.

The *CoreMedia XML Importer* clients serve to transfer content from external sources into *CoreMedia CMS*. They can handle any XML format, and even non XML legacy formats can be transformed into importable XML files. For each different XML format you want to import, a specialized importer instance is required to transform the relevant components of the source documents into corresponding fields of CoreMedia content items.

The importers are non-interactive and, according to their configuration, stop processing after a single import or run and import continuously at periodic intervals. To this end, the importer can be deployed in two different modes:

- A command-line utility for single import
- A web application to run continuously

Before you start an *XML Importer*, it must be configured, for example with the location of its input data.

3. Administration And Operation

This chapter describes the basic administration tasks with the *CoreMedia Importer*.

The sections describe the following tasks:

- Section 3.1, “General Configuration” [16] describes how to configure importers
- Section 3.3, “Deployment and Operation of a Standalone Importer” [21] describes how to deploy and start an importer as a standalone applications
- Section 3.4, “Deployment and Operation of a Web Application Importer” [22] describes how to deploy and start an importer as a web applications

3.1 General Configuration

Each importer needs a configuration file named after the importer and with the extension `properties` in the directory `properties/corem` or `WEB-INF/properties/corem`, if installed as web application. The *Blueprint* example module `importer-config` contains the file `cm-xmlimport.properties`, which serves as a template for such configuration files.

To install an XML importer make a copy of the file `cm-xmlimport.properties` in the same directory and rename the copy to `<name>.properties` where *name* is the name of the importer. The importer name must match the name of the `jpif` start file if the importer is used as command-line or the servlet context name if deployed as web application. See Section 3.3, "Deployment and Operation of a Standalone Importer" [21] and Section 3.4, "Deployment and Operation of a Web Application Importer" [22] for a description of the different deployment modes. The following table describes the general configuration in the properties file.

NOTE

Any Java classes referenced in the properties configuration file (for example the `multiResultGeneratorFactory` or transformers) must be specified with fully qualified names.

`import.user`	
Value	String
Default	importer
Description	The name of the CoreMedia user with which the importer logs on. Make sure that the user has the rights required to carry out operations triggered by the import process, for example, creating a new document, editing, approving, publishing. For this purpose, the standard CoreMedia installation offers a predefined user called importer (password also importer).

`import.password`	
Value	String
Default	importer
Description	The password of the user to log in with.

`import.autoLogoutSeconds`	
Value	int
Default	-1
Description	This property defines the time of inactivity in seconds after which the importer should log out. When the importer is active again, it will log in at the server automatically. A value of "-1" means that the importer will not log out.

`import.multiResultGeneratorFactory.property.sleepingSeconds`	
Value	int
Default	-1
Description	An importer remains logged in per default, whether data are imported or not. When configuring `SubDirGenerators`, the property defines the number of seconds for the importer to be inactive after the completion of the import. If the number of seconds is very large, it is reasonable to log out the importer automatically. In this case, the released importer license can be used by another importer. Note that the special value "-1" will cause the importer to terminate after importing the contents of the inbox directories.

Table 3.1. Properties of the cm-xmlimport.properties file

The following configuration deals with the preparation and transformation of source documents. Both are generic, thanks to the importer API. Since this part of the configuration depends on the source format, this part of the configuration should be conducted by the respective developer himself.

`import.loginTimeoutSeconds`	
Value	long
Default	-1
Description	This property defines the timeout for login attempts after which the importer aborts. If `import.loginTimeoutSeconds=-1`, the importer tries to login forever without abortion.

`import.enforceCompleteVersion`

Value	Boolean
Default	true
Description	This property handles the processing of XML importer files. See Section 4.1, "The CoreMedia XML Format" [27] for details on the CoreMedia XML format. • `import.enforceCompleteVersion=true` For each <version> element in the importer file a new version will be created in the CoreMedia repository. For all properties of a version the values must be given. It is not allowed to omit a property. • `import.enforceCompleteVersion=false` Now it is possible to omit even all property elements of a version. If there are only action elements and the document already exists on the server, then no new version is created and the corresponding actions are applied to the document (delete) or to the latest document version on the server (approve, delete). If there is at least one <property> element in the <version> element then for every property that is specified in the document type but missing in the XML importer file, the property value of the predecessor document version is taken. If there is no predecessor version, then a default value is inserted, that depends on the property type.

`import.validate-textproperty`

Value	Boolean
Default	false
Description	If "true" the importer validates all XML text properties against the associated DTD. If a validation fails, no document is created on the server. For big XML properties the validation may take some time.

`import.removeBrokenLinks`

Value	Boolean
Default	false
Description	If "true" the importer removes broken content links in link list and markup properties. In markup properties only the link tag (`a` or `img`) is removed, not the containing link text. Be careful when enabling this option, as it may lead to invalid XML in markup properties.

`import.entityResolverClass`

Value	class name
Default	see description
Description	Configures the name of a class of type `org.xml.sax.EntityResolver` used to resolve entities in markup properties during XML validation. The default value is `com.coremedia.xml.ClasspathURLEntityResolver`.

Table 3.2. Properties of the cm-xmlimport.properties file

3.2 Security

Because the importer process reads, processes and updates persistent data automatically without further human interaction, it is important to protect not only the importer installation, but also the directories holding the files to import from unauthorized access. On the other hand, the importer should run with minimal privileges to avoid data leaks.

3.3 Deployment and Operation of a Standalone Importer

In the *CoreMedia Blueprint* workspace in `global/examples` you will find a `coremedia-application` module `importer-template` with two submodules `importer` and `importer-config`.

Building and Deploying the Importer

Before you can start the importer, you have to configure and build your own importer application as follows:

1. Take the `importer-template` example module and integrate it into your workspace. If you want, you can merge `importer` and `importer-config` into a single `coremedia-application` module (like the theme-importer-application does). The separation of the template modules is only related to the web application deployment as shown in Section 3.4, “Deployment and Operation of a Web Application Importer” [22].
2. Replace the `cm` prefix of `cm-xmlimport.jpif` and `cm-xmlimport.properties` with your own one, for example, `my-xmlimport.jpif`. The configuration file must be located in the directory `properties/corem`.

   Example: The configuration file `properties/corem/my-xmlimport.properties` belongs to the file `bin/my-xmlimport.jpif`.
3. Configure the importer in the `my-xmlimport.properties` file as described in Section 3.1, “General Configuration” [16].
4. Build your importer with `mvn install`.

Starting the Importer

When you have build the importer, you can start it with the following command, where <ImporterName> corresponds to the name of the importer JPIF file, `my-xmlimporter`, for example:

```
bin/cm <ImporterName>
```

The importer will check the inbox once and then terminates. To constantly check the inbox the importer can be deployed as web application instead. See Section 3.4, “Deployment and Operation of a Web Application Importer” [22].

3.4 Deployment and Operation of a Web Application Importer

The web application importer, in contrast to the `coremedia-application` importer, will continuously scan the inbox.

Building and Deploying the Importer

In order to get an importer as a web application, create a new web application module as follows:

1. Take the `importer-template` example module and integrate it into your workspace, for example into `modules/cmd-tools`.
2. Remove the `importer` module and create a new `importer-webapp` module with a POM file similar to the following example:

```
<?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">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>xml-importer-webapp</artifactId>
  <packaging>war</packaging>

  ...

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

    <dependency>
      <groupId>com.coremedia.blueprint</groupId>
      <artifactId>importer-config</artifactId>
      <type>coremedia-application</type>
      <version>${project.version}</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>com.coremedia.blueprint</groupId>
      <artifactId>importer-config</artifactId>
      <version>${project.version}</version>
      <type>pom</type>
      <scope>runtime</scope>
    </dependency>
 </dependencies>

  <build>
    <finalName>importer</finalName>
    <plugins>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <configuration>
```

```
          <failOnMissingWebXml>false</failOnMissingWebXml>
        <overlays>
            <overlay>
              <groupId>com.coremedia.blueprint</groupId>
              <artifactId>importer-config</artifactId>
              <type>coremedia-application</type>
              <targetPath>WEB-INF</targetPath>
            </overlay>
          </overlays>
        </configuration>
      </plugin>

    </plugins>
  </build>

  ...

</project>
```

Example 3.1. Sample POM excerpt for an importer web application

Your web application module depends on the `importer-component` component and on the `importer-config` module. The configuration is put as an overlay over the web application.

3. Rename the `cm-xmlimport.properties` file in the `importer-config` module to the name of the servlet context of the importer (in the example POM, this would be importer.properties).
4. Configure the importer in the properties file as described in Section 3.1, "General Configuration" [16].

> **NOTE**
>
> When you deploy the Importer as a web application, you have to define all Java classes (for example the `multiResultGeneratorFactory` or transformers) in the properties configuration file with fully qualified names.

5. Build your importer web application with `mvn install`.

After the build, you will find an importer WAR file in the `target` directory. You can deploy the WAR file into your own Tomcat, or simply use the Maven Tomcat plugin as described below.

Starting the Importer

When you have build the importer web application, you can start the importer using the Maven Tomcat plugin from the module's directory.

```
mvn tomcat7:run <ImporterName>
```

The importer starts importing all content from the configured inbox.

3.5 Troubleshooting

After import, content items remain checked out by the import user, or their content is empty.

Possible causes:

The import does not operate strictly transactionally. If an error occurs between the actions *Create document*, *Edit document*, *Check in document*, the repository remains in an intermediate state.

Possible solutions:

The actual cause of the problem is indicated in the log file. Import the documents again after solving the problem. After a completely successful import, the repository again has the expected state.

4. XML Importers

XML as an exchange format

The Extensible Markup Language, XML, is a standard for platform- and software-independent description of structured files and data published by the World Wide Web Consortium (W3C). Most content suppliers recognize XML as a simple yet powerful exchange format and deliver their content in XML format.

CoreMedia naturally also supports XML and provides an importer for XML files of any format and map them to your CoreMedia content types.

An XML import consists of the following steps:

1. Read in the XML files
2. Transform to CoreMedia XML (configurable)
3. Check consistency
4. Submit the documents to *CoreMedia CMS*

Stand-alone or application importer

The importer can either be started as a command-line tool to run a single import or as an application that runs in the background permanently and checks at regular intervals whether new documents for import have arrived. If there are new documents, they are read in. See Section 3.3, “Deployment and Operation of a Standalone Importer” [21] and Section 3.4, “Deployment and Operation of a Web Application Importer” [22] for details

Source documents

For every individual import the source documents to be imported must first be gathered together. A source document can exist as a file for instance, be downloaded over the net or be generated dynamically. See Section 4.2, “Source Documents” [36] for details about the source documents. This is configurable via a Java programming interface (in the following called Importer API or API for short). The standard case is that documents exist as files. It is already covered by the classes of the API. If source documents were not originally created for CoreMedia import, they do not yet correspond to the CoreMedia XML format directly supported by the importer.

Convert into the CoreMedia XML format

After the source documents have been read in, a configurable step of conversion into the CoreMedia XML Format is carried out. See Section 4.1, “The CoreMedia XML Format” [27] for details about the CoreMedia XML format. As standard, the importer supports XSLT transformations and transformations based on regular expressions. However, the Importer API also enables you to insert your own transformers which correspond to the "Java API for XML Processing" (a standard API from Oracle, called JAXP in the following).

Process in DOM format or as a stream

The transformers can process the source documents either in Stream or in DOM format. The Stream format in particular enables parsers for the insertion of documents which

are not in the XML format. Furthermore, a transformer can either process each source document individually (such as an XSLT transformer) or transform all the source documents in one step. (Unfortunately, the latter exceeds the possibilities of JAXP, so that some expansions have been defined.) See Section 4.3, "XML Transformation" [42] for details about transformers.

Chaining transformers

When configuring an importer, you can place multiple transformers one after the other, which are then executed in sequence on import and each receive the result of the previous transformer as input document. For a non-XML format you can enter a thin parser as the first transformer, which transforms the document into an XML format close to the source format. Next, an XSLT transformer can transform this XML format into CoreMedia XML, and finally a CoreMedia filter can allocate the document to the correct repository path.

After processing by the last transformer, the documents must be in CoreMedia XML format. The details of this format depend on the content type. The transformer must be suitably adjusted.

Check consistency of created XML

The configurable phase of the importer ends when the CoreMedia XML format has been created. Now the structure of the content items created is validated. This also includes consistency checks which go beyond the DTD, especially conformity of the content items to the corresponding content types and the referential integrity.

If the documents are proven consistent the import is executed, that is the documents are incorporated into the *CoreMedia CMS*.

Section 4.4, "Example" [48] gives you a complete example of the import and transformation of an XML file.

4.1 The CoreMedia XML Format

XML is a generic format. However, the importer cannot guess the meaning of various tags, but requires a certain format - the CoreMedia XML format.

To achieve a visual separation between XML elements and attributes, the elements are written in the following in pointed brackets: <element>. The attributes are written in italics: *attribute*.

Files here are XML files or XML documents while document means documents in the sense of *CoreMedia CMS*.

4.1.1 Structure of the CoreMedia XML Format

The following example shows the construction of a CoreMedia XML file:

```
<coremedia>
  <document name="MyDocument" id="1" type="Text"
   path="/News/Financial/">
    <version number="1">
      <string name="Headline" value="New market on downward
       trend"/>
      <text name="Text">
        <div>
          <p>Nemax closed with losses.</p>
        </div>
      </text>
      <linklist name="Images"><link idref="pic1"/></linklist>
      <action name="approve"/>
    </version>
  </document>

  <document name="NemaxChart" id="pic1" type="Image"
  path="/News/Financial/Charts">
    <version number="1">
      <blob name="original" mimetype="image/jpeg"
      href="nemaxchart.jpg"/>
    </version>
  </document>
</coremedia>
```

Example 4.1. CoreMedia XML

CoreMedia XML files essentially match the CoreMedia DTD but the importer does not support the full extent of the DTD. There are both limitations and expansions.

The root element is `<coremedia>`. As child elements, multiple <document> elements can arise. Each <document> element describes a document. The <document> elements contain in turn one or more <version> elements. The <version> elements contain child elements which can be divided into two groups: field elements and action elements. They describe the content and status of the document. Section 4.1.4, “Field Ele-

ments" [31] describes the field elements and Section 4.1.5, "Action Elements" [35] the action elements in detail.

4.1.2 IDs

Each document is uniquely determined by the value of its *id* attribute. It can be referenced to via this ID by other documents. In the example below, you can see that the document called `MyDocument` contains a link `<link idref="pic1"/>` to the second document.

The importer supports two types of ID:

- target IDs
  Target IDs refer to documents which already exist in the *CoreMedia CMS* repository. They begin with the prefix **target:** followed by the actual document ID which is always a positive even number.
- internal IDs
  They are only valid within the consistency check of an import procedure and are later mapped to target IDs by the importer. They begin with the prefix **internal:** followed by the actual document ID which can be a string, containing any character allowed for NMTOKEN. The internal document ID must be unique in the set of documents to import.

If the prefix is missing, the type of the ID is determined in the following way:

1. The system checks whether the ID occurs in the import set or not. If a document can be found, the ID is treated as internal.
2. The system checks for the ID in the CoreMedia repository. If a document can be found, the ID is treated as target.
3. If no document with this ID was found at all, the import will fail.

The documents for import must form a closed set with regard to their referential integrity, that is, no documents can be imported which refer to unknown IDs. For internal IDs, this means that the referenced document must also be contained in the import group. A document which is referenced with a target ID must already exist in the *CoreMedia CMS* repository.

The following example is intended to illustrate this - for clarity reasons it is limited to the elements and attributes relevant in this context.

```
<coremedia>
  <document name="MyDocument" id="1" >
    <version>
      <linklist name="Images">
         <link idref="pic1" />              <!-- (1) -->
         <link idref="not_here" />          <!-- (2) -->
         <link idref="target:2468" />       <!-- (3) -->
```

```
        <link idref="3456"/>                <!-- (4) -->
      </linklist>
    </version>
  </document>

  <document name="pic1" id="pic1">
  </document>
</coremedia>
```

Example 4.2. IDs

Case 1: The document with the internal ID *pic1* is also contained in the import group: the link is valid.

Case 2: A document with the internal ID *not_here* is not contained in the import group. It cannot be a target ID either, because it is not a numerical: the import would therefore fail.

Case 3: If, at the time of the import, there is already a document with the ID 2468 in the repository, then the link is correct, otherwise the import would fail.

Case 4: A document with the internal ID *3456* is not contained in the import group. If, at the time of the import, there is already a document with the ID *3456* in the repository, then the link is correct, otherwise the import would fail.

The documents to import can be distributed over multiple XML files. Documents which are connected via a reference do not necessarily have to be in the same file. It is only important that all files with referenced documents are available at the start of the import process.

4.1.3 Container Elements

The main components of CoreMedia XML are the elements *<coremedia>*, <document> and <version>. These elements provide the structure of documents. The content is held in field elements below <version> elements.

`<coremedia>`

`<coremedia>` is the root element. It contains, as children, the documents for import.

Attributes:

`xml:base` gives the basis URL for the `href` attributes which refer to resources. The `href` attributes can then be given relative to this URL. Setting `xml:base` is optional; the default value is the file URL of the XML file.

<document>

<document> stands for a document in *CoreMedia CMS*. <document> elements can be direct children of `<coremedia>` or specified so to speak "inline" in <linklist> elements.

Attributes:

name and *path* give the name of the document under which it is stored in *CoreMedia CMS*. Both attributes must be set. *path* is always interpreted as an absolute path. If the importer finds a document with the same name and the same document type in the target directory a new document version is created. If the existing document has a different type, the import fails. If no such document exists, a new document is created.

type describes the content type of the document and must also be set. The permissible values are the types which you have defined when configuring *CoreMedia CMS*.

id identifies the document. Other documents may use the ID to reference this document (see Section 4.1.2, "IDs" [28]). To satisfy the DTD, the ID must be entered, even when the document is not referenced. For handling IDs, the following rules apply:

- Internal IDs are automatically mapped to the corresponding target IDs by the importer. How this mapping occurs in detail depends on whether the document is mapped, via *name* and *path*, to an existing or to a new document.
- The explicit allocation of new target IDs is not possible via import and is rejected by the importer.
- If the value is a valid target ID and the given *name* and *path* does not match the position of the existing document, the document is moved to the new position. If the new position is already occupied by another document, this is not overwritten and the import fails.
- A target ID which designates an existing document of a different content type results in an import failure.

<version>

A document in *CoreMedia CMS* usually consists of several versions. On import of a document, new versions of the document are created. All content information of a document belongs to a certain version. Correspondingly, <version> is the only permissible type for child elements of <document>. A <document> can contain multiple <version> elements, that is the complete history of a document can be imported in one step. The <version> elements contain the actual content and the state in the form of field and action elements.

Attributes:

number is irrelevant, since new versions are always created on import and the successive number results automatically from the existing versions of the document. Reimport of an existing version via the number is therefore not possible. Due to the DTD, *number* must be set nevertheless. If you have only one version element use "1" as the attribute value.

4.1.4 Field Elements

CoreMedia CMS does not run without specifying content types and their fields (see the *Content Server Manual*), for example

```
<XmlGrammar Name="coremedia-richtext-1.0" Root="div"
SystemId="lib/xml/coremedia-richtext-1.0.dtd"/>
<DocType Name="Text">
    <StringProperty     Name="Headline" Length="200"/>
    <XmlProperty   Name="Text" Grammar="coremedia-richtext-1.0"/>
    <IntProperty        Name="Priority"/>
    <StringProperty     Name="Source" Length="20"/>
    <DateProperty       Name="AutoDeletedate"/>
    <DateProperty       Name="AutoPdate"/>
    <LinkListProperty   Name="Image" LinkType="Image"/>
</DocType>
```

Example 4.3. Content type

If the property `import.enforceCompleteVersion` in the configuration file of the importer is set to "true" (default), then the values for all fields must be provided for an import of a document of this type. For each document property (`StringProperty`, `XmlProperty`, ...), there must be a corresponding XML element (`string`, `text`, ...). If one declared field is missing an error message is generated, and if there are two values for one field, the first is overwritten by the last.

If the property `import.enforceCompleteVersion` in the configuration file of the importer is set to "false", it is not necessary to provide all elements. See the Administrator Manual for a description of the property.

Before the format of each field element is described in detail, the following example shows a typical document of the above content type:

```
<document name="MyDocument" id="1" type="Text"
  path="/News/Financial/">
  <version number="1">
     <string name="Headline" value="New Market on descent"/>
     <text name="Text">
       <div>
         <p>Nemax closed with losses.</p>
       </div>
     </text>
     <integer name="Priority" value="42"/>
     <string name="Source" value="Stockresearch"/>
     <date name="AutoDeletedate"
           date="2001-01-29" time="00:00"
           timezone="Europe/Berlin" />
     <date name="AutoPdatum"
           date="2001-01-22" time="22:30"
           timezone="Europe/Berlin"/>
     <linklist name="Images"><link idref="pic1"/></linklist>
  </version>
</document>
```

Example 4.4. A document

The order of the field elements in the example corresponds to the order of the fields in the definition of the content type *Text*, but this is not required, as any order can be used.

CoreMedia XML supports the document fields of type structured and unstructured text, number, date, blob and linklist. These are now described in detail. All field elements have an attribute *name* for the document field. The attribute is not explicitly listed below.

<text>

Corresponds to an `<XmlProperty>` in the document type definition. The CoreMedia DTD allows PCDATA as content, the format of the `<text>` content deviates at this point from the DTD: the content of `<text>` must conform to the grammar DTD defined for the field.

<string>

Corresponds to a `<StringProperty>` in the document type definition. The attribute *value* holds the string content.

```
<string name="Heading" value="New market on descent/>
```

<integer>

Corresponds to an `<IntProperty>` in the document type definition. The attribute `value` contains the number.

```
<integer name="Priority" value="42"/>
```

<date>

Corresponds to a `<DateProperty>` in the document type definition. In addition to the `name` attribute, there are three further attributes:

- `date`: the date must be entered in the format *yyyy-mm-dd*, such as 2001-11-06
- `time`: the time must be entered in the format *hh:mm*, such as **09:00** or *hh:mm:ss*, such as **23:59:59**
- `timezone`: the format corresponds to the Java 2 API (`java.util.TimeZone`), such as Europe/Berlin or GMT.

All three attributes are optional. If `date` is left out, the whole date is undefined. On the other hand, the default values *00:00* and `java.util.TimeZone.getDefault()` are used for `time` and `timezone, resp`.

```
<date name="AutoPdate" date="2001-01-01" time="12:15"
timezone="Europe/Berlin"/>
```

<blob>

Corresponds to a `<BlobProperty>` in the document type definition. The element has two further attributes:

- `mimetype`: the MIME type must match the MIME type defined in the corresponding `<BlobProperty>` element in the definition of the document type. Wildcards, like `image/*` are not allowed.
- `href`: the URL of a blob. It is given either absolute or relative to the value of the `xml:base` attribute of the `<coremedia>` element.

```
<blob name="onlineImage" mimetype="image/jpeg"
href="chart.jpg"/>
```

<linklist>

Corresponds to a `<LinkListProperty>` in the document type definition. It can have both `<document>` and `<link>` child elements. `<document>` children are imported like other `<document>` elements. The empty element `<link>` has an attribute `idref` with an ID value. The consistency rules given above apply to this `idref` attribute. All documents of a `<linklist>` must have a type that corresponds to the type specified with the `LinkType` attribute of the corresponding `<LinkList Property>`.

```
<linklist name="Images"><link idref="pic1"/></linklist>
```

The content of `<text> is` included in two ways:

- Embedded text
  Place the content in the `<text>` element.
  Example:
  `<text name="Text"> <div> <p>...Flowing text...</p> </div></text>`
- A separate XML resource
  The content is stored in a separate resource and referred to by URL given in the `href` attribute. The XML in the resource must be well-formed. The root element depends on the DTD specified for the field type. The URL in the `href` attribute is relative to the main XML document containing the `<text>` element, or absolute. If the resource is a file, do not store the file in the importer `inbox` directory when the default `SubDirGenerator` is configured in `cm-xmlimport.properties`. Otherwise, the importer will try to import the file and fail.
  Example:
  `<text name="Text" href="../href/text.txt"/>`

The importer selects the embedded content if <text> element contains embedded content and a `href` attribute.

Links in XML fields

You can define XML fields that conform to any DTD. To link to documents from this XML fields use the attribute `xlink:href`. The attribute value must start with the prefix `coremedia:///cap/resources/` followed by an internal or target id as shown below:

```
<text name="Text">
 <div><p>See<a xlink:href="coremedia:///cap/resources/info">info
 </a></p></div>
</text>
```

Example 4.5. Example for the `xlink:href` attribute

This example contains a link to a document with the internal id `info`. The internal id must not contain a "/" character. To link to a field of a document append a "/" character and the name of the field.

```
<text name="Text">
 <div><p>See<a xlink:href="coremedia:///cap/resources/info/xml">
 info</a></p></div>
</text>
```

Example 4.6. Example for a property link

This example contains a link to the field *`xml`* in the document with the internal id info. The field name may not contain a "/" character. If you use a link in an `<img>` tag, the link must point to the `BlobProperty` of the document. The following example contains links with target ids:

```
<text name="Text">
 <div><p>see
 <a xlink:href="coremedia:///cap/resources/target:2468">info1
 </a>and
 <a xlink:href="coremedia:///cap/resources/target:1234/xml">
 info2</a>
 </p></div>
</text>
```

Example 4.7. Example for links with target ids

XML fields that conform to the now deprecated `coremedia-sgmltext.dtd` may link to documents with the elements `<HOX.LINK>` and `<HOX.IMAGE>` as shown in the following example:

```
<text name="Sgmltext">
 <ROOT> <P>See<HOX.LINK ID="info">info</HOX.LINK>
 </P></ROOT>
</text>
```

Example 4.8. Example for the `HOX.LINK` element

Both internal and target ids can be used in the attribute `ID`.

4.1.5 Action Elements

While the field elements describe the contents of the document, the <action> elements define its state. Action elements have the following attributes:

- *name* is the name of the action and can have one of the following values: **publish**, **approve**, **delete** (see `constant-values` for the publication result codes). According to the DTD the **edit** value is allowed but the importer ignores it.
- *user* is ignored as well. All actions are carried out by the "importer" user (see the Content Server Manual).
- *date*, *time*, *timezone* are ignored as well.

4.2 Source Documents

Having dealt with the CoreMedia XML format in the previous section, you will see here how to provide source documents for import.

This section contains important points which must always be considered, followed by the description of the standard configuration. If the standard configuration is not suitable in your case, the last section tells you how to use the Importer API to meet your requirements.

CAUTION

Please do not use special characters in resource names. They can cause trouble in publication (esp. /, \, :, *, ?, !,", <, >).

The server checks for the following resource names:

- Names, which start or end with a space,
- Names, which only consist of one or two dots,
- Names, which are empty,
- Names, which contain the slash "/".

You can't create resources with such names.

4.2.1 General

Since the term "document" is inevitably used rather often in this manual, it is important for understanding this chapter to distinguish between the following types of documents:

- Source document: a document of any format for import, for example a file or a generated DOM tree. This term includes both the input data and the intermediate results of the transformers.
- XML document: a document of any XML format, for example CoreMedia XML.
- CoreMedia document: a document in the sense of the *CoreMedia CMS*. For example, a <document> in the CoreMedia XML format represents a CoreMedia document.

There is no one-to-one relationship between source documents and CoreMedia documents. As you can see from the example of CoreMedia XML above, a source document can represent several CoreMedia documents. On the other hand, a transformer can generate a single CoreMedia document from the contents of several source documents.

Due to the requirements for consistency over documents (especially the referential integrity), in many cases it is not possible to import individual CoreMedia documents. Often, a group of CoreMedia documents which can only be imported together results from many source documents. The importer therefore offers the possibility of processing any number of source documents in one operation.

On the other hand, a large number of source documents significantly increases both the amount of memory required and the time required for consistency testing. Many operations with fewer documents are more efficient than fewer operations with many documents. Therefore, ideally, the source documents should be imported individually and then be combined into the smallest possible groups, if required by the referential integrity.

4.2.2 Source files

The standard configuration of the importer is set up for the situation that the source documents exist as files in certain directories. These directories are set in the configuration file, separated by semicolons:

```
### Path to inbox (may be relative to $COREM_HOME):
import.multiResultGeneratorFactory.property.inbox =
<my/inbox/directory1>;<my/inbox/directory2>;
<my/inbox/directory3>
```

Example 4.9. Inbox directories

Files which lie directly in the inbox are imported individually. If several source files should be imported in one operation, they must be combined in a subdirectory. Such subdirectories can have any desired name, only "bak" and "err" are reserved. The importer creates these two subdirectories and moves successfully imported source files and subdirectories to bak and failures to err.

At the time of the import, the files must be completely ready. In particular, all files must be present in subdirectories which should be imported in one operation. Therefore, both individual files and complete subdirectories should (under Unix) only be moved to the inbox directories in one complete step with `mv`, not by means of successive copying or writing.

If the inbox directories are empty, the importer goes to sleep. You can configure the sleeping time in seconds, using the property `import.multiResultGeneratorFactory.property.sleepingSeconds`. The special value "-1" means that the importer does not wait for new files, but only imports the current contents of the inbox directories once and then ends.

```
### Seconds to sleep between importer runs
import.multiResultGeneratorFactory.property.sleepingSeconds = -1
```

Example 4.10. Sleeping seconds

It may be the case that files are present in the inbox directories which should not be imported as source documents. A typical example for this are graphics which are referenced without path by a blob property in a CoreMedia XML document. On the one hand, such graphics must lie in the same directories, so that the importer finds them, but on the other hand should not be imported as independent source documents. For such cases, a further property can be entered in addition to `inbox` and `sleepingSeconds`: `filenameFilterClass`. The value of this property must be the name of a Java class which implements the `java.io.FilenameFilter` interface. If this property is specified, a file is only imported if its name is accepted by a `FilenameFilter` of this class. If your files have meaningful names, it is usually possible to decide according to the filename extension whether it is a source document or another file.

4.2.3 Document Sets

If it is not possible to supply the importer with source documents according to the principle described in the previous section, you can implement your own mechanism on the basis of the Importer API. Understanding this section requires knowledge of Java; in particular, the Importer API is based on JAXP.

You can find the Importer API on the CoreMedia documentation site at https://documentation.coremedia.com/cmcc-11.

Unless the package is explicitly given, the classes and interfaces mentioned in this section come from the `javax.xml` hierarchy or from `com.coremedia.publisher.importer`. There are no name conflicts between these packages.

The importer internally functions not with files but with document sets. Such a document set contains the source documents to be imported in one operation. A source document can be represented in various ways, indirectly via a URL or directly as a series of bytes or as a DOM tree.

The combination of these document sets is controlled by a class which implements the `MultiResultGeneratorFactory` interface. The name of this class is entered in the configuration file:

```
### The com.coremedia.publisher.importer.MultiResult
### Generator interface
### implementation to use
import.multiResultGeneratorFactory.class=
```

```
com.coremedia.publisher.importer.SubDirGeneratorFactory
```

Example 4.11. MultiResultGeneratorFactory

The class `SubDirGeneratorFactory` creates generators which implement the file logic described in the previous section. `SubDirGeneratorFactory` is included with the delivery of the importer and is preset in the example configuration file `properties/corem/cm-xmlimport.properties`. Instead, you can use your own `MultiResultGeneratorFactory`.

The class is instantiated by the importer with `java.beans.Beans.instantiate`. If the class supports `Properties` in the sense of `java.beans.BeanInfo` (see Oracle JavaBeans API Specification), these properties can be specified in the configuration file and are then set by the importer. For example, `SubDirGeneratorFactory` requires the properties `inbox` and `sleepingSeconds`:

```
### Path to inbox (may be relative to $COREM_HOME):
import.multiResultGeneratorFactory.property.inbox =
<my/inbox/directory>

### Seconds to sleep between importer runs
import.multiResultGeneratorFactory.property.sleepingSeconds = -1
```

Example 4.12. MultiResultGeneratorFactory

Such property entries have the format

`import.multiResultGeneratorFactory.property.<propertyName> = <propertyValue>`

and are specific in their meaning for the particular `MultiResultGeneratorFactory` implementation. Therefore, when configuring your own factory, you have free choice of names and number of properties. You are not confined to `inbox` and `sleepingSeconds`.

All properties are set by the importer as strings. For example, the class `SubDirGeneratorFactory` must transform the value of `sleepingSeconds` into a number.

After the properties have been set, the importer obtains the actual generator for the document set from the factory with `getMultiResultGenerator()`. In this situation, the use of factories has the advantage that the factory can use the properties to configure the generator in any desired way.

`getMultiResultGenerator()` must return a `MultiResultGenerator` object which supports the methods `fail`, `next` and `success`. `next` is called by the importer in order to create a new set of source documents. With `success` and `fail`, the importer informs the generator about the success or failure of the import

of the source documents delivered by the previous `next` command. After calling success or `fail`, the importer no longer accesses the source documents.

For example, on each `next` command, the generator created by `SubDirGeneratorFactory` delivers a file directly from the inbox directory or all files of a subdirectory. After the complete contents of the inbox directory have been imported, the generator delays the next execution of the `next` method by the time determined in the property `sleepingSeconds`, and then delivers the files which have newly arrived in the meantime to the importer. The `success` method moves the file or subdirectory to the `bak` directory, `fail` to the `err` directory.

If the next method of the generator returns null, the importer ends. In the normal case, however, it delivers a new document set in the form of a `MultiResult`. `MultiResult` implements the `Result` interface and therefore fits into the concept of JAXP next to `StreamResult`, `SAXResult` and `DOMResult`. A new empty `MultiResult` is created via `MultiResultFactory.getInstance().getMultiResult()`. New documents are added to `MultiResult` via `addNewResult`. There are two variants of this method:

```
void addNewResult(String systemId) throws Exception;

Result addNewResult(String format, String systemId)
throws Exception;
```

Example 4.13. `MultiResult.addNewResult`

The first variant is used for entering a document via a reference. The `systemId` must be an URL which can be read by the importer as an input stream, for example a file path.

With the second variant, the document data can be entered directly. The method returns a `Result` in which the data can be deposited. The parameter `format` determines whether the result is a `DOMResult`, a `StreamResult` or again a `MultiResult`. (SAX is not yet supported in this version). Valid values for format are `StreamResult.FEATURE`, `DOMResult.FEATURE` or `MultiResult.FEATURE`. In this variant, the `systemID` is not used as a data source, but, according to the JAXP concept, only as the basis for resolution of relative URLs. Depending on the result type, the generator can store the source document with `DOMResult.setNode` or `StreamResult.setOutputStream().write`, or construct a more deeply nested document hierarchy with `MultiResult.addNewResult`. (The nesting plays no role for the importer; it could at most be used by special transformers.)

A final example shows a simple `next` method which returns one file of a directory on each call.

```
File inbox = new File("/tmp/inbox");
File[] files = inbox.listFiles();
int index = 0;
```

```
public MultiResult next() {
    try {
        if (index < files.length) {
            MultiResult mr = MultiResultFactory.getInstance().
 getMultiResult();
            mr.addNewResult(files[index++].getAbsolutePath());
            return mr;
        }
    } catch (Exception e) {
        System.out.println("Something went wrong!");
    }
    return null;
}
```

Example 4.14. next

If no transformers are entered in the configuration file, the sets of source documents delivered by `next` are imported directly. Of course, this only works if the documents are CoreMedia XML documents matching the content types of the *CoreMedia CMS*. Typically, however, a transformation of the documents is necessary to achieve the correct format. This is the subject of the next section.

4.3 XML Transformation

If your XML files were not explicitly created for CoreMedia import, they probably have a different format and must be transformed into the CoreMedia XML format first. The importer supports this with a multi-stage configurable transformation.

If no transformers are inserted in the configuration file, the sets of source documents delivered by next are imported directly. Of course, this only works if the documents are CoreMedia XML documents matching the content types of the *Content Server*. Typically, a transformation of the documents is necessary to achieve the correct format. This is subject of the next section.

If your documents are not in XML format and have a regular text structure, you can create XML documents out of the texts using regular expressions as known from the Perl5 programming language. Furthermore, you can use regular expressions to structure PCDATA sections into XML documents.

If your XML documents do not yet correspond to the CoreMedia XML format, they can be transformed with an XSLT style sheet. You only have to provide the style sheet and the importer carries out the transformation automatically.

Limitations of XSLT

The power of XSLT also has its limits. At the latest when the transformation has to be carried out over multiple documents (for example to realize relationships between articles and their teasers) or when the source documents are not XML documents, XSLT does not help any further. In addition, some transformations within an XML document can only be carried out awkwardly with XSLT, if they run counter to the declarative paradigm. In such cases you have the possibility with the Importer API of integrating your own special transformers into the importer. Your transformer can either process each source document separately or all of them at once. Furthermore, you can access the source documents in the Stream or DOM format and then return the transformed document as Stream or as a DOM tree. However, access to a source document in the DOM format requires that it is an XML document. On the other hand, Stream access is possible for all documents.

Using multiple transformers

As already mentioned, you can combine multiple transformers in order to achieve the desired end result in the form of CoreMedia XML. The order of the transformers is determined in the configuration file of the importer, and during operation each transformer starts with the result of its predecessor. The first transformer directly accesses the document delivered by the document generator (see previous section). The flexibility regarding the access (Stream or DOM, individual or complete set) is unaffected by the order. For example, the first transformer can return a deeply nested `MultiResult` with DOM trees, but the second can start with individual Stream documents. The necessary reformatting is carried out automatically by the importer.

The following sections deal with these transformation possibilities in detail.

4.3.1 Configuration

Configuration of the transformer is done in the importer configuration file. All properties relating to transformers have hierarchical names and begin with the prefix `import.transformer`, for example

```
import.transformer.10.class=XsltTransformerFactory
```

Example 4.15. Transformer Configuration

The configuration of a transformer mostly is too complex to store all necessary information in a single property, and therefore consists of several properties. The next component of the name after the prefix is a number. All transformer properties with the same number refer to the same transformer.

The number has a further function: it determines the order in which the transformers are executed during operation.

The number is followed (possibly hierarchically subdivided further) by the actual name of the property. For each transformer, the corresponding Java `class` must be specified with "class". Furthermore, for logging purposes, each transformer should be given a name. All further properties depend on the particular type of the transformer and describe, for example, style sheets or filter data. These details will be dealt with in later sections in relation to the specific transformers.

The file `corem/properties/cm-xmlimport.properties` which serves as an example for importer configuration, contains a complete (commented out) example for the configuration of an XSLT transformer and a user-defined transformer.

CoreMedia recommends not to use too many transformers, since errors can easily creep in when you make changes to one transformer and overlook effects of the intermediate format for the following transformer. Typically, you should use one or more XSLT style sheets.

NOTE

When you deploy the Importer as a web application, you have to define all Java classes (for example the `multiResultGeneratorFactory` or transformers) in the properties configuration file with fully qualified names.

4.3.2 XSLT

The Importer allows you to define XSLT style sheets which transforms XML files of your format into the CoreMedia format. This style sheet is automatically executed on import.

The XSLT interpreter used in *CoreMedia* requires style sheets corresponding to the W3C Recommendation of the 16.11.1999. This particularly concerns the XSL namespace in the `<xsl:stylesheet>` element, which must be set to

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

Older XSL editors still generate the variant from an earlier Working Draft:

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform/1.0"
```

This is out of date and not supported by *CoreMedia*.

The value of the `class` property in the configuration file for an XSLT transformer is *`XsltTransformerFactory`*. Furthermore, the desired style sheet must be defined with `property.stylesheet`. The path may either be absolute, or relative to the installation directory (*Standalone Importer*) respectively the `WEB-INF` directory (*Web Application Importer*).

```
import.transformer.10.class=
com.coremedia.publisher.importer.XsltTransformerFactory
import.transformer.10.name=My Stylesheet
import.transformer.10.property.stylesheet=/path/to/stylesheet.xsl
```

Example 4.16. XSLT configuration

4.3.3 User-defined Transformers

If transformations with regular expressions or XSLT are not sufficient, you can develop your own transformer in Java based on the Importer API. In this section you can find out more about formatting such transformers. The Importer API is closely related to the Java API for XML processing (JAXP), especially with the `javax.xml.transform` hierarchy. In some places, however, JAXP is not powerful enough, or too XSLT-specific for the requirements of CoreMedia, so that CoreMedia had to define some extensions.

In accordance with both JAXP and the document generator, transformers are not specified directly, but rather indirectly via factories. Since the `javax.xml.transform.TransformerFactory` is very XSL-specific, the Importer API defines a more general factory, the `GeneralTransformerFactory`.

Like the document generator, the transformer factories are instantiated with `java.beans.Beans.instantiate` and can be configured with properties in the sense of `java.beans.BeanInfo`. In the configuration file, such a property is entered with the prefix of the importer (with number), the keyword `property` and the actual name of the property. For example, the `XsltTransformerFactory` introduced above is passed to the style sheet via such a property.

```
import.transformer.10.class=XsltTransformerFactory
import.transformer.10.name=My Stylesheet
import.transformer.10.property.stylesheet=path/to/stylesheet.xsl
```

Example 4.17. Bean Property

The `GeneralTransformerFactory` interface consists of two methods, `getTransformer` and `getFeature`.

The method `getFeature` is used in the sense of JAXP to find out whether the transformers created by this factory support certain source and result formats. For example, if your factory returns "true" for the call

```
factory.getFeature(DOMSource.FEATURE)
factory.getFeature(StreamSource.FEATURE)
```

Example 4.18. `getFeature`

this means that the transformers created with `factory.getTransformer(name)` accept both a `StreamSource` and a `DOMSource` as input documents. Therefore, if your transformer does not contain an XML parser, but is confined to input documents in DOM format, the factory should return "false" for `getFeature(StreamSource.FEATURE)`. On the other hand, if the transformer processes non-XML documents, the factory must return "false" for `getFeature(DOMSource.FEATURE)`, because otherwise the importer tries to parse the supposed XML document, naturally leading to an error.

This is also true for `DOMResult.FEATURE` and `StreamResult.FEATURE`. If your transformer works internally with a DOM tree, it should return it as such, and not as Stream. If the next transformer in the chain expects a DOM tree as input, this saves a new parsing of the document.

In this version, SAX is supported neither on the source nor on the result side.

If a transformer should be only called once for the whole document set rather than for each source document individually, its factory must return "true" for `getFeature(MultiSource.FEATURE)`. In contrast to `DOMSource` and `StreamSource`, `MultiSource` does not belong to JAXP, but is an extension by CoreMedia. If the transformer should return multiple documents, the factory must return "true" for `getFeature(MultiResult.FEATURE)`. `MultiResult` has already been introduced in connection with the document generator.

The source and result formats of a transformer are completely independent from each other. For example, you can develop transformers which create a single Stream document from all source documents, or which produce a set of DOM trees from one Stream document.

To make the creation of factories easier, the Importer API contains the class `GeneralTransformerFactoryImpl`, which implements the `GeneralTransformerFactory` interface. `GeneralTransformerFactoryImpl` can be configured with three properties: `transformerclass`, `sourceformat` and `resultformat`. `transformerclass` sets the class of the actual transformer. This class must be a derivative of `javax.xml.transform.Transformer` (for more details see below), and must have a default construction without parameters. `sourceformat` and `resultformat` give the source and result format. Valid values are *`stream`*, *`dom`* and *`multi`*. (Note: these values do not match those of the corresponding FEATURE constants. The latter are opaque and are therefore not suitable for configuration via property files.)

The configuration of a transformer of the class `MyTransformer`, which should be called individually for each document, which processes the source documents as Stream and which produces multiple result documents, therefore appears as follows:

```
import.transformer.20.class=GeneralTransformerFactoryImpl
import.transformer.20.name=My special transformer
import.transformer.20.property.transformerclass=
com.mycompany.MyTransformer
import.transformer.20.property.sourceformat=stream
import.transformer.20.property.resultformat=multi
```

Example 4.19. Configuration of a transformer

`GeneralTransformerFactoryImpl` has further features: the transformers instantiated with this class automatically receive some parameters without these having to be explicitly configured. In particular, these are

- the name of the transformer (that is the value of the `import.transformer.xx.name` property)
- a log object which the transformer can use for log outputs
- a CoreMedia object which enables access to the CoreMedia repository

Details of the classes of these objects can be found in the Importer API.

By using `getTransformer` the importer calls up an instance of the transformer from the factory. As name argument, the importer passes `getTransformer` the name entered in the configuration file for this transformer with the `name` property (in the example above, therefore, "My special transformer"). The factory can use this name, for example, for log outputs. However, the name is not intended for information that is semantically more important. For this purpose there are properties.

The transformer itself is an object of the `javax.xml.transform.Transformer` class. The decisive abstract method of this class which you must implement within the framework of a derivation of `Transformer` in order to realize your transformation is `transform`. In addition to this, `Transformer` has a few other abstract methods whose function, however, is precisely specified by JAXP. To save you work, these methods are already implemented in the Importer API: if you derive your transformer from `com.coremedia.publisher.importer.AbstractTransformer`, rather than directly from `javax.xml.transform.Transformer`, you only need to implement the `transform` method.

According to the `getFeature` information of the factory, the importer calls the `transform` method either individually for each source document in the form of a `StreamSource` or `DOMSource`, or once for all source documents in the form of a `MultiSource`. In both cases, the importer calls the `getTransformer` method of the factory only once and transforms all documents with the same instance of the transformer.

4.4 Example

This chapter, contains a complete example which illustrates the use of the different transformers. The source documents correspond to the DTD http://xmlnews.org/dtds/xmlnews-story.dtd. The example includes two transformers:

- A special transformer will generate the <base> element (optional according to the DTD) in each document if it does not exist.
- An XSLT style sheet will transform the `XMLNews-Story` documents into CoreMedia XML documents of type Text.

4.4.1 DOM Transformation

In the first step of the example a special transformer is created that generates any missing `<base>` elements in the source documents. These elements are optional according to the DTD, but in the framework of this example are vital for further transformation. The following section of an example document illustrates the position at which the `<base>` elements are located:

```
<?xml version="1.0"?>

<nitf>

<head>
<title>Snow, Freezing Rain Batter U.S. Northeast</title>
<base href="xmlnewssample.xml"/>
</head>

</nitf>
```

Example 4.20. Element

In this example, the `href` attribute of the `<base>` element becomes the ID of the document. Therefore, all generated `<base>` elements within an importer process must have different `href` values. The transformer should ensure this by means of a configurable prefix, extended with a consecutive number. The transformer processes all source documents at once. Here is the Java code of this transformer:

```
import javax.xml.transform.*;
import javax.xml.transform.dom.*;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.w3c.dom.*;

import com.coremedia.publisher.importer.*;
```

```
public class BaseMaker extends AbstractTransformer {

   private static final Logger LOG
     = LoggerFactory.getLogger(BaseMaker.class);

    // Returns the leftmost child of parent which is an Element
   // of the specified tag name.
    protected Element getChild(Node parent, String tag) {
        NodeList children = parent.getChildNodes();
        for (int i=0; i<children.getLength(); i++)
             if (children.item(i).getNodeType() ==
Node.ELEMENT_NODE &&
                tag.equals(children.item(i).getNodeName()))
                return (Element)children.item(i);
        return null;
    }

    protected void makeBase(MultiSource src, MultiResult res)
     throws Exception {
        // loop over the source documents
        for (int i=0; i<src.size(); i++) {
            // get a source document in DOM format
            Source domsrc = src.getSource(i, DOMSource.FEATURE);

            // check for a base Element
            Document doc = (Document)((DOMSource)domsrc).getNode();
            Element nitf = doc.getDocumentElement();
            Element head = getChild(nitf, "head");
            Element base = getChild(head, "base");

            if (base == null) {
                // no base Element yet, generate one
                base = doc.createElement("base");

                // don't forget to configure the factory to
                //pass the prefix
                base.setAttribute("href",
 getParameter("prefix").toString() + i);
                head.appendChild(base);
            }

            // modified or not, append the document to the
            // MultiResult
            String systemId = domsrc.getSystemId();
            Result domres = res.addNewResult(DOMResult.FEATURE,
 systemId);
             ((DOMResult)domres).setNode(doc);
        }
    }

    public void transform(Source src, Result res) throws
     TransformerException {
        // Say "Hi!"...
        String name =
         getParameter(TransformerParameters.NAME).toString();
        LOG.info(name);

        try {
            // The factory assures that you have a MultiSource and
            // a MultiResult
            makeBase((MultiSource)src, (MultiResult)res);
        } catch (TransformerException exc) {
            throw exc;
        } catch (Exception exc) {
            throw new TransformerException(exc);
        }
    }
```

```
}
```

Example 4.21. `BaseMaker.java`

The `transform` method first creates a log output. The fact that the name is available to the transformer is a feature of the `GeneralTransformerFactoryImpl` factory, which you will use for instantiation of this transformer. The actual transformation is transferred to the `makeBase` method. `transform` only catches any exceptions and transforms these, if necessary, into `TransformerExceptions`.

Instead of transforming all documents at once, it would also be possible to administrate the consecutive numbers with a `static` variable and call the transformer individually for each document. This would make the loop over the source documents and the construction of the MultiResult unnecessary. However, this example is intended to show the use of the Importer API.

Next, a factory is required which instantiates and configures the transformer. `GeneralTransformerFactoryImpl` is not sufficient, because it does not support passing of the prefix to the transformer. The following extension puts this right:

```
import javax.xml.transform.*;
import com.coremedia.publisher.importer.*;

public class BaseMakerFactory extends
 GeneralTransformerFactoryImpl {
    private String prefix = null;

    public BaseMakerFactory() throws ClassNotFoundException {
      super(MultiSource.FEATURE, MultiResult.FEATURE, "BaseMaker");
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public Transformer getTransformer(String name)
    throws Exception {
        Transformer trf = super.getTransformer(name);
        trf.setParameter("prefix", prefix);
        return trf;
    }
}
```

Example 4.22. `BaseMakerFactory.java`

The transformer now only has to be entered in the configuration file of the importer. Since you had to change `GeneratorTransformerFactoryImpl` anyway, in order to support the prefix, the class of the transformer (`BaseMaker`), as well as the source and result formats, have also coded straight into `BaseMakerFactory`. In the configuration file, therefore, only a prefix needs to be specified in addition to the class and the name.

```
import.transformer.10.class=BaseMakerFactory
import.transformer.10.name=Create missing base elements
import.transformer.10.property.prefix=XmlNews
```

Example 4.23. Configuration

4.4.2 XSLT Transformation

In this step, a style sheet for the transformation of `XMLNewsStory` documents into CoreMedia XML documents with the document type *Text* is created. With the following example document you find out which information from the source format should be transferred to the target format, and then a style sheet is created.

```
<nitf>
  <head>
    <title>Snow, Freezing Rain Batter U.S. Northeast</title>
    <base href="http://cool.dot.com/news/xmlnews.xml"/>
  </head>

  <body>
    <body.head>
      <hedline>
        <hl1>
          Snow, Freezing Rain Batter
          <location>
            <country>U.S.</country>
            <region>Northeast</region>
          </location>
        </hl1>
      </hedline>
      <byline>
        <bytag>By Matthew Lewis</bytag>
      </byline>
      <dateline>
        <location>
          <city>HARTFORD</city>
          ,
          <state>Conn.</state>
        </location>
        <story.date>Friday January 15 12:27 PM ET</story.date>
      </dateline>
    </body.head>
    <body.content>
      <p>Snow and freezing rain punished the
        <location>
          northeastern
          <country>United States</country>
        </location>
         for a second straight day on
        <chron norm="19990115">Friday</chron>
         , causing at least five weather-related deaths,
         closing airports and spreading misery from
        <location>
          <city>Washington</city>
           ,
          <state>D.C.</state>
        </location>
         , to
        <location>
          <country>Canada</country>
```

```
        </location>
          .
        </p>
      </body.content>
    </body>
</nitf>
```

Example 4.24. An `XmlNews` document

In order to keep the style sheet simple, it is assumed that some elements are generally available although they are optional according to the DTD. The style sheet should execute the following obvious mappings:

- The heading results from the contents of `/nitf/body/body.head/hedline/hl1`.
- The contents of `nitf/body/body.content` should be imported as text.
- For importing a document, you need a name and an ID. Since the document contains no element with suitable content for this, you simply take the filename.

The content should not simply be adopted as pure text but be sensibly matched to the structures of our `coremedia-richtext-1.0.dtd` DTD:

- The `<p>` paragraphs of the `xmlnews` document are adopted 1:1 as <p> in `coremedia-richtext-1.0.dtd`.

While paragraphs and headings are general standard building bricks of any document, the `xmlnews` inline markup within <p>, for example <location>, is application-specific. Therefore, there are no adequate `coremedia-richtext-1.0.dtd` elements for this. Nevertheless, you want to save the information:

- The `xmlnews` inline markup is matched to `<SPAN>` elements whose `CLASS` attribute is set to the name of the original element (such as **location**).

This should be enough for an example of the functionality. The remaining components of our **Text** document will be filled with default values.

Before you deal with the actual transformation, define a utility function and a variable in which you first save the filename:

```
<xsl:template name="fetchFilename">
  <xsl:param name="filename">unkown</xsl:param>
  <xsl:choose>
    <xsl:when test="contains($filename,'/')">
      <xsl:call-template name="fetchFilename">
        <xsl:with-param name="filename">
          <xsl:value-of select="substring-after($filename,'/')"/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select ="$filename"/>
    </xsl:otherwise>
```

```
  </xsl:choose>
</xsl:template>

<xsl:variable name="filename">
  <xsl:call-template name="fetchFilename">
    <xsl:with-param name="filename">
      <xsl:value-of select="/nitf/head/base/@href"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:variable>
```

Example 4.25. Filename

The attribute `href` of the element `/nitf/head/base` contains the complete URL of the document. The `fetchFilename` function recursively cuts off one level of the path using the "/" sign, until only the filename is left.

Now you begin top-down with the transformation templates. Use the root as the entry point for generating the *<coremedia>* element:

```
<xsl:template match="/">
  <coremedia>
    <xsl:apply-templates select="nitf"/>
  </coremedia>
</xsl:template>
```

Example 4.26. coremedia element

The `<nitf>` element is mapped to a `<document>` element:

```
<xsl:template match="nitf">
  <document>
    <xsl:attribute name="type">Text</xsl:attribute>
    <xsl:attribute name="path">Test/Xmlnews</xsl:attribute>
    <xsl:attribute name="name">
      <xsl:value-of select="$filename"></xsl:value-of>
    </xsl:attribute>
    <xsl:attribute name="id">
      <xsl:value-of select="$filename"></xsl:value-of>
    </xsl:attribute>

      <xsl:apply-templates select="body"/>
  </document>
</xsl:template>
```

Example 4.27. nitf/ document

You decided on the document type `Text` at the beginning. For simplicity reasons, set the fixed path `Test/Xmlnews` as the target directory. Set the name and ID to the already extracted filename.

Proceed down to `<body>` and generate a `<version>`.

```
<xsl:template match="body">
  <version>
    <xsl:attribute name="number">1</xsl:attribute>
    <xsl:apply-templates select="body.head/hedline/hl1"/>
    <xsl:apply-templates select="body.content"/>
    <xsl:element name="integer">
      <xsl:attribute name="name">Priority</xsl:attribute>
      <xsl:attribute name="value">42</xsl:attribute>
    </xsl:element>
    <xsl:element name="string">
      <xsl:attribute name="name">Source</xsl:attribute>
      <xsl:attribute name="value">known to editor</xsl:attribute>
    </xsl:element>
    <xsl:element name="date">
      <xsl:attribute name="name">AutoDeletedate</xsl:attribute>
    </xsl:element>
    <xsl:element name="date">
      <xsl:attribute name="name">AutoPdatum</xsl:attribute>
    </xsl:element>
    <xsl:element name="linklist">
      <xsl:attribute name="name">Images</xsl:attribute>
    </xsl:element>
  </version>
</xsl:template>
```

Example 4.28. body / version

The version number, *number*, is irrelevant for import and is simply set to 1.

In the version a corresponding field element must be generated for every field of your document type `Text`. Set `Priority` and `Source` to default values, while `AutoDeletedate`, `AutoPdate` and `Images` are left empty. Within this example, only the heading and the actual content should be taken from the source document. For this purpose, suitable templates for `body.head/hedline/hl1` and for `body.content` are called.

```
<xsl:strip-space elements="hl1"/>

<xsl:template match="hl1">
  <string>
    <xsl:attribute name="name">Heading</xsl:attribute>
    <xsl:attribute name="value">
      <xsl:value-of select="."/>
    </xsl:attribute>
  </string>
</xsl:template>
```

Example 4.29. Heading

The heading results directly from the textual content of the `<hl1>` element. Inline Markup is not taken into consideration here.

```
<xsl:template match="body.content">
  <text>
    <xsl:attribute name="name">Text</xsl:attribute>
    <div>
      <xsl:apply-templates select="p"/>
    </div>
```

```
  </text>
</xsl:template>
```

Example 4.30. Content

At this point, the transition from CoreMedia DTD to `coremedia-richtext-1.0.dtd` occurs. The template generates the CoreMedia field element for the document field *Text* and the `coremedia-richtext-1.0.dtd` element `<div>`, which is filled with `<p>` elements.

Due to the XSLT default templates working through elements recursively and copying text, the style sheet already produces correct CoreMedia XML in this version. However, you still want to transform the Inline Markup into `<span>` elements, and need a further template for that.

```
<xsl:template match="p/*">
  <span>
    <xsl:attribute name="class">
      <xsl:value-of select="local-name()"/>
    </xsl:attribute>
    <xsl:apply-templates/>
  </span>
</xsl:template>
```

Example 4.31. Inline Markup

If you are familiar with XPath, you will have noticed that only the direct child elements of <p> are handled with `match="p/*"`. This is deliberate, because `span` elements in `coremedia-richtext-1.0.dtd` may not be nested and therefore you cannot take nested markup into account with such simple means.

Our style sheet is ready now. Even if you have little experience with XSLT it should now be quite simple to obtain the author from the `<bytag>`, for example, and place it in a string field element of your document.

To make the importer automatically executing the style sheet enter it in the configuration file of the importer:

```
import.transformer.20.class=XsltTransformerFactory
import.transformer.20.name=XmlNews to CoreMedia
import.transformer.20.property.stylesheet=/path/to/xmlnews.xsl
```

Example 4.32. Configuration

When the style sheet is applied to our XML example document, the following file results:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<coremedia>
  <document type="Text"
      path="Test/Xmlnews" name="xmlnews.xml" id="xmlnews.xml">

      <version number="1">
      <string name="Ueberschrift"
            value="Snow, Freezing Rain Batter U.S.Northeast"/>
      <text name="Text">
            <div>
              <p>Snow and freezing rain punished the
           <span class="location">northeastern United States
      </span> for a second straight day on
          <span class="chron">Friday</span>, causing at least
         five weather-related deaths, closing airports and
         spreading misery from <span class="location">
         Washington, D.C.</span>, to
         <span class="location">Canada</span>.
              </p>
            </div>
      </text>
      <integer name="Prioritaet" value="42"/>
      <string name="Quelle" value="d. Red. bekannt"/>
      <date name="AutoLoeschdatum"/>
      <date name="AutoPdatum"/>
      <linklist name="Bilder"/>
    </version>
  </document>
</coremedia>
```

Example 4.33. A document

5. Configuration Property Reference

Different aspects of the Importer 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 contain the properties that are relevant for the *Importer*:

- Table 3.41, "Properties of the cm-xmlimport.properties file" in *Deployment Manual* contains properties for the configuration of the connection to the *Content Server*.
- Table 3.42, "Properties of the cm-xmlimport.properties file" in *Deployment Manual* contains properties for the configuration of the transformation process of the *Importer*.

Glossary

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

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

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

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

MXML	MXML is an XML dialect used by Apache Flex (formerly Adobe Flex) for the declarative specification of UI components and other objects. Up to CMCC 10 (2107), CoreMedia Studio used the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 7. Starting with CMCC 11 (2110), a new, Node.js and npm based version of Jangaroo is used that supports standard TypeScript syntax instead of MXML/ActionScript, still compiling to Ext JS 7 JavaScript.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	With projects you can group content and manage and edit it collaboratively, setting due dates and defining to-dos. Projects are created in the Control Room and managed in project tabs.
Property	In relation to CoreMedia, properties have two different meanings: In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content item depends on the content type. In connection with the configuration of CoreMedia components, the system behavior of a component is determined by properties.
Replication Live Server	The aim of the *Replication Live Server* is to distribute load on different servers and to improve the robustness of the *Content Delivery Environment*. The *Replication Live Server* is a complete Content Server installation. Its content is an replicated image of the content of a *Master Live Server*. The *Replication Live Server* updates its database due to change events from the *Master Live Server*. You can connect an arbitrary number of *Replication Live Servers* to the *Master Live Server*.
Resource	A folder or a content item in the CoreMedia system.
ResourceURI	A ResourceUri uniquely identifies a page which has been or will be created by the *Active Delivery Server*. The ResourceUri consists of five components: Resource ID, Template ID, Version number, Property names and a number of key/value pairs as additional parameters.
Responsive Design	Responsive design is an approach to design a website that provides an optimal viewing experience on different devices, such as PC, tablet, mobile phone.
Site	A site is a cohesive collection of web pages in a single locale, sometimes referred to as localized site. In *CoreMedia CMS* a site especially consists of a site folder, a site indicator and a home page for a site. A typical site also has a master site it is derived from.
Site Folder	All contents of a site are bundled in one dedicated folder. The most prominent document in a site folder is the site indicator, which describes details of a site.

Site Indicator	A site indicator is the central configuration object for a site. It is an instance of a special content type, most likely `CMSite`.
Site Manager	Swing component of CoreMedia for editing content items, managing users and workflows. The Site Manager is deprecated for editorial use.
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, predefined layout and even predefined content.
Translation Manager Role	Editors in the translation manager role are in charge of triggering translation workflows for sites.
User Changes web application	The *User Changes* web application is a *Content Repository* listener, which collects all content, modified by *Studio* users. This content can then be managed in the *Control Room*, as a part of projects and workflows.
Variants	Most of the time used in context of content variants, variants refer to all localized versions within the complete hierarchy of master and their derived sites (including the root master itself).
Version history	A newly created content item receives the version number 1. New versions are created when the content item is checked in; these are numbered in chronological order.
Weak Links	In general *CoreMedia CMS* always guarantees link consistency. But links can be declared with the *weak* attribute, so that they are not checked during publication or withdrawal. Caution! Weak links may cause dead links in the live environment.
Workflow	A workflow is the defined series of tasks within an organization to produce a final outcome. Sophisticated applications allow you to define different workflows for different types of jobs. So, for example, in a publishing setting, a document might be automatically routed from writer to editor to proofreader to production. At each stage in the workflow, one individual or group is responsible for a specific task. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process.

Workflow Server	The *CoreMedia Workflow Server* is part of the Content Management Environment. It comes with predefined workflows for publication and global-search-and-replace but also executes freely definable workflows.
XLIFF	XLIFF is an XML-based format, standardized by OASIS for the exchange of localizable data. An XLIFF file contains not only the text to be translated but also metadata about the text. For example, the source and target language. *CoreMedia Studio* allows you to export content items in the XLIFF format and to import the files again after translation.

Index