CoreMedia Content Cloud

Unified API Developer Manual

CoreMedia

Copyright CoreMedia GmbH © 2021

CoreMedia GmbH

Ludwig-Erhard-Straße 18

20459 Hamburg

International

All rights reserved. No part of this manual or the corresponding program may be reproduced or copied in any form (print, photocopy or other process) without the written permission of CoreMedia GmbH.

Germany

Alle Rechte vorbehalten. CoreMedia und weitere im Text erwähnte CoreMedia Produkte sowie die entsprechenden Logos sind Marken oder eingetragene Marken der CoreMedia GmbH in Deutschland. Alle anderen Namen von Produkten sind Marken der jeweiligen Firmen.

Das Handbuch bzw. Teile hiervon sowie die dazugehörigen Programme dürfen in keiner Weise (Druck, Fotokopie oder sonstige Verfahren) ohne schriftliche Genehmigung der CoreMedia GmbH reproduziert oder vervielfältigt werden. Unberührt hiervon bleiben die gesetzlich erlaubten Nutzungsarten nach dem UrhG.

Licenses and Trademarks

All trademarks acknowledged.
September 19, 2021 (Release 2104)

1. Preface 1
1.1. Audience 2
1.2. Typographic Conventions 3
1.3. CoreMedia Services 5
1.3.1. Registration 5
1.3.2. CoreMedia Releases 6
1.3.3. Documentation 6
1.3.4. CoreMedia Training 10
1.3.5. CoreMedia Support 10
1.4. Changelog 12
2. Unified API Overview 13
2.1. Features and Design Goals 14
2.2. Use Cases 15
3. An Introductory Example 17
4. Common Concepts 19
4.1. Connection 20
4.1.1. Creating a Connection 20
4.1.2. Lifecycle and Caching 23
4.1.3. Connection Listener 26
4.1.4. Server Control 26
4.2. Repositories and Services 28
4.3. Objects 30
4.4. Values 32
4.4.1. XML Texts 32
4.4.2. Blobs 33
4.4.3. Lists 34
4.4.4. Structs 34
4.5. Types 37
4.6. Identifiers and Equality 39
4.7. Listeners 43
4.8. Exceptions 45
4.9. Sessions 46
4.10. Caching 49
4.11. Serialization 50
4.12. Further Reading 51
5. The Content Repository 52
5.1. Objects 53
5.2. UUIDs 58
5.3. Types 59
5.4. Access Control 60
5.5. Publication Service 62
5.6. Observed Property Service 65
5.7. Query Service 66
5.8. Search Service of the Unified API 76
5.9. Workflow Content Service 79
5.10. Property Service 80
5.11. Listeners 81
5.12. Further Reading 82
6. The Workflow Repository 83

6.1. Objects 85
6.2. Workflow States 88
6.3. Differences to the Classic Workflow API 94
6.4. The Work List Service 95
6.5. Workflow Variables and Views 97
6.6. The Access Control Service 100
6.7. Managing Process Definitions 102
6.8. Events 103
6.9. Timers 105
6.10. Writing Own Plugins 108
6.10.1. Programming Restrictions 108
6.10.2. Serialization 110
6.10.3. Actions 110
6.10.4. Long Actions 111
6.10.5. Expressions 112
6.10.6. Performer Policies 114
6.10.7. Rights Policies 115
6.10.8. Remote Client Actions 117
6.10.9. Managers 118
6.11. Examples 120
6.11.1. Example Clients 120
6.11.2. Example Plugins 121
6.11.3. Example Code of the Mail Action 127
6.12. Guide to the API Documentation 131
7. The User Repository 132
7.1. Objects 133
7.2. UUIDs 135
7.3. Retrieving Objects 136
7.4. Listeners 137
7.5. Further Reading 138
Glossary 139
Index 145

List of Figures

4.1. Class Diagram: Repositories and Services 29
4.2. Class Diagram: Blobs 34
4.3. Class Diagram: Types 38
4.4. Class Diagram: Repositories and Identified Objects 42
4.5. Class Diagram: Listeners 43
5.1. Class Diagram: Content and Versions 53
5.2. Statechart: Checked In and Out 55
5.3. Statechart: Place Approvals 56
5.4. Statechart: Deleting 56
5.5. Statechart: Version 57
5.6. Statechart: Content Publication 62
6.1. Workflow Class Diagram 85
6.2. States of a process 89
6.3. States of an automated task 90
6.4. States of a Task 91
6.5. Workflow Object and View Definitions 97
6.6. Workflow views 98
7.1. Class Diagram: Users and Groups 134

List of Tables

1.1. Typographic conventions 3
1.2. Pictographs 4
1.3. CoreMedia manuals 7
1.4. Changes 12
4.1. Connection properties 21
4.2. Parameters of connection's management bean 24
4.3. ID formats for CapObject 39
4.4. ID formats for CapType 40
4.5. ID formats for other objects 41
5.1. Rights for the Unified API 60
5.2. Types in subexpressions 71
6.1. WfAPI signal names and UAPI event classes 94

List of Examples

3.1. Create a new folder 17
4.1. Open a session 46
4.2. Log in another session 47
4.3. Using a session pool 47
6.1. `AbortAllProcesses` 120
6.2. Suspend My Processes 120
6.3. Create Process Example 121
6.4. The SendMail action 128

1. Preface

This book introduces and explains the *Unified API*, which is the recommended API for most applications that use *CoreMedia CMS*.

The following chapters are organized as follows:

- An overview of the API and its uses is given in Chapter 2, *Unified API Overview* [13].
- Afterwards, Chapter 3, *An Introductory Example* [17] introduces you to the *Unified API* by the way of a simple example.
- Concepts of the *Unified API* that are independent of the accessed repository are explained in Chapter 4, *Common Concepts* [19].
- Afterwards, the individual repositories are dealt with, starting with the content repository in Chapter 5, *The Content Repository* [52].
- The workflow repository is the topic of Chapter 6, *The Workflow Repository* [83].
- In Chapter 7, *The User Repository* [132] the user repository is documented.

1.1 Audience

This manual is addressed to developers of CoreMedia projects who want to develop content applications using the *Unified API*. They'll find a description of ideas and concepts, building blocks, and detailed examples.

1.2 Typographic Conventions

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

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

Table 1.1. Typographic conventions

In addition, these symbols can mark single paragraphs:

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

Table 1.2. Pictographs

1.3 CoreMedia Services

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

NOTE

CoreMedia User Orientation for CoreMedia Developers and Partners

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

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

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

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 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* .

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 and Javadoc as PDF files and as online documentation at the following URL:

https://documentation.coremedia.com/cmcc-10

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.
Content Server Manual	Developers, architects, administrators	This manual describes the concepts and administration of the main CoreMedia component, the *Content Server*. You will learn about the content type model which lies at the heart of a CoreMedia system, about user and rights management, database configuration, and more.

Manual	Audience	Content
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.
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application or the usage of the watchdog component.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.

Manual	Audience	Content
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.
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. The location where component log output is written is specified in its `logback.xml` file.

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, log files can be found in the CoreMedia component's installation directory in `/var/logs` or for web applications in the `logs/` directory of the servlet container. See Section 4.7, “Logging” in *Operations Basics* for details.

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. Unified API Overview

In this chapter you will get a very high-level overview of the *Unified API* and a sketch of its possible applications.

The *Unified API* is the preferred API for interfacing with the *CoreMedia CMS* when writing custom tools and agents and when writing content delivery applications. In particular, it is tightly integrated with the *CoreMedia Content Application Engine* and it is available in the *Workflow Server* for modifying content and for implementing plugin classes.

The *Unified API* allows you to access both the *Content Server* and the *Workflow Server* from custom code. It presents the properties of documents, versions, folders, users, groups, processes, and tasks in a uniform and object-oriented way.

While the *Unified API* is comprehensive, care has been taken to isolate various aspects of the API, so that each individual aspect remains of moderate size. To this end, a connection mediates between multiple repositories. In turn, each repository is augmented by services and provides access to stateful objects. Stateful objects share a common metamodel and are identified in a common name space. This style is applied throughout the API and promotes uniformity.

Typical applications of the *Unified API* are:

- content delivery through a servlet engine;
- form-driven web applications for content modifications;
- administrative command line tools;
- background processes whose actions are triggered by events;
- periodically scheduled processes that carry out custom actions;
- workflow actions, expressions, performers policies, and rights policies.

2.1 Features and Design Goals

The Unified API supports programmers and makes the administrator's life simple.

- *Programming is easy.*
  While the API is comprehensive, a simple application might still not use more than half a dozen classes. This greatly reduces the initial learning effort for using the API.
  Creating a connection to *Content Server* and *Workflow Server* is as simple as calling a single method. Afterwards, the connection provides quick access to the entire system.
  As different parts of the API share a common style and in fact a common metamodel, it is comparatively easy to acquire knowledge about new parts of the API.
  The API explicitly specifies preconditions and postconditions and indicates the possible events and exceptions, leaving little room for ambiguities.
  Convenience methods simplify common tasks.
- *Deployment is easy.*
  Deploying an application that uses the *Unified API* is straightforward. Adding a few jars to the class path is all that is needed. The API does not demand special configuration files.
  Through a management interface it is possible to control *Unified API* applications at runtime.
- *The API is memory-efficient.*
  Multiple sessions per connection are possible, sharing a common cache while providing individual rights checks.
  All stateful objects are thin wrappers that use little memory and fetch their state through the common cache as needed.
- *The API is robust.*
  The *Unified API* can survive server restarts, providing a continuous event stream and maintaining cache consistency.
  The cache size is configurable in bytes, virtually eliminating fluctuations of memory usage by the API.

2.2 Use Cases

Here you will find typical use cases for the *Unified API*.

Content Delivery

Situation: You want to deliver content that is stored in the *CoreMedia CMS*, for example, when generating a website.

Solution: The *Unified API* is used inside the *CoreMedia Content Application Engine* to access persistent data. The engine is used for efficient caching on higher levels. JSPs render your content.

Form-driven Content Modification

Situation: You want to create a web application that allows certain recurring modifications of the content, for example, changing a price information.

Solution: Again, you use the *CoreMedia Content Application Engine*, this time augmented with the write functionality of the *Unified API*.

Command Line Tool

Situation: You want to create a command line tool that automates certain administrative tasks, for example, the creation of users with a predefined set of query documents.

Solution: You program the tool using the *Unified API*, possibly starting with the base client provided as a code example.

Automated Agents

Situation: You want to create background processes that perform automated actions when certain events occur, for example, starting a workflow when a document is moved into a certain folder.

Solution: You create an appropriate repository listener using the *Unified API* and add the required actions in Java code.

Workflow Actions

Situation: You want to perform very complicated actions during certain workflow tasks.

Solution: You program a workflow action using the *Unified API*, updating content objects and workflow variables as needed. You might want to create a user-specific session for modifications.

Performers Policies

Situation: You want to control the set of users to whom a certain task is offered.

Solution: You program a performer policy using the *Unified API*, evaluating the state of workflow variables and referenced content while determining one or more users who may execute the task. Possibly, you also create a right policy to limit the permissible activities of the chosen users.

3. An Introductory Example

The following example shows how to create a new folder with a fixed name. While not interesting in itself, it contains all the steps needed to establish a connection and to perform some work.

```
package com.coremedia.examples.capclient;

import com.coremedia.cap.Cap;
import com.coremedia.cap.common.CapConnection;
import com.coremedia.cap.content.*;

public class HelloWorld {
  public static void main(String[] args) {
    String url = "http://localhost:40180/ior";
    CapConnection con = Cap.connect(url, "admin", "admin");
    ContentRepository repository = con.getContentRepository();
    try {
      Content root = repository.getRoot();
      ContentType folderType = repository.getFolderContentType();
      folderType.create(root, "hello world");
    } finally {
      con.close();
    }
  }
}
```

Example 3.1. Create a new folder

Look at the example line by line.

```
String url = "http://localhost:40180/ior";
```

The *Content Server* to use is indicated by its URL. If you are connecting to a *Content Server* on a different host, you may want to change `localhost` to the name of the configured host and `40180` to the configured port.

```
CapConnection con = Cap.connect(url, "admin", "admin");
```

Besides the URL, only user name and password are required to log on to the server. Here you use the admin account, assuming that a test environment has been set up and left basically unchanged. A connection object is returned from the connect call.

```
ContentRepository repository = connection.getContentRepository();
```

The connection object is a mediator that provides access to all parts of the *CoreMedia CMS*. There are separate repositories for content access, user management, workflows and so on. Here you only deal with the content repository.

```
Content root = repository.getRoot();
```

The root folder of the content repository is retrieved and stored locally as a content object. Both folders and documents are summarized under the common concept of content. While there are some differences between folders and documents, they share many common traits, which allows you to use a common abstraction in the *Unified API*.

```
ContentType folderType = repository.getFolderContentType();
```

Every content is equipped with a content type. Types of documents may be freely defined, but for folders there is a single well-known content type.

```
folderType.create(root, "hello world");
```

The content type is instructed to create a new instance of itself. You have to provide two arguments: the folder in which the new content is created and the new content's name.

```
try {
  ...
} finally {
  con.close();
}
```

Ultimately, you want to close the connection in order to free licenses that were allocated on the server and to release local resources that were obtained when opening the connection. If you had forgotten to close the connection, the program would not terminate, waiting for background threads started for the duration of the connection.

It is generally a good idea to close the connection in a `try/finally` block, so that it is closed in all cases. For example, run the example again and you should receive an error due to a duplicated content name. Nevertheless, the program exits cleanly.

You will notice debug output on the console. See Section 4.7, “Logging” in *Operations Basics* for more details about logging. If the log output bothers you, redirect the standard error output stream to a file or the null device.

4. Common Concepts

The *Unified API* applies to three functional areas:

- content,
- workflow,
- user management.

Each area is accessible through a *repository*. A repository provides access to persistent objects and offers various services. Many tasks can be performed while only accessing a single repository, but at times you need access to the full functionality. For each repository, you will find in the following an entire chapter containing a detailed discussion. This chapter, however, is limited to topics that apply regardless of the repository at hand.

First, the connection object is discussed. It mediates between the individual repositories. Because the connection is the primary entry point when working with the *Unified API*, it is explained in detail how a connection can be obtained and configured.

Then, key concepts are described that apply equally to all three repositories. The basic structure of all repositories is essentially the same and also the persistent objects share many features. Moreover, one should be aware of certain design principles that apply throughout the *Unified API*.

4.1 Connection

In this section, details of the connection object are discussed. It is shown how a connection can be created and which services it offers.

While the connection also provides access to the three repositories, repositories are not viewed as integral parts of the connection. They will be discussed one by one in the following chapters.

4.1.1 Creating a Connection

Before working with the *Unified API*, a connection to the server must be opened. The connection object implements the interface `com.coremedia.cap.common.CapConnection`. There are a number of static methods in the class `com.coremedia.cap.Cap` that allow you to specify various sets of parameters for logging on to *Content Server* and *Workflow Server*.

Passing Parameters Directly

The most common way of opening a connection is provided by a method of the class `com.coremedia.cap.Cap` with four parameters:

- The IOR URL of the *Content Server*
- The name of the user who logs in
- The user's domain
- The user's password

All parameters are passed as string values. The IOR URL is explained in more detail in the *Operations Basics Manual*. It is a means for bootstrapping the CORBA protocol.

```
String url = "http://localhost:40180/ior";
CapConnection connection = Cap.connect(url,
  "user", "domain", "secret");
```

The login call will fail with an exception if the *Content Server* is not reachable. A connection to the *Workflow Server* is also opened, if the *Workflow Server* is reachable, but its absence does not abort the login sequence.

Because the IOR URL is cumbersome to write, the *Unified API* uses some rules for determining this parameter when it is omitted.

```
CapConnection connection = Cap.connect(null,
  "user", "domain", "secret");
```

If the system property `coremedia.content.server.url` is set, its value is used as the URL. Else, if the system property `coremedia.configpath` is set, the system tries to determine the URL from the file `capclient.properties`. Because the latter property is automatically set by the `cm` start script, there is no need to configure the URL when the *Unified API* client is started by means of a `.jpif` file.

When you use the built-in user repository of *CoreMedia CMS* and not an LDAP server for managing your users, you can set the domain parameter to `null` or omit it entirely.

```
CapConnection connection = Cap.connect(url, "user", "secret");
```

Passing Parameters as a Map

When you want to pass more parameters than available to the standard login methods or when you want to determine the parameters in a more flexible way, you can pass a `java.util.Map` to the login method. The keys must be chosen from a number of constants defined in the class `Cap`. The values in the map are normally strings.

```
Map<String,?> params = new HashMap<String,?>();
params.put(Cap.CONTENT_SERVER_URL,
  "http://localhost:40180/ior");
params.put(Cap.WORKFLOW_SERVER_URL,
  "http://localhost:40380/ior");
params.put(Cap.USER, "admin");
params.put(Cap.DOMAIN, "");
params.put(Cap.PASSWORD, "admin");
CapConnection connection = Cap.connect(params);
```

In the previous example, you can see that the initial workflow server URL is passed as one parameter. Normally this is not required, because the *Content Server* acts as a naming service and provides the necessary information for connecting to other servers. However, in complex setups with multiple firewalls and connection redirection, it may be necessary that different clients connect via different URLs.

In the following, you will find summarized the available properties.

Name	Value	Default	Description
`CONTENT_SERVER_URL`	URL string	(determined heuristically)	the IOR URL of the *Content Server*
`WORKFLOW_SERVER_URL`	URL string	(fetched from the *Content Server*)	the IOR URL of the *Workflow Server*
`USER`	string	N/A	the name of the user to log in

Name	Value	Default	Description
`DOMAIN`	string	""	the domain of the user to log in
`PASSWORD`	string	N/A	the password of the user to log in
`USE_WORKFLOW`	"true", "false", ""	""	whether the *Workflow Server* should be connected; if "true", the connection is required; if "", the connection is optional; if "false", no connection attempt is made
`ORB`	an `org.omg.CORBA.ORB` object	(created automatically)	the ORB for setting up the CORBA connection
`CONNECTION_FACTORY_CLASS`	string	(built-in factory)	the name of a class implementing the interface `CapConnection.ConnectionFactory`

Table 4.1. Connection properties

You can also create a connection without opening it immediately. Here you may pass a number of parameters by means of a map, but you can set additional parameters later before opening the connection.

```
Map params = Collections.singletonMap(Cap.CONTENT_SERVER_URL,
 "http://localhost:40180/ior");
CapConnection connection = Cap.prepare(params);
connection.setUser("admin");
connection.setPassword("admin");
connection.open();
```

The methods that are available for setting the parameters of a connection are

- `setUrl(..)`,
- `setUser(...)`,
- `setDomain(...)`, and
- `setPassword(...)`.

Passing Parameters as a URL

While flexible, the creation of a map takes some lines of code, so that CoreMedia provides a simple method that works in many cases. The additional parameters beside the Content Server URL are inlined as URL parameters in that URL. This permits the compact configuration via a single string.

```
String url = "http://localhost:40180/ior"+
  "?user=admin&password=admin&useworkflow=false";
CapConnection connection = Cap.connect(url);
```

Here the workflow component has been disabled entirely by the means of `useworkflow=false`. This reduces the resource requirements when the workflow connection is not needed at all.

Individual parameters are separated by ampersands (`&`), the entire set of parameters is separated from the IOR URL by a question mark (`?`). Possible parameters are:

- *workflowurl*,
- *user*,
- *domain*,
- *password*,
- *useworkflow*.

Note that the well-known parameters are removed from the URL before it is resolved over the network. In particular, the password is not transmitted in clear text.

4.1.2 Lifecycle and Caching

After being created using the `Cap.connect(...)` methods, a connection is open immediately, that is, its methods can be invoked and all read and write accesses to the associate repositories are possible, too. A connection that was created through `Cap.prepare(...)` starts off closed. It has to be opened by a call to `open()`.

An open connection will stay open until closed explicitly. In particular, an open connection does not become eligible for garbage collection simply by discarding references to it. There are a number of active threads inside a *Unified API* connection that will keep the connection alive until explicitly closed.

After you have closed the connection, all stateful objects that were retrieved from the connection become nonfunctional, in particular the repositories, services and `CapObject`s. Immutable objects like strings or markup objects generally remain intact, but blobs become unusable.

Operations on Closed Connections

The only operations that are possible on a closed or not yet opened connection are calls to setters and getters for the connection parameters like user name or password.

Reopening a Connection

You can reopen a closed connection using the method's `connection.open()`. This should only be done in special cases. Normally, a connection is expected to stay open until the application terminates.

Care must be taken when reopening connections under an Oracle JDK, whose ORB implementation does not properly release its memory and TCP sockets when being closed. Since the *Unified API* connection must instantiate an ORB for managing the CORBA connection to the servers, this ORB bug can lead to resource problems after repeated sequences of open and close operations. In order to avoid this, you can inject a singleton ORB into the connection, which will then be used continually without being shut down at the close of the connection.

```
org.omg.CORBA.ORB orb =
  org.omg.CORBA.ORB.init(new String[0],
    System.getProperties());
Map<String,?> params = new HashMap<String,?>();
params.put(Cap.ORB, orb);
params.put(Cap.USER, "admin");
params.put(Cap.PASSWORD, "admin");
CapConnection connection = Cap.connect(params);
```

Management of Open Connections

While a connection is open, you can also access the connection's management bean as provided by the `getMBean()` method. These are the configurable parameters:

Property	Value	Default	Description
`heapCacheSize`	long	20000000	the number of bytes to use by the main memory cache
`blobCacheSize`	long	10737418240	the number of bytes to use by the disk cache
`blobCachePath`	string	N/A	the location of the disk cache in the file system; this property maps directly to the system property `java.io.tmpdir`

Property	Value	Default	Description
`maxCachedBlobSize`	int	`Integer.MAX_VALUE`	the maximum size of a blob that can be cached. Note that the maximum size of a cached blob is implicitly limited by `blobCacheSize`
`blobStreamingSizeThreshold`	int	131072	the threshold for blob sizes above which blobs are streamed instead of being completely downloaded first
`blobStreamingThreads`	int	2	the maximum number of threads that is used for streaming large blobs
`eventChunkSize`	int	1000	the maximum number of events that is fetched at once from the *Content Server* when attaching a listener with a historic time stamp

Table 4.2. Parameters of connection's management bean

Reopening Connections

The *Unified API* also supports the reopening of closed connections. After a connection has been reopened, the listeners have all been removed from the listener sets and blobs may have been rendered unusable, but the repositories, services and `CapObject`s have returned to their previous state, allowing reads and writes.

The cache object that is associated with the connection does not remain stable. Instead, a new cache object is created whenever the connection is opened.

This makes it possible to create a perpetually running client that releases its licenses when it is idle for an extended period. Of course, the reacquisition of contested licenses may fail, so that this pattern is not suitable for system components with strict availability requirements.

Automatic Reconnect after Server or Network Problems

The *Unified API* supports reconnects to servers after network problems and even after the servers are restarted. The connection remains open while a reconnect is attempted, but read and write accesses may fail with an exception.

In the case of the content and user repository, the event streams are reestablished and no events are lost. In the case of the workflow repository, events may be lost, but all caches are properly invalidated after the reconnect.

If the document types are changed in any way while the *Content Server* is down, a reconnect may fail in unexpected ways. Always shut down all clients before modifying the document type declarations.

4.1.3 Connection Listener

The *Unified API* supports one listener type that can be directly attached to the connection: the `CapConnectionListener`. A connection listener is notified about important events that affect the status of the connection.

In particular, the listener is notified whenever the connection detects a problem in the communication with the server. In this case, the `connectionUnavailable` method is called. As soon as the server or the network recovers, a `connectionAvailable` is sent.

When the run level of the server is changed, there may be a warning that the connection has to be closed. This is done through the method `connectionWillBeUnavailable`. In the case that a run level switch is aborted, the method `connectionWillNotBeUnavailable` is called to signal this condition.

The method `connectionDisrupted` indicates the rare event that the connect has become permanently unavailable, so that no reconnect is attempted. Possibly the connection's user was deleted in the database or the connection was shutdown by an explicit invocation of `cm killsession`.

4.1.4 Server Control

The `CapConnection` provides one service object: the `ServerControl`, which is reachable through the method `getServerControl`. It provides means for inspecting and controlling the login process on the *Content Server*.

In particular, it provides methods for inspecting the license information, for inspecting and tracking the set of currently opened sessions, for requesting trace level logging, for killing individual sessions and for changing the run level of the *Content Server*.

4.2 Repositories and Services

Having obtained the connection object as described in Section 4.1.1, "Creating a Connection" [20], you can access three repositories: content repository, workflow repository and user repository. A repository encapsulates functionality that pertains to one category of domain objects. All repositories implement the common superinterface `CapRepository`.

A repository offers methods for the following tasks:

- Lookup existing objects.
- Modify existing objects.
- Create new objects.
- Inspect objects.
- Inspect types.
- Provide access to services.
- Add and remove listeners that are informed about all events in the repository.
- Get information about the connected server and about the local machine.
- Obtain a reference to the associated connection.

In the previous list, objects are identifiable persistent objects. The content repository is concerned with documents and folders. The workflow repository is concerned with processes and task. The user repository is concerned with users and groups. Depending on the stateful objects that have to be processed, you choose the appropriate repository.

The term *services* referred to objects that exist once per connection and that can be obtained through the repositories. In some sense, services are small repositories whose functionality is very limited. They might perform any of the tasks listed above by accessing objects and types or handling listeners, but their methods concern only one specific aspect of the repository, for example, only content publication or only the computation of rights to workflow objects.

In fact, the methods provided by such services might have just as well been provided by their repository, but at the expense of clarity. By grouping methods in service objects, you can get a quick overview of the system, while getting closely accustomed to the relevant services, only.

A typical method that is reachable directly on the repository level is `ContentRepository.getRoot()`, which returns an object representing the root folder. It is not appropriate for an individual object and it is not easily grouped with other methods to form a service. A typical method on the service level is `PublicationService.approvePlace(Content)`. It matches nicely with other publisher-related methods and there is no need why it would absolutely have to be placed in the object-level class `Content`. After all, many applications do not care about publication at all, so that it is preferred to make it a little less visible.

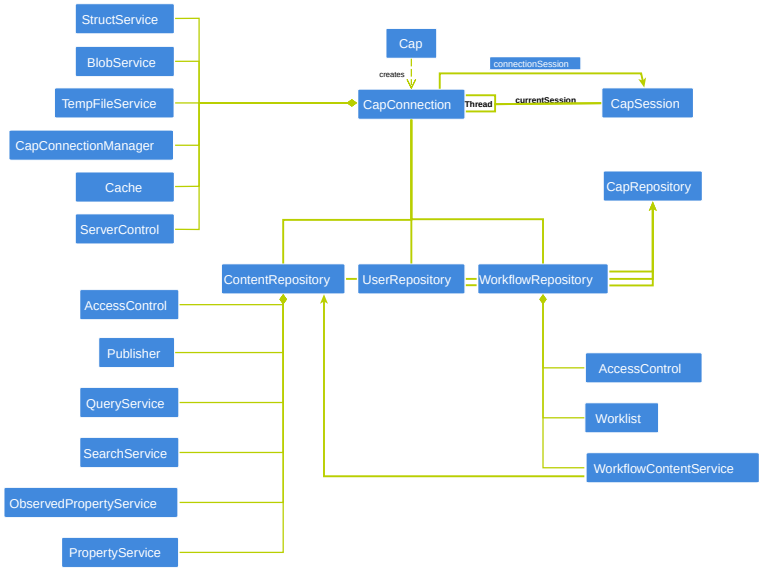

Figure 4.1. Class Diagram: Repositories and Services

In Figure 4.1, "Class Diagram: Repositories and Services" [29] you can see an UML class diagram of the connection and all repositories and services that are reachable through the session.

See Chapter 5, *The Content Repository* [52], Chapter 6, *The Workflow Repository* [83], and Chapter 7, *The User Repository* [132] for detailed discussions of the individual repositories. The services are also described in the chapter that is devoted to their repository.

4.3 Objects

The *Unified API* provides a common superinterface for all persistent entities: `CapObject`. A `CapObject` can be thought of as being contained by a repository. Within that repository, it is made unique by an identifier. The available object classes have already been named in this text: folders and documents, users and groups, processes and tasks.

Folders and documents are jointly presented through the interface `Content`. Documents may exist in more than one `Version`. The `Version` and `Content` interfaces are subsumed under the `ContentObject` interface. Likewise, `User` and `Group` objects share a common superinterface `Member` and the interfaces `Process`, `Task` and `WorkflowView` are derived from the interface `WorkflowObject`. All of these interfaces extend `CapObject`.

Two `CapObject`s refer to the same persistent entity if they are equal as per `Object.equals(Object)`. In general, there may be more than one Java object for the same persistent entity.

> **CAUTION**
>
> Never compare two objects of the *Unified API* using the `==` operator. This operator will typically return `false` even though two objects refer to the same persistent entity. Always use object equality instead.

`CapObject`s are also providing access to properties of that object. To that end, `CapObject` extends the interface `CapStruct`, which defines a generic abstraction of an entity with named properties of various types.

You can obtain either a map with all properties or individual property values using the getters of a `CapStruct`. When getting a map, an immutable snapshot of the object's properties is returned. When getting one property value multiple times, however, concurrent writes will be visible immediately.

All structs provide a struct type through the method `getType()`. The type is immutable and constitutes a model of the possible property values for the struct. Properties can themselves be of different types as will be described in Section 4.5, “Types” [37]. There are typed getter methods for returning the current values of properties. If a typed getter is applied to a property with a different type, the *Unified API* specifies an automatic conversion in many cases. Please see the Javadoc of `CapStruct` for details. If there is no possible conversion algorithm, an exception is thrown.

When setting a property of a `CapObject`, make sure that you use a value that is appropriate for the property type used, because no automatic conversion takes place.

The values returned by a getter are always immutable. In the case of `String` or `Integer` objects this is obvious, but it is even true for collection-valued properties that return an instance of `java.util.List`. When you want to modify a collection-valued property, you have to create a new collection and set that entire collection as the new value. Modifying the returned value is not possible.

Having set any property of a `CapObject`, that change is not immediately made persistent on the server-side. Changes are collected until either an operation occurs that cannot be delayed or the method `CapConnection.flush()` is called on the current connection. See also Section 4.9, “Sessions” [46] for details about the session handling.

4.4 Values

Objects of the *Unified API* can store property values of various types. Not all property types are available for all repositories, however. Please see the documentation of the individual repositories for an overview of the supported property types.

Most of the values are well-known in the Java language: `String`, `Integer`, `Calendar`, and the like. There are some special property values for which the existing classes are not sufficient. These values are discussed in more detail now. All values share the common feature that they are unmodifiable in the sense that they will not change spontaneously and that they do not provide methods to change their state.

4.4.1 XML Texts

For XML properties, a `Markup` object is provided as the property value. A `Markup` represents an immutable XML document fragment. It consumes less memory than a DOM representation and can generate SAX events faster than a SAX parser. Conversion and interaction with the standard APIs SAX 2, DOM 2, JAXP, and TRaX is possible.

Note that while the memory footprint of a `Markup` is comparatively small, such objects are still kept entirely in main memory. If you handle many large XML texts, it becomes important that you make them eligible for garbage collection as soon as possible.

`Markup` instances are read-only and encourage a functional programming style like in `Markup m2 = m.transform(...)`. SAX-based and XSLT-based transformations are available. The class `MarkupFactory` allows the creation of `Markup` objects from an `InputStream`, a `Reader`, an `InputSource`, a JAXP `Source`, a DOM `Node` or a `String`.

`Markup` instances carry an optional grammar name as a hint regarding the structure of the XML text.

Note that unlike other value objects, `Markup` s do not declare a special `Object.equals(Object)` method, so that they cannot be easily compared. If required, you should design your own comparison algorithm that takes the actual XML format into account.

Please refer to the Javadoc of the package `com.coremedia.xml` for details about the `Markup` interface and the associated classes.

4.4.2 Blobs

Blob properties take `Blob` objects as values. Like `Markup` objects, they are API objects that are immutable. They provide access to metadata and to an input stream that contains the actual binary content.

When a blob is read from the content repository, it is cached on disk and not in main memory. It is even possible for the disk cache to be cleared while you still hold a reference to the `Blob` object. Therefore, a content repository blob is a comparatively cheap object.

The workflow repository supports blobs, too. Such blobs are always loaded into main memory and they cannot be garbage collected as long as they are directly or indirectly referenced from client code. Normally, this is not a problem, because workflow repository blobs often serve very special needs, being used for the compact storage of complex data structures. Workflow blobs are generally not recommended for storing large images or audio stream.

When you want to set a blob property, it is possible to use a `Blob` object that you obtained by a previous read operation. The class `BlobService` allows the creation of `Blob` objects from either a file, a URL, an `InputStream`, or a byte array. It returns a blob object that you can pass into the setter.

Normally, you obtain a blob by calling the method `CapObject.getBlob(String)`. When you call `CapObject.getBlobRef(String)` instead, you get a reference to the blob instead, encapsulated as a `CapBlobRef` object. While ordinary blobs are immutable, blob references can change over time, reflecting concurrent changes to the `CapObject`. Blob references are cheaper than blobs, reducing resource requirements. They can also be useful when you want to indicate the origin of a blob as compared to its content, for example, when generating URLs that link to image properties.

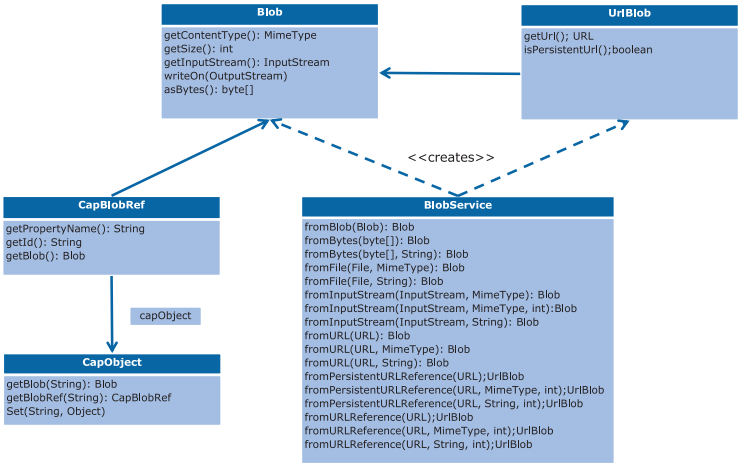

Figure 4.2. Class Diagram: Blobs

4.4.3 Lists

Some properties contain an entire list of values instead of a single value. For content objects, only lists of `Content` are possible. For workflow objects, all kinds of property types are also available as aggregation properties. Such properties always use implementations of `java.util.List` for representing values.

When retrieving an aggregation from the repository, the resulting object is dead and unmodifiable: it will not change due to concurrent actions and it cannot be changed by the client. When you want to change the value of a list-valued property, you have to provide a new list with the correct state, possibly copying the previous list into a new collection.

When reading a property with a typed getter, lists are automatically converted to atomic values and vice versa. Lists of different types are automatically converted by converting the individual entries. See the Javadoc of `CapObject` for details.

4.4.4 Structs

In Section 4.3, "Objects" [30] the interface `CapStruct` was introduced as a superinterface of `CapObject` for providing readable properties of an entity. While `CapOb-`

`ject`s are mutable and thus not suitable as values, the interface `Struct` also inherits from `CapStruct` and represents an unmodifiable structured object.

Structs can be stored in markup properties of the content repository, if the markup property uses the predefined grammar `coremedia-struct-2008`. The Unified API will transparently convert instances of `Struct` to and from XML. The storage format is compatible with the struct abstraction that used to be provided with *CoreMedia Starter Kit*.

`Struct`s support only a limited range of primitive property types, namely strings, integers, Boolean, links to `Content` and lists thereof. However, structs may also contain arbitrarily nested structs and lists of structs as complex property values. While structs themselves are immutable, they provide the `builder()` method that returns a builder object that can be used to create other similar structs.

CAUTION

`StructBuilder`s are not structs. They cannot be used as property values.

A `StructBuilder` provides methods to set property values and to declare new properties. The method `set(String, Object)` sets a single property, whereas the method `setAll(Map)` sets multiple properties at once. The methods `declare...` take varying arguments depending on the type of property they define. For list properties, the methods `set(String, int, Object)`, `add(String, Object)` and `add(String, int, Object)` provide ways to replace a list element or to add a new list element. Likewise, `remove(String, int)` removes a single element from a list.

When building nested structs, a struct builder always considers either the top-level struct or one of the substructs as the current struct. Set and declare operations are always performed on the current struct. Using the methods `enter(String)` a substruct of the current struct can be selected as the new current struct. In the case of struct lists, use `enter(String, int)`. When calling `up()`, the current struct can be set back one level towards the top-level struct. Calling `at(...)`, you can navigate directly to a deeply nested substruct ignoring the previous current struct. The method `currentPath()` returns the current path, allowing you to return to a given substruct later on.

The method `mode(...)` requests one of three different behaviors that are represented by the enumeration class `StructBuilderMode`. The mode determines how the struct builder reacts when a declare or set operation conflicts with the existing declaration of a property. By default, a new property can be directly set without declaring it, as long as the value is not `null` or a list containing values of mixed types, because a suitable property descriptor can be inferred. But that is not allowed in all modes.

- `STRICT`: Declare operations fail if the property already exists. Set operations fail if no property descriptor exists and they fail if the existing property descriptor does not allow the value. Property descriptors are never inferred.
- `DEFAULT`: Declare operations fail if the property already exists. Set operations fail if the existing property descriptor does not allow the value or if a new property descriptor cannot be inferred.
- `LOOSE`: Declare operations never fail. Set operations fail only if the desired property descriptor cannot be inferred. If a new value does not match an existing property descriptor, the existing descriptor is replaced by another descriptor that allows the value.

You can use the method `remove(String)` to remove a property declaration from the current struct. In strict and default mode, this is necessary before a property can be redeclared. Using `removeAll()` the current struct be reset to an empty struct.

The method `defaultTo(Struct)` can be used to extend the current struct with those property declarations of the argument struct that were not previously present in the current struct. This is useful to set default values when initializing a struct or when merging multiple levels of struct-based configurations. When an existing struct property is defaulted to another struct property, the default is applied recursively. When an existing struct list property is defaulted to a struct property (not a struct list property), each list element is augmented with default values individually.

Finally, when the struct is completely built, you can retrieve it from the builder by means of the `build()` method. The builder remains usable to build additional similar structs. At any time, you can also retrieve the current struct using `currentStruct()`.

CAUTION

`StructBuilder` instances are not thread-safe. Builders must not be accessed concurrently by multiple threads.

4.5 Types

Every `CapObject` is an instance of a type. A type defines the properties that are appropriate for that object. Types are represented as `CapType` objects. Types are named and they may be put into a subtype hierarchy, which can be queried through the `CapType` objects.

For each property, a type aggregates a `CapPropertyDescriptor` object. There is one subclass of `CapPropertyDescriptor` for every kind of property value: `IntegerPropertyDescriptor`, `LinkPropertyDescriptor`, and so on.

Property descriptors provide further information about the property. In particular, the method `isCollection()` indicates whether the descriptor belongs to a collection-valued property.

The type and descriptor objects allow you to inspect the structure of the type system algorithmically. This is not required for many applications, but it allows you to write reusable algorithms that are supposed to act on `CapObject`s regardless of their actual internal structure.

Often, types act as factories. Using create methods, it is possible to build additional instances of a type. The methods for doing this are defined in the sub interfaces, though. They require additional information that depends on the repository that is used.

For more details on the type system, see the Javadoc of the mentioned classes.

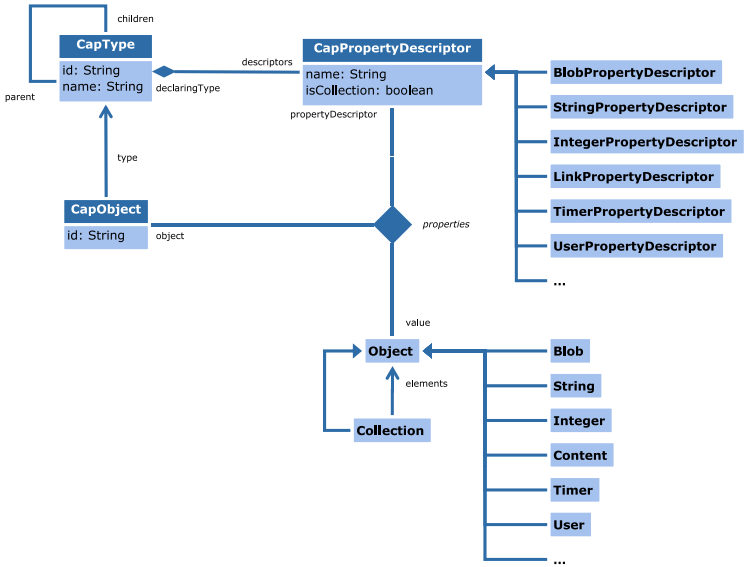

Figure 4.3. Class Diagram: Types

4.6 Identifiers and Equality

Every `CapObject` and `CapType` is equipped with a stable string ID that can be persistently stored and converted back into an object reference as needed. To this end, `CapObject` has got a method `getId()` for retrieving the ID. Methods for converting IDs into object references are typically provided by the repository objects.

Every `ContentObject` and every `Member` on the *Content Management Server* also has an additional UUID, a stable and universally unique identifier as defined in RFC 4122. UUIDs are not meant as replacement of simple string IDs, but make sense in certain scenarios. For details on `ContentObject` UUIDs, have a look at Section 5.2, "UUIDs" [58]. For details on `Member` UUIDs, have a look at Section 7.2, "UUIDs" [135].

It is recommended that you treat the string IDs as opaque strings, because the exact format of the strings might change in future releases of *CoreMedia CMS*. Still, CoreMedia provides detail information about the IDs for the purposes of debugging and for interfacing the *Unified API* with legacy clients which might insist on using numeric IDs.

The class `com.coremedia.cap.common.IdHelper` is provided for formatting and parsing all sorts of ID strings. Note that all methods in that class may be redefined arbitrarily in the next *CoreMedia CMS* release.

The following table summarizes the various ID formats for `CapObject`s.

ID string	Interface	Description
coremedia:///cap/content/<int>	Content	document or folder
coremedia:///cap/version/<int>/<int>	Version	version of document parameters: numeric content ID/version number
coremedia:///cap/process/<int>	Process	process
coremedia:///cap/task/<int>/<int>	Task	task parameters: numeric process ID/numeric task ID
coremedia:///cap/initialview/<int>	WorkflowView	initial process view parameter: numeric process ID

ID string	Interface	Description
coremedia:///cap/process-view/<int>	WorkflowView	ordinary process view parameter: numeric process ID
coremedia:///cap/task-view/<int>/<int>	WorkflowView	task view parameters: numeric process ID/numeric task ID
coremedia:///cap/user/<int>	User	user
coremedia:///cap/group/<int>	Group	group

Table 4.3. ID formats for CapObject

The `CapType`s are also identified by an ID.

ID string	Interface	Description
coremedia:///cap/content-type/<string>	ContentType	content type
coremedia:///cap/grammar/<string>	XmlGrammar	XML grammar
coremedia:///cap/processdefinition/<int>	ProcessDefinition	process definition parameter: numeric process definition ID
coremedia:///cap/taskdefinition/<int>/<int>	TaskDefinition	task definition parameters: numeric process definition ID/numeric task definition ID
coremedia:///cap/initialviewdefinition/<int>	WorkflowViewDefinition	initial process view definition parameter: numeric process definition ID
coremedia:///cap/process-viewdefinition/<int>	WorkflowViewDefinition	ordinary process view definition parameter: numeric process definition ID

ID string	Interface	Description
coremedia:///cap/taskviewdefinition/<int>/<int>	WorkflowViewDefinition	task view definition parameters: numeric process definition ID/numeric task definition ID

Table 4.4. ID formats for CapType

There are some other objects that are also assigned an ID, but that do not implement `CapObject` or `CapType`. Such objects implement the method `getId()`, but they do not provide getters and setters for properties.

ID string	Interface	Description
coremedia://cap/publication/<int>	Publication	a publication that has been enqueued
coremedia://cap/publicationtarget/<string>	PublicationTarget	a publication target
coremedia://cap/session/<int>	CapSessionInfo	a session that has been opened on the *Content Server*
coremedia://cap/service/<int>	CapServiceInfo	a service of the *Content Server* for which logins are possible

Table 4.5. ID formats for other objects

Unified API objects that define an ID are equal in the sense of `Object.equals(Object)`, if and only if their IDs are equal and if they belong to the same *Unified API* connection.

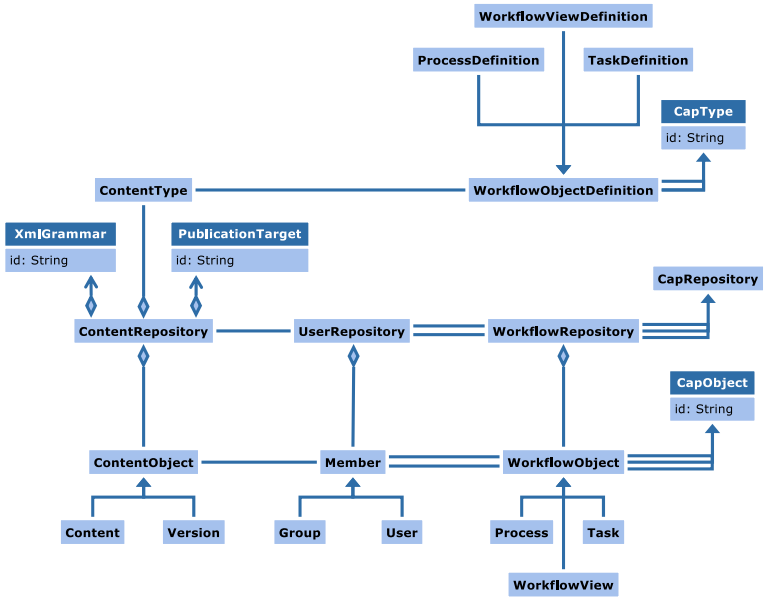

Figure 4.4. Class Diagram: Repositories and Identified Objects

It is recommended to use the IDs for external storage only. The identified objects of the *Unified API* are lightweight, so that it makes no sense to store IDs in main memory for a long time. IDs are more difficult to handle and often larger than their object counterparts. But for storage in a database or a file, IDs come in handy.

IDs are also used by some of the administrative command line tools and for generating debugging output.

4.7 Listeners

The *Unified API* allows you to attach listeners to the repositories and certain services. The base interface of all listeners is `CapListener`. The base class of all events is `CapEvent`.

In Figure 4.5, "Class Diagram: Listeners" [43] you see the type hierarchy of the class `CapListener`. Normally, you will want to implement one of the repository listeners, but there are occasions when you need the events of a service listener or a connection listener.

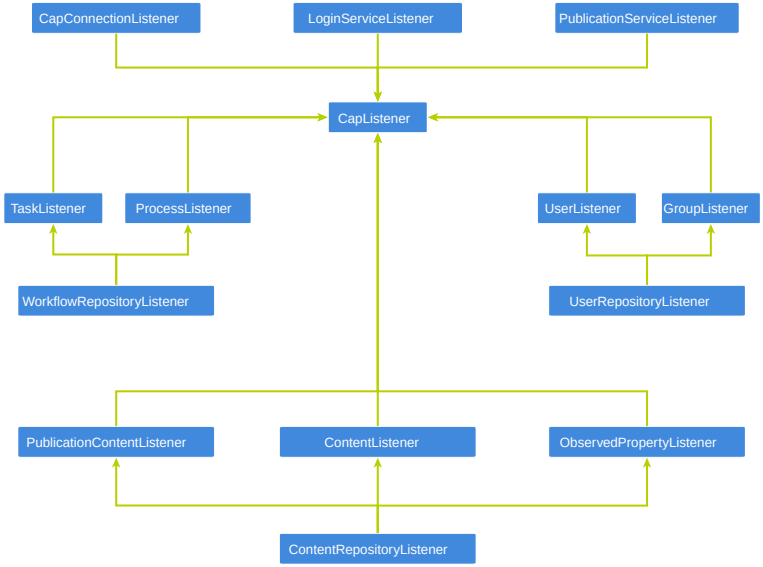

Figure 4.5. Class Diagram: Listeners

Most listener classes come with an abstract handler class whose name can be derived by adding `Base` to the interface's name. You can inherit from these classes when you want to handle only a small subset of the events provided. For example, a `ContentRepositoryListener` might be based on the class `ContentRepositoryListenerBase`.

Listeners are informed about changes asynchronously. No guarantees are made about the possible delays. However, it is assured that a listener will receive exactly those events that arise out of operations that are executed after the listener is added and

before the listener is removed. Furthermore, all changes of an operation are visible through the *Unified API* before a listener is notified about the changes. In particular, the internal cache is invalidated as needed. It may be, however, that subsequent operations have already overwritten the state that was generated by the operation that caused the event. For example, a renamed document may have already been re-renamed before the event of the first rename operation is processed.

Listeners are called in a single thread. Events are processed in order and each event is delivered to all interested listeners before the next event is handled. This means that a slow listener can create a backlog of unprocessed events, even for other listeners. It also implies that listeners must not wait for events to arrive at other listeners.

You can set a listener priority to define the order of notification when adding a listener. A listener with a higher priority will be notified about a single event before all listeners with a lower priority. The default priority is `CapListener.DEFAULT_PRIORITY`.

Listeners may access the *Unified API* for processing events. They may even make write calls that cause additional events. However, a listener must not add or remove listeners. It may not even remove itself from the set of listeners. Spawn a separate thread if you have to do this.

When a `CapConnection` is closed, all listeners that are attached to the connection or its repositories are automatically removed. No more events are delivered, even when the connection is reopened. If desired, new listeners have to be attached.

4.8 Exceptions

All exceptions that are thrown by the Unified API are derived from the single class `CapException`, which is a runtime exception.

Because runtime exceptions are used, you do not have to catch the exceptions explicitly. The exceptions are documented in the Javadoc, however, so that you can easily catch those exceptions that you expect and can handle reasonably. You will find the list or error codes linked on the Online Documentation site.

Exceptions are equipped with error codes that simplify the analysis of the actual problem. However, these error codes are not supposed to be used algorithmically. The codes may change at any point of time in the future. They are solely intended for debugging purposes.

Instead, for the most important problems and groups of problems, own exception classes were created. These exceptions can be treated specially in order to recover from errors. They will not change, although new exceptions may enter the hierarchy.

Individual exception classes can provide further hints about the problem at hand. For example, a `ContentException` references the content that was involved into the failed operation.

As it is possible for a write buffer flush to occur almost everywhere, it is possible that the associated `FlushFailedException` is thrown at almost every point in the code. If an application cannot be made robust with respect to such exceptions, care must be taken to flush all writes as soon as possible after the setters were called.

Some method calls involve bulk operations, that is they operate many resources at one time. When such an operation fails, a `BulkOperationFailedException` is thrown. From that exception you can retrieve the `BulkOperationResult` that provides more details on the failed operation. Bulk operations only return normally when they succeed completely. This ensures that a problem is detected reliably as soon as possible.

4.9 Sessions

Having opened a `CapConnection`, all actions are executed on behalf of the single user whose credentials where provided when logging in. In some contexts, it is desirable to use different users for different tasks while maintaining a shared cache. To this end, the *Unified API* allows you to use multiple sessions per connection.

Every session is represented by an instance of `CapSession`. The session that is created while the connection is opened is also known as the *connection session*. Additional sessions can be opened by the connection's login methods. Having obtained a session, this session can replace the default connection session by calling the method `setSession(CapSession)` on the connection. Alternatively, you can call `activate()` on the session. Afterwards, all accesses in the same thread are performed on behalf of the new session.

```
CapSession session = connection.login(user, password);
try {
  session.activate();
  ...
} finally {
  session.close();
}
```

Example 4.1. Open a session

The previous code fragment shows how a second session is created from an existing connection. Notice that the call to `activate` was necessary, because the login call does not automatically set the session. Only between `activate` and `close` you can see the newly created user as the user of the current session. In fact, in other threads the original session still applies. After closing the session, the connection session is again active.

The call to `activate()` returns the previously set session. The above code assumes that the previous session is not worth remembering. After closing a session, the thread's session automatically returns to the connection session. Another example at the end of these sections shows how the old session can be reestablished.

In other cases you might want to save the original session and reestablish it after the work of the second session is complete, without closing the second session. That way you can save the time that is required for opening the session. Of course, a session that is held open consumes a concurrent license all the time.

All accesses to the repositories are subject to the limitations of the requested session. During reads and writes, the rights check is based on the identity of the session's user. Write rights may happen to be reduced, but it is also possible that additional rights are gained by switching to another user. However, the read rights available to any session

are at most the read rights of the connection session. This is required in order to ensure efficient caching and to avoid accidental information leaks. Due to this restriction, it is recommended that the connection session's user should be allowed to read all repositories in their entirety, if additional sessions are expected to be created.

When attaching a listener using the *Unified API*, the current session is recorded. Before events are delivered to the listener, that session is reestablished as the current session. This way, listeners inherit the privileges of the code that attaches them.

Note that it is always possible to reset the current session to the connection session. Therefore, setting the current session is not sufficient for enforcing access restrictions when a `CapConnection` object is passed to untrusted code. Multiple sessions show their greatest potential in trusted applications which receive help in restricting user views while maintaining a shared cache.

Certain privileged connections have the ability to create new sessions for arbitrary users without providing a password. In particular this is true for the workflow service. In this case, logging in another session might be as simple as:

```
User user = ...;
CapSession session = connection.login(user);
...
```

Example 4.2. Log in another session

Note that it is not possible for ordinary user code to create a privileged connection. Instead, a privileged connection is returned by framework methods like `WfServer.getConnection()`. The default connection in the *Studio Server* is also privileged.

In the case of a privileged connection, you may also use a `com.coremedia.cap.common.pool.CapSessionPool` to obtain sessions temporarily. This class keeps a pool of sessions, which can greatly speed up your application if you change sessions often. Note that you still have to activate a session after it has been retrieved from the pool.

```
CapSessionPool pool = new CapSessionPoolImpl();
pool.setConnection(connection);
...
CapSession session = pool.acquireSession(user);
CapSession oldSession = session.activate();
try {
  ...
} finally {
  pool.releaseSession(session);
  oldSession.activate();
}
```

Example 4.3. Using a session pool

See `CapSessionPool` for further configuration options.

Write Buffering

When writing properties of a `CapObject`, these writes are initially buffered per thread and not sent to a server. Afterwards, the accumulated changes are sent to the server during a `flush()` call on the `CapConnection` object.

Buffering the changes per thread and not per session simplifies concurrent programming using the *Unified API* and reduces lock contention when a session is reused across threads.

The write buffers are also flushed when a call is made that cannot be handled locally by the *Unified API*. Currently, all calls except setters and getters will flush the write buffers, but this may change in future versions.

It is a good practice to flush the write buffers before any user interaction is resumed, before long delays are expected, and before returning from public methods that may be called from arbitrary code.

4.10 Caching

As long as a connection is open, it maintains an internal cache to avoid unnecessary refetches of persistent data from the servers. You can configure the size of the data that is cached in behalf of the connection by means of the connection's management bean.

You are free to use the cache for your own purposes, in particular for maintaining aggregate views on persistent data. Typically, this is done using the framework of the *CoreMedia Content Application Engine* as described in the *Content Applications Developer Manual*. The *Content Application Engine* includes code generators for the rapid implementation of custom cacheable beans. You can also access the cache directly by means of the `getCache()` method of the connection object. Please refer to the Javadoc of the class `com.coremedia.cache.Cache` for details about this class.

Almost every read call is cache-aware, meaning that the cache will timely invalidate cache entries that performed some operations by means of the *Unified API*.

There are, however, some exceptions to this rule. Results of queries or search requests will never be cacheable. Such computations are invalidated right away after being completed. Therefore, these operations tend to be relatively expensive. When accessing user data that is stored in an LDAP repository, invalidations are time-based. That is, computed values will eventually be removed from the cache, but they may be present for a while in order to improve performance. Other than that, caching and automated invalidation is fully available.

Please note that each time the connection is closed and reopened, a new instance of the cache is build. The cache cannot be used after the connection is closed, not even for tasks unrelated to the *Unified API*.

4.11 Serialization

Most objects returned by the *Unified API* support object serialization as per `java.io.Serializable` for persistent storage. While you should normally keep all persistent CMS data in the content and workflow repositories, serialization might be appropriate for short term storage, for example when maintaining conversational state in a web application.

Serialization itself requires no additional setup. Mutable objects will write their identity to the `ObjectOutputStream`, while values write their value. One piece of information is lost, however: the connection is not written to the stream. This is because a connection maintains a complex dynamic state and because it keeps security credentials that should not be externally accessible.

Therefore, you have to provide a connection when deserializing a *Unified API* object. This is done by registering a connection at the class `DefaultConnection`. You can register a connection for the entire JVM. However, CoreMedia recommends that you register a connection specifically for the thread that deserializes the objects. For an example, see the following code fragment:

```
CapConnection old = DefaultConnection.setLocal(myConnection);
  try {
    object = objectInputStream.readObject();
  } finally {
    DefaultConnection.set(old);
  }
```

Here a specific connection `myConnection` is set before accessing the stream. By resetting the connection after deserializing, you avoid unexpected side effects to calling code.

Besides the connection, also its sessions, its repositories, and its services cannot be serialized. Moreover, `Blob` objects do not support serialization. While blobs provide a value semantics, storing them in the object stream would be undesirable due to their size, so that a write of a blob normally indicates an error. If you want to serialize blobs, you can do it manually by converting the blob to a byte array during a `writeObject` method.

CAUTION

Serialization is not recommended for long term storage. Future CMS releases might make incompatible changes to the stream format.

4.12 Further Reading

If you want to read more about setting up and configuring the various servers with which the *Unified API* interacts, CoreMedia recommends the Content Server Manual as further reading. In that manual, you will also find information on how to create users and grant the users access to the repositories.

The Javadoc provides much more detailed information about the interfaces and methods that make up the *Unified API*. It is suggested that you use the Javadoc as a reference while programming, but it is also useful for getting a more detailed overview.

Look at the class `com.coremedia.cap.Cap` in more detail to find out about the methods for establishing a connection. Now inspect `com.coremedia.cap.common.CapConnection`, but upon first reading view it solely as a means to get access to various repositories and to `close()` the connection after you are done.

Afterwards, you should have a look at the other classes in `com.coremedia.cap.common`. In particular, make yourself comfortable with `CapObject`, `CapType, CapEvent, CapListener, CapException`, and the type hierarchy of `Blob`. The package `com.coremedia.xml` is also recommended for dealing with XML properties.

The subsequent chapters will deal with the individual repositories and their functionality in more detail.

5. The Content Repository

The content repository stores versioned documents that are organized in a folder tree. It allows the user to create, retrieve, read and update stored documents and folders while checking access rights. It also ensures that content can be published from the *Content Management Server* to the *Master Live Server*.

The content repository is augmented by the following services:

- `AccessControl` for determine rights
- `PublicationService` for controlling the publication process
- `ObservedPropertyService` for accessing contents which have a given value in an observed property
- `QueryService` for performing structured queries
- `SearchService` for performing full text searches
- `PropertyService` for accessing persistent properties of the *Content Server*
- `WorkflowContentService` for finding workflows that access a given content

The `PublisherService`, the `WorkflowContentService`, and all modifying methods are only available on the *Content Management Server*.

5.1 Objects

The content repository is concerned with the handling of folders and documents. The *Unified API* presents folders and documents jointly through the interface `Content`, which is a sub interface of `CapObject`. In releases prior to *CoreMedia CMS 2005*, the term *resource* was used to refer jointly to folders and documents. But that term was meant to indicate a very much reduced signature that allowed only for those methods that are common to folders and documents. The interface `Content`, however, provides all methods that are applicable to either documents or folders. Besides `Content`, there is the `Version` interface, which represents a historic version of a `Content`.

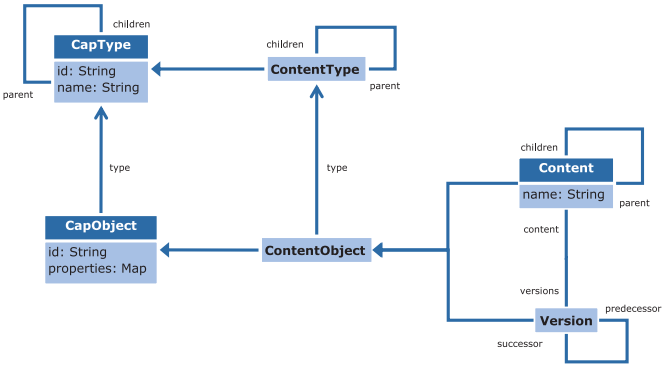

Figure 5.1. Class Diagram: Content and Versions

In the class diagram from Figure 5.1, “Class Diagram: Content and Versions” [53], you can see the above mentioned classes and their associations. The `ContentType` interface will be discussed later in Section 5.3, “Types” [59].

A document may have an arbitrary number of versions, which are linked in a predecessor/successor chain. You can get the versions of a document by means of `getVersions()`.

Besides these regular versions, checked-out documents have got a *working version* that represents their current state. The working version differs significantly from other versions. Most notably, its properties may change over time as the checked-out content is changed. Normally, you should not need to access the working version, as the associated content itself provides a richer and conceptually cleaner interface. For migrating legacy code, however, it might be natural to use the working version, so that a uniform interface is available.

Folders do not have any versions and they do not define any properties. Instead, they provide access to their children, which may be either documents or folders. You can retrieve all children or a child with a specific name by using the appropriate methods defined in `Content`.

There are quite a few methods that allow you to inspect the state of a content. You can query whether a document is deleted, whether it is checked out, who created it, and the like. This information is available as regular properties of the `CapObject`. You have to call the individual getter methods for obtaining this information.

A content supports many updating operations. In particular, it inherits the methods for setting properties from `CapObject`. Before changing the properties, a document must be checked out. After changing the properties, it may be checked in or, more rarely, be reverted to the original state. Keep in mind that, as noted in Section 4.9, "Sessions" [46], property changes are buffered and sent to the server only when the CapConnection is flushed explicitly.

Additionally, there a several other methods that deal with moving, renaming, copying, and deleting content. Currently, these operations are executed immediately. They are not buffered.

A `Content` object may enter various states during its lifetime. The full state space is quite large with over 50 different states. However, there are a number of orthogonal views that can be more compactly presented and that define the possible transitions completely.

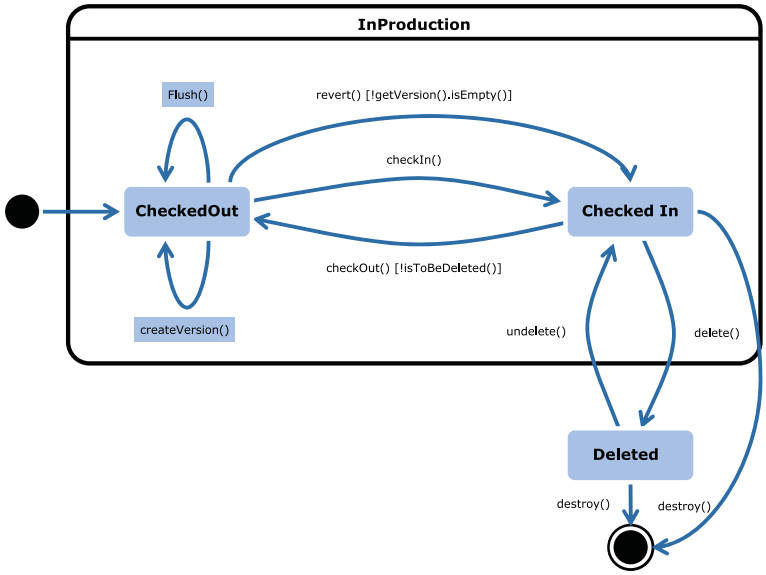

Figure 5.2. Statechart: Checked In and Out

Only documents may be checked in and out as described in Figure 5.2, “Statechart: Checked In and Out” [55]. Folders are always checked in.

The next figures apply to the publication process, which is handled by the `PublicationService` as described in Section 5.5, “Publication Service” [62]. Please refer to that section for details about the mentioned methods.

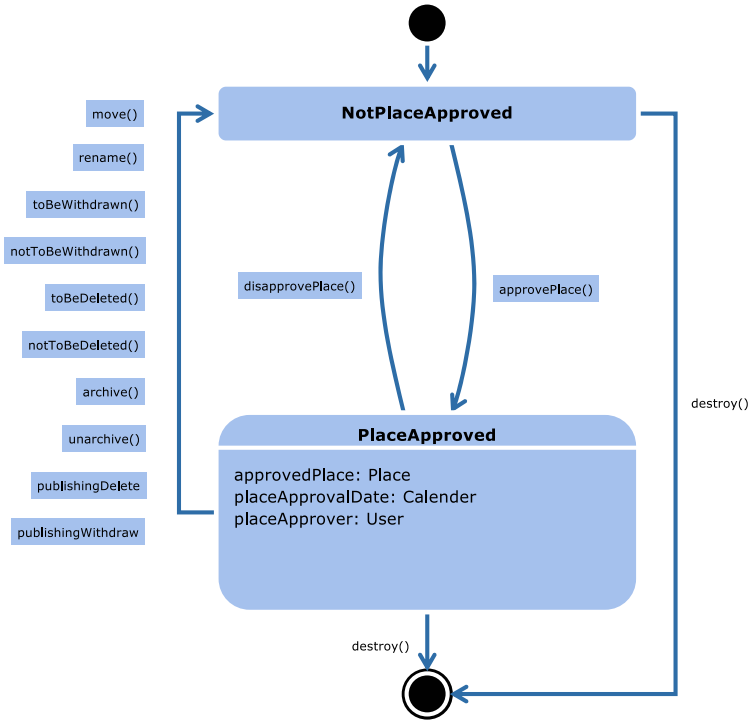

Figure 5.3. Statechart: Place Approvals

The place approval states of a content are quite simple, but they are shown in Figure 5.3, "Statechart: Place Approvals" [56] to indicate that a place disapproval can happen implicitly during a number of operations.

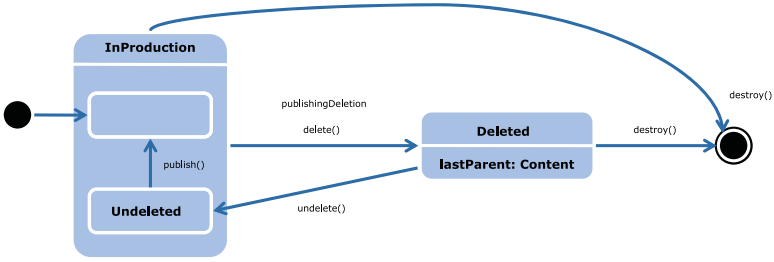

Figure 5.4. Statechart: Deleting

As shown in Figure 5.4, “Statechart: Deleting” [56], a content becomes deleted, when a deletion is published or when the content is deleted explicitly. It can be moved out from the *`Deleted`* state, reaching the *`Undeleted`* state, which it keeps until being deleted again or published.

One last state chart refers to the state of `Version` objects.

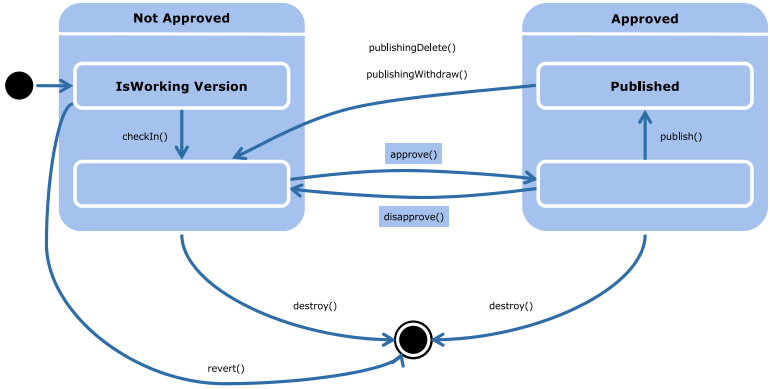

Figure 5.5. Statechart: Version

A version of a document is created when the document is created or checked out. In Figure 5.5, “Statechart: Version” [57] you can see the lifecycle of a version. Typically, the content is checked in, so that the version is promoted to a regular version and is no longer a working version. The version is then approved and published, so that it appears on the live system.

When the diagram references the destroy operation, this applies either to an explicit `destroy()` call of the version or the content, to an action of the document collector or version collector, or to a cleanup during publication when the publisher is configured to destroy intermediate unpublished versions.

5.2 UUIDs

In addition to the simple string identifiers described in Section 4.6, "Identifiers and Equality" [39], every `ContentObject` on the *Content Management Server* has a UUID since version 2004.1. UUIDs are stable and universally unique identifiers as defined in RFC 4122 and are represented as `java.lang.UUID`. They are not meant as replacement of simple string IDs, and should not be used where a simple ID is sufficient. UUIDs make sense in certain scenarios where uniqueness across multiple repositories is important, or when content objects may be transferred to another repository and should keep their identity. For details see: Section "Serverimport/Serverexport" in *Content Server Manual*.

Similar to string IDs, the API provides a `getUuid()` method in class `ContentObject` to retrieve a UUID, and methods to look up a `Content` or `Version` for a given UUID. A `Content` with a given UUID can be retrieved from the `ContentRepository` with method `getContent(UUID)`. A `Version` with a given UUID can be retrieved from its containing `Content` with method `getVersion(UUID)`. It is important to note, that a UUID does not encode information about the location of the `ContentObject`. By itself, it cannot be used to identify the repository or even the containing `Content` of a `Version`.

Note, that as of version 2004.1, UUIDs are only available on the *Content Management Server*. If a connection is made to a live server, or to a server of a previous release, then all methods that would return a UUID will return null instead.

5.3 Types

The types of both `Content` and `Version` objects are defined by `ContentType` objects. `ContentType` inherits from `CapType`, but not all kinds of properties are supported. Only integer, string, date, blob, XML, struct, and link list properties are provided.

Documents and versions are using the types that are configured at the *Content Server*. For folders there is a special pseudo-type without property descriptors. Two other abstract pseudo-types are provided: one for documents of any type and one for content in general, including folders and documents.

You can obtain a reference to a type by calling `ContentRepository.getContentType(String)` with the name of the type. The pseudo-types are provided by the methods `getFolderContentType()`, `getDocumentContentType()`, and `getContentContentType()`. The pseudo-types are properly integrated into the type hierarchy.

Types also allow you the creation of new content objects. To this end, you have to call one of the create methods and pass parameters that will allow the server to determine at least a name and a folder for the content.

5.4 Access Control

The `AccessControl` service of the content repository is responsible for maintaining the set of rights rules and for evaluating the rules to determine whether a user is allowed to perform a certain operation on content objects or not.

Overview Of Rights

The following rights are defined for the *Unified API*:

Right	Affected Operations
READ	read content
WRITE	write content
DELETE	move content to or from the recycle bin; destroy content; mark or unmark content for deletion or withdrawal
APPROVE	approve places and versions
PUBLISH	publish content
SUPERVISE	assign rights rules to content

Table 5.1. Rights for the Unified API

Instances of the class `com.coremedia.cap.content.authorization.Right` represent the rights defined here. `Right` objects are readily provided as constants, but also be created from shorthand characters. The rights `SET_TO_BE_WITHDRAWN` and `SET_TO_BE_DELETED` are aliases for the `DELETE` right.

Please have a look at the *Content Server Manual* for a more detailed discussion of rights and for a specification of how rights are derived from rules. That manual refers to the so-called *folder right*, which is represented in the *Unified API* as a combination of the write right and the delete right in rules that apply to the folder document type.

Checking Rights

The rights checks are performed by the methods `mayRead(Content)`, `mayApprove(Content)`, and the like. While most checks depend only on the given content object, the `mayCreate(...)` method must also be informed about the document type to be created.

Some of the methods also take the content's current state into account when computing the rights. For example, `mayCheckIn(Content)` will only return true when the content in question is actually checked-out and it takes into account that the user who checked out the content has special rights when it comes to checking it in.

There are convenience methods for checking an entire collection of content objects with one call. Such methods only grant a right if it would be granted on each individual content. There are generic `mayPerform(...)` methods, which are passed a `Right` object that denotes the actual operation to check.

Normally, the rights are checked for the user of the current session, but it is possible to specify a set of groups and compute the rights assuming the user is a member of exactly these groups.

Setting Rights Rules

Rights checks are based on rules. The `AccessControl` service offers methods for retrieving all rules or a subset thereof as a collection of `Rule` objects. Rule objects are a compact representation of all parameters that make up a rule: a content, a type, a group, and a rights mask. They do not provide modifying operations themselves. Instead, the `AccessControl` service provides methods for creating, modifying, and deleting rules.

Using the `AccessControl` service, it is also possible to check whether a rule already exists. Furthermore, you can retrieve all rules that apply to a certain content or group, respectively.

5.5 Publication Service

The `PublicationService` allows you to control the publication process and to inspect the state of the publication queue.

When a content is created, it exists on the *Content Management Server* only. The process of transferring the content to the *Master Live Server* is referred to as *publishing* the content. Before a content can be published, it has to be *approved*. In general, the approval of a content refers to its location in the folder tree. It is approved that the content appear on the *Master Live Server* at a given place, hence the name *place approval* for this type of approval which can be performed by the method `approvePlace(Content)`. When a document is published, a version must be created on the *Master Live Server*, too. To this end, the version itself must be approved using the method `approve(Version)`. Only an approved version can be published. Even if a content is published, subsequent movements, renames, and property changes happen on the *Content Management Server* only. New places or new versions must be published explicitly.

When a content is supposed to leave the *Master Live Server*, it must be marked for *withdrawal* or *deletion* using the methods `toBeWithdrawn(Content)` and `toBeDeleted(Content)`. After that operation is place approved, the content can be included in a publication set. During the subsequent publication, the content is removed from the *Master Live Server* instead of being updated. In the case of a mark for deletion, it is also moved into the archive on the *Content Management Server*.

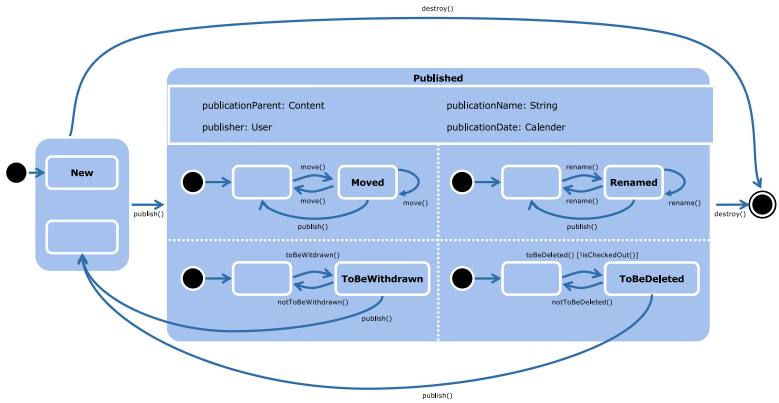

Figure 5.6. Statechart: Content Publication

A rather complex state chart is shown in Figure 5.6, "Statechart: Content Publication" [62]. It depicts the various states with respect to publication. Being published, a content gains access to a number of attributes that are only available in this state. Its state space is fragmented into four sub spaces:

- it might be moved
- it might be renamed
- it might be marked for being withdrawn
- it might be marked for being deleted

A publication is initiated by the `publish(...)` methods. You can also request a publication preview by means of the `preview(...)` methods. A preview does not actually copy information to the *Master Live Server*, but makes all checks to determine whether a publication would be successful. Possible arguments to the publish and preview calls are a single content, a collection of contents, or a `PublicationSet`.

When contents are given as argument, the actual publication set is determined heuristically. To this end, the publication service selects versions to be published with the content, if that is appropriate given the current marks and approvals. You can also create a publication set by providing collections of contents and versions explicitly, taking care that no versions are included whose content is marked for withdrawal or deletion.

After a publication has completed successfully, a `PublicationResult` is returned. The publication result informs about all contents that were involved in the publication and about the actions that were performed. If the publication is unsuccessful, a `PublicationFailedException` is thrown, which wraps a publication result that details the cause of the error.

As an example, let us look at an excerpt from the class `PublicationServiceExample` that is available as a source code example:

```
PublicationService publisher = repository.getPublicationService();
publisher.approvePlace(folder);
publisher.publish(folder);
publisher.toBeDeleted(folder);
publisher.approvePlace(folder);
publisher.publish(folder);
```

A folder that has been created before is approved, published, marked for deletion, approved again, and deleted by publishing. This example summarizes the entire lifecycle of content publication in a few lines. Obviously, real applications will not use all of these methods in one place.

The publication service also provides a means to inspect the current state of the publication queue. You can get a list of all pending publications and access a summary of each publication's characteristics. A `PublicationServiceListener` informs about changes to the publication queue.

If you have enabled *Multi-Master Management* for your *CoreMedia CMS*, there may be more than one publication target. Each publication target represents one *Master Live*

Server and includes any number of base folders. You can retrieve all `PublicationTarget` objects from the publication service. The *Content Server Manual* provides more information on how to set up publication targets.

5.6 Observed Property Service

The `ObservedPropertyService` allows you to access contents which have a given value in an observed property. A content property is observed when the property in the content type definition is annotated with `extensions:observe`. See Section "Changing the Observe Attribute of a StringProperty" in *Content Server Manual* for the configuration of an observed property. Currently only a string property with a maximum length of 256 is supported. For such an observed property and a given non-empty value the Content Server maintains the set of contents whose observed property has the value. The `ObservedPropertyService` offers methods to retrieve the contents in a cached and dependency-tracked way.

Example: Given is a content type `ExternalProduct` with an observed string property `externalId`. Now, the set of external product contents whose external ID is "acme sportswear 123" can be retrieved by `observedPropertyService.getContentsWithValue("acme sportswear 123", externalIdDescriptor)` whereas `externalIdDescriptor` is the content property descriptor of the `externalId` property.

The same could be achieved using the query service. But the result retrieved by the `ObservedPropertyService` is cached and dependency-tracked, which is more efficient. Additionally, for every change of the set of contents with the observed value the content repository sends a corresponding event.

The `ObservedPropertyEvent` is thrown when a set of observed contents has changed. See the Javadoc for details.

CAUTION

Beware that the `ObservedPropertyEvent` reveals the value of the observed property to the listener. For example, the external IDs of the `ExternalProduct` contents maintained in the CMS could be collected without having the proper access rights to the Content Repository.

5.7 Query Service

The `QueryService` allows synchronous, structured queries against the content repository. A query is a string formulated in the *CoreMedia* query language, with an optional array of parameter objects which may be referenced from the query string. The number of results to return can be limited. There are two variants of queries, those applying to content objects and their current property values, and those applying to versions (including working versions). Queries are initiated with the `poseContentQuery` and `poseVersionQuery`, respectively.

A query string expresses a condition on content objects. The `QueryService` returns all `Content` or `Version` objects in the repository for which the condition is true. The condition is made up of a logical combination [`AND`, `OR`, `NOT`] of type constraints, comparisons and tests, which may refer to the object's properties.

A query may also refer to getters defined in the `Content`, `Version` and `PublicationService` interfaces in the same way it refers to user-defined properties, by giving its name. The names of API methods are transformed as follows:

- `Content`

  For a zero-argument method whose name starts with `get`, the implied property name omits the `get` and starts with a lowercase letter. So the method `getCreationDate()` becomes `creationDate`. For a zero-argument method whose name starts with `is`, the implied property has the same name as the method. So the method `isCheckedIn()` becomes `isCheckedIn`.

- `Version`

  Transformation is similar to `Content`, but to avoid confusion, all getters are prefixed with `version`. So `getEditor()` becomes `versionEditor`. Boolean-valued getters start with `versionIs`.

- `PublicationService`

  A one-argument method that takes a `Content` as its argument is transformed as if it were a zero-argument method defined in class `Content`, and analogously for `Version`. So `isApproved(Version)` becomes `versionIsApproved` and `getPublisher(Content)` becomes `publisher`.

An implied property based on a `Version` getter is only defined for documents, not for folders. In a content query, it implicitly refers to the working version for a checked out document, or to the latest version for a checked in document.

There are three implied properties with a deviating semantics:

- The word `id` represents the current content (and not the string returned by the method `getId()` defined in `Content`).
- For version queries, the word `version` represents the current version. For content queries, it represents the working version for checked-out content, and the latest version for checked-in content.
- The implied property `containsWideLink` is true if the content or version contains a link (in a link list or in XML) to a content that belongs to a different base folder.

The following implied properties are currently defined:

- `baseFolder`
- `containsWideLink`
- `creationDate`
- `checkedInVersion`
- `checkedOutVersion`
- `creator`
- `editor`
- `id`
- `isDeleted`
- `isCheckedIn`
- `isCheckedOut`
- `isDocument`
- `isFolder`
- `isInProduction`
- `isMoved`
- `isNew`
- `isPlaceApproved`
- `isPublished`
- `isRenamed`
- `isToBeDeleted`
- `isToBeWithdrawn`
- `isUndeleted`
- `lastParent`
- `latestApprovedVersion`
- `latestPublishedVersion`
- `modificationDate`
- `modifier`
- `name`
- `parent`
- `placeApprovalDate`
- `placeApprover`
- `publicationDate`
- `publicationName`
- `publicationParent`
- `publisher`

- `versionApprovalDate`
- `versionApprover`
- `versionEditionDate`
- `versionEditor`
- `versionIsApproved`
- `versionIsPublished`
- `version`
- `versionPublicationDate`
- `versionPublisher`
- `workingVersion`

When an `ORDER BY` clause is given, the query result is sorted according to the values of the given properties. These properties must be defined for all content types for which the condition may be true.

Please refer to the Javadoc of class `QueryService` for a comprehensive list of implied properties available in query expressions.

The query syntax is as follows:

```
query ::=
  conditional_expression [ order_by ] [ limit ]
  ;

order_by ::=
  ORDER BY order_entry {"," order_entry}
  ;

limit ::=
  LIMIT numeric_literal
  ;

order_entry ::=
  property [ ASCENDING | ASC | DESCENDING | DESC ]
  ;

conditional_expression ::=
  TYPE ["="] type { "," type } [":" conditional_expression]
  | conditional_expression OR conditional_expression
  | conditional_expression AND conditional_expression
  | NOT conditional_expression
  | ( conditional_expression )
  | BELOW content
  | REFERENCES content
  | property REFERENCES content
  | REFERENCED
  | REFERENCED BY versionOrContent
  | property IS [NOT] NULL
  | comparison_expression
  | contains_expression
  | value_expression
  ;

type ::=
  identifier
  ;

comparison_expression ::=
  value_expression comparison_operator value_expression
  ;

comparison_operator ::=
```

```
  "=" | ">" | ">=" | "<" | "<="
  ;

contains_expression ::=
  property CONTAINS literal_expression
  | property CONTAINS EXACT literal_expression
  | property CONTAINS PREFIX literal_expression
  | property CONTAINS STEM literal_expression
  ;

value_expression ::=
  property
  | literal_expression
  ;

property ::=
  implied_property
  | identifier
  ;

content ::=
  literal_expression
  ;

version ::=
  literal_expression
  ;

versionOrContent ::=
  literal_expression
  ;

literal_expression ::=
  string_literal
  | numeric_literal
  | boolean_literal
  | DATE string_literal
  | PATH string_literal
  | USER string_literal
  | ID string_literal
  | input_parameter
  ;

boolean_literal ::=
  TRUE
  | FALSE
  ;
```

NOTE

The operator "=" in a comparison expression of String literals is handed over to the database. Thus, it depends on the database if the operator is case-sensitive or not.

Identifiers consist of Java identifier characters. Where the name of an identifier collides with a keyword or an implied property, the identifier can be enclosed in double quotes to preserve its meaning as an identifier. Examples:

- `Article`
- `title`
- `Document_`
- `"parent"`

- `"DATE"`

String literals are delimited by single quotes. A single quote inside a string literal is represented by two successive single quotes. Examples:

- `'hello world'`
- `'banker''s baguette'`

Numeric literals conform to Java syntax. Essentially, a numeric literal is a sequence of digits, optionally preceded by a minus sign. Examples:

- `123`
- `-3`

As date literals, the String has to be of the form recognized by `DateConverter`. This class generates and parses a subset of ISO8601 strings, namely those matching `yyyy-MM-dd'T'HH:mm:ssTZD` where the time zone distance `TZD` is expressed as `+hh:mm` or `-hh:mm`. Examples:

- `DATE '2004-09-08T13:47:07-02:00'`
- `DATE '2004-12-31T23:59:59+00:00'`

`PATH` literals denote a content by giving its path, beginning at the root folder. It is an error if no content exists at the given path. Examples:

- `PATH '/Home/admin'`
- `PATH '/'`

`USER` literals denote a user name and a domain name separated by an `@` character. If the domain name is empty, the `@` character may be omitted. Examples:

- `USER 'admin'`
- `USER 'fred@msad'`

`ID` literals denote a content, version or user by giving its ID, as returned by `CapObject.getId()`. Examples:

- `ID 'coremedia:///cap/content/1'`
- `ID 'coremedia:///cap/version/4/2'`
- `ID 'coremedia:///cap/user/0'`

An input parameter refers to an object passed along with the query string. Input parameters are represented by a question mark immediately followed by a sequence of digits, which represents the zero-based index of the parameter object. Examples:

- `?0`
- `?1`
- `?42`

Type rules

Queries are strongly typed. For example, when you try to compare a String with an Integer, an error (MalformedQueryException) will be reported. There are rules that govern the required types for subexpressions, and the resulting type of each expression in the grammar above.

The following tables show all possible types that occur in subexpressions of a query. The second column shows corresponding Java types, which is relevant for parameter objects and implied properties. The third column shows the corresponding CapPropertyDescriptor type of a content property, which is relevant for non-implied properties.

Type	Java type	CapProperty-Descriptor type
Blob	n.a.	`BLOB`
Boolean	`java.lang.Boolean`	`BOOLEAN`
Date	`java.util.Calendar`	`DATE`
Integer	`java.lang.Integer`	`INTEGER`
String	`java.lang.String`	`STRING`
Content	`com.coremedia.cap.content.Content`	n.a.
Version	`com.coremedia.cap.content.Version`	n.a.
Content List	n.a.	LINK list
Markup	`com.coremedia.xml.Markup`	`MARKUP`
User	`com.coremedia.cap.user.User`	n.a.

Table 5.2. Types in subexpressions

The type of an `implied_property` clause depends on the return type of the corresponding getter method. For example, the return type of `Content#getName()` is `java.lang.String`, so the type of the expression `name` is `String`. There are three exceptions: For `id`, the type is `Content`. For `version`, the type is `Version`. For `containsWideLink`, the type is `Boolean`.

An `identifier` in a `property` clause is resolved as a property of the content type in the closest surrounding `TYPE` clause. The property type is then mapped to a query expression type using the table above. For example, assuming a string property called `headline` in a content type `Article`, the subexpression `headline` in the query "`TYPE Article: headline CONTAINS 'foo'`" would have type `String`. If there is no surrounding `TYPE` clause, the content type `Content` is assumed, which does not define any properties. If the `TYPE` clause lists multiple content types, the type of the property with the given name has to be same in all listed content types.

A property name in an `order_entry` clause is resolved as a property of the most specific type that can fulfill the query. Only properties of type `Boolean`, `Date`, `Integer`, or `String` are allowed.

In a `REFERENCES` clause, the property (if given) must be a Markup or Link List property.

In a `comparison_expression`, the types of both subexpressions must be the same, and must be one of `Boolean`, `Date`, `Integer`, or `String`, or one subexpression must be of type `Integer` and the other type must be integer compatible. `User`, `Version` and `Content` are integer compatible, by using the user id, content id, or version number for comparison.

The property in a `CONTAINS` expression must be a String or Markup property. The literal must be a `String`.

Where a `value_expression` is used as a `conditional_expression`, the `value_expression` must be a `Boolean`.

The expression type of an `input_parameter` depends on the class of the java object passed as a parameter. The mapping from Java type to expression type is given in the table above. For example, when passing in an instance of `java.lang.String`, the corresponding parameter expression will have the type `String`.

Where a `content`, `version`, or `contentOrVersion` clause is used in the grammar above, the `literal_expression` must have the respective type.

Interpretation

So far, you have seen when a query is syntactically correct, and when its types are correct. This section describes what the query expression actually means, where it was not explained before.

The following description is geared towards content queries (`QueryService#poseContentQuery`). In a version query, where the following description refers to a "content", the version's content is understood. Where the description refers to a "content object", the version itself is understood.

A "`TYPE =`" condition is true for a content if the content's type is one of the types listed, and the content fulfills the condition on the right hand (if given). This form does not take type inheritance into account.

A "`TYPE`" condition (without "=") is true if the content is a (direct or indirect) instance of at least one of the types listed, and the content fulfills the condition on the right hand (if given). This form takes type inheritance into account.

A "`BELOW folder`" condition is true for a content if the content is a child of the given folder. For the purpose of this condition, a folder is a child of itself.

A "`REFERENCES target`" condition is true for a content object if the content object contains a link to the given target in any markup or link list property.

A "`property REFERENCES target`" condition is true for a content object if the named property of the content object contains a link to the given target. The property must be a markup or link list property.

A "`REFERENCED`" condition is true for a content if it has at least one referrer, that is a content containing a link to this content in any markup or link list property.

A "`REFERENCED BY contentOrVersion`" condition is true for a content if the given content or version has a reference to this content in any markup or link list property.

A "`property CONTAINS literal`" condition is true for a content object for a string property, if the literal's string value is a substring of the content object's property's string value. For a markup property, all XML markup is discarded, and the string is searched for in the concatenated `cdata` elements.

`CONTAINS EXACT`, `PREFIX`, and `STEM` are only available if your database supports them. Currently only Oracle databases with the special module "multimedia" (formerly named *interMedia*) support this.

In an `order_by` clause, the first `order_entry` takes priority. If contents compare equal according to the first order entry, the next order entry is considered, etc. The ordering of `String` values is database dependent. The ordering of `Date` values ignores the time zone. The `Boolean` value FALSE is considered less than TRUE.

The `limit` clause limits the number of results that will be returned, and may improve performance, especially if only one result is required, and if sorting is not requested. It is equivalent to passing a limit argument to the query service method.

Examples

Search for a specific ID

This query returns the object with the given ID.

```
Collection<Content> result =
  qs.poseContentQuery("id = 1486")
```

Items reference another content item

This query gives all content items that reference a given item, defined by its content ID:

```
Collection<Content> result =
  qs.poseContentQuery("REFERENCES ID 'coremedia:///cap/content/1486'")
```

Articles that contain specific text

Consider that you are searching for all content items of type `CMArticle` that are not deleted and that contain the word 'Gin' in its `detailText` property:

```
Collection<Content> result =
  qs.poseContentQuery("TYPE CMArticle: NOT isDeleted AND detailText CONTAINS
 'Gin'")
```

> **NOTE**
>
> This query will find all occurrences of 'Gin' even it is part of, for example, 'Ginger'. There is the clause CONTAINS EXACT which would only find 'Gin', but at the moment, it is only supported by Oracle databases with the multimedia module.

Search for the latest published versions

This query will find the last published versions of all CMChannel content items.

```
Collection<Content> result =
  qs.poseVersionQuery("TYPE CMChannel: versionIsPublished AND
version=latestPublishedVersion")
```

Search for the latest published version of a specific content item

This query will find the last published versions of content item with ID 586.

```
Collection<Content> result =
  qs.poseVersionQuery("TYPE Document_: versionIsPublished AND
version=latestPublishedVersion AND id=586")
```

Search for all content items lastly edited by a user

Consider that you are searching for all documents that were lastly edited by the user `admin`. Given that the variable `qs` holds a reference to the query service, you could issue the following statement:

```
Collection<Content> result =
  qs.poseContentQuery("TYPE Document_: editor = USER 'admin'")
```

All content items checked out by a user

Consider that you are searching for all documents that are checked out by the user `admin`.

```
Collection<Content> result =
  qs.poseContentQuery("TYPE Document_: isCheckedOut AND editor = USER 'admin'")
```

Published content below a specific folder

This statement retrieves arbitrary published content that is stored in the folder `/Home`. At most 50 results are returned.

```
Collection<Content> result =
  qs.poseContentQuery("isPublished AND BELOW PATH '/Home' LIMIT 50")
```

Items of type Articles, marked for deletion or withdrawal, approved before the given date

A parametrized query is shown that retrieves all documents of the type `Article` that are marked for deletion or withdrawal and were approved before the given date.

```
Collection<Content> result =
  qs.poseContentQuery("TYPE Article: placeApprovalDate < ?0 AND "+
  "(isToBeDeleted OR isToBeWithdrawn)",
  maxApprovalDate);
```

Search in structs

As structs are a subset of XML properties you can search for a specific text using CONTAINS. However, it is not possible to address one specific inner property of a struct.

5.8 Search Service of the Unified API

The `SearchService` provides full-text search capabilities. You can use its methods to quickly find contents based on their current property values and some of their implied properties (such as content creator).

Client search requests are routed through the *CoreMedia Content Server* to the *CoreMedia Search Engine*. The *CoreMedia Search Engine* is a facade to a configured third-party search server which by default is an Apache Solr instance. The search server returns the search result back to the *CoreMedia Content Server* and the requesting client.

NOTE

Note, that the index of the *CoreMedia Search Engine* is updated asynchronously and therefore does not always represent the current state of the content repository. Note further, that the *CoreMedia Search Engine* does not allow searching in old document versions.

If you need up-to-date results or want to search for document versions, you should consider using the `QueryService`.

The `SearchService` has the following methods:

- `isSearchEnabled` returns true, if search service is enabled, false otherwise.
- `search` methods to search for not deleted contents using a simple query language as described below.
- `searchNative` to search in a search server specific search query language, like Apache Solr Query Language described below.

CAUTION

Note, that `search`, and `searchNative` methods may return contents for which the user of the current session does not have read rights. You must handle rights yourself and filter out unreadable contents if needed.

Search with Simple Query Language

Use one of the `search` methods to perform a simple search for not deleted contents. There are multiple search methods with different parameters to restrict the query to contents of a given type and below a given folder. See the API documentation for details.

These methods take a query string in a simple query language which consists of terms and/or phrases separated by white space. The terms and/or phrases are combined with a logical AND.

A query term is basically a word to search for. Only alphanumeric characters are allowed here. You can prefix the term with a minus operator ('`-`') to indicate a NOT expression, that is the word must not appear in the search results. Likewise, you can use a plus operator ('`+`') as prefix but it is the default and will be ignored. The following example query will search for contents which contain the word `news` but not the word `sport`:

```
news -sport
```

The query term may end with an asterisk ('`*`') to perform a wildcard query which matches all words that start with the characters before the asterisk. Note, that the asterisk may appear at the end of the term only. The next example returns all contents containing words that start with `test`:

```
test*
```

A phrase to search for is enclosed in double quotes. Wildcards are not allowed in phrases and plus and minus operators are ignored. A search for contents containing the phrase `Hello World` can be performed with:

```
"hello world"
```

The following example searches for documents of any type (but not folders) which contain the word `hamburg` below a folder `/Site`. The list obtained from the `SearchResult` contains the found `Content` objects sorted by their name. Sorting is explained in a following section.

```
Content site = contentRepository.getChild("Site");
ContentType type = contentRepository.getDocumentContentType();
SearchResult result =
 searchService.search("hamburg",     // the query string
                      "name", true, // sort ascending by name
                      site, true,   // below folder site
                      type, true,   // documents only
                      0, 100);      // max. 100 hits
```

```
List<Content> hits = result.getMatches();
```

Search with Solr Query Language

If you need a more powerful search with Apache Solr directly, you can use the more generic `searchNative` method and perform a query in the Solr Query Language. For details on the query language refer to the Apache Solr documentation.

One of the fields in the Solr schema is `creator` which contains the ID of the user who created the content. The following example searches for all contents created by user `admin` that are located below the folder `/Site` and contain the word `test`.

```
User user = userRepository.getUserByName("admin");
int userId = IdHelper.parseUserId(user.getId());

Content folder = contentRepository.getChild("/Sites");
int folderId = IdHelper.parseContentId(folder.getId());

String query = "feederstate:SUCCESS" +
               " creator:" + userId +
               " folderpath:" + folderId +
               " test";

SearchResult result = searchService.searchNative(
  query,
  "name", true,      // sort ascending by name
  0, 100);           // max. 100 hits

List<Content> hits = result.getMatches();
```

An important thing to note is the term `feederstate:SUCCESS` within the query string. You must specify this term in every query except when searching for contents that were not successfully indexed. In the latter case you must include the term `feederstate:ERROR`. If you don't want to find contents in the recycle bin, you must either search below a given folder, as shown in the example above, or include the term `isdeleted:false`.

The term `test` is not prefixed with the name of an index field. In that case the default field `textbody` is used and the search is performed on the full-text content.

Sort search results

You can use method parameters of the `search` and `searchNative` methods to specify the sorting of the returned results. The search methods take the name of the search field and whether sorting should be ascending or descending as parameters and return the results sorted accordingly. Sorting is handled by the search engine which is much more efficient than client-side sorting could be.

5.9 Workflow Content Service

Content objects are often edited and published in formalized business processes. To this end, the `WorkflowRepository` as described in Chapter 6, *The Workflow Repository* [83] provides a means to define your own processes, which may refer to documents and folder using their variables. Accessing content through workflow objects is well supported by the instances of the `Task` and `Process` interfaces. But sometimes it is interesting to know the processes in which a certain document or folder is processed.

To this end, the `WorkflowContentService` provides the method `getProcesses(Content)` for obtaining a collection of processes that reference a given content. It is up to you to determine the exact variable and possibly task that is currently dealing with the content, if desired. Usually obtaining a reference to the process is enough to perform the remaining operations efficiently.

5.10 Property Service

A *Content Server* can store an arbitrary number of persistent key/value pairs in a dedicated database table. These values can be used to store global resource-independent states. For example, some values are used internally to store the current database schema version and other information.

The `PropertyService` allows you to access this persistent store. The `PropertyService` presents the table as a map from strings to strings. You can get individual entries or the entire map and you can set and remove key/value pairs.

It is advisable to keep the number of key/value pairs moderate. It would be possible to store the path of a configuration folder or perhaps the time of the last run of a certain script. It is not recommended to store individual values per document and folder.

Please refer to the documentation of `PropertyService` for information about reserved keys that may not be used for arbitrary purposes.

5.11 Listeners

The ContentRepository supports two different listeners: the `ContentRepositoryListener` and the `PublicationServiceListener`.

The `ContentRepositoryListener` can be registered directly at the `ContentRepository`. It receives events about content creation, update and destruction and about operations of the `PublicationService` on content, e.g, approvals, deletions or publications. Additionally, it receives events about the observed properties. The listener methods of these three categories of events are separated into three sub interfaces `ContentListener`, `PublicationContentListener` and `ObservedPropertyListener`. In addition, methods for handling rights rule changes are defined directly in `ContentRepositoryListener.`

While these interfaces highlight the conceptual differences between the various events provided to a `ContentRepositoryListener`, the full implementation of the entire `ContentRepositoryListener` is allowed when registering a listener. The class `ContentRepositoryListenerBase` helps with an abstract implementation when you want to react to a small subset of events.

When attaching a `ContentRepositoryListener`, you can provide a timestamp. The timestamp has to lie in the past, you might have obtained it, for example, when listening to an earlier event. Exactly those events that occurred after that timestamp will be propagated. Once the past events have been delivered, the event stream switches transparently to the live stream of events.

A special timestamp constant `Timestamp.SYNTHETIC_REPLAY` indicates that a synthetic sequence of events should be delivered instead of the real events that occurred in the past. A synthetic replay is typically shorter than a full historic replay, but the load and memory requirements on the *Content Server* while generating the synthetic events can be significant. If possible, it is recommended that you attach listeners with relatively recent timestamps or with no timestamp at all

The `PublicationServiceListener` is provided with events about the state of the publisher itself. An event is sent whenever a publication is enqueued, started, completed or aborted. The listener is also informed when the publication targets of a *Content Management Server* with *Multi-Master Management* are redefined.

5.12 Further Reading

To learn more about the design of a document type system, see the corresponding chapter in the *Content Server Manual* it will also provide you with additional information on rights and rights rules.

The Javadoc of the *Unified API* is the recommended source for in-depth descriptions of individual classes and methods. If you want to skim the Javadoc at this point of time, you will get some hints now.

The most common use cases involve the access and modification of content. For this, look at the package `com.coremedia.cap.content`, particularly at the interfaces `ContentRepository` and `Content`. Look at `Version` for getting an idea of how to handle the historic states of content. Afterwards, it will be interesting to inspect the content meta model, which is represented by the class `ContentType`.

In the case of content, the following service objects are provided: `PublicationService`, `AccessControl`, `PropertyService`, `QueryService`, `SearchService`, `ObservedPropertyService`, and `WorkflowContentService`. Have a look at the individual services and their getters in `ContentRepository`, so you can refer back to them when the functionality is required. You may want to investigate the `PublicationService` in more detail right away, because the approval of content and its transfer to the *Live Servers* are important in many contexts.

6. The Workflow Repository

The `WorkflowRepository` stores tasks and processes, and the definitions that describe their structure. Tasks and processes in the repository are executed by the server-side workflow engine, up to the point where user interaction is required. Users can select the tasks they wish to work on, modify the respective workflow variables and related content, and finally pass back control to the *Workflow Server*.

To support user interaction, the *Unified API* allows to

- create and start new processes,
- observe the current state of the computation (states and variables of tasks and processes),
- observe the progress of the computation (events),
- provide rights policies and performers policies for determining authorized users,
- determine where user interaction is required (work list management), and
- feed back a user's inputs (values, and commands like accept, complete) to the workflow engine.

To perform automated actions, the *Unified API* allows to

- define actions and expressions for execution in the Workflow Server.

For administrative purposes, the *Unified API* allows to

- monitor the running and the escalated processes and tasks,
- interrupt processes by suspending or aborting,
- upload and download process definitions, and
- inspect the structure of a process definition

In the following, these aspects will be described in some more detail. In Section 6.1, “Objects” [85] you will find a description of the basic types and objects stored in the workflow repository, and their relationships. Their lifecycle states are described in the next section, Section 6.2, “Workflow States” [88]. All objects relevant to a user in their current state, such as offered tasks, can be tracked using the work list service described in Section 6.4, “The Work List Service” [95]. Access to task-specific subsets of a workflow's variables is implemented using views, described in Section 6.5, “Workflow Variables and Views” [97]. Read Section 6.6, “The Access Control Service” [100] to learn how to determine the permissions granted to a user. The upload and download of process definitions is covered in Section 6.7, “Managing Process Definitions” [102]. All changes taking place in a workflow repository can be observed as events, as described in Section 6.8, “Events” [103].

Now, the plugin interfaces (also called service provider interfaces) are described, which you may implement.

Finally, a few examples are given in Section 6.11, “Examples” [120], and you will find pointers for further reading in Section 6.12, “Guide to the API Documentation” [131].

6.1 Objects

The workflow repository, like the content repository and the user repository, provides access to objects stored persistently on a CoreMedia Server, in this case, the *CoreMedia Workflow Server*. The objects to be accessed are modeled as subclasses of `CapObject`, and their structure is modeled by workflow-specific subclasses of `CapType`.

Like all persistent objects in the *CoreMedia CMS*, workflow objects carry an ID that uniquely identifies the object within the system, across users and sessions. This ID can be used to retrieve an object encountered before. The workflow repository offers a number of getter methods taking an ID argument, which differs only in the expected type of the retrieved object.

There are also methods to retrieve all objects of a certain kind (which, depending on the repository's size, can be quite expensive).

Many client applications will navigate the repository beginning from a process or task that is relevant for the current user. These tasks and processes can be determined using the work list service described in Section 6.4, “The Work List Service” [95]. If you want to navigate from a given content object to the processes that affect it, use the `WorkflowContentService` as described in Section 5.9, “Workflow Content Service” [79].

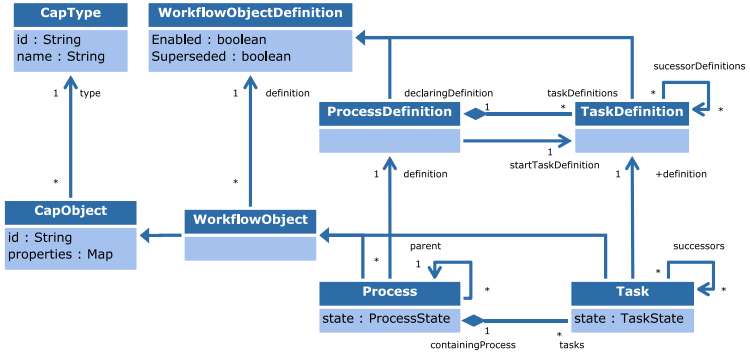

Figure 6.1. Workflow Class Diagram

The objects stored in the workflow repository can be discriminated into processes and tasks. Each process is composed of a number of tasks, which will be executed in a defined order. As can be seen in Figure 6.1, “Workflow Class Diagram” [85], the *Unified API* represents these objects using the classes `Process` and `Task`.

All names of association end in Figure 6.1, “Workflow Class Diagram” [85] correspond to getter methods in the *Unified API*. For example, the aggregation between Process

and Task can be navigated in both directions: The `Process` containing a `Task` can be determined using `Task#getContainingProcess()`, and the set of `Task`s contained in a `Process` can be determined using `Process#getTasks()`.

The set of tasks of a process conforms to the process' definition. For each task definition contained in the process' definition, the process contains a task, which is created at the time the process is created. The successor relation between tasks conforms to the structure of the task definitions, as expressed by the `successorDefinition` relation. The task structure is also fixed at process creation time. The first task to be executed when a process is started is determined by the process definition's `startTaskDefinition`.

Both processes and tasks pass through various states during their lifetime. These states are modeled using the enumeration types `ProcessState` and `TaskState`, respectively, and are described in further detail in Section 6.2, "Workflow States" [88].

In addition to the states predefined by CoreMedia, information about the progress of a process or task can be stored in workflow variables. Workflow variables can also be used to pass information from and to automated tasks, and may be seen or edited by users in task-specific forms.

Workflow variables are represented just like content properties and user attributes. As described in Section 7.1, "Objects" [133], each `CapObject` carries a property value for each property declared by its type. Both `ProcessDefinition` and `TaskDefinition` inherit from `CapType` the ability to declare properties. In the context of workflow objects, the `type` association between object and type is called `definition`, and differs only in its more explicit typing: When a `WorkflowObject` is asked for its type, the result can only be a `WorkflowObjectDefinition`; even more specifically, a `Process` is always defined by a `ProcessDefinition`, and a `Task` is always defined by a `TaskDefinition`.

The types of properties generally available are described in Section 4.5, "Types" [37]. The *Workflow Server* allows more kinds of properties than the *Content Server*, especially, lists can be declared of all element types (not just `Content` links), and additional atomic property types are available: `User`, `Group`, `Boolean`, `ContentType`, and `Timer`. Also, properties can be declared to be read-only.

When a `Timer` property is declared, this has two effects: Firstly, it creates a property that can hold a `TimeLimit`, and secondly, it creates a `Timer`, which, when enabled, notifies the application when the time limit is reached. See section Section 6.9, "Timers" [105] for details.

Being a `CapType`, each process definition has a name. The repository remembers the most recently uploaded process definition for each name, and automatically disables earlier definitions carrying the same name. A disabled process definition cannot be used to start new processes, but continues to be available for running processes. A process definition can be disabled explicitly even if it is not superseded by a newer version. The repository can be queried for a process definition by name, and again to support expres-

sion languages, a Map containing all enabled process definitions by name can be obtained.

A new process is created either by calling the method `ProcessDefinition#create` on an enabled process definition, or automatically by a server-side fork task (see the *Workflow Manual* for details on task types). A process that is started as a sub process of another process can be asked for its parent process.

The parent relation between processes must not be confused with the containment relation between processes and tasks; a process' tasks are created at the same time the process is created, but a sub process is only created when the corresponding fork task is executed (which, among other considerations, allows for recursion).

6.2 Workflow States

Since processes and tasks are dynamic, interacting entities, their lifecycle needs to be explained in some detail.

Process States

The state chart of a process is shown in Figure 6.2, “States of a process” [89]. After being created, a process is started, and may be suspended and resumed a number of times. Ultimately, the final task is completed and the process closes.

When a process is created, it does not immediately start running. Instead, the process remains "not started" until its `start` method is invoked. This way, the process' variables can be initialized at leisure. Note that as long as the process is not started, its initial view is active [see Section 6.5, “Workflow Variables and Views” [97]]. For example, this allows some variables to be writable only during initialization, and allows different validation rules to apply during setup and during the process' runtime.

When the `start` method is invoked, the process becomes "running" and starts with its first task. All relevant automated action and user interaction happens during the execution of tasks. The process itself mostly serves to structure and coordinate their execution.

While a process is running, it may be suspended at any time by invoking its `suspend` method. This stops all progress, be it in automated or user tasks. The process can be continued by invoking the `resume` method.

A process can terminate either normally, by reaching and completing its final task, or it may be aborted by an invocation of its `abort` method. Either way, after a short cleanup delay, the process and all its sub processes are destroyed, and all state and variable values are irretrievably lost. If some part of the process' state is still of interest, a process should first be suspended and inspected before aborting it.

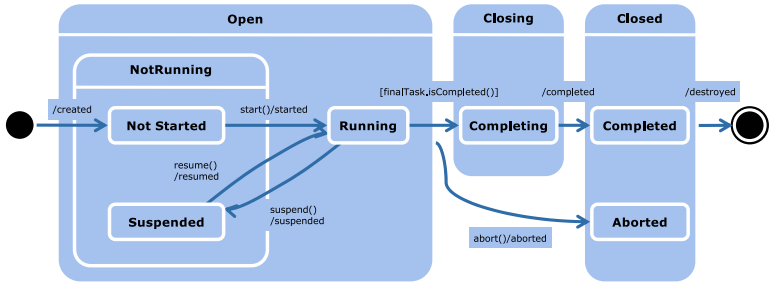

Figure 6.2. States of a process

automated tasks

Automated Tasks

Now you will learn the lifecycle of tasks.

For simplicity, begin with an automated task. In the normal case, the task's state progresses linearly from left to right, shown in the state diagram in Figure 6.3, "States of an automated task" [90]. The task is started by its process, or by the completion of its predecessor. It waits for its (optional) guard condition to become true. Then the automated actions are executed. When the automated actions are finished, the task becomes "completing". As soon as control is successfully transferred to the successor task, the task enters the "completed" state. The task structure of a process definition may contain loops, so a task that has been executed once may later be reached and start again. A task's lifecycle terminates when the containing process terminates.

Since the guard condition as well as the automated actions can contain customized code, error conditions must be modeled explicitly. When the evaluation of a condition or the execution of an action fails, or if a timer expires, the task is escalated, and will not automatically make any further progress. The previous state before escalation is recorded (denoted as history state (`H*`) in the state diagram) and can be inquired using `Task#getEscalatedState()`. If the failure was caused by external circumstances, it may make sense to retry the task after resolving the problem. When the `retry` method is invoked, the task goes back to the state before escalation and tries to execute the condition or actions once more.

As described above, a process may be suspended. This operation cascades to all tasks contained in the process, which will all be suspended. Each task's state before suspension is recorded (denoted by the lower history state in Figure 6.3, "States of an automated task" [90]), and can be inquired using `Task#getSuspendedState()`. The task can only continue when the complete process is resumed, which will move each task back to the state before it was suspended.

When a process is aborted, each of its tasks will be marked as aborted (making most methods unusable) and will be destroyed soon after.

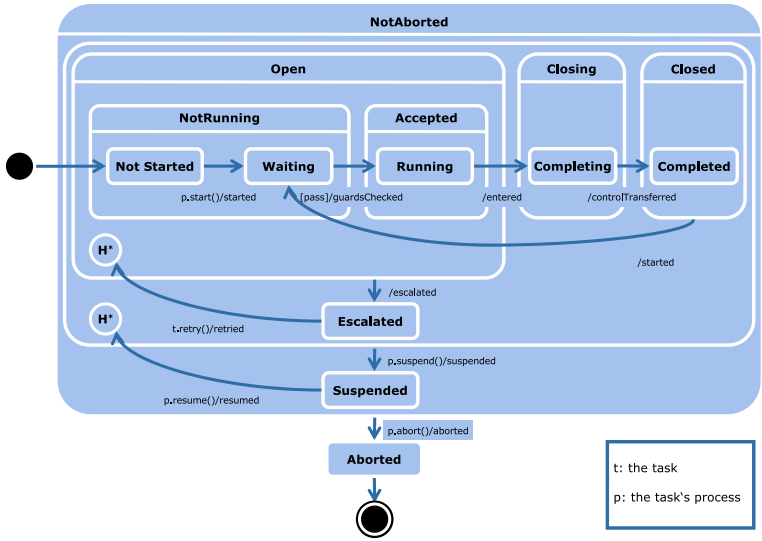

Figure 6.3. States of an automated task

User Tasks

While the execution of an automated task only consists of server-side actions, a user task's execution is split into several steps. As soon as the guard condition is true, a user task is activated, and waits for a user to accept the task. When a user accepts, on the server, the task's preconditions are checked, and the task's entry actions are executed. When the entry actions are finished, the task becomes running, and responsibility for further actions passes to the user. When the user has completed his or her part, the server checks the task's postconditions and runs the task's exit actions.

Figure 6.4, “States of a Task” [91] is a combined state chart for automated and user tasks. Look out for `[isUserTask()]` conditions which annotate the differences between the task types.

There are several transitions where customized server-side code is executed. In each of these cases, when something goes wrong, the task becomes escalated. Another potential cause for escalation is a timer expiring, for example because the user does not complete a task in the expected period. The mechanism for retry, suspend/resume and abort is the same as described for automated tasks above.

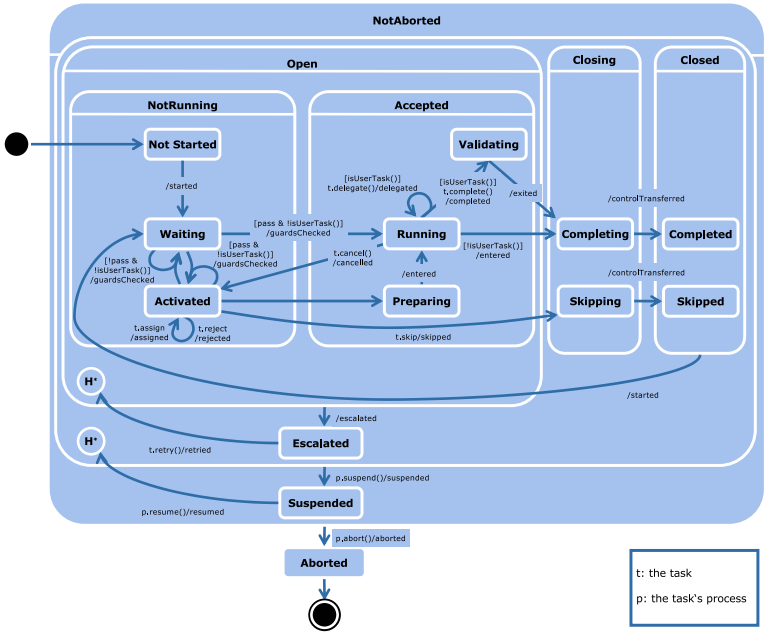

Figure 6.4. States of a Task

When a task is activated, that is its guard check has been passed, it may be offered to several users. By default, all users that have the right to accept the task (see Section 6.6, "The Access Control Service" [100]), and have not rejected the task yet, appear in the set of offered users. A task may also be assigned directly to a user or to a group, or a certain performer may have been forced by a previous task. The strategy for offering tasks to users can be overridden by providing a customized performer policy (see the *Workflow Manual* for details), or by changing the handling of the accept right in a custom rights policy (see Section 6.10.7, "Rights Policies" [115]).

The set of users a task is offered to may be inquired using the method `Task#getOfferedTo()`. All tasks that are offered to the current user can be determined using the work list service (see Section 6.4, "The Work List Service" [95]). Changes to a task's set of offered users are signaled by `TaskOfferedEvent` and `TaskRevokedEvent` instances (see Section 6.8, "Events" [103]). There are no events for changes to the work list. Instead, when working inside the *CoreMedia CAE* caching infrastructure, your code simply calls the work list getters, and can rely on the correct dependencies being registered behind the scenes. In this way, your code will be automatically reexecuted when any accessed work list changes. See the *CoreMedia CAE* Developer Manual and Section 4.10, "Caching" [49] for further details.

A task's guard condition may become false before the task is accepted by any eligible user. In this case, the task goes back to the waiting state.

The user who accepts a task becomes the *performer* of the task. This entails certain privileges required to perform the task, namely the ability to read and write the task's variables, and the ability to cancel, complete or retry the task.

Before passing control to the user, first, the task's preconditions are checked. This feature can be used to verify assumptions by the workflow designer. If a condition is not met, the task is escalated. If all checks are passed, the task's entry actions are executed. This may include GUI-based remote client actions, which will be executed in the name of the user [see Section 6.10.8, "Remote Client Actions" [117]].

The Unified API offers the method `Task#acceptAndEnter()`, which waits until the task has safely arrived in the running state. Any exceptions thrown by failing preconditions or entry actions are passed on to the method's caller. This allows for a synchronous programming model: When `acceptAndEnter` returns normally, you can be sure that the task is running. In contrast, `accept` supports an asynchronous programming model, insofar as it only triggers the server-side computation. When `accept` returns, the server-side code may not have finished yet.

A task can be passed directly from one performer to another using the method `Task#delegate()`. The task remains in the running state, no conditions are checked or actions executed.

A task may also be canceled, sending it back to the activated state. The user ceases to be the task's performer. Again, postconditions are not checked, and exit actions are not executed.

Note that these methods may also be invoked by a different user than the performer, assuming the respective rights are granted. For example, when a user is on vacation and has left behind some running task, an administrator or process owner may still lead the process to conclusion by delegating or canceling the task. An additional option for a user task is to skip the task, in order to make progress even when no suitable performer can be found.

A call to `Task#complete()` indicates to the workflow server that the user has finished his or her work. All configured postconditions are checked. If any post condition fails, the user probably has not fulfilled his task as planned. The task becomes escalated, and may be retried by the performer, returning it to the running state. Note that the current performer is remembered while the task is escalated and/or suspended.

After all postconditions are successfully checked, the configured exit actions are run, and the task changes to state completing. Similar to `acceptAndEnter`, the method `Task#completeAndExit()` synchronously waits until the task including all post conditions and server-side actions has completed, and passes any exceptions on to its caller.

The remaining lifecycle is as described for automated tasks, above.

6.3 Differences to the Classic Workflow API

There are currently two APIs for accessing workflow objects: the classic *Workflow API* (or *WfAPI* for short) and the *Unified API*. The classic API is intended solely for use in the *Site Manager* today. While it is still supported for legacy stand-alone clients and server-side extensions, CoreMedia recommends that such code be ported to the *Unified API*.

The *Unified API* covers all workflow-related functionality required for developing client-side applications. In comparison to the *WfAPI*, it is integrated much better with the content repository, and provides a simpler model for accessing workflow variable values.

When migrating a *WfAPI* client to the *Unified API*, note that what is a process in the *Unified API* used to be called a "process instance" in the *WfAPI*, while a process definition in the *Unified API* used to be called a "process" in the *WfAPI*, and similarly for tasks.

The state hierarchy has been reshuffled slightly (compare the state charts in the *Workflow Manual* and in this manual). Note, that some events have been renamed, shown in Table 6.1, “WfAPI signal names and UAPI event classes” [94]. There are no per-object listeners in the *Unified API*, only the `WorkflowRepositoryListener`. The *WfAPI*'s directory service functionality is covered completely by the *Unified API*'s user repository.

WfAPI name	UAPI name
CHECK	`TaskGuardsCheckedEvent`
ACCEPT	`TaskAcceptedEvent`
RUN	`TaskEnteredEvent`
VALIDATE	`TaskCompletedEvent`
FINISH	`TaskExitedEvent`
RESET	`TaskDeactivatedEvent`
TIMEOUT	`TaskTimerExpiredEvent`
COMPLETE	`TaskControlTransferredEvent`

Table 6.1. WfAPI signal names and UAPI event classes

6.4 The Work List Service

The work list service is probably the most useful part of the workflow API, as it tells a user what work there is to do for her.

A user interacts with the *Workflow Server* in several ways:

- selecting tasks to accept
- working on, and eventually completing accepted tasks
- resolving problems, represented by escalated tasks
- starting new processes
- monitoring the started processes

The work list service is implemented as a separate interface, and can be accessed using `WorkflowRepository#getWorklistService()`. All methods in the interface perform their computation for the current user. See Section 4.9, “Sessions” [46] for information on how to switch between different user sessions.

The first request by a certain user needs some time to initialize and retrieve the required information from the server. Subsequent requests are much faster, because the work lists are cached, and updated incrementally. The work lists are kept in memory until the user logs out, so especially when dealing with work lists, be sure to log out each user you have logged in.

All methods of the work list service are cache-aware. This means that when the work list service is accessed from within the *CoreMedia CAE*, the calling method's result will only be recomputed if the contents of the accessed work list actually changed. See the *Content Application Engine* section in the *Delivery Developer Manual* for further details.

The names of user-aware methods follow the pattern "get<Objects>< Predicate>", which should be read as "return all <Objects> that fulfill the condition <Predicate> for the current user".

The specific work lists available are:

- tasks offered

  Contains all tasks that the current use can accept. In order to decide which task to accept, a user might want to inspect the task's variables.

- tasks accepted

  Contains all tasks that the user has accepted, and is currently performing. These are the tasks whose variables a user might want to inspect and modify. The user may finish working on this task by delegating, canceling or completing it, or by aborting the whole process. This list includes suspended tasks.

- tasks escalated

  Contains all tasks that are escalated, that the user might want to get running again (retry), possibly after fixing the work environment. These include the tasks that the user has performed (or was about to perform), and also all tasks in processes owned by the user.

- process definitions of which new processes may be created

  Contains all types of processes that the user may want to instantiate. For example, this list is suitable for the selection in a "create new workflow" GUI action.

- processes not started

  Contains all freshly created processes owned by the current user. For these processes, the user will want to fill out the initial view, before starting (or aborting) the process.

- processes running

  Contains all running processes owned by the current user. This list also includes suspended processes. A user might want to inspect the process' running view in order to observe the current state of the process global variables. Processes in this list may be suspended and resumed (or aborted) by the current user.

- tasks with warning

  Contains all tasks that have a warning, and whose process is owned by the current user. This aids the user in tracking the progress of his/her processes.

There are some methods that may only be called by an administrator. The names of these methods look like `getAll<Objects><Predicate>`. This should be read "return all <Objects> that fulfill the condition <Predicate> for any user".

The administrative methods serve to give an overview of everything that is going on in the system. However, if the system is busy, the resulting lists can be quite large, so care must be taken to access them sensibly.

6.5 Workflow Variables and Views

Workflow variables may be defined directly in the process, as well as locally in any of the tasks. While performing a task, a user needs to inspect, and possibly modify, some of those variables. Which variables the user needs, depends on the concrete task.

Therefore, each task definition contains a view definition, which specifies the variables to be accessed, and the kind of access required (read or write), while performing the task. The view definitions of different task definitions may reference the same variables, for example to share a common process description, or a common list of content objects to operate on.

The *Unified API* represents a view as a special kind of `CapObject`. A view appears to have properties that can be read and written, while in fact, each of those properties is stored in some task or process. Each read and write access to a view is redirected according to the view definition. By inspecting the view definition, which is a subclass of `CapType`, the available properties can be listed.

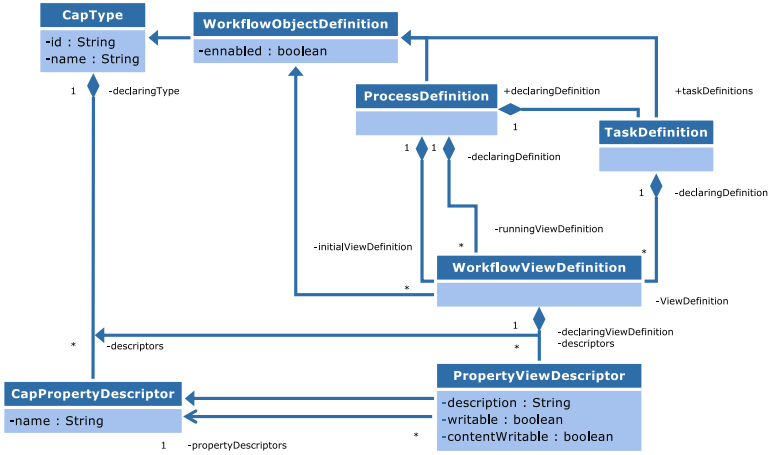

Figure 6.5. Workflow Object and View Definitions

The *Unified API* representation for view definitions is shown in Figure 6.5, "Workflow Object and View Definitions" [97]. The diagram shows all subclasses of `CapType` managed in a workflow repository. Their instances form a hierarchy, which corresponds to the nesting in the workflow definition XML file (see the *Workflow Manual* for details). A process definition contains a number of task definitions. Views can be defined by both process definitions and task definitions. There is one view definition per task definition, and two

per process definition, where the initial view applies before the process starts running, and the running view applies after the process has started.

At the bottom of the diagram, you can see the property descriptors. Each `CapType` aggregates a number of property descriptors. In the case of workflow and task definitions, these are the workflow variables. So for each process or task, a value is stored for each descriptor of its definition.

A view definition also is a `CapType`, its property descriptors are `PropertyViewDescriptors`. In addition to being regular property descriptors, they provide information on how to represent the view during user interactions, and information on where the actual value is stored. A view as an instance of a view definition does not itself store any values. Instead, the name of the property view descriptor determines where the actual value is stored, relative to the view's origin. The origin of a view is the process or task the view was obtained from.

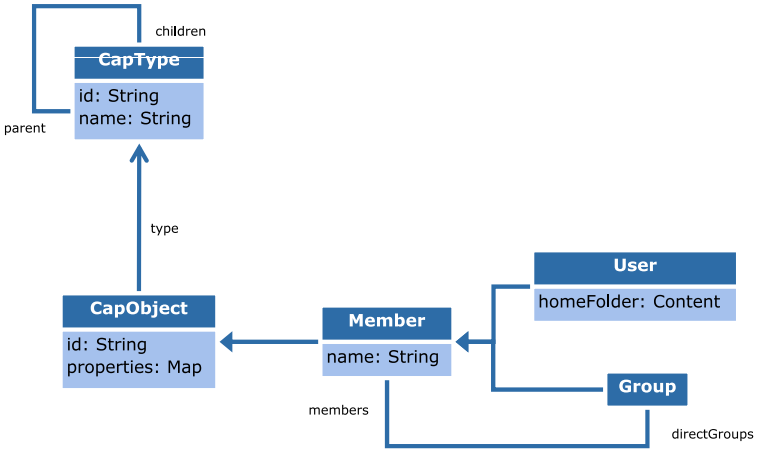

Figure 6.6. Workflow views

The concrete mechanism for variable resolution should not bother a Unified API client, because the complete mapping from view property to workflow variable is managed transparently by the API. A view can be treated just like a `CapObject` that just has slightly unusual property names. For completeness, the mapping from view property to target object and target property works as follows:

A qualified property view descriptor name, of the form *<task>.<property>*, is resolved to the property with the given name of the task with the given name in the same process as the view's origin.

An unqualified property view descriptor name, of the form <*property*>, refers to the named property of the view's origin. If the origin is a task and does not contain such a property, the property view descriptor refers instead to the named property of the task's containing process.

Two additional flags are specified per property view descriptor. Firstly, a property is declared as read-only or writable. This overrides the value of the writable flag in the target property declaration. For example, a workflow variable declared as read-only in the process may be writable from the process' initial view.

Secondly, for a property containing content links, the `contentWritable` flag determines how the referenced content should be presented to the user. If the flag is false, the content should be displayed as read-only. For example, assume one task in a process that has the goal of selecting a set of documents for later tasks to work on. This task will only need to add and remove links, not change the documents themselves. Therefore, this task view should contain a property mapping that is writable, but not content writable.

Workflow views also affect the interpretation of READ and WRITE rights. See Section 6.6, “The Access Control Service” [100] for further details.

6.6 The Access Control Service

The `AccessControl` service of the `WorkflowRepository` allows you to determine whether a user has the right to execute some operation on some workflow object. There is one right for each operation defined in the Task and Process interfaces, plus a read and a write right, plus the right to create instances of a process definition.

A right can be queried using the generic method `AccessControl#mayPerform`, which expects as arguments the object to which the operation would apply, and the operation, represented as a `Right` object. The user for whom the right should be checked can optionally be passed as a third parameter, and defaults to the connection's current user.

Note that in contrast to the `ContentRepository`'s `AccessControl`, in the workflow access control, there is no method signature expecting a set of groups. Whereas content rights can only be granted at group granularity, workflow rights can be granted per user. Therefore, group memberships are not sufficient to compute workflow rights.

For each right, there is a method `mayOperation`, where *operation* is the name of the right, which serves as a shortcut for `mayPerform` with the respective right as argument. So for example, the following two statements are equivalent:

```
ac = wr.getAccessControl();
allowed = ac.mayPerform(task, Right.TASK_ACCEPT, user);
allowed = ac.mayAccept(task, user);
```

The Javadoc of each operation specifies the required rights. Note that even when a user has the right to execute an operation, he may still be unable to do so. For example, some operations are only applicable in certain object states. Such "physical" requirements are expressed as preconditions, whereas "legal" requirements are expressed as rights.

There are two rights whose names do not directly correspond to operations, namely `Right.READ` and `Right.WRITE`. These rights govern access to the properties of a `WorkflowObject`. The `WorkflowObject` in question can be a `Process`, a `Task`, or a `WorkflowView`. When the `READ` right is given, all declared properties can be read, for example using `WorkflowObject#get(String)`. When the `WRITE` right is given, all properties that are not read-only can be assigned to, for instance using `WorkflowObject#set(String,Object)`.

As explained in Section 6.5, “Workflow Variables and Views” [97], the properties of a view may actually be stored in various tasks or in the process. However, for rights computation, the rights defined on the view's origin are considered, not the rights on the workflow object that stores the variable. In this way, a view can be used to grant access to a controlled subset of the process variables. This is especially interesting because a

task's current performer is granted READ and WRITE rights on the task, which applies to all properties in the task's view.

The actual rights computation is performed by a rights policy, which is described in section Section 6.10.7, “Rights Policies” [115].

6.7 Managing Process Definitions

The *Unified API* offers some administrative functionality for dealing with process definitions. As described in the *Workflow Manual*, a process definition consists of some XML, which may contain references to custom Java classes. The byte code of custom Java classes has to be supplied either in the workflow server's class path, or in an accompanying JAR file.

Code that is deployed in the workflow server's class path is shared between all versions of all workflows. It can only be changed by shutting down and restarting the workflow server, which disrupts service and causes significant delays for reinitializing users' work lists etc. In contrast, custom code that is uploaded together with the process definition's XML is used only by this version of this process definition. Such code runs in a separate class loader, and therefore will not share classes and static fields with other process definitions.

Currently, custom code on the workflow server only has access to the content and user parts of the *Unified API*. For accessing workflow objects, server-side code uses the traditional *Workflow API*. Find more details on the server-side *Workflow API* and the interfaces for server-side code in the *Workflow Manual* and in the Javadoc.

The method `WorkflowRepository#createProcessDefinition` is used to upload an XML process definition together with an optional JAR file to the *Workflow Server*. Later, the data originally uploaded can be retrieved using `ProcessDefinition#getProcessDefinition` and `ProcessDefinition#getProcessClasses`.

6.8 Events

Similar to the content repository, the workflow repository allows you to add listeners that get informed of every state transition, property change, task offer or warning, expired timer, and process definition update.

A workflow repository listener can be added by invoking the method `WorkflowRepository#addWorkflowRepositoryListener`. To assist in implementing custom listeners, the *Unified API* offers the class `WorkflowRepositoryListenerBase`, which can be used to handle selected types of events (by overriding the respective methods), or to handle all workflow events uniformly (by overriding the method `#handleWorkflowRepositoryEvent`)

When a listener is invoked, the same session is active that was active when the listener was registered. See Section 4.7, "Listeners" [43], and Section 4.9, "Sessions" [46] for details.

Events are delivered serially to all registered listeners. That is, the next event will only be delivered after the current event was processed by all registered listeners. If you need to do any time-consuming computations, you should transfer the relevant events to a separate queue.

At the time the listener is invoked, the effect of the event is guaranteed to have reached the *Unified API* objects belonging to the same connection. So for example, in the method handling a `TaskEnteredEvent`, you can assume that `isRunning` returns true for the affected user task if the user has not completed the task yet. Note that the event may be significantly delayed, so the state may well have been changed in the meantime.

In contrast to the content repository, a workflow repository listener may lose events when the server connection is lost. All listeners will resume their work, but an application generally has to assume that the workflow server state has changed completely while the connection was lost. So in addition to workflow repository events, you may also want to observe connection events (see Section 4.1.3, "Connection Listener" [26]).

When mixing reaction to events with accesses to the current repository state, it is often easier to use the cache and invalidation based programming model offered by the *CoreMedia CAE* (see the *Delivery Developer Manual*). The caching framework also handles connection failures: When the connection is reestablished, all cached values that depend on the workflow repository are automatically invalidated (or a recomputation is triggered).

The events related to process and task states can be gleaned from the state charts shown in Section 6.2, "Workflow States" [88]. In the task state chart, an event name *accepted* on the transition label corresponds to an instance of `TaskAcceptedEvent` dispatched to the workflow repository listener. Similarly, a *started* label in the process state diagram corresponds to a `ProcessStartedEvent`.

When a task is newly offered to some users, or when the offer to some users has been revoked, a `TaskOfferedEvent` or `TaskRevokedEvent` is signaled, respectively. See the subsection on user tasks in Section 6.2, "Workflow States" [88] for more information, or see the *Workflow Manual*.

A `ProcessCreatedEvent` is signaled when a new process is created. The event indicates the performer - the user who invoked the `ProcessDefinition#create` method - as well as the new process' owner.

The remaining kinds of events are related to process definitions. A `ProcessDefinitionCreatedEvent` is sent when a process definition is uploaded, for instance using `WorkflowRepository#createProcessDefinition`. When a process definition is uploaded as described in Section 6.7, "Managing Process Definitions" [102], it supersedes any previous process definition with the same name. In addition to the identity of the created process definition, the event therefore carries information about the process definition's name, and the identity of the superseded process definition. The superseded process definition is implicitly disabled. A process definition can also be disabled or enabled explicitly by calling `ProcessDefinition#enable` or `#disable`, which causes a `ProcessDefinitionEnabled` event to be sent.

6.9 Timers

The workflow server supports timers, which implement a reaction when some part of the workflow takes too long. There are system defined timers, which observe the time it takes to complete a process or a task, and the time it takes until an offered task is accepted; and there are user defined timers, which are enabled and disabled according to the process definition.

At any time, a timer can be in one of three states: *off*, *running*, or *suspended*. A timer is initially off, and starts running when it is enabled. A running timer is suspended exactly when the containing process is suspended, and becomes running again when the process is resumed. When a running timer is disabled, it is turned *off* again.

As long as a process is suspended, its timers cannot be enabled or disabled. Also, suspending a timer that is off has no effect on the timer's state.

In addition to its lifecycle state, a timer holds a time limit and an expiration flag.

A time limit can be given either in relative or in absolute form. The relative form indicates a time distance, represented as a number of seconds, and is used for system-defined timers, and (usually) when initializing a timer inside a process definition. The absolute form indicates a fixed point in time, represented as date and time, and makes most sense when a time limit is set interactively by a user ("This article is needed by next Friday").

When a timer first becomes running after setting a time limit, the timer's expiration date is computed, either adding the relative time limit to the current time, or directly using the absolute time limit. The expiration date remains fixed even if the process is suspended and resumed again.

When a running timer reaches or exceeds its expiration date, it expires. This has the following effects:

- The expiration flag is set.
- A `TimerExpiredEvent` is sent.
- All server-side timer handlers registered for this timer are invoked. Timer handlers are defined in the process definition.

As long as the timer's expiration flag is set, no further `TimerExpiredEvent`s or handler invocations take place. The expiration flag is cleared each time the time limit is modified.

If a timer's expiration date is reached while the timer is suspended, the timer will expire as soon as the process is resumed.

In the following, the different system-defined timers and the handling of user-defined timers are described.

Process Completion Timer

The process completion timer observes the total execution time of a process. Its relative time limit can be set in the process definition using the `defaultTimeout` attribute, as described in the *Workflow Manual*. The process completion timer is enabled when the process is started, and is cleared when the process completes or is aborted.

By default, the process completion timer adds a warning to the process when the timeout expires. Different handlers can be defined in the process definition.

Task Completion Timer

The task completion timer is defined for user tasks only, and measures the total time the task is waiting for users: from when the task is first offered, to when the task is completed by a user. The relative time limit is set using the task definition's `defaultTimeout` attribute.

If the guard of an activated task becomes false, the task goes back to the `WAITING` state, and the timer is disabled. However, the timer's expiration date is not changed.

The timer is cleared when the task is completed, interrupted, or skipped, or when an alternative task in an implicit choice is accepted.

By default, the task completion timer adds a warning to the task when the timeout expires, causing the task to appear on the `tasks-with-warning work` list. Different handlers can be defined in the process definition.

Task Acceptance Timer

The task acceptance timer is also specific to user tasks, and measures the time from when the task was first offered to when it is accepted. It is configured using the `defaultOfferTimeout` attribute in the task definition.

The task acceptance timer is enabled exactly when the containing task is in the `ACTIVATED` state. The timer is enabled when the task first changes from `WAITING` to `ACTIVATED`. It is disabled when the task is accepted, or becomes `WAITING` again because the guard condition becomes false. When the task becomes `ACTIVATED` again (because the guard condition becomes true, or because a previously accepted task is canceled), the timer is enabled again, and continues running with unchanged expiration date.

Like the task completion timer, the timer is eventually cleared when the task is completed, interrupted, or skipped, or when an alternative task in an implicit choice is accepted.

By default, the task acceptance timer adds a warning to the task when the timeout expires, causing the task to appear on the `tasks-with-warning` work list. Different handlers can be defined in the process definition.

User-defined Timer

Additional timers can be defined in the process definition, by defining a timer variable. These timers are not enabled or disabled automatically, but need to be handled explicitly using `EnableTimer` and `DisableTimer` actions.

A timer variable consists of two parts: The time limit is a value like all other property values that can be freely read and written to the containing workflow object property. The timer object itself observes the value of this variable, and can be accessed using `WorkflowObject#getTimer(name)` [or `#getTimers` or `#getTimersByName`].

```
t.get("MyTimer"); // property access
t.getTimeLimit("MyTimer"); // typed property access
t.getTimer("MyTimer").getLimit(); // this works, too
```

Absolute and relative time limits are implemented using the value classes `AbsoluteTimeLimit` and `RelativeTimeLimit`, which implement the `TimeLimit` interface, and may be freely constructed by an application programmer:

```
t.set("MyTimer", new RelativeTimeLimit(300));
Calendar abs = DateConverter.convertToCalendar(
                              "2004-09-15T21:59:00+01:00");
t.set("MyTimer", new AbsoluteTimeLimit(abs));
```

6.10 Writing Own Plugins

In this section you will see how *Workflow Server* plugins are written using the *Unified API*. There are seven possible types of plugins: actions, long actions, expressions, rights policies, performer policies, client action handlers, and managers. While these plugins can still be written using the classic *Workflow API* introduced with *CoreMedia CAP 4.0*, it is generally simpler to use the *Unified API*.

Of these plugins, the client action handlers live purely in *Unified API* clients, rights policies are needed both in the clients and on the *Workflow Server*, and the remaining three interfaces are instantiated on the server, only.

When implementing plugins, the comments from the *Workflow Developer Manual* generally carry over directly to the Unified API. In particular, restrictions on the permitted operations for the various plugin types are also applicable when using the *Unified API*.

In the following, you will learn about actions in Section 6.10.3, “Actions” [110], long actions in Section 6.10.4, “Long Actions” [111], expressions in Section 6.10.5, “Expressions” [112], performers policies in Section 6.10.6, “Performer Policies” [114], rights policies in Section 6.10.7, “Rights Policies” [115], client action handlers in Section 6.10.8, “Remote Client Actions” [117] and managers in Section 6.10.9, “Managers” [118].

6.10.1 Programming Restrictions

All workflow plugins can be executed in the *Workflow Server*. While rights policies and remote action handlers are also executed on the client-side, they still should be coded according to the server-side rules in order to be executable everywhere. In the following, the restrictions are listed that apply when programming code for the *Workflow Server*.

Limitations of the API

The main restriction arises from the fact that the server calls workflow plugins in the context of a transaction. In the *Workflow Server*, one transaction may write the variables of at most one process and its tasks. Accessing multiple processes, even if they are instances of the same definition, is not allowed. All server-side plugins are passed a workflow object in the signature of their main business methods. It is this workflow object that should be read or possibly modified by the plugin.

There are also some parts of the API that are not supported. Normally, these parts are not needed for writing plugins.

- No rights checks are performed when writing workflow variables. This ensures compatibility with the old *WfAPI*. The `AccessControl` service is still available without restrictions.
- When opening lightweight sessions by means of `Connection.login(...)`, these sessions will have no influence on the objects of the workflow repository. This is because the plugin is already running inside a transaction whose owner cannot be changed later on.
- No state modifying operations like `accept()` or `suspend()` are permitted. This is because server-side plugins are typically executed exactly *during* such state transitions and state transitions cannot be nested.
- Worklists are unavailable.
- Workflow repository events are not currently delivered inside the workflow server. Already adding a listener results in an exception. Only events regarding content and users are delivered. However, even such listeners should not normally be used, because plugins are supposed to terminate quickly without waiting for external conditions.
- As a consequence of missing events, your own cache entries should only access objects of the content and user repositories. When accessing workflow objects, no invalidations will be generated, resulting in outdated cache entries later on.

General Remarks

A plugin should not engage in user interactions. It may still connect to external processes, for example when sending a mail message or when accessing an external database, but it should not freeze when a user does not respond.

A plugin should be able to complete without requiring progress other parts of the workflow in order to avoid potential deadlocks.

In order to resolve concurrent accesses to shared data, the server may restart a transaction. This may also happen during a system failure, but that is far less likely. In any case, a restart amounts to a repeated execution of your plugin. Therefore, your plugins should be robust to handle such a situation. Usually expressions, right policies, and performers policies do not result in side effects, so that it is irrelevant whether they are executed once or twice, but action are a more difficult matter.

Finally, keep in mind that your plugin will run in a server with an expected uptime of weeks or months. Therefore, any memory leak should be avoided. Preferably, your plugins do not use mutable fields except those that are used for configuration and they do not use mutable static fields at all. When you create own threads, make sure that they are guaranteed to terminate. When you use system resources like sockets or file handles, make sure they are released sooner rather than later.

6.10.2 Serialization

The plugin interfaces `Action`, `Expression`, `PerformersPolicy`, and `RightsPolicy` inherit from the interface `java.io.Serializable`. That means that you must take care to make your implementations serializable, in particular by marking all non-serializable fields as transient. Remember that *Unified API* objects are serializable, so that it is alright to reference such objects from your plugin, for example when configuring folders or groups.

If you want to add special code for restoring transient fields after read, you can do so in a `readObject` method.

It is advisable to define a serial version UID for your class to be able to indicate the compatibility of serialized versions appropriately. Note that you may make changes that break serialization compatibility, but that you must invoke the tool `cm workflow converter` while the server is down after such changes.

6.10.3 Actions

Actions are executed during the entry and exit phases of a user task, during the execution of an automated task, or during the processing of a `RunActionTimerHandler`. This means that an action is typically executed in the context of a task, but that it may be executed in the context of a process, too, if used with a timer handler.

By means of the interface `Action`, you can only implement server-side actions, that is, actions that run completely within the *Workflow Server*. Actions are run on the server on behalf of the workflow user as configured in the Workflow Server properties.

The main method of an action is `execute(WorkflowObject)`, where the argument is either a task or a process depending on the context of the action. While executing, the action implementation should only read and write variables of the argument workflow object and its view. It is recommended that the exact variable names are made configurable by means of bean-style getters and setters.

The method `isExecutable(WorkflowObject)`, should return false, when it is not currently possible to execute an action. Normally, you should always return `true` from this method, but there are cases where you might want to wait for a workflow variable to be set correctly before processing an action.

After execution, you may return a new instance of `ActionResult` in order to indicate success or failure. If you use the attributes *`successVariable`* and/or *`resultVariable`* in your XML workflow definition, the action result is automatically evaluated to set those variables. The action result can also take exceptions that are

interpreted as warnings. If you include warnings in your result, they are added to the list that is returned from the method `getWarnings()` of the affected workflow object.

The method `abort()` should be implemented to let all running `execute` calls return early, possibly by throwing an `ActionAbortedException`. This method is called when the Workflow Server is shutting down. There is no need to implement special logic if the execute method always returns early. If execution takes some time, you should also consider implementing a long action instead.

The name returned by the method `getName()` of an action is used solely for logging and for parameterizing exceptions. It does not carry any semantic meaning, so that you may choose it as you like.

In order to simplify the development of an action, you may derive your class from the predefined classes `AbstractAction` or `SimpleAction`. Thereby, it is enough to implement a single method, namely `execute(Process)` in the former and `doExecute(Process)` in the latter case. Because the exact task in which the action is executed is not included in the signature of these methods, this approach requires that all relevant variables are defined at the process level. This is the typical use case. A detailed example of an action sending mail implemented as `SimpleAction` is given in Section 6.11.3, “Example Code of the Mail Action” [127].

The server may run an action more than once, in particular when a transaction has to be restarted due to concurrent activity. Therefore, you should design your actions in such a way that either the second execution detects that the action has already been executed or that a repeated execution is acceptable. For example, it is preferable to set a variable to a certain value rather than to increment an integer or to toggle a flag

6.10.4 Long Actions

Long actions are very similar to actions, but they are executed in three separate phases. Only the first and the last phase are permitted to access the containing process and its variables. The second phase runs completely outside of any database transaction. Therefore, the second phase does not consume system resources and there is no need to finish it quickly. Long actions are particularly well suited for accessing remote servers that may not respond immediately.

The first phase consists of the method

```
Object extractParameters(Task task);
```

which must read all task and process variables that are needed for processing. Afterwards, all relevant data must be packaged into an object of arbitrary type, which is returned from the method. If multiple values have to be returned, either an object array or a custom class can be used for aggregating these values. Because a long action al-

ways runs in the context of an automated task, the method is passed a correctly typed task object immediately. Often you will have to retrieve the containing process before reading any variables. Afterwards the method

```
Object execute(Object params);
```

is executed. It is passed the object that was returned from `extractParameters`. It may perform arbitrary computations for an extended period before it returns its result as an object. The method may not, however, access any objects of the workflow repository. Finally

```
ActionResult storeResult(Task task, Object result);
```

is called with the result from `execute`. It may write task and process variables as needed. The returned action result is processed as by an ordinary action.

The class `LongActionBase` implements the `LongAction` interface and provides some convenience code. Instead of `execute` and `storeResult` you simply implement the method

```
Object doExecute(Object params) throws Exception;
```

If that method throws an exception, that exception forms the basis of a failed action result. If a value is returned, that value is wrapped in a successful action result. Note that you must implement the `extractParameters` method even if you base you action on the `LongActionBase` class.

The method `abort()` should be implemented to let all running `extractParameters`, `execute` and `storeResult` method calls return early, possibly by throwing an `ActionAbortedException`. This method is called when the Workflow Server is shutting down.

Like an ordinary action, a long action must be reentrant and it must be robust against being rerun in the case of a problem.

6.10.5 Expressions

Expressions are executed when guards are evaluated, when they are nested in actions, and for other configurable computations. Like actions, expressions are typically executed in the context of a task, but may occasionally be executed in the context of a process.

The main method of an expression is `evaluate(WorkflowObject,Map<String,Object>)`, which receives as arguments the workflow object in which the expression is evaluated and a number of local properties. These properties must not be confused with the variables present in the workflow object or its view. In-

stead, the properties are purely local to the expression, without any form of persistence. They are typically set in the predefined expressions `Let`, `Exists`, or `ForAll`.

If an expression modifies the given map, it should make sure to return it to its previous state before returning from the `evaluate` method. Preferably, this is done in a `try/finally` construction.

Please see the Javadoc for more details regarding the data types that are permitted as return values of the expression and for the parameter map.

If you are creating an expression that will only return Boolean values, you can implement the interface `BooleanExpression`. Thereby you indicate the reduced set of return values and make your expression usable in a greater number of contexts, in particular in guards and as a subexpression of predefined Boolean connectives.

Mixing Unified API and WfAPI Expressions

You can include *Unified API* expressions and *WfAPI* expressions in one process definition. This allows a stepwise migration of existing plugins to the new APIs. You may even use, for example, old-style expressions as subexpressions of *Unified API* expressions. In this case, the expressions are automatically wrapped so that they appear as objects of the API that is used by the containing object. The wrappers will take care of converting argument values and return values when calling methods of the wrapped expression.

When using *Unified API* expression inside *WfAPI* actions, care has to be taken with respect to the correct treatment of null values. Because the *WfAPI* uses typed nulls and the *Unified API* expressions may return an ordinary Java `null`, a special subtype of `WfValue` has been introduced to the *WfAPI*: `NullValue`.

This value should not be used when working purely inside the *WfAPI*. Only when a *Unified API* expression returns a `null` and when that value must be propagated to a *WfAPI* action or expression, the above mentioned wrapper objects convert the value to a `NullValue`. Of course a plugin that was written for *CoreMedia CMS 2005* or earlier may not expect a value of that type, possibly failing during a type cast. Therefore, existing plugin implementations may have to be hardened against the new value type `NullValue`, before you can use them with *Unified API* subexpressions.

Note that it is often desired to port the entire set of plugins to the new API anyway, so that this paragraph applies to a few specific cases, only. It is possible to use the built-in actions and expressions without restrictions. They will neither produce untyped nulls nor misbehave when they come across a `NullValue`.

6.10.6 Performer Policies

Performer policies are used when determining the users who are offered a certain task on their to-do lists. This set of users is then stored persistently with the task in order to reduce server startup times. When users reject the task from their to-do lists, the performer policy will be invoked again to update the lists.

The interface to implement is `PerformersPolicy`. When implementing a policy, it is advisable to start with the class `AbstractPerformersPolicy`, which takes care of managing the policy state, namely:

- the forced user, who is set by means of the predefined action `ForceUser`,
- the excluded users, who are set by means of the predefined action `ExcludePerformer` or `ExcludeUser`.
- the preferred users, who are set by means of the methods `assignTo(User)` and `assignTo(Group)` in the interface `Task`,
- the rejected users, who are set by means of the method `reject()` in the interface `Task`.

The policy state is maintained in four variables, which are defined in the method `addInternalProperties(PropertyBuilder)` in the interface `PerformersPolicy`. Such variables do not need to be declared in the XML definition file. The interface `PropertyBuilder` provides one method per variable type. In own implementations, you may create as many variables as needed.

This simplifies the encapsulation of the internal working of custom performers policies. Note that these variables reside in the same name space as variables defined in the XML process definition, so that you should choose names that are unlikely to occur as ordinary workflow variable names.

When using the class `AbstractPerformersPolicy`, you will have to implement only four methods and of those, the methods `getName()` and `getDescription()` are used for logging purposes, only.

The method `calculatePerformers` is the most important method of the policy. When called, the policy return a `Performers` object, which is essentially a collection of users together with a Boolean flag that determines whether the current task is being forced onto a user. Some clients, most notably the *Site Manager* may choose to accept a forced task automatically on behalf of the current user. In principle, you can use any algorithm to compute the collection of users, but you should normally respect at least exclusions and rejections in that computations.

When no users are found, the policy is free to take appropriate measures to resolve this situation, for example by clearing the set of rejections. When an empty set of users is returned from the method, the task will be escalated.

A collection of users is passed into the method `calculatePerformers`. This collection contains all users that are permitted to accept the task in question. By taking this collection into account when determining the performers, you avoid duplicating the rights rules in the performer policy.

The method `mayDelegateTo` is called when a user tries to delegate a task to another user. While the permission check for the executing user is done by a `RightsPolicy`, the `PerformersPolicy` checks whether a designated user may receive the task, typically taking the set of excluded users into account.

6.10.7 Rights Policies

A rights policy governs, for a certain workflow object, which users have permission to exercise which rights. The rights policy is configured for each process and for each task in the workflow definition.

The rights policy can be retrieved and accessed directly from the `AccessControl` service (though this is rarely necessary), and is also used for various internal purposes.

- All methods in the `AccessControl` service eventually delegate to the rights policy.
- All client-side access checks (for lightweight sessions) are based on the rights policy. For the connection session, rights are checked on the server.
- The work list computation is based on the rights policy, on the client as well as on the server.

In order to avoid costly network round-trips, each connection to the workflow repository obtains a local copy of the rights policy configuration, and performs all rights computations using client-side code.

The default rights policy is described in the *Workflow Developer Manual*. In the following, you will learn how to deploy a custom rights policy.

A custom rights policy is often used to influence the "can do" work list, also known as "tasks offered". As described in Section 6.4, “The Work List Service” [95], the `TASK_ACCEPT` right forms the basis for the computation of the `tasksOffered` work list, which can therefore be customized by changing the rights policy.

Marshalling

Responsibility for computing work lists is split between server and client. It is therefore essential that both the server-side and client-side code behave exactly the same. The server-side implementation has to implement either the `WfRightsPolicy` interface as described in the *Workflow Developer Manual*, or the `RightsPolicy` interface of the *Unified API*. The client-side implementation in the *Unified API* has to implement `RightsPolicy`.

The server-side rights policy is created and configured when the workflow definition is uploaded and parsed, and is stored in the database in serialized form. Since Java serialization is unsuitable as a cross-platform network protocol, the rights policy needs to supply a marshaller implementation, which encodes the rights policy configuration into a portable format such as XML, together with an ID (the policy ID) identifying the format. This encoded form is transmitted to the client when the client loads the process or task definition.

On the *Unified API* client side, the policy ID is used to select a `RightsPolicyMarshaller` which parses the transmitted configuration, and creates the client-side counterpart as an instance of a class implementing `RightsPolicy`.

The code for the client-side rights policy and its marshaller must be deployed in a JAR file in the client's class path. The first implementation of `RightsPolicyMarshaller` with a matching policy ID is used. Names of implementing classes must be listed in a file called `META-INF/services/com.coremedia.cap.workflow.plugin.RightsPolicyMarshaller` inside the JAR file, as described in the "Service Provider" section of the JAR File Specification.

For example, the `cap-client.jar` contains a file with the above name, consisting of the following line:

```
com.coremedia.cotopaxi.workflow.authorization.
ACLRightsPolicyMarshaller
```

This instructs the *Unified API* implementation to consult the class `ACLRightsPolicyMarshaller` in the given package when unmarshalling rights policies. The method `RightsPolicyMarshaler#getPolicyId` of this class returns the string `"coremedia:///cap/workflow-rights-policy/ACL"`, which is exactly the policy ID marshalled by the server side default rights policy.

When a rights policy with this policy ID is received on the client, the ACL rights policy marshaller is invoked and reads the ACL configuration sent from the server. The marshaller creates and returns an instance of `ACLRightsPolicy` interpreting this configuration. As the name suggests, `ACLRightsPolicy` is the implementation of the `RightsPolicy` interface for this rights policy.

Configuration

If you want to design a *Unified API* rights policy that can be used on the server side, your policy must be configurable from the XML definition. The rights policy is configured in the element `<Rights>` that may be included in the definition of any task of process. Traditionally, configuration is done by including `<Grant>` and `<Revoke>` subelements in the `<Rights>` element. If you want to support this configuration style, you should implement the interface `ConfigurableRightPolicy` instead of `RightsPolicy`.

That interface provides a number of grant and revoke methods that are called by the server while it parses the definition file. It also contains the method `setRights(Set)`, which informs the policy about those rights that it is supposed to handle. Note that this set is different for tasks and processes.

Rights Computation

The interface `RightsPolicy` contains four `mayPerform` methods that must return true or false as a certain operation is allowed or forbidden. The `getUsers` methods allow the policy to determine those users who are allowed to perform a certain operation. Similarly, the `getGroups` methods return collections of groups. This is particularly useful when showing a selection of potential users or groups during user interaction.

The rights for a certain task may depend on more than just a set of groups and their membership relations, as is true for the default implementation. For example, the right for an approval task may depend on the documents that are about to be approved. Note that when accessing content in a rights policy, the "can do" work list will only be re-evaluated after changes to content that is directly referenced by the process. Therefore, it is not recommended to access other content from the rights policy.

6.10.8 Remote Client Actions

A workflow definition includes actions in several places, for example as entry and exit actions of user tasks, and in automated tasks. Specific kinds of action are the so-called remote client actions. While normally actions are executed on the *Workflow Server*, a remote client action is invoked on the user's computer.

Remote client actions can be used to invoke functionality that is not easily emulated using customized workflow variable editors. For example, a remote client action might open a publication window, it might present a modal dialog, or it might start a native application installed on the user's computer.

A remote client action can only execute when a session is open from the client to the workflow server. Similar to receiving an event, the session is informed that a remote client action should be executed. Since the server needs to know which session to inform, this is only possible for task entry or exit actions. The session that accepted or completed the task is recorded and used for the following remote client action.

For a remote call, the name and parameters of the action are passed to the client. There, the call is presented to each registered `RemoteActionHandler` in turn. If a `RemoteActionHandler` cannot handle the call, it returns null instead of an action result, and the next handler will be invoked. The *Unified API* includes a remote action handler for all predefined workflow actions. Additional handlers for custom remote client actions can be added using `WorkflowRepository#addRemoteActionHandler`. See the Javadoc for details.

Remote Actions That Are Not Remote

The same XML fragments and action names used for remote action invocation can also be used in automated tasks, where no client connection is available. In this case, the "remote" action is not actually remote, but is executed on the server, by a server-local remote action handler.

It is also possible to execute an action in the name of a user without requiring the user to be currently logged in to the workflow server. When using the XML attribute *`userVariable`* for an action in an automated task, a content repository session for the given user is established before invoking the server-local remote action handler. The attribute *`userVariable`* contains the name of a `UserVariable` of the task or process, which is read every time the action is executed.

In order to deploy a custom remote action handler on the *Workflow Server*, you need to add its class name to the property `workflow.server.remote-action-handler` which contains a comma separated list of fully qualified class names. Your class should provide a public no-args constructor and should implement the `RemoteActionHandler` interface. The built-in remote action handler provided by CoreMedia, `com.coremedia.cotopaxi.workflow.BuiltInRemoteActionHandler`, should usually come last. The JAR file containing your classes must be in the workflow server's class path. New or changed server-side handlers only become effective when the server is restarted.

6.10.9 Managers

Managers are components that are deployed in the *Workflow Server*, becoming globally available for use by other plugins. Managers may encapsulate global state that is relevant for multiple processes. They may also coordinate the interaction of processes with external entities, possibly acting as a connection pool. They may react to an external event

by requesting a recheck of one or more processes. During a recheck, all guards that were evaluated to false are reevaluated, so that waiting tasks may start running.

Managers must be registered by indicating at least the class of the manager in a property named `workflow.server.managers.<name>.class`, where `<name>` becomes the name of the manager. Using the optional properties `workflow.server.managers.<name>.order` it is possible to control the startup order of the managers. Additional properties with the prefix `workflow.server.managers.<name>.` may be given in the configuration file and retrieved by the manager during setup.

All managers must implement the interface `Manager`. Typically, it is simpler to base own implementations on the abstract base class `AbstractManager`, though. This class provides utility methods for obtaining the *Unified API* connection and for reading manager-specific configuration parameters.

The life cycle of a manager starts with its creation by means of a no-arg constructor. Afterwards, the setters `setName` and `setConnection` are called, providing context information to the manager. Afterwards `init()` is called, which allows the manager to set up itself based on configuration. It may not yet start asynchronous behavior, because some parts of the server might not be fully set up. That in turn is allowed after the method `start()` has been called.

The method `stop()` is called when the server is shut down, just before the *Workflow Server* stops to execute automatic tasks and to accept external requests. The manager should stop all asynchronous behavior before returning from this method. Finally, the method `dispose()` is called. It provides the manager with an opportunity to release any system resources, in particular if custom workflow action were still accessing those resources.

In order to react to external events, processes should register themselves with an appropriate manager. That manager reacts to events by calling `recheck(Process, ...)` with the affected process as an argument. Note that this is the only way for a manager to influence a process directly while operating asynchronously out of the scope of a call from the server. Particularly, it is not allowed for the manager to update process variables itself. This must be done by the process after requesting information from the manager or in a call from the process to the server during the execution of an action.

Using the `ManagerService`, which is an aspect of the workflow repository that is only available in the *Workflow Server*, other plugins may request a reference to a manager by providing the manager's name. Note that one manager class may be registered multiple time using different names, if that is required. Clients will have to make a cast to be able to call the business methods of the manager.

6.11 Examples

In this section you will find examples for both the client API and the API for writing *Workflow Server* plugins.

6.11.1 Example Clients

As a very simple application, here is a little tool that aborts all running workflows.

```
WorklistService worklist =
    connection.getWorkflowRepository().getWorklistService();
  Set<Process> running = worklist.getAllProcessesRunning();
  for (Process process: running) {
    process.abort();
  }
```

Example 6.1. `AbortAllProcesses`

The code deals with the workflow repository and one of its aspects, the worklist service. The worklist service maintains a number of collections of workflow objects. Each collection contains those objects that match a certain predicate: running processes, escalated tasks, available process definitions, and so on.

In Example 6.1, "`AbortAllProcesses`" [120], one of the administrative worklists is used. The method `getAllProcessesRunning()` returns the set of all processes that are started, but not yet finished.

Elaborating on this example, a minor variant follows. Example 6.2, "Suspend My Processes" [120] operates only on processes that were started by a single user. More precisely, only processes of the current session's user are returned during a call to `getProcessesRunning`.

```
WorkflowRepository repository = connection.getWorkflowRepository()
  WorklistService worklist = repository.getWorklistService();
  Set<Process> running = worklist.getProcessesRunning();
  for (Process process: running) {
    process.suspend();
  }
```

Example 6.2. Suspend My Processes

As you can see, the processes are suspended instead of being aborted. Suspended processes keep their state and can be resumed later on.

In the next example, you will see how to create and control a process using the *Unified API*. Before running the Example 6.3, "Create Process Example" [121], the standard three-

step publication workflow must have been uploaded. Please see the *Administration Manual* for details.

The example code starts with retrieving the process definition via the well-known name and creates a process instance afterwards. The process has to be started before its first task can be started.

```
ProcessDefinition processDefinition =
    repository.getProcessDefinition("ThreeStepPublication");
  Process process = processDefinition.create();
  process.start();
```

Example 6.3. Create Process Example

When the process is started, the process definition on the server takes control. In the case of the three step publication, this leads to the Compose task eventually being offered to the process' owner. The code repeatedly tries to accept the task, which may fail because the task is not offered yet, or because it was already automatically accepted by a connected *Site Manager*.

```
Task composeTask = process.getTask("Compose");
  while (!composeTask.isAccepted()) {
    try {
      composeTask.accept();
    } catch (IllegalTaskStateException e) {
      // not offered yet, or race condition with editor
    }
    Thread.sleep(1000);
  }
```

Now that you are sure that the task is accepted, you can freely access its variables.

```
composeTask.getView().set("subject", "a subject");
  connection.flush();
  Thread.sleep(5000);    // let the user have a good look
  composeTask.complete();
```

Here one variable is set to a new value and the change is flushed to the server. Just as with content, writes are buffered for workflows, too. After providing you with some time to inspect the new variable value in the editor, the compose task is completed. Because the change set was not changed and is still empty, the three-step publication process terminates automatically.

6.11.2 Example Plugins

This section provides some examples of various types of plugins. All classes must be deployed on the *Workflow Server* in order to become functional.

Expression

The first example show how to create a reusable expression for performing queries in the *Content Server*. The expression starts by extracting the *Unified API* connection from the argument workflow object. (Alternatively, the connection could have been injected by implementing the CapConnectionAware interface)

```
public Object evaluate(WorkflowObject wo,
  Map<String,Object> localVariables)
{
  CapConnection connection = wo.getRepository().getConnection();
  QueryService queryService = connection.getContentRepository()
    .getQueryService();
```

Afterwards, all subexpressions are evaluated. Note that the `localVariables` are passed to the subexpression unchanged.

```
Object[] parameters = new Object[expressions.size()];
  for (int i = 0; i < parameters.length; i++) {
    Expression expression = expressions.get(i);
    parameters[i] = expression.evaluate(object, localVariables);
  }
```

Lastly, you can pose the actual query.

```
return new ArrayList<Content>(queryService.
    poseContentQuery(query, parameters));
}
```

In the XML definition, the subexpressions occur as XML subelements and the query as an attribute of the `<Expression>` element.

```
<Expression class="com.coremedia.examples.plugin.QueryExpression"
  query="REFERENCED BY ?0">
  <Get variable="document"/>
</Expression>
```

The query string is passed into the `QueryExpression` object by means of a specific setter `setQuery(String)`. The subexpression `Get` is parsed and handed to the example expression and stored in a list named `expressions` by means of the following method:

```
public void add(Object o) {
  if (o instanceof Expression) {
    expressions.add((Expression)o);
  } else {
    throw new RuntimeException("don't know how to add "+o);
  }
}
```

Even if you do not intend to use subexpressions, you might want to implement a similar method when requiring a highly flexible configuration mechanism. Every nested XML element that cannot be handled by a more specific setter method is passed to the `set(Object)` method.

Action

The next example introduces a custom action that is capable of moving and renaming a content atomically. The class is named `MoveAndRenameAction`. It is derived from the base class `SimpleAction`, which further reduces its complexity.

The arguments for the action are taken from three process variables, whose names are configured in the XML definition and stored in three fields in the action. The action is configured as follows:

```
<Action class="com.coremedia.examples.plugin.MoveAndRenameAction"
  contentVariable="content"
  targetVariable="target"
  nameVariable="name" />
```

Except for the three fields and the setters, the implementation consists of a single method.

```
public boolean doExecute(Process process) {
  Content content = process.getLink(contentVariable);
  Content target = process.getLink(targetVariable);
  String name = process.getString(nameVariable);
  content.moveTo(target, name);
  return true;
}
```

By returning true, the action indicates that it completed normally.

Another example of an action implemented as `SimpleAction` that sends emails is listed in Section 6.11.3, “Example Code of the Mail Action” [127]. Because the mail server is an external component that might take long to respond this action is a good candidate to be implemented as a `LongAction` as described below.

LongAction

In the following, an action that sends a mail is implemented. Because the mail server is an external component that might not respond immediately, a long action is created. You omit the definition of various string fields that hold configuration values for the action and skip immediately the methods for executing the action.

During the first phase the receiver, subject and body text of the mail are determined.

```
public class MailAction implements LongAction {
  public Object extractParameters(Task task) {
    com.coremedia.cap.workflow.Process process =
      task.getContainingProcess();
    String receiver = process.getString(receiverVariable);
    String subject = process.getString(subjectVariable);
    String text = process.getString(textVariable);
    return new Object[]{receiver, subject, text};
  }
  ...
```

Afterwards, the mail is actually sent outside of a DB transaction.

```
public Object execute(Object params) {
    Object[] paramArr = (Object[]) params;
    String receiver = (String) paramArr[0];
    String subject = (String) paramArr[1];
    String text = (String) paramArr[2];
    boolean result = false;
    try {
      result = send(host, user, password,
        from, receiver, subject, text);
    } catch (Exception e) {
      return e;
    }
    return result;
  }

  protected boolean send(String host, String username,
    String password, String from,
    String receiver, String subject, String text)
  throws MessagingException, AddressException {
    ...
  }
}
```

Please see the full source code for details of the mail delivery, which is outside the scope of this manual. Finally, the result is converted into an action result.

```
public ActionResult storeResult(Task task, Object result) {
    if (result instanceof Boolean) {
      return new ActionResult(((Boolean)result).booleanValue());
    } else {
      return new ActionResult((Exception)result);
    }
  }
}
```

Assuming there are process variables `receiver`, `subject`, and `text`, the `LongAction` could be used in a process definition as follows:

```
<AutomatedTask name="SendMail" final="true">
  <Action class="com.coremedia.examples.plugin.MailAction"
    host="smtp.company.com"
    user="automailer"
    password="secret"
    from="noreply@company.com"
    receiverVariable="receiver"
    subjectVariable="subject"
    textVariable="text"/>
</AutomatedTask>
```

PerformersPolicy

One example of a performer policy is the `DefaultPerformersPolicy`, which is distributed together with the *Unified API* sources. The main method of that class will be discussed here.

First, the users that may execute the task are calculated.

```
public Performers calculatePerformers(Task task,
  Collection permittedUsers)
{
  Set<User> users = new HashSet<User>();
```

```
  users.addAll(permittedUsers);
  users.removeAll(getExcludedUsers(task));
```

If the task is forced to a user, that user is chosen.

```
User forcedUser = getForcedUser(task);
  if(forcedUser != null) {
    if(users.contains(forcedUser)) {
      return new Performers(forcedUser, true);
    }
  }
```

Otherwise, you look for users who are preferred, but have to rejected the task.

```
users.removeAll(getRejectedUsers(task));
  Set<User> preferredUsers =
    new HashSet<User>(getPreferredUsers(task));
  preferredUsers.retainAll(users);
  if(preferredUsers.size() > 0) {
    return new Performers(preferredUsers, false);
  }
```

If you failed due to rejections, those rejections are cleared before recomputing the set of users. Note that this is a side effect that is explicitly allowed during the `calculatePerformers` method.

```
if(users.size() == 0 && getRejectedUsers(task).size() > 0) {
    removeAllRejections(task);
    return calculatePerformers(task, permittedUsers);
  }
```

If there are no rejections to be cleared, you have to go with the users that are not preferred.

```
return new Performers(users, false);
}
```

RightsPolicy

In the following, you will see the *Unified API* half of a custom rights policy. That policy assigns rights to exactly that user who created a process and grants rights for the creation of new processes to all members of a single group.

The server half, as presented in the *Workflow Manual*, is only sufficient for use in the *Workflow Server and* for the editor. The *Unified API* needs its own implementation.

First, you need some code to deal with serialization and configuration.

```
public class OnlyOwnerRightsPolicy implements RightsPolicy {
  private static final long serialVersionUID =
   7465148942676430339L;

  private Group group = null;

  public void setGroup(Group group) {
    this.group = group;
```

```
  }

  public Group getGroup() {
    return group;
  }

  public void setGroup(String groupAtDomain) throws WfException {
    UserRepository userRepository = WfServer.getConnection().
     getUserRepository();
    Group group = userRepository.getGroupByName(groupAtDomain);

    if (group == null) {
      throw new RuntimeException("Could not find
      group "+groupAtDomain);
    }
    setGroup(group);
  }
  ...
```

The last method is called only in the *Workflow Server* while an XML process definition using the new policy is parsed. You are therefore allowed to obtain the server's *Unified API* connection through the `WfServer` singleton.

Now you can look at some of the methods that compute the rights of individual users.

```
private User getOwner(WorkflowObject workflowObject) {
    if (workflowObject instanceof Task) {
      workflowObject =
       ((Task)workflowObject).getContainingProcess();
    }
    return ((Process)workflowObject).getOwner();
  }
  public boolean mayPerform(WorkflowObject workflowObject,
    Right right, User user)
  {
    if (user.isSuperUser()) return true;
    User owner = getOwner(workflowObject);
    return owner != null && owner.equals(user);
  }
  public boolean mayPerform(WorkflowObjectDefinition
    definition, Right right, User user)
  {
    return user.isMemberOf(group);
  }
  ...
```

Skipping some parts of the code that are very similar to the server-side code as presented in the *Workflow Manual*, you observe that there is also a weight method that estimates the main memory size of the policy in bytes. It is used for caching policies. Here 12 bytes for the policy and 16 bytes for the referenced group are estimated.

```
public int getWeight() {
    return 28;
  }
```

Finally, there is the unmarshalling process that is needed to create a policy instance in the client VM.

```
public RightsPolicyMarshaller getMarshaller() {
    return new OnlyOwnerRightsPolicyMarshaller();
```

```
  }
}
```

The marshaller itself resides in yet another class. Let us look at the unmarshal method, only.

```
public class OnlyOwnerRightsPolicyMarshaller
implements RightsPolicyMarshaller {
  ...
  public RightsPolicy unmarshal(CapConnection connection,
    byte[] data)
  {
    OnlyOwnerRightsPolicy result = new OnlyOwnerRightsPolicy();
    if (data[4] == 1) {
      int groupId = (data[0] & 0x000000ff) +
        (data[1]<<8 & 0x0000ff00) +
        (data[2]<<16 & 0x00ff0000) +
        (data[3]<<24);
      result.setGroup(connection.getUserRepository().
        getGroup(IdHelper.formatGroupId(groupId)));
    }
    return result;
  }
  ...
}
```

Notice how a connection is passed into the unmarshaller, so that it can be used to build *Unified API* objects for use in the policy.

6.11.3 Example Code of the Mail Action

Here you find the partial process definition and the simple implementation of an action sending emails. The action is implemented as a `SimpleAction` with predefined timeout. If you need to increase the timeout, you should implement interface `LongAction` instead, which is better suited for long-running actions.

Assuming there are process variables `field` and `document` the mail action can be defined as follows:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Workflow>
  <Process name="SendMailProcess" startTask="SendMail">

   <Variable name="receiver" type="String">
      <String value="default@test.com"/>
   </Variable>

   <Variable name="field" type="String"/>
   <Variable name="document" type="Resource"/>

   <Variable name="delivered" type="Boolean">
      <Boolean value="false"/>
   </Variable>

   <AutomatedTask name="SendMail"
                  description="mail-task" final="true">
     <Action class="com.coremedia.extension.workflow.mail.SendMail"
             receiverVariable="receiver"
```

```
            documentVariable="document"
            fieldVariable="field"
            successVariable="delivered"/>
   </AutomatedTask>

   ...
  </Process>
</Workflow>
```

The implementation then looks as follows:

```
package com.coremedia.extension.workflow.mail;

import com.coremedia.cap.content.Content;
import com.coremedia.cap.workflow.Process;
import com.coremedia.cap.workflow.plugin.SimpleAction;
import com.coremedia.xml.Markup;
import org.slf4j.*;
import javax.mail.*;
import javax.mail.event.*;
import javax.mail.internet.*;

public class SendMail extends SimpleAction {

  private static final Logger LOG =
                          LoggerFactory.getLogger(SendMail.class);
  static final long serialVersionUID = 1258062873454333627L;

  protected String transportType = "smtp";
  protected String host = "smtp.coremedia.com";
  protected String user = "testuser";
  protected String password = "testpassword";
  protected String from = "testuser@coremedia.com";
  protected String subject = "This is a test mail";

  protected String receiverVariable;
  protected String fieldVariable;
  protected String documentVariable;

  protected Message createMessage(Session session,
                                  String from, String to,
                                  String subject, String text)
                   throws MessagingException {

    MimeMessage message = new MimeMessage(session);
    message.setFrom(new InternetAddress(from));
    message.addRecipient(Message.RecipientType.TO,
                         new InternetAddress(to));
    message.setSubject(subject);
    message.setText(text);
    message.saveChanges();
    return message;
  }

  protected boolean send(String host, String username,
                         String password, String from,
                         String to, String subject, String text,
                         String transport_type)
            throws MessagingException {

    Session session = Session.getDefaultInstance
                                  (System.getProperties(), null);

    Message message = createMessage(session,from,to,subject,text);
    MessageDelivery delivery = new MessageDelivery();

    Transport transport = session.getTransport(transport_type);
    transport.addTransportListener(delivery);
    transport.connect(host, username, password);
```

```
  transport.sendMessage(message, message.getAllRecipients());
  transport.close();
  transport.removeTransportListener(delivery);

  return delivery.isMailDelivered();
}

@Override
protected boolean doExecute(Process process) {

  String to = process.getString(receiverVariable);
  Content content = process.getLink(documentVariable);
  if (content == null) {
    return false;
  }
  String field = process.getString(fieldVariable);
  Markup markup = content.getMarkup(field);
  String body = markup == null ? "" : markup.toString();
  try {
    return send(host, user, password, from, to, subject,
                                      body, transportType);
  } catch (MessagingException e) {
    LOG.error(e.getMessage());
  }
  return false;
}

// Setters for configuring the action in a process definition.
public void setReceiverVariable(String receiverVariable) {
  this.receiverVariable = receiverVariable;
}

public void setFieldVariable(String fieldVariable) {
  this.fieldVariable = fieldVariable;
}

public void setDocumentVariable(String documentVariable) {
  this.documentVariable = documentVariable;
}

protected static class MessageDelivery
                    implements TransportListener{

  // wait a second for delivery
  // (If you need to increase the timeout, you should instead
  // implement interface LongAction which is better suited
  // for long-running actions. You should also implement method
  // #abort correctly so that the execution of the action does
  // not interfere with the shutdown of the Workflow Server.)
  private static final long TIMEOUT = 1000;

  private Boolean delivered = null;

  protected synchronized boolean isMailDelivered() {
    long timeout = System.currentTimeMillis() + TIMEOUT;
    while (delivered == null) {
      long now = System.currentTimeMillis();
      if (now >= timeout) {
        break;
      }
      try {
        wait(timeout - now);
      } catch (InterruptedException e) {
        LOG.error(e.getMessage());
      }
    }
    return Boolean.TRUE.equals(delivered);
  }

  private synchronized void deliverySuccess(boolean state) {
    delivered = state;
    notifyAll();
```

```
    }

    public void messageDelivered(TransportEvent e) {
      deliverySuccess(true);
    }

    public void messageNotDelivered(TransportEvent e) {
      deliverySuccess(false);
    }

    public void messagePartiallyDelivered(TransportEvent e) {
      deliverySuccess(false);
    }
  }
}
```

Example 6.4. The SendMail action

6.12 Guide to the API Documentation

The `WorkflowRepository` in `com.coremedia.cap.workflow` handles processes and tasks. It provides two aspects `WorklistService` and `AccessControl`.

You may want to start with the interface `Task` and inspect its methods and its state diagram. The access to variables of a task is exactly the same as the access to properties of a content (see `CapObject`). Keep in mind, however, that there are additional data types available in the workflow context.

Afterwards, consult the interface `Process`, memorize that although there are some similarities in its state diagram, equal state names mean different things for processes and tasks. See Section 6.2, “Workflow States” [88] for further details.

Conclude the first look at the workflow repository with the `WorklistService` aspect and examine the various collections of workflow objects it provides. Some background information is provided in Section 6.4, “The Work List Service” [95].

7. The User Repository

The user repository stores information about users and groups. It allows you to create, retrieve, read and update user and groups that are stored in the built-in user management of the *Content Server*. It also provides read access to additional users and groups that are managed in an LDAP server that may be associated with the *Content Server*.

7.1 Objects

The user repository manages `User` objects and `Group` objects. A `Group` can contain an arbitrary number of `Member` objects, which may be users or groups. The *Unified API* distinguishes between membership and direct membership. Only the latter is directly stored, the former is computed dynamically. A `Member` object is a member of a certain group, if there is a chain of direct member associations that ultimately leads from the group to the member.

Every member has a name and a domain. There are typically only very few domains in any given *CoreMedia CMS* installation, leaving the name as the main identifying feature of a member. A user is often designated in the `<name>@<domain>` format, for example, `joe@mydomain` or `admin@`. As you can see, for built-in users, the domain part is left empty.

The domain that is represented by the empty string provides access to the built-in user management of the *Content Server*. For members of this domain this is also indicated by the method `isBuiltIn()`. Only members of the built-in user management may be changed under direct control of the *Unified API*. Users of other, external domains are mapped into the system from external servers by means of the LDAP protocol. Only read access is allowed for external domains. You can access the distinguished name of an external user through the *Unified API* in case you need to connect back to the LDAP repository.

For users of external domains, the getter methods of `CapObject`, which is a super interface of `Member`, may be used to access custom string attributes stored in the LDAP server. The built-in user management does not support member attributes. Note that there is no fixed set of `CapType` objects for members, because LDAP does not enforce a strict typing. Instead, there is one artificial type per member that describes the available properties for these objects.

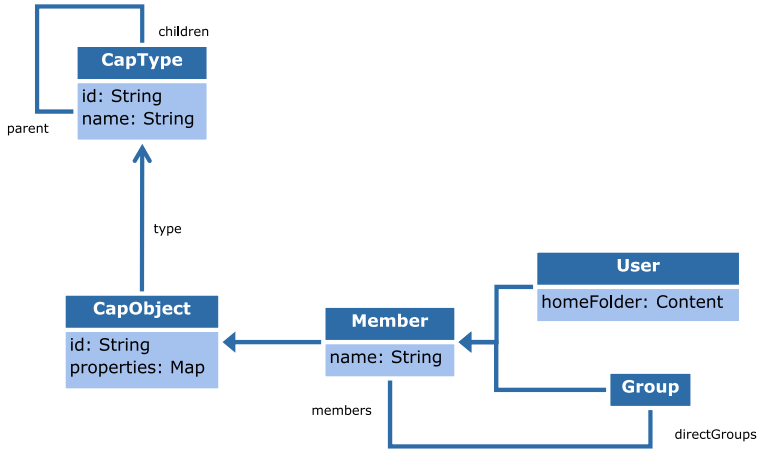

Figure 7.1. Class Diagram: Users and Groups

In Figure 7.1, "Class Diagram: Users and Groups" [134] you can see an overview of all classes involved in the representation of users and groups.

A group is called administrative, if its direct and indirect members are supposed to gain special privileges while working in the *CoreMedia CMS*. A user is called administrative, if at least one of its direct or indirect groups is administrative.

For the purposes of assigning rights to users, groups may be designated as content groups or live groups or both. Only rules of content groups affect the computation of rights on the *Content Management Server*. Only read rights rules that are defined for live groups are published to the *Live Servers*.

CAUTION

CoreMedia Live Server Group management is not used anymore and has therefore been deprecated.

CoreMedia is planning to remove Live Server Group management together with Site Manager from the product portfolio with the next major release.

For users, the home folders can be retrieved as a content object. As already explained, setting the home folder is only possible for built-in users.

7.2 UUIDs

In addition to the simple string identifiers described in Section 4.6, "Identifiers and Equality" [39], every `Member` on the *Content Management Server* has a UUID since version 2007.1. UUIDs are stable and universally unique identifiers as defined in RFC 4122 and are represented as `java.lang.UUID`. They are not meant as replacement of simple string IDs, and should not be used where a simple ID is sufficient. UUIDs make sense in certain scenarios where uniqueness across multiple repositories is important, or when users or groups may be transferred to another repository and should keep their identity. While the latter is not yet possible in version 2007.1, later releases may add such features relying on UUIDs. External user provider implementations can provide the same UUIDs for identical users and groups on different *Content Server* installations. For example, users and groups from CoreMedia's ActiveDirectory UserProvider implementation receive their UUIDs from Active Directory. For details on user providers, see Section 3.12, "LDAP Integration" in *Content Server Manual*. The `LdapUserProvider` class has methods that take or return UUIDs, and that can be overridden by a custom user provider. See its API documentation for details.

Similar to string IDs, the API provides a `getUuid()` method in class `Member` to retrieve a UUID, and methods to look up a `User` or `Group` for a given UUID. A `User` or `Group` with a given UUID can be retrieved from the `UserRepository` with method `getMember(UUID)`, `getUser(UUID)`, or `getGroup(UUID)`. It is important to note, that a UUID does not encode any further information about the referred object. It cannot be used to identify the type of the referred object, or the repository that contains it.

Note, that as of version 2007.1, UUIDs are only available on the *Content Management Server*. If a connection is made to a live server, or to a server of a previous release, then all methods that would return a UUID will return null instead.

7.3 Retrieving Objects

The `UserRepository` interface contains a number of methods that allow you to get access to `User` and `Group` objects. As in all repositories, you can obtain a `Member` with a given string id using `getMember(String)`. It is also possible to retrieve members based on their name and optionally their domain using the `getUserByName` and `getMemberByName` methods.

You can issue a query for users or groups that provides only a substring of the actual name. In this case, the methods `findUsers` and `findGroups` return a collection of matching objects. Using these methods, you can set an upper bound on the number of results, reducing the load on the repository.

7.4 Listeners

The `UserRepository` provides listeners with events about all changes to user and groups. The interface `UserRepositoryListener` is partitioned into three main parts.

- The super interface `UserListener` contains methods regarding the creation, update and destruction of users.
- The super interface `GroupListener` is concerned with groups.
- A `UserRepositoryListener` defines two additional methods to be called when a group gains or loses a member.

The class `UserRepositoryListenerBase` provides an empty default implementation that can be overridden as needed. Attach your listener using the `addUserRepositoryListener` method of the `UserRepository`.

Because LDAP does not provide an event mechanism, the *Content Server* has no immediate means to detect changes to members that are imported from an LDAP server. When being accessed for the first time, a creation event is generated. Note that the first access may be seconds, days, or years after the user was actually created.

LDAP data is cached for a certain amount of time and not refetched from the server. During that time, changes to the LDAP server are not detected. If an LDAP member is accessed again after it expires from the cache or if it is explicitly updated using the calls `invalidate()` or `refresh()`, the *Content Server* may detect some changes and send appropriate events. No events are sent for custom properties of members. No events are sent for changes of the group-member association. In short, while you may find it convenient to monitor the events for LDAP members, the events are incomplete and may arrive late.

7.5 Further Reading

Refer to the *Content Server Manual* for more information on how to connect *CoreMedia CMS* with an LDAP server and on how to create users by means of the built-in user management.

The access control services of the content and workflow repositories must take the structure of users and groups into account when computing rights. The `WorklistService`, too, is dependent on the specific user who accesses the worklist.

The Javadoc of the *Unified API* is the recommended source for in-depth descriptions of individual classes and methods. Look at the interfaces `UserRepository`, `User`, `Group`, and `Member` in the package `com.coremedia.cap.user` primarily.

Glossary

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

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

CoreMedia Studio

CoreMedia Studio is the working environment for business specialists. Its functionality covers all 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.

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, Smartwatches/Wearable Devices, Out-of-Home or In-Store Displays or Internet-of-Things use cases.

Home Page

The main entry point for all visitors of a site. Technically it is often referred to as root document and also serves as provider of the default layout for all subpages.

IETF BCP 47

Document series of *Best current practice* (BCP) defined by the Internet Engineering Task Force (IETF). It includes the definition of IETF language tags, which are an abbreviated language code such as en for English, pt-BR for Brazilian Portuguese,

	or nan-Hant-TW for Min Nan Chinese as spoken in Taiwan using traditional Han characters.
Importer	Component of the CoreMedia system for importing external content of varying format.
IOR (Interoperable Object Reference)	A CORBA term, *Interoperable Object Reference* refers to the name with which a CORBA object can be referenced.
Jangaroo	*Jangaroo* is a JavaScript framework developed by CoreMedia that supports ActionScript as an input language which is compiled down to JavaScript. You will find detailed descriptions on the Jangaroo webpage http://www.jangaroo.net.
Java Management Extensions (JMX)	The Java Management Extensions is an API for managing and monitoring applications and services in a Java environment. It is a standard, developed through the Java Community Process as JSR-3. Parts of the specification are already integrated with Java 5. JMX provides a tiered architecture with the instrumentation level, the agent level and the manager level. On the instrumentation level, MBeans are used as managed resources.
JSP	JSP (Java Server Pages) is a template technology based on Java for generating dynamic HTML pages. It consists of HTML code fragments in which Java code can be embedded.
Locale	Locale is a combination of country and language. Thus, it refers to translation as well as to localization. Locales used in translation processes are typically represented as IETF BCP 47 language tags.
Master Live Server	The *Master Live Server* is the heart of the *Content Delivery Environment*. It receives the published content from the *Content Management Server* and makes it available to the *CAE*. If you are using the *CoreMedia Multi-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. CoreMedia Studio uses the Open Source compiler Jangaroo 4 to translate MXML and ActionScript sources to JavaScript that is compatible with Ext JS 6.
Personalisation	On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.
Projects	A project is a collection of content items in CoreMedia CMS created by a specific user. A project can be managed as a unit, published or put in a workflow, for example.

Property	In relation to CoreMedia, properties have two different meanings: In CoreMedia, content items are described with properties (content fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and XML for the textual content. Which properties exist for a content 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.
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

A

abort all running workflows, 120
Access Control, 60
access control servcie, 100
administrative group, 134
Audience, 2
automated task states, 89

B

Blob object, 33

C

caching, 49
CapException, 45
CapListener, 43
    asynchronous information, 43
CapObject, 30
CapSession, 46
CapType, 37
comparing objects, 30
connection, 20
    create, 20
    Lifecycle, 23
    map parameters, 21
    passing parameters as a Map, 21
    passing parameters as a URL, 23
    passing parameters directly, 20
    ServerControl, 26
connection listener, 26
content repository, 52
ContentRepositoryListener, 81
create new folder, 17

I

ID, 39
    formats for CapObject, 39
    formats for CapType, 40
    formats for other objects, 41

L

List, 34

M

Markup object, 32

O

ObservedPropertyService, 65

P

property service, 80
publication preview, 63
PublicationService, 62

Q

query service, 66

R

remote client actions, 117
repository, 19, 28
rights policies, 115

S

search service, 76
server side workflow API, 94
Simple Query Language, 77
system defined timer, 105

U

Unified API, 13
    use cases, 15
User object, 133
UUID
    Content, 58
    Group, 135
    User, 135
    Version, 58

V

values, 32

W

work list service, 95
workflow content service, 79
workflow events, 103
workflow repository, 83
workflow variables, 97
working version, 53
write buffering, 48