CoreMedia Content Cloud

Workflow Manual

CoreMedia

Copyright CoreMedia GmbH © 2024

CoreMedia GmbH

Altes Klöpperhaus, 5. OG

Rödingsmarkt 9

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.
November 07, 2024 (Release 2310)

1. Introduction 1
1.1. Audience 2
1.2. Structure Of The Manual 3
1.3. Typographic Conventions 4
1.4. CoreMedia Services 6
1.4.1. Registration 6
1.4.2. CoreMedia Releases 7
1.4.3. Documentation 8
1.4.4. CoreMedia Training 11
1.4.5. CoreMedia Support 11
1.5. Changelog 14
2. Overview of CoreMedia Workflow 15
3. Configuration And Operation 20
3.1. Starting the Workflow Server 21
3.2. Uploading Workflows 22
3.3. Converting Workflows 23
3.4. Using JMX Management 24
3.5. Workflow Server Utilities 25
3.5.1. Start 25
3.5.2. Download 25
3.5.3. Enable 27
3.5.4. Upload 27
3.5.5. Workflowconverter 28
3.5.6. Processdefinitions 30
3.5.7. Processes 31
4. Customize Workflow Definitions 32
4.1. Defining Workflows 33
4.1.1. The BeanParser 35
4.1.2. Elements of Activity Diagrams 37
4.1.3. Processes 38
4.1.4. Tasks 40
4.1.5. Flow Control 45
4.1.6. Workflow Variables 52
4.1.7. Expressions 52
4.1.8. Actions 54
4.1.9. Rights 55
4.1.10. Subworkflows 56
4.1.11. Timers 56
4.2. Upload Workflow Definitions 58
4.3. Example of Workflow Definition 59
4.4. Reference of Predefined Classes 66
4.4.1. Predefined Action Classes 66
4.4.2. Predefined FinalAction Classes 79
4.4.3. Predefined TimerHandler Classes 80
5. Implementing Extensions 83
5.1. Update Workflows 84
5.2. Variable Values 85
5.3. Programming Actions 86
5.3.1. General Rules 86

5.3.2. Repeated Execution of Actions 87
5.3.3. Server-Side Actions 88
5.3.4. Client-Side Actions 88
5.3.5. Access Workflow Variables from the Action 92
5.3.6. Example Action 93
5.4. Programming Expressions 94
5.4.1. General Rules 94
5.4.2. Generic Expressions 95
5.4.3. Boolean Expressions 96
5.4.4. Example Expression 96
5.5. Programming Rights Policies 99
5.5.1. Example Rights Policy 100
5.6. Programming Performer Policies 104
5.7. Programming Clients 107
5.8. Spring in the Workflow Server 108
5.8.1. Using Spring Beans 108
5.9. Pitfalls of Implemented Classes 109
6. Reference 111
6.1. Configuration Reference 112
6.1.1. Configuration of Workflow Client Properties 112
6.1.2. Configuration of Workflow Server Properties 112
6.1.3. Managed Properties 113
6.2. XML Element Reference 116
6.3. Studio Simple Publication Workflow Definition 170
Glossary 172
Index 179

List of Figures

2.1. CoreMedia architecture with integrated workflow 16
2.2. Control Room with workflow start window 17
2.3. Workflow in the Workflow App 18
2.4. Workflow window 19
4.1. Activity diagram of a simple workflow 34
4.2. Elements of activity diagrams 37
4.3. State diagram of a process 39
4.4. State diagram of a task 40
4.5. Example of a sequence diagram 46
4.6. Example of a choice diagram 47
4.7. Example of an implicit choice 48
4.8. Example of a loop 49
4.9. Example of a concurrency diagram 51
4.10. Expired timer with AbortTaskTimerHandler 81
4.11. Expired timer with `AddWarningTimerHandler` 81

List of Tables

1.1. Typographic conventions 4
1.2. Pictographs 5
1.3. CoreMedia manuals 8
1.4. Changes 14
3.1. Options of start 25
3.2. Options of download 26
3.3. Options of enable 27
3.4. Options of upload 28
3.5. Parameters of the *workflowconverter* utility 29
3.6. Options of the *processdefinitions* tool 30
3.7. Options of the processes utility 31
4.1. Workflow elements vs. programming language 35
4.2. Status of Tasks 40
4.3. Attributes common to all actions 66
4.4. Attributes of client-side actions. 67
4.5. Attributes of the ApproveResource action. 68
4.6. Attributes of the CheckInDocument action. 68
4.7. Attribute of the CeckOutDocument action. 68
4.8. Attributes of the CopyResource action. 69
4.9. Attributes of the CreateDocument action. 69
4.10. Attributes of the CreateFolder action. 70
4.11. Attribute of the DeleteResource action. 70
4.12. Attribute of the DisapproveResource action. 70
4.13. Attributes of the MoveResource action. 71
4.14. Attribute of the OpenDocument action. 71
4.15. Attributes of the PublishResources action. 71
4.16. Attributes of the RenameResource action. 72
4.17. Attribute of the SaveDocument action. 72
4.18. Attribute of the StoreProperties action. 73
4.19. Attribute of the UncheckOutDocument action. 73
4.20. Attribute of the UndeleteResource action. 73
4.21. Attributes of the ArchiveProcess action. 74
4.22. Attribute of the DisableTimer action. 75
4.23. Attribute of the EnableTimer action. 76
4.24. Attribute of the ExcludePerformer action. 76
4.25. Attribute of the ExcludeUser action. 76
4.26. Attributes of the ForceUser action. 77
4.27. Attributes of the Log action. 77
4.28. Attribute of the PreferPerformer action. 78
4.29. Attributes of the RegisterPendingProcess action. 78
4.30. Attribute of the CancelUserTask action. 79
4.31. Attribute of the SkipUserTask action. 79
4.32. Attributes of the ArchiveProcessFinalAction 80
6.1. Managed Workflow Server properties 113
6.2. Workflow Server operations properties 115
6.3. Attributes of Action element 119
6.4. Attributes of the AggregationVariable element. 119

6.5. Attribute of the Assign element 121
6.6. Attributes of the Automated Task element 122
6.7. Attribute of the Blob element 123
6.8. Attribute of the Boolean element 124
6.9. Attributes of the Case element 124
6.10. Attributes of the Choice element. 125
6.11. Attributes of the Condition element 125
6.12. Attribute of the ContentType element 126
6.13. Attribute of the Date element 126
6.14. Attributes of the Document element. 127
6.15. Attribute of the DocumentType element 127
6.16. Attribute of the Else element 128
6.17. Attributes of EntryAction element 128
6.18. Attributes of the Exists element 129
6.19. Attributes of the ExitAction element 130
6.20. Attributes of the Expression element 131
6.21. Attributes of the FinalAction element 131
6.22. Attributes of the Folder element. 132
6.23. Attributes of the ForAll element 132
6.24. Attributes of the Fork element 134
6.25. Attributes of the ForkSubprocess element 134
6.26. Attributes of the Get element 136
6.27. Attributes of the Grant element 137
6.28. Attributes of the Group element.name 139
6.29. Attributes of the If element 140
6.30. Attribute of the Integer element 142
6.31. Attributes of the IsDocument element 143
6.32. Attributes of the IsDocumentVersion element 143
6.33. Attributes of the IsEmpty element 144
6.34. Attributes of the IsExpired element 145
6.35. Attributes of the IsFolder element 145
6.36. Attributes of the Join element 146
6.37. Attributes of the JoinSubprocess element 146
6.38. Attributes of the Length element 147
6.39. Attributes of the Let element 149
6.40. Attributes of the NotEmpty element 150
6.41. Attributes of the Performers element 151
6.42. Attributes of the PostCondition element 152
6.43. Attributes of the Precondition element 153
6.44. Attribute of the Predecessor element 153
6.45. Attributes of the Process element 154
6.46. Attributes of the Property element 155
6.47. Attributes of the Read element 156
6.48. Attributes of the Reads element 156
6.49. Attributes of the Resource element. 157
6.50. Attributes of the Revoke element. 158
6.51. Attributes of the Rights element 159
6.52. Attribute of the String element 160
6.53. Attribute of the Successor element 160

6.54. Attributes of the Switch element. 161
6.55. Attribute of the Then element 162
6.56. Attributes of the Timer element 162
6.57. Attributes of the TimerHandler element 163
6.58. Attributes of the User element. 163
6.59. Attributes of the UserTask element 165
6.60. Attributes of the Validator element 166
6.61. Attributes of the Variable element 168
6.62. Attributes of the Writes element 169

List of Examples

4.1. Example of a BeanParser XML file 36
4.2. Example listing of a sequence 46
4.3. Example listing of a choice 47
4.4. Example listing of an implicit choice 48
4.5. Example listing of a loop 49
4.6. Example listing of concurrency 51
4.7. Example of a Guard 54
4.8. Example of the ACL for a process 55
4.9. Example of a self-defined timer which expires after 100 seconds 57
4.10. General definitions of the workflow 59
4.11. Automated task "Assign User" 61
4.12. User Task Compose 61
4.13. If Task 62
4.14. User Task "Publish" 63
4.15. If Task "CheckPublication" 64
4.16. Example of automated task Finish 64
4.17. Example of ArchiveProcessFinalAction 65
4.18. Example of the ArchiveProcess action 74
4.19. Example of the AssignVariable element 75
4.20. How to force a user 77
4.21. How to use a log action 78
4.22. Example of the ArchiveProcessFinalAction 80
4.23. Example of TimerHandler usage 80
5.1. Example of the server-side stub for a client-side action 90
5.2. Example of an action listener 91
5.3. Command executable on the Site Manager 91
5.4. How to configure an action bean 92
5.5. Example of an action 93
5.6. Use a generic expression in the workflow definition 95
5.7. Example of a generic expression 95
5.8. Example of a Boolean expression 96
5.9. Including expressions in the workflow definition 97
5.10. Example Expression 97
5.11. Integrate own rights policy in the workflow definition 100
5.12. Defining a performer policy in the workflow definition 105
5.13. Invoking a performer policy 105
5.14. Create a workflow client 107
6.1. Example of the variable usage 118
6.2. Action with a Guard used in a UserTask 119
6.3. Example of an aggregation variable 120
6.4. Example of an And element. 120
6.5. Example of an AutomatedTask 122
6.6. Example of an Assignment task 123
6.7. Example of a Blob variable 123
6.8. Example of a Boolean variable 124
6.9. Example of a Choice element 125

6.10. Example of a Condition element. It is checked whether the document variable is null or not. 126
6.11. Example of a ContentType variable 126
6.12. Example of a Date variable 127
6.13. Example of a Document variable. 127
6.14. Example of an EntryAction which checks out a document 128
6.15. Example of an Equal expression 129
6.16. Example of an Exists expression which checks if one of the documents in the variable Articles has the entry Sports in Topics 129
6.17. Example of an Exit Action which checks whether the document is null or not 130
6.18. Example of an Expression element 131
6.19. Example of a Folder variable 132
6.20. Example of a ForAll element which checks if all documents are checked in before approving them 133
6.21. Example of a Fork task 133
6.22. Example of a ForkSubprocess task 135
6.23. Example of a Get element 136
6.24. Example of a Grant element 138
6.25. Example of a Greater expression 138
6.26. Example of a GreaterEqual expression 139
6.27. Example of a Group variable 140
6.28. Example of a Guard 140
6.29. Example of an If task 141
6.30. Example for an Implies expression 141
6.31. Example of an InitialAssignment element 142
6.32. Example of an Integer Variable 142
6.33. Example of an IsDocument expression 143
6.34. Example of an IsDocumentVersion expression 144
6.35. Example of an IsExpired expression 145
6.36. Example of an IsFolder expression 145
6.37. Example of a Length element 147
6.38. Example of a Less expression 148
6.39. Example of a Let element which is needed to check whether the headline of an article is longer than 50 characters or not 149
6.40. Example of a Not element 149
6.41. Example of a NotEqual expression 150
6.42. Example of an Or expression 151
6.43. Performers element 152
6.44. Example of a PostCondition element 152
6.45. Example of a PreCondition 153
6.46. Example of the Process element 155
6.47. Example of a Property element 155
6.48. Example of a Reads element 157
6.49. Example of a Resource variable 157
6.50. Example of a Revoke element 158
6.51. Example of a Rights element 160
6.52. Example of a String variable 160
6.53. Example of the Switch element. 161

6.54. Example of a Timer variable 162
6.55. Example of a TimerHandler element 163
6.56. Example of a User variable 164
6.57. Example of a UserTask task 166
6.58. Example of a Validator element 167
6.59. Example of a Variable element 168
6.60. Example of the Workflow element 168
6.61. Example of a Writes element 169
6.62. Listing of the direct publication workflow 170

1. Introduction

The use of the *CoreMedia CMS* covers a range from sites maintained by a single editor to very large portals edited by many users in different roles. The more users are involved in editing, approving and publishing content items, the more difficult it becomes to coordinate tasks and schedules. IT support can greatly enhance productivity because the users do not have to deal with organizational issues.

This goal can be achieved by introducing automated workflows. These workflows do not precisely prescribe how tasks have to be performed, but coordinate and support the timely execution of different tasks by different users with as much flexibility as possible and as necessary. The *CoreMedia Workflow* has a non-restrictive, supportive approach: users are given access to the right resources at the right time via tasks. In contrast to restrictively controlling users, the *CoreMedia Workflow* focuses on progress of the overall business processes.

The workflow manual does not cover all eventualities, but introduces concepts, ideas and the technology. Our manuals undergo permanent revision, and CoreMedia is closely tracking progress in development and experience.

To make our manuals valuable tools in development and implementation of the *CoreMedia CMS*, do not hesitate to contact us for ideas and suggestions via `<documentation@coremedia.com>`.

1.1 Audience

This manual is intended for administrators, who configure and operate the system, and for developers, who want to create own workflow definitions or who want to program own extensions to the workflow system. You will find further information on the usage of the predefined workflows in the Studio User Manual.

1.2 Structure Of The Manual

This manual provides information on the principles of the CoreMedia Workflow, on how to configure and operate the system, write own workflows and on how to develop extensions for the workflow.

- In Chapter 2, *Overview of CoreMedia Workflow* [15] you will find a short introduction into the GUI and components of the Workflow.
- In Chapter 3, *Configuration And Operation* [20] you will learn how to configure and operate the workflow system.
- In Chapter 4, *Customize Workflow Definitions* [32] you will learn how to develop your own workflow definitions. It explains the syntax of relevant XML files.
- In Chapter 5, *Implementing Extensions* [83] you will learn how to implement own extensions of the workflow.
- In Chapter 6, *Reference* [111] you will find a list of the XML elements existing for workflow definitions and some code examples and workflow definition examples.

1.3 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

1.4 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 1.4.1, “Registration” [6] 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 1.4.1, “Registration” [6] describes how to register for the usage of the services.
- Section 1.4.2, “CoreMedia Releases” [7] describes where to find the download of the software.
- Section 1.4.3, “Documentation” [8] describes the CoreMedia documentation. This includes an overview of the manuals and the URL where to find the documentation.
- Section 1.4.4, “CoreMedia Training” [11] describes CoreMedia training. This includes the training calendar,the curriculum and certification information.
- Section 1.4.5, “CoreMedia Support” [11] describes the CoreMedia support.

1.4.1 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 1.4.5, “CoreMedia Support” [11]) by email to request further access depending on your customer, partner or freelancer status so that you can use the CoreMedia services.

1.4.2 CoreMedia Releases

Downloading and Upgrading the Blueprint Workspace

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

https://releases.coremedia.com/cmcc-11

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

NOTE

If you encounter a 404 error then you are probably not logged in at GitHub or do not have sufficient permissions yet. See Section 1.4.1, “Registration” [6] for details about the registration process. If the problems persist, try clearing your browser cache and cookies.

Maven artifacts

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

https://repository.coremedia.com

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

npm packages

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

https://npm.coremedia.io

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

License files

You need license files to run the CoreMedia system. Contact the support (see Section 1.4.5, “CoreMedia Support” [11]) to get your licences.

1.4.3 Documentation

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

https://documentation.coremedia.com

The manuals have the following content and use cases:

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

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

Manual	Audience	Content
Operations Basics Manual	Developers, administrators	This manual describes some overall concepts such as the communication between the components, how to set up secure connections, how to start application.
Search Manual	Developers, architects, administrators	This manual describes the configuration and customization of the *CoreMedia Search Engine* and the two feeder applications: the *Content Feeder* and the *CAE Feeder*.
Site Manager Developer Manual	Developers, architects, administrators	This manual describes the configuration and customization of *Site Manager*, the Java based stand-alone application for administrative tasks. You will learn how to configure the *Site Manager* with property files and XML files and how to develop your own extensions using the *Site Manager API*. The Site Manager is deprecated for editorial work.
Studio Developer Manual	Developers, architects	This manual describes the concepts and extension of *CoreMedia Studio*. You will learn about the underlying concepts, how to use the development environment and how to customize *Studio* to your needs.
Studio User Manual	Editors	This manual describes the usage of *CoreMedia Studio* for editorial and administrative work. It also describes the usage of the *Adaptive Personalization* and *Elastic Social* GUI that are integrated into *Studio*.
Studio Benutzerhandbuch	Editors	The Studio User Manual but in German.
Supported Environments	Developers, architects, administrators	This document lists the third-party environments with which you can use the CoreMedia system, Java versions or operation systems for example.
Unified API Developer Manual	Developers, architects	This manual describes the concepts and usage of the *CoreMedia Unified API*, which is the recommended API for most applications. This includes access to the content repository, the workflow repository and the user repository.

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

Table 1.3. CoreMedia manuals

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

Email: documentation@coremedia.com

1.4.4 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

1.4.5 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 1.4.1, "Registration" [6]. The support email address is:

Email: support@coremedia.com

Create a support request

Support request

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

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

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

Support checklist

To put it in a nutshell, CoreMedia needs:

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

Log files

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

Which Log File?

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

Where do I Find the Log Files?

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

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

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

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

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

1.5 Changelog

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

Section	Version	Description

Table 1.4. Changes

2. Overview of CoreMedia Workflow

NOTE

The *Workflow Server* is installed as a Spring Boot application. So you have to use the standard Spring Boot mechanisms to start and stop the server. The workflow server utilities described in Section 3.5, “Workflow Server Utilities” [25] on the other hand are started with the `cm` utility.

The *CoreMedia Workflow* consists of two components:

- The Workflow Server
  This component is a complete server that communicates with the *Content Management Server* and the *Site Manager*. The *Workflow Server* executes the workflow instances.
- The Client GUI
  The Client GUI is what the user works with: by means of the Client GUI tasks are offered and processed.

See the illustration below for grouping and interaction of the components:

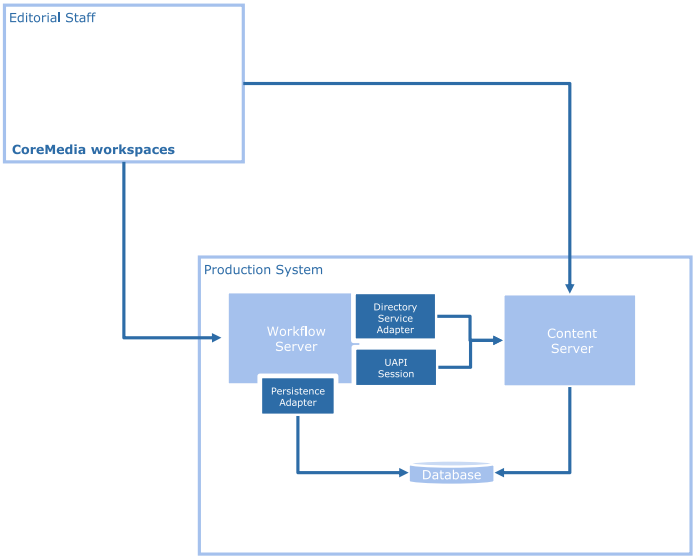

Figure 2.1. CoreMedia architecture with integrated workflow

CoreMedia CMS has two user interfaces for the creation and administration of workflows integrated into *CoreMedia Studio* and the *Site Manager*.

Studio workflow support

You can start and manage workflows from the Control Room in *Studio* and in the Workflow App. For details please consult Section 4.7.2, "Publishing Content" in *Studio User Manual*.

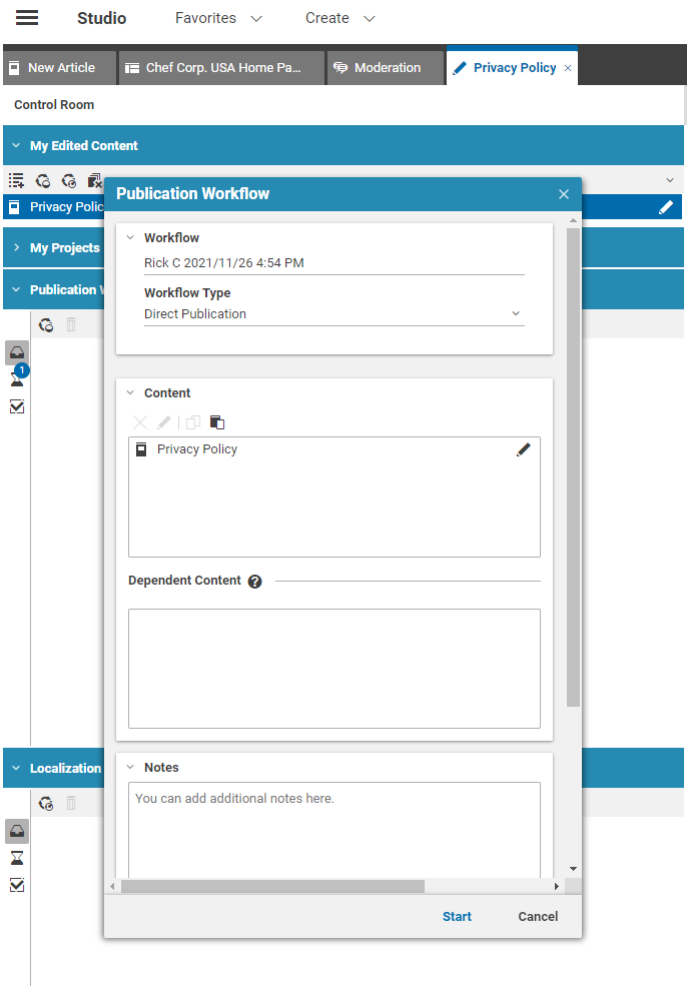

Figure 2.2. Control Room with workflow start window

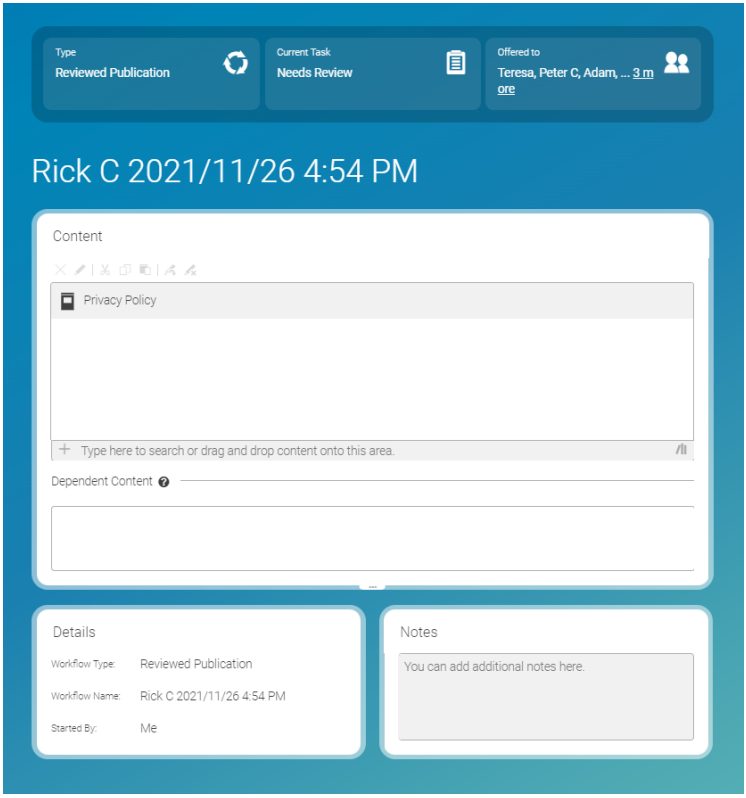

Figure 2.3. Workflow in the Workflow App

Site Manager workflow support

The main workflow window holds three sections below menu (1) and toolbar (2):

- Task and workflow overview (3)
  Tasks or workflows to be edited are displayed here.
- Detail information window (4)
  Here you find all relevant information about the workflow, partly editable. A content item selected here opens in the content window.
- Document window (5)
  Displays a content item selected in the detail information window before.

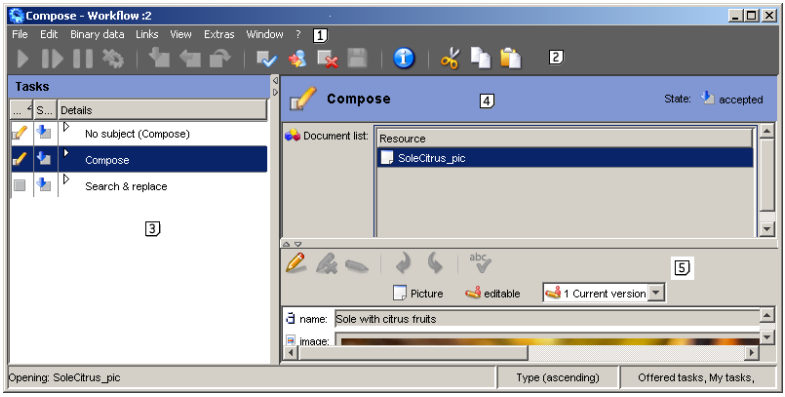

Figure 2.4. Workflow window

For a detailed description of *task and workflow overview[3]* and *detail information[4]* window see Section "Site Manager | Workflow Window" of the Site Manager User Manual.

The *CoreMedia Workflow* comes with three predefined workflows. Two of these workflows deal with the approval and publication of resources, the third workflow handles translation.

- simple-publication
  A user (who needs approval and publish rights) creates a workflow with all necessary resources. The resources will be published (and implicitly approved) by the same user.
- two-step-publication
  A user creates a workflow with all necessary resources. A second user (who needs approval and publish rights) can approve the resources. After the successful completion of this task, the resources will be published automatically.
- Translation Workflow
  Workflow to translate content from the master site to derived sites.
- Synchronization Workflow
  Merges changes from the master site to derived sites.

3. Configuration And Operation

This chapter describes the configuration and operation of *CoreMedia Workflow*.

- Section 3.1, “Starting the Workflow Server” [21] describes how you start the *Workflow Server* and how you can upload workflow definitions.
- Section 3.2, “Uploading Workflows” [22] describes how you can upload your own workflow definitions.
- Section 3.3, “Converting Workflows” [23] describes how you can convert uploaded workflows when classes have changed.
- Section 3.4, “Using JMX Management” [24] describes where you find information for JMX management of the *Workflow Server*.
- Section 3.5, “Workflow Server Utilities” [25] describes how you can use the workflow tools. Small utilities for process overviews or uploading workflow definitions.

3.1 Starting the Workflow Server

Start the Workflow Server

In order to start the *Workflow Server*, start the corresponding Spring Boot application. See https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-running-your-application.html for details.

This will also create groups required by the standard workflows. In order to upload a workflow definition you can use the upload utility (see Section 3.5.4, “Upload” [27]).

The names have to be suffixed with `.xml`. When you would use the *simple-publication* workflow, For example, you have to execute the following command when you want to use the *simple-publication* workflow.

```
cm upload -u admin -p <Password> -n simple-publication.xml
```

3.2 Uploading Workflows

Upload workflows

You can create your own workflow definitions. In order to make these definitions available to the users you need to upload them. For this purpose, you can either use the `upload` utility (see Section 3.5.4, "Upload" [27] for a detailed description) or the menu item **File|Load** in the workflow window of the *Site Manager*.

3.3 Converting Workflows

Convert uploaded workflows

Uploaded workflow definitions are stored in the database as serialized objects. If incompatible changes in classes occurred, you need to convert these workflows. For this purpose, you have to use the `workflowconverter` utility (see Section 3.5.5, “Workflowconverter” [28] for a detailed description).

3.4 Using JMX Management

The *CoreMedia Workflow Server* provides JMX access for management and monitoring. Read the following chapters for further information:

1. In the *CoreMedia Operations Basics Manual* read the *Basics of Operations/JMX Management* chapter with general information about JMX and its configuration in CoreMedia applications.
2. Read Section 6.1.3, “Managed Properties” [113] in order to see the managed properties of the *Workflow Server*.

CAUTION

Note that configuration changes made via JMX are not persisted, that is they are effective only until the next server restart.

3.5 Workflow Server Utilities

There are some tools that help you to work with the *Workflow Server*.

General usage in a Windows 64-bit environment

The server utilities can be started using the `cm64.exe` command in a Windows 64-bit environment with a JVM 64-bit, as described in the Operations Basics.

3.5.1 Start

With the `start` tool you can start new workflows.

Usage of start

```
usage: cm start -u <user> [other options]
         [-pn <name1> <name2> ... | <id1> <id2> ...]
available options:
 -d,--domain                   domain for login
                               (default=<builtin>)
 -pn,--processdefinition-name  names of workflows to start
 -p,--password                 password for login
 -u,--user                     user for login (required)
 -url                          url to connect to
```

The options have the following meaning:

Parameters	Description
`-pn`	The names of the workflows to be started.

Table 3.1. Options of start

`start` creates a new workflow for each specified name or ID. You can start multiple workflows of the same type by specifying the name or the ID several times. Use the `processdefinitions` tool [see Section 3.5.6, “Processdefinitions” [30]] to list the available process definitions. Note that you can only start workflows of process definitions which are enabled.

3.5.2 Download

The `download` tool fetches a process definition and, when existing, an associated JAR from the Workflow Server and writes them into files.

Usage of download

```
usage: cm download [-?] [-d <domain name>] [-f <file>] [-j <jar-file>]
       [-p <password>] -u <user name> [-url <ior url>] [-v]

available options:

 -?,--help                    Print usage information and quit.
 -d,--domain <domain name>    domain for login (default=<builtin>)
 -f,--definition <file>       file name for the workflow definition to
                              download (default=processdefinition-<id>.xml)
 -j,--jar <jar-file>          file name for the workflow jar to download
                              (default=processdefinition-<id>.jar)
 -p,--password <password>     password for login; you will be prompted for
                              password if not given
 -u,--user <user name>        user for login (required)
 -url,--url <ior url>         url to connect to
 -v,--verbose                 enables verbose output
```

The options have the following meaning:

Parameters	Description
`--jar, -j`	The name of the file into which the JAR file should be written. The default is `processdefinition-<name>-<id>.jar`. If there is no custom JAR file associated with the process definition, this option is irrelevant.
`--definition, -f`	The name of the file into which the process definition should be written. The default is `processdefinition-<name>-<id>.xml`.

Table 3.2. Options of download

The downloaded process definition corresponds to the `coremedia-workflow.dtd`.

Example

You can use the Section 3.5.6, "Processdefinitions" [30] tool to get the IDs of all workflow definitions that are uploaded to the Workflow Server. Then use, for example, the following call, where "1" is the ID of one of the uploaded workflow definitions:

```
./cm download -u admin -p admin 1
```

The output will tell you about the process definition identified from input (such as its name and ID) as well as the files written.

The written file(s) can be found by default in the directory of the *download* tool. To change the download location, consider providing a different path via `--definition` parameter and possibly `--jar` parameter, for additional process definition classes download.

3.5.3 Enable

With the `enable` tool you can enable or disable process definitions.

Usage of enable

```
usage: cm enable -u <user> [other options]
          [-n <name1> <name2> ... | -i <name1> <name2> ...]
available options:
 -d,--domain <domain name>   domain for login (default=<builtin>)
 -i,--disable <disable>      names of workflows to disable
 -n,--enable <enable>        names of workflows to enable
 -p,--password <password>    password for login
 -u,--user <user name>       user for login (required)
 -url <ior url>              url to connect to
```

The options have the following meaning:

Parameters	Description
`-i`	Disable the specified workflows.
`-n`	Enable the specified workflows.

Table 3.3. Options of enable

Editors cannot start new workflows from disabled process definitions. Initially uploaded process definitions are enabled.

3.5.4 Upload

With the `upload` tool you can add new process definitions to the workflow server.

Usage of upload

```
usage: cm upload -u <user> [other options]
          [-f <definition path> [-j <jar path>] |
          -n <name1> <name2> ...]
available options:
 -n,--names <names>          names of built-in workflows to upload
 -d,--domain <domain name>   domain for login (default=<builtin>)
 -f,--definition <def>       file name of the workflow definition to
                             upload
 -j,--jar <jar>              file name of the workflow jar to upload
 -p,--password <password>    password for login
```

```
-u,--user <user name>       user for login (required)
-url <ior url>              url to connect to
```

The options have the following meaning:

Parameters	Description
`-n`	Specify workflows by filename (such as `studio-two-step-publication.xml`). This works only for the standard workflows which are delivered with the *CoreMedia CMS*.
`-f`	Specify the XML file which contains the process definition. This option is available only if your *CoreMedia CMS* license includes the usage of custom workflows.
`-j`	Specify a JAR file which contains all resources (esp. custom actions) your workflow needs. You need this option only in combination with the `-f` option for custom workflows. The standard workflows don't need additional resources.

Table 3.4. Options of upload

If a process definition with the name of the uploaded process definition exists already, that definition is superseded by the uploaded definition. Process instances of the old definition run to completion, but additional instances are built using the new definition.

If your process definition references custom Java classes, such classes are preferentially loaded from the JAR files located in the Workflow Server's `lib` directory. Only if a class with a given name is not found there, the server will read the uploaded JAR.

If you upload all custom classes with the process definition and refrain from deploying jars at the Workflow Server, it becomes easier to use updated versions of the classes. In this case the new classes will only be used with the new definition, while the existing definitions and instances use the original versions. Therefore, it is not necessary to run the tool `cm workflowconverter` to resolve possible serialization issues.

3.5.5 Workflowconverter

Uploaded workflow definitions are stored in the database as serialized objects. You can customize workflows by programming own extensions, for example actions, expressions, handlers. So every time, you have made incompatible changes in classes, which are used in already uploaded workflows, you need to convert these workflows. In case of an update of the *CoreMedia Workflow Server*, the workflows have to be converted, too.

Otherwise, object deserialization errors can occur (see Oracle JDK documentation for details).

Active process definitions and inactive process definitions for which there are still running processes can be converted during every Workflow Server start. This automatic conversion can be enabled by the `workflow.server.enable-workflow-converter` flag (see Section 6.1.2, "Configuration of Workflow Server Properties" [112]). Alternatively, this conversion can be executed manually with the workflowconverter tool before starting the workflow server: `cm workflowconverter -c`

The *workflowconverter* utility has the following syntax:

```
cm workflowconverter [ -v | -c [processID]* | -f [pro
cessID]* | -X [processID]* | -r processID jar]
```

The parameters have the following meaning:

Parameter	Description
`-v`	Checks which workflows can not be deserialized and have to be converted.
`-c [processID]*`	If you use `-c` without a process ID parameter, all uploaded workflows will be converted if necessary. If you enter process IDs, only the workflows with the given process IDs will be converted if necessary.
`-f [processID]*`	Like `-c`, but the workflows are converted unconditionally. This is useful, if group IDs used in the serialized workflows have become invalid. Even though this should be an exceptional case, sometimes it happens that external groups (like LDAP groups) vanish and reappear, for example by a temporary misconfiguration of the user provider, and then get a new ID in the CMS. The workflow converter does not detect this, because it is not a matter of deserialization, so you have to enforce the conversion.
`-X [processID]*`	Similar to `-c` the workflow converter converts the uploaded workflows if necessary. If the conversion fails, the workflow process and all corresponding workflow instances are removed from the workflow server.
`-r processID jar`	Replace a custom made JAR file for a workflow with a new version (see Section 3.5.4, "Upload" [27] for the upload of a JAR file).

Table 3.5. Parameters of the workflowconverter utility

To convert the workflows, use the `cm workflowconverter` utility as follows:

1. Make sure that the *CoreMedia Workflow Server* is stopped.

2. Make sure that the *Content Server* to which the *Workflow Server* is attached is running. If necessary, start the *Content Server*.
3. Copy the changed classes (if any) into the appropriate directories.
4. Start the `workflowconverter` utility. Note that the conversion only takes place, if the `-c` or `-X` flag is given.
5. Finally, start the workflow server again.

The *Content Server* must run so that user names and groups names can be resolved while reparsing the workflow definitions.

3.5.6 Processdefinitions

The `processdefinitions` tool shows all uploaded workflow process definitions.

Usage of the process-definitions tool

```
usage: cm processdefinitions -u <user> [other options] [-v]
available options:
 -d,--domain <domain name>   domain for login (default=<builtin>)
 -p,--password <password>    password for login
 -u,--user <user name>       user for login (required)
 -url <ior url>              url to connect to
 -v                          verbose
```

The `processdefinitions` tool has only one additional option:

Parameter	Description
`-v`	Verbose output, prints out additional information

Table 3.6. Options of the processdefinitions tool

The non-verbose output of `processdefinitions` shows the names and IDs of all uploaded process definitions, for example:

```
process definitions:
  id: coremedia:///cap/processdefinition/1,
    name: ThreeStepPublication, enabled: true
  id: coremedia:///cap/processdefinition/6,
    name: SimplePublication, enabled: true
  id: coremedia:///cap/processdefinition/5,
    name: SimplePublication, enabled: false
```

This overview is useful to find out appropriate arguments for other server tools like `start`, `download` or `enable`. The IDs of the process definitions are unique. The names are not unique (see `SimplePublication` in the above example), but only one process definition of a certain name can be enabled at a time.

The verbose output provides detailed information about the process definitions.

3.5.7 Processes

The `processes` utility shows all running workflow processes.

```
usage: cm processes -u <user> [other options] [-v|-v2]
available options:
 -d,--domain <domain name>   domain for login (default=<builtin>)
 -p,--password <password>    password for login
 -u,--user <user name>       user for login (required)
 -url <ior url>              url to connect to
 -v,--verbose                enables verbose output
 -v2,--very-verbose          include task details
```

Usage of the processes utility

The `processes` tool has the following additional options:

Parameter	Description
`-v`	Verbose output, prints out additional information
`-v2`	Even more verbose output, includes task details

Table 3.7. Options of the processes utility

The following sample output of the `processes` utility shows two simple-publication workflows:

```
processes:
  id: coremedia:///cap/process/46, definition: SimplePublication
    (coremedia:///cap/processdefinition/3)
  id: coremedia:///cap/process/26, definition: SimplePublication
    (coremedia:///cap/processdefinition/3)
```

Use the `-v` option or the `dump` tool (see the Content Server Manual) to obtain details about a process.

4. Customize Workflow Definitions

This chapter is about the definition and description of workflows. Definition means that a desired workflow (or business process) is described by means of UML activity diagrams. Then, description means the translation of a UML workflow description into a workflow XML file and probably some Java classes.

- Section 4.1, "Defining Workflows" [33] gives a short survey of how to analyze and define a workflow by means of activity diagrams and the syntactical elements of the XML workflow description language.
- Section 4.2, "Upload Workflow Definitions" [58] describes how you can upload your workflow definition to the workflow server.
- Section 4.3, "Example of Workflow Definition" [59] gives an example on how to define a workflow.
- In Section 6.2, "XML Element Reference" [116], all elements of the XML workflow description language are described as a reference.

NOTE

The BeanParser, that is used to parse the *CoreMedia Workflow* definition allows you to configure all bean properties of the beans that are introduced in the following. Since not all configuration hooks will be explained, it's always a good idea to consult the Javadoc and discover all configuration possibilities.

4.1 Defining Workflows

A useful notation for defining workflows are activity diagrams as specified by the Unified Modeling Language (UML). *CoreMedia Workflow* definitions are based on activity diagrams. They have to be converted to a *CoreMedia CMS* specific XML format for the workflow engine.

After presenting a small example, the notation of activity diagrams is presented and the translation into the *CoreMedia Workflow* XML is shown.

Figure 4.1, "Activity diagram of a simple workflow" [34] describes the following simple workflow with an activity diagram:

A resource is created by one user (an editor) and approved and published by another user (the chief editor). More precisely, the users fill the roles editor and chief editor, respectively. The workflow "edit and publish resource" consists of the following tasks:

- A user of the role editor creates and edits a content item.
- A user of the chief editor role now has to read the resulting content item and judge whether to approve or disapprove it.
- If the content item is approved, the chief editor is requested to publish it.
- If the resource is not approved, the resource has to be changed again by the first user.

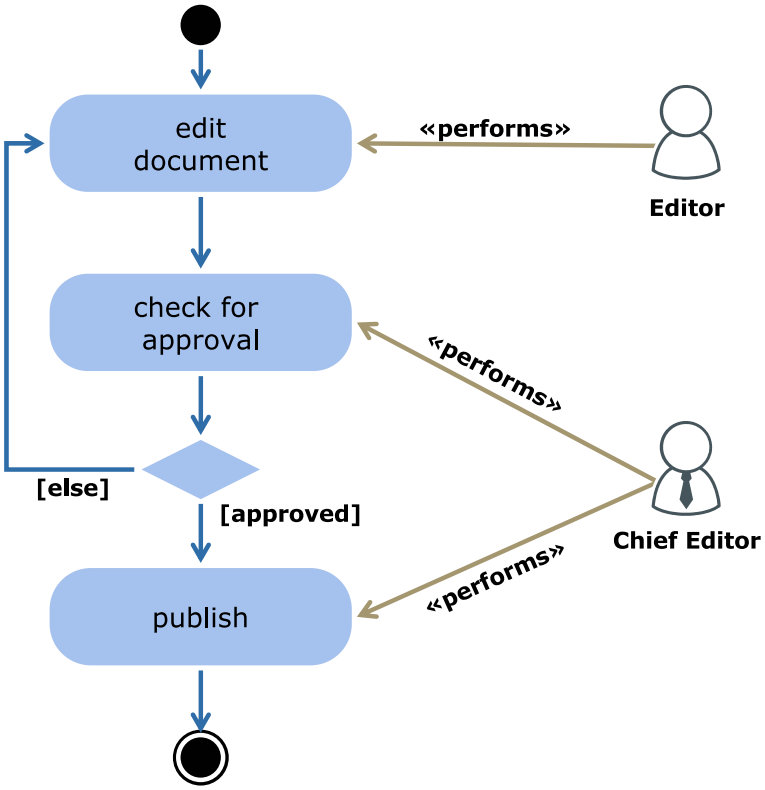

Figure 4.1. Activity diagram of a simple workflow

In the following you will find a description of the UML elements used for the definition of workflows and their mapping to the XML format used by *CoreMedia Workflow*. The details of the XML elements are given in the Section 6.2, "XML Element Reference" [116], the workflow XML reference.

In the *CoreMedia Workflow*, a workflow is defined in a file using XML syntax. A formal description of the syntax of this XML file can be found in the corresponding DTD `coremedia-workflow.dtd` which is located in the zipped `xml` folder of the `lib/cap-schema-bundle-<version>.jar` file. In principle, the workflow file must obey the DTD, but cannot be validated against the DTD in all cases. The reason is that *CoreMedia Workflow* XML can be customized by using your own extensions. It is not possible to capture all future extensions in a static DTD, so the DTD only describes the basis for *CoreMedia Workflow* XML.

In the following sections the important syntactical concepts of the workflow description are explained. The elements of the workflow definition can be seen as elements of a programming language. The following table shows this correlation (not all XML elements are included):

Syntax element of programming language	Respective elements of the workflow definition
variable	`Variable,AggregationVariable`
expression, comparator, function	`Equal, NotEqual, Greater, GreaterEqual, Less, LessEqual, And, Or, Implies, Not, ForAll, Exists, Let, Get, Read, Length, IsEmpty, NotEmpty, IsFolder, IsDocument, IsDocumentVersion`
data type	value classes: `Blob, Boolean, Content, ContentType, Date, Document, Folder, Group, Integer, String, Timer, User`
flow control	`Fork, Join, If, Choice, Switch, Case`
precondition, postcondition	`PreCondition, PostCondition`
procedure	Action, EntryAction, ExitAction
sub program	`ForkSubprocess, JoinSubprocess`

Table 4.1. Workflow elements vs. programming language

4.1.1 The BeanParser

The XML files used to configure *CoreMedia CMS* components are processed by the *BeanParser*, which is a basic part of the system. As such, it is used to

- read the license,
- define content types and workflows,
- configure *Site Manager*.

The *BeanParser* processes the XML files as follows:

- For each XML element it tries to instantiate an object of a class, which is determined by a factory or via the `class` attribute. The object is created via Java Reflection and a zero-argument constructor.
- If the XML element occurs inside another XML element, it tries to set the object created by the inner element on the object created by the outer element. For this, it calls a setter method and passes the object. The setter method may be named `set<Element Name>()`, `add<ElementName>()` or simply `set()` or `add().`
- For each attribute of an element it calls a setter method on the object that was created when parsing the element start tag. The setter method may be named `set<AttributeName>()`, `add<AttributeName>()` or simply `set()` or `add().`

Example:

Assume the following XML file:

```
<FirstElement class="com.example.FirstElement" attribute1="Ho">
    <SecondElement class="com.example.SecondElement"
      attribute="Hi"/>
</FirstElement>
```

Example 4.1. Example of a BeanParser XML file

The BeanParser will execute the following steps:

1. Create an instance of class `com.example.FirstElement`.
2. Call `setAttribute1("Ho")` on that instance.
3. Create an instance of class `com.example.SecondElement`.
4. Call `setAttribute("Hi")` on that second instance.
5. Call `firstElement.setSecondElement(secondElement)`, that is, set the object created in step 3 on the object created in step 1.

Advanced features:

The class attribute has a special meaning as it determines the name of the class to instantiate objects from. For this attribute, no setter methods has to be defined inside the class.

The *BeanParser* works without an XML Document Type Definition (DTD), but in connection with a DTD, it makes use of `ID` and `IDREF` feature of the XML parsers. The object, that has been created by the element with the `IDREF` attribute, is substituted by the object that is defined the corresponding `ID` attribute. Again, no setter methods have to be defined inside the involved classes.

4.1.2 Elements of Activity Diagrams

The following Unified Modeling Language (UML) activity diagram symbols may be translated in elements of CoreMedia Workflow definitions like this:

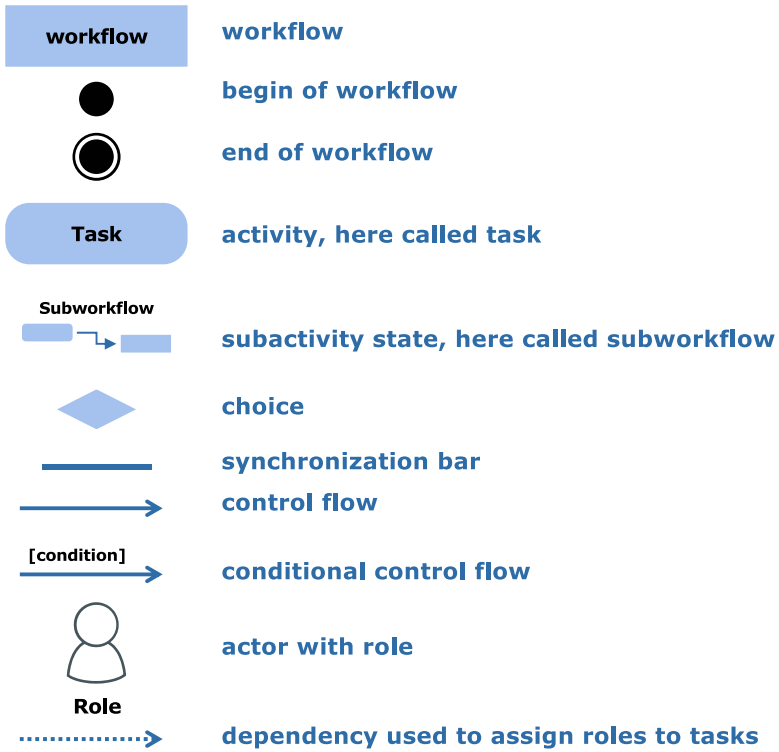

Figure 4.2. Elements of activity diagrams

- Begin of workflow

  This symbol marks the begin of the workflow. For this node, only outgoing transitions are allowed.

- End of workflow

  This symbol marks the end of the workflow. For this node, only incoming transitions are allowed.

- Activity / Task

  This symbol denotes an activity, which is called a task in the *CoreMedia Workflow*.

- Sub activity state / Subworkflow

  A separate workflow can be called from a task of another workflow. Thus, the separate workflow can be called a subworkflow task.

- Decision node / Branch / Choice

  This symbol stands for a node where the control flow branches, depending on a decision. In a workflow definition, a decision-based branch is usually called an `If` task.

- Synchronization bar

  This symbol is used for splitting or synchronizing the control flow. In the splitting case the control flow *forks* in more than one followup task. In the synchronization case, multiple tasks executed in parallel are *joined* together.

- Control Flow

  Transitions specify the control flow from a node to its successor. Nodes can be any of begin or end of workflow, task, choice and synchronization bar.

- Conditional Control Flow

  Transitions can be inscribed with a condition in square brackets. Such edges are usually used as outgoing edges of a decision node (called a Choice task).

- Actor with Role

  An actor is used in UML to denote a participant in a use case. CoreMedia introduces actors to specify rights of users of certain groups (roles) for user tasks.

- Dependency used to assign Roles to Tasks

  A dashed arrow denotes a UML dependency. CoreMedia uses special dependencies to connect roles (see above) with user tasks in order to assign rights.

4.1.3 Processes

Each workflow definition describes one process. A process can take several states as shown in Figure 4.3, “State diagram of a process” [39].

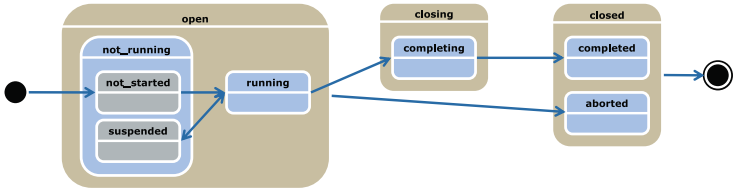

Figure 4.3. State diagram of a process

There are five operations which can be applied to a process, depending on its state:

- create a process

  If a process is created, the variables of the process are initialized. The *Site Manager* opens a form for this, where the user can enter the values of the workflow variables depending on the tasks client view. The workflow is in the state not_started, so no task is activated yet.

- start a process

  If a process is started, it switches to the state running and starts executing with its start task.

- suspend a process

  A running process may be suspended by an authorized user. The further execution of all tasks is paused until the process is resumed again. Thus, tasks can neither be accepted nor delegated or completed if a process is in state suspended.

- resume a process

  If a process was suspended it may be resumed by an authorized user and continues where it had paused before.

- abort a process

  A process may be aborted by an authorized user in any substate of the state open. Aborting a process means deleting it. The actions which took place as part of the workflow so far are not rolled back, so, for example, approved resources remain approved.

4.1.4 Tasks

Tasks are the main building blocks of workflows. There are `UserTasks` and `AutomatedTasks`, as well as auxiliary control flow tasks like `If`, `Choice`, `Fork`, `Join`, Switch, `ForkSubprocess` and `JoinSubprocess`. All mentioned different types of tasks can be defined using the *CoreMedia Workflow* XML format.

Like a process definition is a template for concrete process instances, a task definition is a template for specific task instances. While being executed by the workflow engine, a task instance can take several states as shown in the state diagram in Figure 4.4, "State diagram of a task" [40].

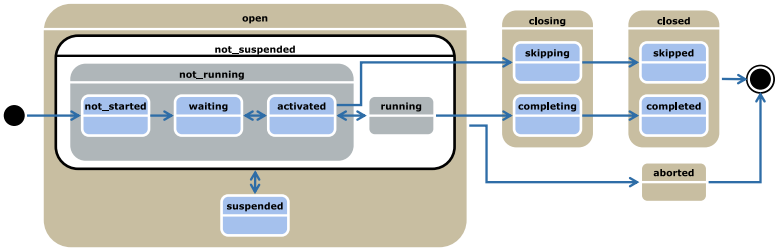

Figure 4.4. State diagram of a task

Different operations are possible or mandatory during the execution of a task instance to enter or leave the different states. A rights policy defines which operations are allowed to a user. You can configure this rights policy. The following table shows how to leave or enter the different task states. A user task always requires its performing user to have the appropriate rights to perform an action which changes the state of a task.

State	Enter State	Leave State
not_started	This is the starting state of all task instances after process creation.	The state is left automatically after the workflow server has entered the task.
waiting	This state is entered automatically, after the task is reached by the workflow server. It is also entered from the activated state if instance context changes have been made. So the guards are recalculated.	The state is left automatically when the task is ready for activation, that is if the following conditions have been fulfilled: • The control flow of the workflow has reached the task. • The optional guard specified in the task definition evaluates to "true".

State	Enter State	Leave State
activated	This state is entered automatically, after the waiting state has been left. Assigning a task If this state has been entered, you can nominate a user or group for this task via `Task.assignTo()`. So only these users will see the task in their task list (if they have the appropriate rights). This operation will not change the state of the task. Rejecting a task A user can also reject the task via `Task.reject()`, so it will not be offered to him anymore. If all appropriate users have rejected the task, it will be offered again to all these users (this is the default performers policy). Canceling a task The state activated is also entered if a task was accepted by a user and then canceled by this user via `Task.cancel()`. All changes made so far by the user are saved, but the task is offered again to all appropriate users like before it was accepted.	A user task must be accepted by the user in order to leave the state 'activated' via `Task.accept()`. Then preconditions and entry actions are performed. After successfully running the actions, the task is performed by the user and no more available to other users. Another way to leave the state is to skip the task via `Task.skip()`, switching to the state 'skipping'. A fallback to waiting is possible.
suspended	This state can only be entered via `Process.suspend()` which suspends the workflow. All task are withdrawn from the task list (GUI specific).	This state can be left via `Task.resume()`. The workflow will restart at the same task where it was suspended.
running	If an automated task has been activated it automatically leaves the state 'activated' and changes to 'running'. A user task must be accepted by the user via `Task.accept()` in order to enter the state 'running'. The task is then performed by the user and is no more available to other users.	An automated task leaves the state 'running' and enters one of the states 'completed' (via `Task.complete()`) and 'aborted' depending on the success of the actions and `preconditions and Postconditions` performed.

State	Enter State	Leave State
		A user task can leave the state 'running' and enter one of the states 'waiting', 'completed' [via 'completing'] and 'aborted'. 'Activated' is reached, when the user cancels the task via `Task.cancel()`. All changes made so far by the user are saved, but the task is offered again to all appropriate users. 'Completed' is reached, when the task is completed via `Task.complete()` and the exit actions execute successfully and the post-conditions evaluate to "true". 'Aborted' is reached, when one of the exit actions and postconditions fails.
skipping	Intermediate state.	Intermediate state.
skipped	This state is entered if the task has been skipped by a user via `Task.skip()`. The process continues with the following task.	This state can only be left, when the flow of operation returns to the task. That is, there is a loop in the workflow definition which returns to the task.
completing	Intermediate state.	Intermediate state.
completed	An automated task enters this state when all actions have been successfully performed and the preconditions and postconditions have been evaluated to "true". A user task enters this state when the user completes the task, the exit actions have been successfully executed and the post-conditions evaluated to "true".	This state can only be left, when the flow of operation returns to the task. That is, when there is a loop in the workflow definition, which returns to the task.
aborted	This state is entered if the process is aborted via `Process.abort()`.	Final state.

State	Enter State	Leave State
escalated	This state is entered automatically when an error occurs, if, for example, a postcondition fails. The previous user is still the performer if there was a performer (depends on the former state).	You can leave this state retrying the task via `Task.retry()`. This will retry the last operation, which has failed: for example, if a precondition has failed, the task will restart with the state transition from activated to running, or if a postcondition has failed the task will restart with the state transition from running to completing and repeating all actions.

Table 4.2. Status of Tasks

4.1.4.1 Common Features of All Tasks

User tasks, automated tasks and control flow tasks have many features in common. They are presented in this section.

The most important common feature of all tasks is that each must be assigned a *name*, which identifies it uniquely within the process. The name has to be an identifier according to the usual XML rules for names [`NMTOKEN`].

Since the name is only a symbolic identifier, a task may also contain a *description*. Although any task may contain a description, it makes most sense for user tasks. If you want to provide localized versions of descriptions, put an identifier instead of the text itself into the description attribute in the workflow definition. In a resource bundle [`.properties` file, see the editor configuration in the *Administrator Manual*], you can map the identifier to the localized text, depending on the chosen locale.

Tasks that finish a workflow process are declared *final*. There has to be at least one task in a process definition, which is declared *final*. Only user tasks and automated tasks can be declared *final*.

A task refers its *successor* by name. Each task must either have at least one successor or be final. Forking tasks may have multiple successors. Joining task may have multiple predecessors.

Variables in the task scope define the local state of a task instance. However, task variables do not have restricted visibility. A variable in a task may be referred to from other tasks by prefixing the variable name with the task name and a dot. A variable defined in the process can be referred to by simply using its name without a prefix. For the definition of variables, see section Section 4.1.6, “Workflow Variables” [52].

A *guard* defines an expression that delays activation of a user or automated task until the expression evaluates to `true`. The expression is re-evaluated each time the state of process- or task instances changes or the content, name, or place of referred resources in the *Content Management Server* changes.

A *precondition* defines requirements which have to be fulfilled before the task itself is executed. A *postcondition* defines requirements which will be evaluated after the exit action has been executed. If more than one precondition or postcondition is provided, then the conditions are evaluated in the order specified. The result of such an evaluation operation is equivalent to define an `And` expression with an ordered set of conditions.

Note that violating a condition is considered an error. If you want to delay execution until a condition is true, use a guard. If you want to check a condition and allow correction of wrong data entry within a user task, use a *validator* (see below).

4.1.4.2 User Tasks

The most common kind of task is the user task, which is executed by participants of the workflow.

When defining a user task, first consider the rule that selects which users to offer the task. Usually, the appropriate users are selected from their groups. For each group, a list of rights on the task is given, where *accept* is the most important one for user tasks. For special requirements, you can implement your own business logic in a `WfPerformersPolicy`.

For a user task a *client* view has to be given. A client defines a view on the variables of the workflow that may be read and/or modified. For resource variables, you can additionally determine whether the referred content may be editable.

To make workflow more convenient for the participants, user task's actions have access to various functions of the *Site Manager*. While an automated task can change resources (check out a content item for instance), a user task can even open a content view or start a publication with graphical feedback. For a list of possible actions, see Section 6.2, “XML Element Reference” [116] and Section 4.4.1, “Predefined Action Classes” [66].

Validators (see Section 4.1.7.4, “Validators” [54]) have a special feature in the context of a client view. If a validator fails and provides a description, it is displayed as an error message in a client view. Like task descriptions, validator error messages may be localized (see Section 4.1.4.1, “Common Features of All Tasks” [43]).

4.1.4.3 Automated Tasks

Automated tasks usually consist of an action sequence, an optional guard and preconditions or postconditions. They are executed by the workflow server.

A guard is used to activate the automated task depending on some condition. For details about when conditions are reevaluated, see Section 4.1.4.1, “Common Features of All Tasks” [43].

Actions within an automated task usually modify workflow variables, manipulate resources, perform calculations and/or access external systems. However, they may not access the Client GUI, since they are not executed on the client side, as the workflow server uses a direct connection to the *Content Management Server* for automated tasks. If you want GUI interaction, you have to use a user task.

Several actions which are to be executed sequentially should be given as an action sequence within a single automated task, not as a sequence of automated tasks. This is easier to understand and will be executed faster. The general rule of identifying different tasks by potentially different users can also be applied here, if you consider automated tasks as being accepted and performed by a "robot".

An automated task completes as soon as all it actions have been executed and its optional postcondition is evaluated. If an action raises an exception or the postcondition evaluates to false, the automated task is aborted. The reason that led to the error should be fixed before the task is retried. As a last resort, the whole workflow can be aborted.

4.1.5 Flow Control

The control flow between the tasks can be defined by Unified Modeling Language (UML) activity diagrams using the following schemes:

Sequence

When tasks are arranged in a sequence, a successor task may start just after its predecessor task has been completed. Since the workflow server uses a pull approach, the task does not run immediately after the predecessor has been completed, as this is delayed until a user accepts it (except for automated tasks). The very first task of a process always runs immediately.

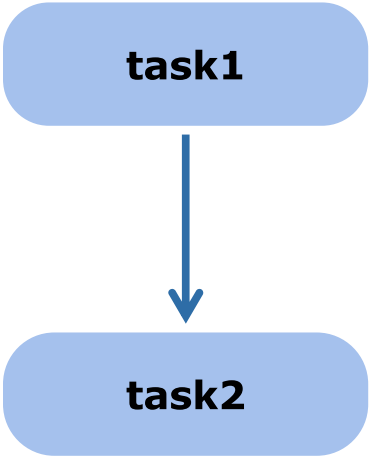

Figure 4.5. Example of a sequence diagram

Respective elements and attributes of the workflow definition: `successor` attribute of all task XML elements.

Example:

```
<UserTask name="task1" successor="task2">
  .
  .
</UserTask>
<UserTask name="task2">
  .
  .
</UserTask>
```

Example 4.2. Example listing of a sequence

Choice

Based upon a condition, the control flow continues at exactly one of two or more followup tasks. This is also called an or-split, since only one task will be performed.

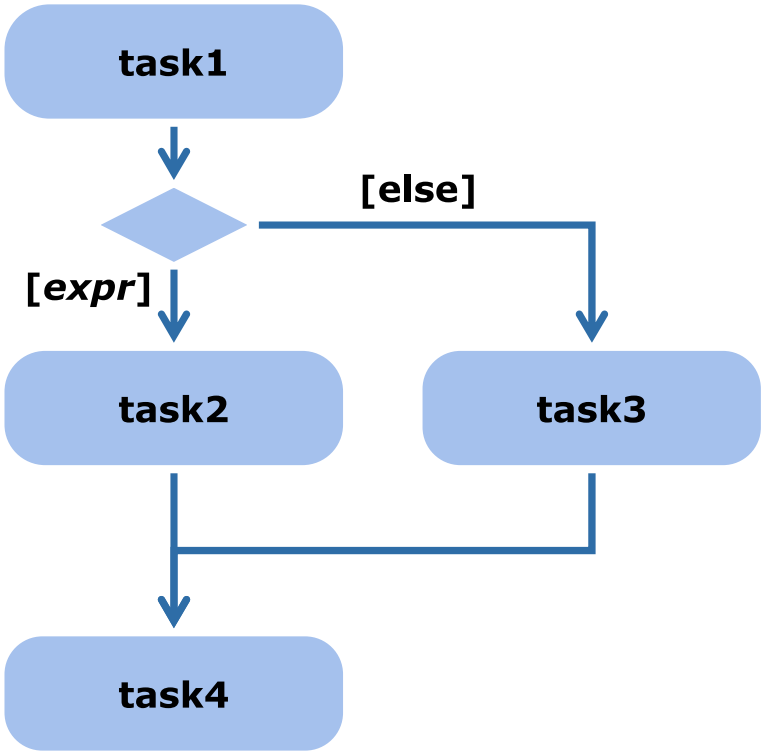

Figure 4.6. Example of a choice diagram

Respective elements of the workflow definition: `<If>[<Condition>`, `<Then>`, `<Else>], <Switch>[<Case>]`

Example:

```
<UserTask name="task1" successor="choice">
  <!-- Code -->
</UserTask>
<If name="choice">
  <Condition>
    <!-- expr -->
  </Condition>
  <Then successor="task2"/>
  <Else successor="task3"/>
</If>
<UserTask name="task2" successor="task4">
  <!-- Code -->
</UserTask>
<UserTask name="task3" successor="task4">
```

```
    <!-- Code -->
  </UserTask>
```

Example 4.3. Example listing of a choice

Implicit Choice

If a choice is used (see above), the workflow engine decides where to continue the control flow based on an explicit expression. An implicit choice lets the workflow users decide where to continue, simply by offering two or more user tasks, from which only one may be accepted. As soon as this one task is accepted, the other task(s) is/are automatically withdrawn and may not be accepted anymore. The notation is to draw two or more outgoing control flow edges *without* a condition inscription. The decision node may be omitted, as in the example diagram.

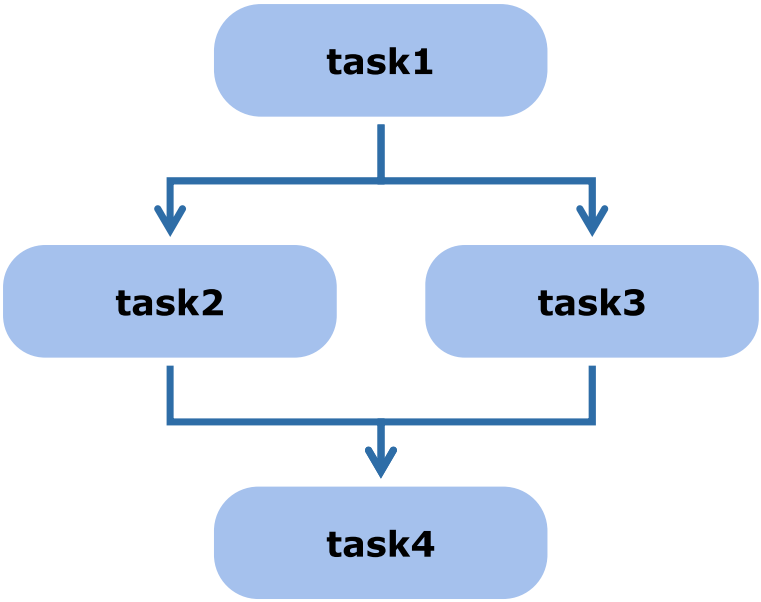

Figure 4.7. Example of an implicit choice

Respective elements of the workflow definition: `<Choice>[<Successor>]`

Example:

```
<UserTask name="task1" successor="implicitChoice">
  <!-- Code -->
</UserTask>
```

```
<Choice name="implicitChoice">
  <Successor name="task2"/>
  <Successor name="task3"/>
</Choice>
<UserTask name="task2" successor="task4">
  <!-- Code -->
</UserTask>
<UserTask name="task3" successor="task4">
  <!-- Code -->
</UserTask>
```

Example 4.4. Example listing of an implicit choice

Loop

The loop is a special case of a choice, where one of the successor tasks is a predecessor of the current task. Thus, a task may be repeatedly performed.

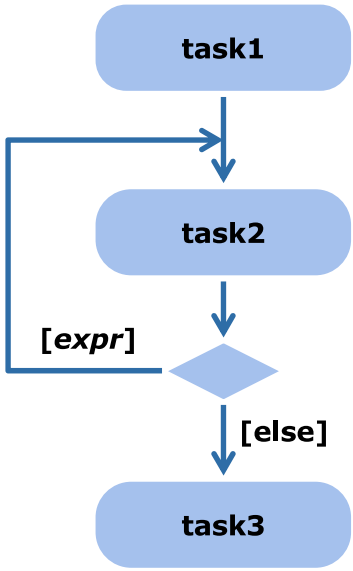

Figure 4.8. Example of a loop

Respective elements of the workflow definition: `<If>[<Condition>`, `<Then>`, `<Else>]`

Example:

```
<UserTask name="task2" successor="loopCondition">
  <!-- Code -->
</UserTask>
```

```
<If name="loopCondition">
  <Condition>
    <!-- expr -->
  </Condition>
  <Then successor="task2"/>
  <Else successor="task3"/>
</If>
<UserTask name="task3">
  <!-- Code -->
</UserTask>
```

Example 4.5. Example listing of a loop

Concurrency/Parallel Execution

After the task before the synchronization bar is completed, *all* followup tasks are activated. This is called a *fork* of the control flow. The resynchronization of parallel executing tasks is called a *join*. This is also called an and-split, since all followup tasks are performed. Each fork must be matched by exactly one join that joins all previously forked tasks.

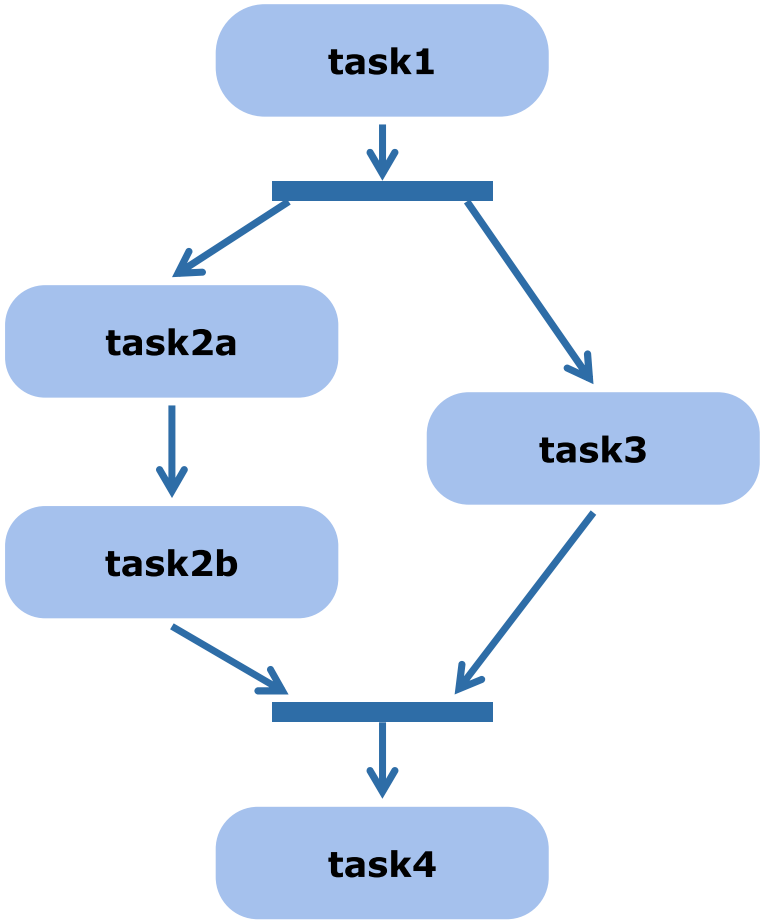

Figure 4.9. Example of a concurrency diagram

Respective elements of the workflow definition: `<Fork>[,<Join>]`

Example:

```
<Fork name="fork">
  <Successor name="task2a"/>
  <Successor name="task3"/>
</Fork>
<UserTask name="task2a" successor="task2b">
  <!-- Code -->
</UserTask>
```

```
<UserTask name="task2b" successor="join">
  <!-- Code -->
</UserTask>
<UserTask name="task3" successor="join">
  <!-- Code -->
</UserTask>
<Join name="join" successor="task4">
  <Predecessor name="task2b"/>
  <Predecessor name="task3"/>
</Join>
```

Example 4.6. Example listing of concurrency

4.1.6 Workflow Variables

Workflow variables are declared within a workflow definition. They contain references to resources or other values. There are single-valued variables (*atomic variables*) and list-valued variables (*aggregation variables)* of a given type. Workflow variables are the main connection between the workflow server and the *Content Management Server*. By assigning resources to workflow variables, these resources may easily be accessed in later tasks of the same workflow instance. Workflow variables provide the context in which a task has to be carried out. If a workflow variable is defined in a task, it can be accessed by another task using the dot syntax `name-of-task.name-of-variable`.

Each Variable is *typed*. A variable can only be bound to a value of the corresponding type or subtype. There is a fixed amount of types for workflow variables:

- basic value types: Boolean, blob, Integer, String, Date, Timer
- CoreMedia resource related types: Content[Folder, Document], ContentType
- CoreMedia-user-manager-related types: Group, User

If a variable should be shown or edited in the client GUI, it must be mentioned in a client view (see Section 4.1.4.2, “User Tasks” [44]). Please note, that for aggregation variables there exists only an editor for resource variables. So by default, you can only edit resource aggregation variables in the variable view.

4.1.7 Expressions

Expressions are used to specify conditions in validators, guards, preconditions or post-conditions and to guard action execution.

Simple expressions return constants, access variables, read properties of resources, or the like. More complex expressions can be build up from the simple ones by comparison operators, logical connectives, logical quantors, and so on. It is possible to specify

custom expressions via `WfExpression`, if the predefined expressions are not sufficient.

4.1.7.1 Conditions

Conditions are used to define how processing should proceed. They are expressions which evaluate to a Boolean value. There meaning depends on the action in which the expression is specified.

- Specified in an `Action`, `EntryAction`, or `ExitAction`, a condition determines whether the action should be executed or skipped.
- Specified in an `If` element, a condition determines which branch should be taken.
- Specified in a `Case` element, a condition determines when a branch should be taken.
- Specified in a `Precondition` or `Postcondition` element, a condition determines whether constraints are fulfilled.
- Specified in a `Guard` element, a condition determines when a task is activated.

4.1.7.2 Preconditions and Postconditions

Preconditions and postconditions are Boolean expressions that act as assertions which are evaluated when entering or leaving a task. A task can contain any number of preconditions and Postconditions.

Preconditions and postconditions help the developer to determine error conditions that can not be handled by the normal workflow. If preconditions or postconditions evaluates to "false", the task is escalated. It may be manually restarted when the error condition has been resolved.

4.1.7.3 Guards

Guards are Boolean expressions that must evaluate to "true" before the task is activated. The expression may be based on the current values of workflow variables, on resources in the `Content Management Server` or on external resources. A possible use of guards is to determine the resources that are required for the task. The task then is deactivated until all resources are freely available. Thus, the workflow suspends execution until the guard is fulfilled.

In Example 4.7, “Example of a Guard” [54] you see a guard that checks whether the property `isCheckedOut_` of the resource contained in the variable "document" (`variable="document"`) is set to false (the stored value is negated by `Not`). That is, the task continues when the content item is checked in.

```
<Guard>
  <Not><Read variable="document" property="isCheckedOut_"/></Not>
</Guard>
```

Example 4.7. Example of a Guard

4.1.7.4 Validators

Validators are Boolean expressions that ensure that the variables that may be modified via a client view satisfy certain constraints. For example, they can ensure that values stay within a predefined range or that certain variable values have been entered at all. If a validator expression evaluates to "false", a message is presented to the user who performed the task, so that the error condition may be resolved by continuing work on the task.

Validators can be specified to verify each "save" of variables. When defining the validator, set `validatedOnSave="true"`. In this case, you will get an error message if you try to save and the validator expression evaluates to "false".

4.1.8 Actions

Actions are used to automate or semi automate tasks. To do so, arbitrary actions can be invoked at the start or end of a user task, during an automated task, or at the very end after a process was completed or aborted.

User Task

- Element `<EntryAction>`
  This kind of action is invoked after the task is accepted, but before the user starts to work on the task. Typical start actions are the initialization of resources.
- Element `<ExitAction>`
  These actions are invoked after the task has been completed by the user and after the postconditions are checked, but before the workflow continues. A typical exit action might complete the users work and set some calculated properties, approve resources in the name of a user, show up a publication window etc.

Automated Task

- Element `<Action>`
  An automated task is not performed by a user. The task duration is exactly the duration of the invoked actions plus preconditions and conditions. If preconditions or postconditions are violated, the task is aborted.

Final Action

- Element `<FinalAction>`
  Final actions are invoked after the process completed successfully or was aborted. Typical use cases are cleaning up used resources or archiving data from the process before it gets deleted. Compared to actions running in a task, these actions use a different interface and cannot modify the process itself anymore.

User task actions are executed with the rights and on behalf of the user who accepted the task. Actions in automated tasks and final actions run with the *Workflow Server's* "user" account at the *Content Management Server*.

4.1.9 Rights

Rights determine which operations user and groups may perform on processes and tasks. A rights policy is used to decide whether a concrete user may perform an operation on a workflow object.

The rights policy, which is used by the *CoreMedia Workflow Server* is configurable. By default, the `ACLRightsPolicy` is used. It determines the rights based on Access Control Lists (ACL) for each workflow object. The ACLs are defined by granting and revoking rights for a user or a group. The following rules apply:

- Rights for a user are calculated from concrete rights defined for a user and the rights from all the groups the user is a member of. Users and groups may be specified directly or by storing them into a specified variable.
- A revoke precedes a grant.
- Rights for users and groups read from a variable precede rights granted to a fixed user. These rights again precede rights for a fixed group.

For example:

```
<Rights>
  <Grant user="admin" rights="create,start,suspend,resume,abort"/>
  <Grant group="composer" rights="create,start"/>
  <Grant group="suspender" rights="suspend,resume"/>
</Rights>
```

Example 4.8. Example of the ACL for a process

This ACL for a process gives the user `admin` the right to create, start, suspend, resume and abort the process instance. Whether the user `admin` is in the groups `composer` or `suspender` is not relevant. Users, that are member of the `composer` group, may create and start process instances. If a `composer` group member, is in the group `suspender`, too, he may suspend and resume, the process instance, too. Users that

are not member of the `composer` or `suspender` group have no rights on the process instance.

4.1.10 Subworkflows

Basically a subworkflow is an ordinary workflow started by the task `<ForkSubprocess>` within another workflow. The subworkflow may be passed parameters via the subelements of the `<Parameters>` element.

A subworkflow is always started as a separate process, while the main process continues. There are two different ways in which a subworkflow may be started:

- Synchronously via `<ForkSubprocess detached="false">`
  If the main workflow is suspended, resumed or aborted, the subworkflow is suspended, resumed or aborted, too, but it may finish without affecting the subworkflow.
  The main workflow may wait for the subprocess to complete or to be aborted via the `<JoinSubprocess>` task. Note, that it is not possible to loop (see Section Section 4.1.5, "Flow Control" [45]) a <ForkSubprocess> and join all subprocesses afterwards. Use recursion in this case so that each subworkflow starts exactly one subworkflow.
- Asynchronously via `<ForkSubprocess detached="true">` or simply `<ForkSubprocess>`
  If the main workflow stops, the subworkflow is not affected. Since they are not connected, there is no possibility for the main workflow to wait for the subworkflow to finish.

4.1.11 Timers

Timers can be used to define time spans or moments in the execution of a workflow. For example, the time available for a user task to be accepted. The *CoreMedia Workflow* supports timers which can be initialized with relative (the timeout value is added to current time giving the expiration time) or absolute values.

By default, two timers are attached to UserTask definitions and one to the Process definition which can be set via the following attributes:

- `defaultTimeout`: This is a relative timer which is activated when a process instance is started or a task instance is activated.
- `defaultOfferTimeout`: This is a relative timer which is activated at the first offer of the task after the activation. This means if the task is first accepted by a user, then canceled by the user and again offered to the other users the timer will not be restarted. In contrast, if the task is used in a loop, the timer will be restarted each time the loop reaches this task.

If these timers expire, they will add a warning message to their process or task instance. You might use one of the predefined TimerHandlers (using the `<TimerHandler>` tag) to react differently if timers expire (see Section 4.4.3, "Predefined TimerHandler Classes" [80]). The handler *must* be defined in the same location, that is the process or task definition, where its associated timer variable is defined.

In addition, you may define custom timers: At first you have to define a variable of type `Timer`. Using the attribute `relative` you can define whether the timer is a relative ("true") or absolute one ("false"). Defining an absolute value in the workflow definition might not make much sense, it is more useful in the client GUI.

The timer can be started and stopped using the actions `EnableTimer` and `DisableTimer` (see Section 4.4.1, "Predefined Action Classes" [66]). Using the expressions `IsExpired` or `IsEnabled`, you can check whether your timer has been expired or is enabled and running.

Note that

- Timer values have no identity, they are bound to their variables.
- Aggregations of timers are not supported.

The following example shows an automated task which defines and enables a timer variable. The succeeding user task waits until the timer expires:

```
<AutomatedTask name="StartTimer" description="SimplyStart"
  successor="Wait">
    <Variable name="waiting" type="Timer">
        <Timer value="100"/>
    </Variable>
    <Action class="EnableTimer" timerVariable="waiting"/>
</AutomatedTask>
<UserTask name="Wait" successor="Next">
    <Guard>
        <IsExpired variable="StartTimer.waiting">
    </Guard>
    <!-- Code -->
</UserTask>
```

Example 4.9. Example of a self-defined timer which expires after 100 seconds

4.2 Upload Workflow Definitions

In order to make your workflow definitions available to the users you need to upload them. For this purpose, you can either use the `upload` utility or the menu item **File|Load** in the workflow window of the *Site Manager*.

If you upload a workflow definition with a name (the attribute `name` of the `Process` tag, not the file name) which has already been loaded, then a new instance of the workflow will be created and the old workflow instance will be disabled. So, running workflows will still use the old workflow definition, but you cannot create new workflows from the old definition. This may be a problem if you are using subworkflows.

To manually enable or disable workflow definition, you can use the `enable` utility (see Section 3.5.3, “Enable” [27] for a detailed description).

4.3 Example of Workflow Definition

Here the definition of a workflow is shown by means of the Studio simple publication workflow.

The routine is: An editor creates and edits a change set in the compose task. After completing the compose task, the resources will be approved and published automatically (only if the `forceUser` action succeeds). In Example 6.62, "Listing of the direct publication workflow" [170] you find the complete XML definition of this workflow.

The workflow definition consists of multiple blocks:

- The general definitions of the workflow
- An automated task AssignComposer
- Am `user` task Compose
- An `if` task CheckEmptyChangeset
- A user task Publish
- An `if` task CheckPublication
- An automated task Finish
- A FinalAction

These blocks will be illustrated in detail.

General definitions

```
1:   <?xml version="1.0" encoding="iso-8859-1"?>
2:
3:   <Workflow>
4:    <Process name="StudioSimplePublication"
                description="studio-simple-publication"
               startTask="AssignUser">
5:
6:      <Rights>
7:        <Grant group="administratoren"
                 rights="read, create, start, suspend, resume,
                 abort"/>
8:        <Grant group="composer-role"
                  rights="read, create, start, suspend, resume,
                  abort"/>
9:        <Grant group="approver-role" rights="read"/>
10:       <Grant group="publisher-role" rights="read"/>
11:     </Rights>
12:
13:     <Variable name="subject" type="String"/>
14:     <Variable name="comment" type="String"/>
15:     <AggregationVariable name="changeSet" type="Resource"/>
16:     <AggregationVariable name="comments" type="String"/>
17:
18:     <Variable name="changeSetLockedInStudio" type="Boolean">
19;       <Boolean value="true"/>
20:     </Variable>
21:     <Variable name="publicationSuccessful" type="Boolean">
22:       <Boolean value="false"/>
23:     </Variable>
24:     <AggregationVariable name="publicationResultResources"
                             type="Resource"/>
```

```
25:     <AggregationVariable name="publicationResultCodes"
                             type="Integer"/>
26:     <AggregationVariable name="publicationResultVersions"
                             type="Integer"/>
27:     <AggregationVariable name="publicationResultParams"
                             type="String"/>
28:
29:     <InitialAssignment>
30:       <Writes variable="subject"/>
31:       <Writes variable="comment"/>
32:       <Writes variable="changeSet"/>
33:       <Writes variable="comments"/>
34:     </InitialAssignment>
35:
36:     <Assignment>
37:       <Reads variable="subject"/>
38:       <Reads variable="comment"/>
39:       <Reads variable="changeSet"/>
40:       <Reads variable="comments"/>
41:     </Assignment>
42:       .
43:       .
44:       .
45:   </Process>
46: </Workflow>
```

Example 4.10. General definitions of the workflow

In the general definitions the workflow itself is described.

Line 4 - 5: The process is named 'SimplePublication'. The localized name is displayed in the GUI when selecting a workflow. The first task that is executed after the workflow start is the task 'AssignComposer'.
Line 6 - 11: The rights (see Section 4.1.9, “Rights” [55]) concerning the workflow are assigned to users and groups. The user admin can create, start, suspend, resume and abort a workflow instance. The members of the group *composer-role* are allowed to create, start, suspend, resume and abort the workflow process instance.
Line 13 - 27: Different variables are defined by name and type for storing the state of the workflow. The `changeSet` and `comment` variables store the resources which are processed and the comments of the users. The four aggregation variables which are prefixed with publication are used to store the publication result.
Lines 29 - 34: If a new workflow has been created a dialog box opens up (this can be suppressed) where workflow variables can be initialized. The variables to show or set are defined in this initial client view. The variables `subject`, `comment`, `changeSet` and `comments` will be shown in the initial window, so that the creator of the workflow can change the content of the variable.
Line 36 - 41: If the workflow has been started, the variables defined in this client view will be shown in the variable view of the workflow window. The variables need not to be read only as in the example. The variables `subject`, `comment`, `changeSet` and `comments` will be shown in the variable view (if the workflow is selected in the workflow list), but it is not possible to change the variables, because they are defined as read only via the `<Reads ...>` elements.

Automated Task 'AssignUser'

```
1:  <AutomatedTask name="AssignUser"
                   description="assignuser-task"
                   successor="CheckEmptyChangeSet">
2:    <Action class="ForceUser" task="Publish"
            userVariable="OWNER_"/>
3:    <Action class="ForceUser" task="Compose"
            userVariable="OWNER_"/>
4:    <Action class="RegisterPendingProcess"
            userVariable="OWNER_"/>
5:  </AutomatedTask>
```

Example 4.11. Automated task "Assign User"

The first task in the workflow is an automated task that assigns a user to the main tasks - the user task 'Compose' and 'Publish - of the workflow. The user to assign is the creator and thus owner of the workflow.

Line 1 + 5: The automated task is named 'AssignUser'. The names of tasks are used in the definition of a successor of a task. The task, that is started after task 'AssignUser' completes, is 'CheckEmptyChangeSet'.
Line 2 + 3: The `Action` element defines the action which should be executed in the automated task. Here the predefined `ForceUser` action is used, which assigns the user defined in `userVariable` to the task defined in `task`. Thus, the `Compose` and `Publish` tasks will only be offered and automatically accepted to the user defined in the variable `OWNER_`. `WfVariableOWNER_` is a predefined variable which contains the user, who created the workflow.
Line 4: The `RegisterPendingProcess` registers the workflow process in the user's pending processes list. Users can watch their pending workflows in the Control Room.

User Task 'Compose'

```
1:  <UserTask name="Compose"
              description="studio-simple-publication-compose-task"
              successor="CheckEmptyChangeSet" reexecutable="true"
              autoAccepted="true">
2:    <Rights>
3:      <Grant group="administratoren" rights="read, accept, delegate, skip"/>
4:      <Grant group="composer-role" rights="read, accept, delegate, skip"/>
5:    </Rights>
6:
7:    <Assignment>
8:      <Writes variable="subject"/>
9:      <Writes variable="comment"/>
10:     <Writes variable="changeSet" contentEditable="true"/>
11:     <Writes variable="comments"/>
12:     <Reads variable="publicationResultCodes"/>
13:   </Assignment>
14: </UserTask>
```

Example 4.12. User Task Compose

This task is called when the publication fails so that one might fix problems. The purpose of the task is to enable the user to collect the content items which should be published at once.

Line 1: The user task is named 'Compose'. The localized description is looked up in a resource bundle under the key "simple-publication-compose-taskLabel" (the tooltip key is "simple-publication-compose-taskToolTip") and shown in the workflow window. The task `CheckEmptyChangeSet` is started after task `Compose` has completed.
Line 2 - 5: The rights concerning the task are assigned to groups. The group *administratoren* can read, accept, delegate or skip the task. The members of the *group composer-role* are allowed to read, accept, delegate, or skip the task. Not all the actions associated with a right can be performed in *Studio*, for some of them you need the *Site Manager*.
Line 7 - 13: If the task has been selected, the variables defined in this section will be shown in the variable view of the workflow window if the user has the *read* right. You can change the content of the variables `subject`, `comment`, `changeSet` and `comments` because they are defined in `Writes` elements. In addition, you can change the content of the content items, which are provided by the variable `changeSet` due to the attribute `contentEditable="true"`. The variable `publicationResultCodes` defined in the `<Variable>` section of the workflow, will be shown if you press the appropriate button in the variable view (if the task has been selected in the workflow list). You cannot change the content of the variable because it is defined as `<Reads ...>`.

If Task CheckEmptyChangeset

```
1:      <If name="CheckEmptyChangeSet">
2:        <Condition>
3:          <IsEmpty variable="changeSet"/>
4:        </Condition>
5:        <Then successor="Finish"/>
6:        <Else successor="Publish"/>
7:      </If>
```

Example 4.13. If Task

The second task in the workflow is the 'CheckEmptyChangeSet' task, an `If` task. The aim of the task is to check if the change set is empty. Then, no publication is necessary and the workflow can be finished.

Line 1 - 7: An `If` task is defined with the name 'CheckEmptyChangeSet'. An `If` task is a control flow element, which will be executed automatically. Thus, no visible description is necessary.
Line 2 - 4: A condition is defined that checks, whether the variable `changeSet` contains elements or not.
Line 5: If the condition evaluates to "true" (change set is empty) the workflow should be finished. Thus, the succeeding task is `Finish`.

Line 6: If the condition evaluates to "false" (change set contains elements) the changes should be published. Thus, the succeeding task is `Publish`.

User Task 'Publish'

```
1:  <UserTask name="Publish"
2:     description="studio-simple-publication-publish-task"
3:            successor="CheckPublication" autoCompleted="true"
               reexecutable="true" autoAccepted="true">
4:    <Rights>
5:      <Grant group="administratoren" rights="read,accept,retry"/>
6:      <Grant group="composer-role" rights="read,accept,retry"/>
7:    </Rights>
8:
9:    <Assignment>
10:     <Reads variable="subject"/>
11:     <Reads variable="comment"/>
12:     <Reads description="publish-changeSet"
                    variable="changeSet"
                    contentEditable="false"/>
13:     <Reads variable="comments"/>
14:    </Assignment>
15:
16:
17:   <EntryAction class="ApproveResource" gui="true"
18:       resourceVariable="changeSet"
19:       successVariable="publicationSuccessful"
20:       ignoreErrors="true"
21:       timeout="180"
          userVariable="PERFORMER_">
22:   </EntryAction>
23:
24:   <EntryAction class="PublishResources" gui="true"
25:      resourceVariable="changeSet"
26:      resultVariable="publicationResultResources"
27:      versionVariable="publicationResultVersions"
28:      codeVariable="publicationResultCodes"
29:      parameterVariable="publicationResultParams"
30:      successVariable="publicationSuccessful"
  ignoreErrors="false"
31:      ignorePublicationErrors="true" timeout="600"
          userVariable="PERFORMER_"/>
32:     </UserTask>
```

Example 4.14. User Task "Publish"

The third task of the workflow is a user task called 'Publish', that will publish the changes contained in the change set. This task will be automatically accepted by the composer of the change set due to the exit action `ForceUser` in the 'AssignUser' task.

Line 1 - 3: The user task is named "Publish" and its successor is the task "CheckPublication". The task will automatically be completed after execution of the entry actions because of the attribute `autoCompleted="true"`. This is useful when a set of automated actions should be executed on behalf of a user.
Line 4 - 7: The rights are granted to the groups `administratoren` and `composer-role`.
Line 9 - 15: Like mentioned before, variables are defined which should be shown in the variable view of the workflow window. Nevertheless, automatically completed tasks will only be shown in the case of error in the task list. In contrast to the declaration of these

variables in the `Compose` task no changes at all can be applied to the variables (due to `Reads`) and its content (due to `contentEditable="false"`).
Line 17 - 22: The first action performed in the task is the predefined `ApproveResource` action which approves the content items given via the attribute `resourceVariable`.
Line 24 - 31: After executing the first entry action, the second one will be performed. Here the content items given via the attribute `resourceVariable` will be published by the predefined action `PublishResources`. The other attributes define the variables to store the publication result into, to set timeouts and to ignore publication errors only.

If Task "CheckPublication"

```
1:       <If name="CheckPublication">
2:        <Condition>
3:          <Get variable="publicationSuccessful"/>
4:        </Condition>
5:        <Then successor="Finish"/>
6:        <Else successor="Compose"/>
7:      </If>
```

Example 4.15. If Task "CheckPublication"

The fifth task in the workflow is the 'CheckPublication' task, an `If` task. The aim of the task is to check if the publication was successful. If it was, the workflow will be finished, otherwise the compose task will be started again.

Line 1 + 7: The `If` task is named 'CheckPublication'. An `If` task is a control flow element which will be executed automatically.
Line 2 - 4: A condition is defined which will be evaluated. The value of the Boolean variable `publicationSuccessful`, which has been set in the `Publish` task will be read using the `Get` element.
Line 5: If the condition evaluates to "true" (`publicationSuccessful`="true") the workflow should be finished. Thus, the succeeding task is "Finish" task.
Line 6: If the condition evaluates to "false" (`publicationSuccessful`="false") the `Compose` task should be offered again.

Automated Task 'Finish'

```
1: <AutomatedTask name="Finish" final="true">
2:   <Action class="AssignVariable"
               resultVariable="changeSetLockedInStudio">
3:     <Boolean value="false"/>
4:   </Action>
5: </AutomatedTask>
```

Example 4.16. Example of automated task Finish

The last task of the workflow is an automated task and defines actions that are executed before the workflow completes. The task would also be needed if no such actions were necessary because the previous `If` task may not be the final task of the workflow.

Line 1: The automated task is named 'Finish'. Because the task should be the last one in the workflow, it must be marked as final. If the control flow of the workflow reaches a task with the attribute `final="true"`, it quits the execution of the workflow after the task was successfully executed.
Line 2 - 4: The lock on the change set in *Studio* is removed.

Final Action 'ArchiveProcessFinalAction'

```
1:  <FinalAction class="ArchiveProcessFinalAction"
                 maxProcessesPerUser="100"/>
```

Example 4.17. Example of ArchiveProcessFinalAction

Final actions are executed at the very end, after a workflow completed successfully or was aborted. The `ArchiveProcessFinalAction` archives data of the workflow and moves it from the list of pending workflows to the list of finished workflows for all users for that the `RegisterPendingProcess` action was called before.

4.4 Reference of Predefined Classes

In this chapter you will find a summary of all predefined classes which you can use in the tasks of the *CoreMedia Workflow*.

4.4.1 Predefined Action Classes

These are the predefined action classes which can be performed in tasks. They can be used with the elements `<Action>`, `<EntryAction>` and `<ExitAction>` by specifying the name of the action class as the class attribute of the respective action element.

If an action is described as applying to one resource in an atomic variable, it can be applied to a set of resources in an aggregation variable, too.

All predefined action classes discussed here support the following additional attributes to be specified as part of the action element:

Attribute	Type	Default	Description
class	NMTOKEN	#REQUIRED	the name of the action
successVariable	NMTOKEN	#IMPLIED	the name of a Boolean variable that will represent whether the action was successfully executed
resultVariable	NMTOKEN	`#IMPLIED`	the name of a variable that will receive a possible result of the action, if any

Table 4.3. Attributes common to all actions

Furthermore, every predefined action may contain a `Condition` element, which will be evaluated to determine whether the action should be executed at all.

Actions can be divided into server actions which are running solely on server-side and client actions (based on the class `AbstractClientAction`) which are running on client and server-side.

Client-side actions

Client action classes that are used to modify resources on the GUI Client respond to the following attributes:

Attribute	Type	Default	Description
gui	(*Boolean*)	"true"	Defines whether a GUI element will be shown on execution of the action ("true") or not. For instance, executing `publishResources` with `gui="false"` will not show the publication window.
ignoreErrors	(*Boolean*)	"false"	If set to "true", this attribute makes sure that the task containing the action will continue normally after an error was encountered.
timeout	NMTOKEN	"30"	The timeout in seconds for the action. The default timeout is 30 seconds.

Table 4.4. Attributes of client-side actions.

ApproveResource

Use this action to approve one or more CoreMedia resources referenced by a variable. If the variable stores an explicit version, that version is approved and a place approval

takes place. If no version information is present, only the place of the resource is approved.

Attribute	Type	Default	Description
resourceVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the resource(s) to be approved

Table 4.5. Attributes of the ApproveResource action.

CheckInDocument

Use this action to check-in one or more CoreMedia content items referenced by a variable.

Attribute	Type	Default	Description
documentVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the document(s) to be checked in

Table 4.6. Attributes of the CheckInDocument action.

CheckOutDocument

Use this action to check-out one or more CoreMedia content items referenced by a variable.

Attribute	Type	Default	Description
documentVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the content item(s) to be checked out

Table 4.7. Attribute of the CeckOutDocument action.

CopyResource

Use this action to copy a resource to a specified folder.

Attribute	Type	Default	Description
sourceVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the resource to be copied
destinationVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the folder where the copied resource should be located

Table 4.8. Attributes of the CopyResource action.

CreateDocument

Use this action to create a new content item in a specified folder.

This element may contain any number of Property elements that specify initial property values for the newly created content item.

Attribute	Type	Default	Description
folderVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the folder where the resource should be created
nameVariable	NMTOKEN	#REQUIRED	the name of the string variable that contains the name to be used
typeVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the content type for which a content item should be created

Table 4.9. Attributes of the CreateDocument action.

CreateFolder

Use this action to create a new folder in a specified parent folder.

Attribute	Type	Default	Description
folderVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the existing folder in which the new folder should be created
nameVariable	NMTOKEN	#REQUIRED	the name of the string variable that contains the name to be used

Table 4.10. Attributes of the CreateFolder action.

DeleteResource

Use this action to mark a resource for deletion.

Attribute	Type	Default	Description
resourceVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the resource(s) to be marked for deletion

Table 4.11. Attribute of the DeleteResource action.

DisapproveResource

Use this action to disapprove one or more CoreMedia resources referenced by a variable. If the variable stores an explicit version, that version is disapproved. If no version information is present, the most recent version will be disapproved.

Attribute	Type	Default	Description
resourceVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the resource(s) to be disapproved

Table 4.12. Attribute of the DisapproveResource action.

MoveResource

Use this action to move a resource to another folder.

Attribute	Type	Default	Description
sourceVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the resource to be moved
destinationVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the destination folder for the move

Table 4.13. Attributes of the MoveResource action.

OpenDocument

Use this action to open a content item in the editor.

Attribute	Type	Default	Description
documentVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the content item to open

Table 4.14. Attribute of the OpenDocument action.

PublishResources

Use this action to publish one or more CoreMedia resources referenced by a variable. If the variable stores an explicit version, that version is published. If no version information is present, the most recent version will be published.

The aggregation variables `resultVariable`, `codeVariable`, `parameterVariable`, and `versionVariable` jointly represent the result of the publication.

Attribute	Type	Default	Description
codeVariable	NMTOKEN	#REQUIRED	an integer aggregation variable
ignorePublicationErrors	(*Boolean*)	"false"	Defines whether an unsuccessful publication should be ignored

Attribute	Type	Default	Description
parameterVariable	NMTOKEN	#REQUIRED	a string aggregation variable
resourceVariable	NMTOKEN	#REQUIRED	Defines the name of the variable that contains the resource(s) to be published
versionVariable	NMTOKEN	#REQUIRED	an integer aggregation variable

Table 4.15. Attributes of the PublishResources action.

RenameResource

Use this action to rename a resource.

Attribute	Type	Default	Description
resourceVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the resource to be renamed
nameVariable	NMTOKEN	#REQUIRED	the name of the string variable that provides the new name of the resource

Table 4.16. Attributes of the RenameResource action.

SaveDocument

Use this action to save a content item that has to be opened in the Client GUI.

Attribute	Type	Default	Description
documentVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the content item to be saved

Table 4.17. Attribute of the SaveDocument action.

StoreProperties

Use this action to store property values in a content item. The property name and value are defined using the subelement `Property`.

Attribute	Type	Default	Description
documentVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the content item

Table 4.18. Attribute of the StoreProperties action.

UncheckOutDocument

Use this action to revert the check out of one or more CoreMedia content items referenced by a variable.

Attribute	Type	Default	Description
documentVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the checked-out content item(s)

Table 4.19. Attribute of the UncheckOutDocument action.

UndeleteResource

Use this action to remove the deletion from a resource.

Attribute	Type	Default	Description
resourceVariable	NMTOKEN	#REQUIRED	the name of the variable that contains the deleted resource(s)

Table 4.20. Attribute of the UndeleteResource action.

Server-side actions

While actions on the client-side deal with resources of the *Content Management Server*, actions on the server-side work on workflow objects in the *Workflow Server*.

ArchiveProcess

CAUTION

The `ArchiveProcess` action is deprecated and should not be used anymore. It's recommended to use `ArchiveProcessFinalAction` instead, which can also archive data of aborted processes. See Section 4.4.2, "Predefined FinalAction Classes" [79] for its description.

Use this action in the final task of a process to archive data of the process before it completes and gets destroyed in the *Workflow Server*. If the `RegisterPendingProcess` action was used before to add the process to some users' lists of pending processes, then these users can view the completed process in *Studio*'s *Control Room*.

The action can store the process data to a MongoDB database. To configure it, set the properties `mongodb.client-uri`, `mongodb.prefix` and `repository.caplist.connect` in the *Workflow Server*. See Section 6.1, "Configuration Reference" [112] for a description of these properties.

Attribute	Type	Default	Description
maxProcessesPerUser	NMTOKEN	[unlimited]	The maximum number of processes to show in the list of finished processes in *Studio's Control Room*. This attribute should be set to the same value for all ArchiveProcessFinalAction and ArchiveProcess actions in different workflow definitions, because all processes are stored in the same list.

Table 4.21. Attributes of the ArchiveProcess action.

Example:

The task will escalate if the process cannot be archived successfully, for example if MongoDB is down. Because of this it is recommended to retry the task as in this example:

```
<Variable name="finishTaskRetryTimer" type="Timer"/>
<TimerHandler class="RetryTaskTimerHandler"
              task="Finish"
              timerName="finishTaskRetryTimer"/>
```

```
<AutomatedTask name="Finish" final="true">
  <Action class="AssignVariable"
          resultVariable="finishTaskRetryTimer">
    <Timer value="30"/>
  </Action>
  <Action class="EnableTimer" timerVariable="finishTaskRetryTimer"/>
  <Action class="ArchiveProcess" maxProcessesPerUser="100"/>
</AutomatedTask>
```

Example 4.18. Example of the ArchiveProcess action

AssignVariable

Use this action to assign a new value to a variable. It takes a list of expressions (that evaluate to a `WfValue`) via the `Expression` subelement or `WfValue`s via the `Boolean`, `Date`, `String` etc. subelements.

Example:

This example will assign `Integer` values to the variable defined via the attribute `resultVariable`.

```
<Action class="AssignVariable" resultVariable="resultVariable">
    <Read variable="firstVariable" property="version_"/>
    <Expression class="AddLatestVersion">
      <Get variable="secondVariable"/>
    </Expression>
    <Integer value="4711"/>
</Action>
```

Example 4.19. Example of the AssignVariable element

DisableTimer

Use this action to disable a timer.

Attribute	Type	Default	Description
timerVariable	NMTOKEN	#REQUIRED	the variable that contains the timer that should be disabled

Table 4.22. Attribute of the DisableTimer action.

EnableTimer

Use this action to enable a timer. Note, that a timer has to be enabled before it may expire later.

Attribute	Type	Default	Description
timerVariable	NMTOKEN	#REQUIRED	the variable that contains the timer that should be enabled

Table 4.23. Attribute of the EnableTimer action.

ExcludePerformer

Use this action to exclude the performer of the current task from performing another specified task. When the specified task coincides with the current task, the exclusion will take effect when the task is reached the next time.

Attribute	Type	Default	Description
task	NMTOKEN	#Implied current task	the name of the task for which an exclusion should be established

Table 4.24. Attribute of the ExcludePerformer action.

ExcludeUser

Use this action to exclude a configured user from performing another specified task. When the specified task coincides with the current task, the exclusion will take effect when the task is reached the next time.

Attribute	Type	Default	Description
task	NMTOKEN	#Implied current task	the name of the task for which an exclusion should be established
userVariable	NMTOKEN	#IMPLIED performer	The variable which contains the user who should be excluded.

Table 4.25. Attribute of the ExcludeUser action.

ForceUser

Use this action to preset a user as the performer of a task. The task will be automatically accepted by the Client GUI for the user.

Example:

```
<AutomatedTask name="AssignComposer" description="assignUser"
 successor="Compose">
 <Action class="ForceUser" task="Compose" userVariable="OWNER_"/>
</AutomatedTask>
```

Example 4.20. How to force a user

Attribute	Type	Default	Description
task	NMTOKEN	#IMPLIED current task	The task for which the user is predefined.
userVariable	NMTOKEN	#IMPLIED performer	The variable which contains the user who should accept the task.

Table 4.26. Attributes of the ForceUser action.

Log

Use this action to write output to the log. The log name can be defined using the `facility` attribute. You can write text to the output defined via the attribute `message`. Using the subelement `Get` you can output the content of variables. Define the log level using the attributes `debug`, `info`, `warning`, or `error` (see Section 4.7, "Logging" in *Operations Basics* for details on the logging).

Attribute	Description
debug|info|warning|error	Defines the log level "debug", "info", "warning", or "error". Value must be "true".
message	The message which is printed to the log.
facility	Define a different log facility for the output. The default log facility contains both the process and the task name. For example, the following entry in the *Workflow Server*'s Logback configuration would match all info output of MyProcess workflows: `<logger name="workflow.actions.log.MyProcess"`

Attribute	Description
	additivity="false" level="info"> <appender-ref ref="file" /> </logger>

Table 4.27. Attributes of the Log action.

```
<Task ...>
  <Action class="Log" info="true" message="Enter task with x  ">
    <Get variable="x"/>
  </Action> </Task>
</Task>
```

Example 4.21. How to use a log action

PreferPerformer

Use this action to set the performer of the current task as the preferred performer of another task. When the given task coincides with the current task, the preference will take effect when the task is reached the next time.

Attribute	Type	Default	Description
task	NMTOKEN	#IMPLIED current task	the name of the task for which a preference should be established

Table 4.28. Attribute of the PreferPerformer action.

RegisterPendingProcess

Use this action to add the process to a user's pending processes list that is shown in *Studio*'s *Control Room*.

The action stores the user's pending processes to a MongoDB database. To configure it, set the properties `mongodb.client-uri`, `mongodb.prefix` and `repository.caplist.connect` in the *Workflow Server*. See Section 6.1, "Configuration Reference" [112] for a description of these properties.

Attribute	Type	Default	Description
userVariable	NMTOKEN	#IMPLIED the performer of a UserTask or the process	the variable which contains the user to whose list of pending pro-

Attribute	Type	Default	Description
		owner if not used in a UserTask	cesses the process should be added to

Table 4.29. Attributes of the RegisterPendingProcess action.

CancelUserTask

Use this action to cancel an activated user task.

Attribute	Type	Default	Description
task	NMTOKEN	#IMPLIED current task	The name of the user task to cancel.

Table 4.30. Attribute of the CancelUserTask action.

SkipUserTask

Use this action to skip an activated user task.

Attribute	Type	Default	Description
task	NMTOKEN	#IMPLIED current task	The name of the user task to skip.

Table 4.31. Attribute of the SkipUserTask action.

4.4.2 Predefined FinalAction Classes

These are the predefined action classes which can be executed after a process completed or was aborted. They are used with the element `<FinalAction>` and by specifying the name of the action class as the class attribute.

ArchiveProcessFinalAction

Use this action to archive data of the process after it completed or was aborted and before it gets destroyed in the *Workflow Server*. If the `RegisterPendingProcess` action was used before to add the process to some users' lists of pending processes, then these users can view the completed process in *Studio*'s *Control Room*.

The action can store the process data to a MongoDB database. To configure it, set the properties `mongodb.client-uri`, `mongodb.prefix` and `repository.caplist.connect` in the *Workflow Server*. See Section 6.1, "Configuration Reference" [112] for a description of these properties.

The *Workflow Server* will retry the execution of this action in case of communication problems with the MongoDB database. The exception classes that trigger a retry are defined in the configuration property `workflow.server.archive.retry-exception`, which is described in Table 3.33, "Workflow Server Properties" in *Deployment Manual*.

Attribute	Type	Default	Description
`maxProcessesPerUser`	NMTOKEN	[unlimited]	The maximum number of processes to show in the list of finished processes in *Studio's Control Room*. This attribute should be set to the same value for all ArchiveProcessFinalAction and ArchiveProcess actions in different workflow definitions, because all processes are stored in the same list.

Table 4.32. Attributes of the ArchiveProcessFinalAction

Example:

```
<FinalAction class="ArchiveProcessFinalAction"
             maxProcessesPerUser="100"/>
```

Example 4.22. Example of the ArchiveProcessFinalAction

4.4.3 Predefined TimerHandler Classes

Timer handler classes are invoked when a timer expires.

```
<UserTask name="c0_edit" final="true">
    <Variable name="skipExpiredTimer" type="Timer">
```

```
        <Timer value="30"/>
    </Variable>
    <TimerHandler class="RunActionTimerHandler" name="TimerHandler"
                          timerName="skipExpiredTimer">
        <Action class="Log" debug="true" message="timer expired"/>
        <Action class="CancelUserTask" task="c0_edit"/>
    </TimerHandler>
    <EntryAction class="EnableTimer"
                         timerVariable="skipExpiredTimer"/>
    <EntryAction class="Log"
                        debug="true" message="c0_edit accepted"/>
    <Rights>
        <Grant user="cpesch"
         rights="read,accept,complete,cancel,retry"/>
    </Rights>
    <Client>
        <Reads variable="skipExpiredTimer"/>
    </Client>
</UserTask>
```

Example 4.23. Example of TimerHandler usage

AbortTaskTimerHandler

This timer handler aborts the task instance in which it is defined on expiration (see Figure 4.10, “Expired timer with AbortTaskTimerHandler” [81]).

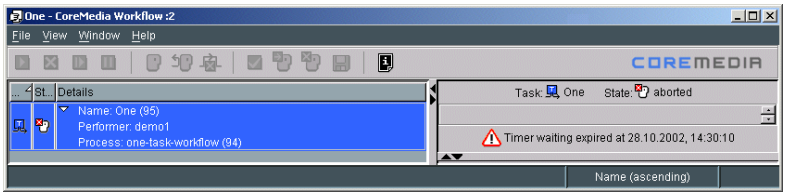

Figure 4.10. Expired timer with AbortTaskTimerHandler

AddWarningTimerHandler

This timer handler adds a timer expiration warning to a process or task instance (see Figure 4.11, “Expired timer with `AddWarningTimerHandler`” [81]).

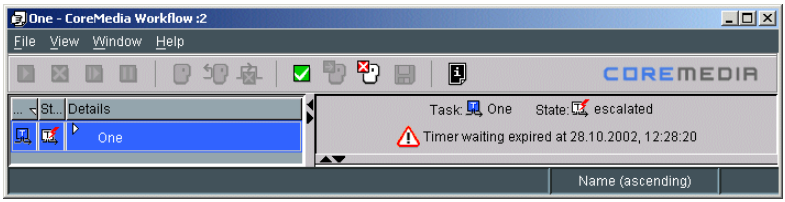

Figure 4.11. Expired timer with `AddWarningTimerHandler`

RetryTaskTimerHandler

This timer handler retries an escalated task. The handler and its timer need to be defined below the Process element. The name of the task to retry is specified in the additional attribute "task".

RunActionTimerHandler

This timer handler runs one or more actions on expiration. The actions can be defined using the sub element `Action`.

SkipUserTaskTimerHandler

This timer handler aborts the *activated* user task on expiration. It does *not* work with a task if it is not activated.

5. Implementing Extensions

This chapter deals with the customizing of the workflow by programming own extensions and configuring the workflow. The following types of workflow beans are supported:

- Actions (server-side and client-side actions)
- Expressions (used in guards, conditions, validators)
- Rights policies
- Performers policies

In addition, you can implement own

- Clients,
- Workflow startups.

You will find some programming guidelines and examples for each bean in the following subsections. Please refer to the *Workflow API* or *Site Manager API* for more details on the classes described in the following chapters.

Note that this manual describes the old *Workflow API* that was the sole means for writing extensions up to *CMS 2005*. From *CMS 2006* on, it is also possible to use the *Unified API* for writing extensions, there called plugins. Please consult the *Unified API Developer Manual* for details regarding this new API. Most information from the following sections carries over to the new API.

Because the old *Workflow API* is still used in the *Site Manager*, it is not formally deprecated. However, using the *Unified API* is recommended for new server-side actions, expressions, rights policies, and performers policies, because development has become easier and faster. In general, old and new extensions mix without problems. Please see the *Unified API Developer Manual* for details.

5.1 Update Workflows

Uploaded workflow definitions are stored in the database as serialized objects. Every time, you have made incompatible changes to your extension classes, which are used in already uploaded workflows, you need to convert these workflows. Use the workflow converter utility for this. In case of an update of the *CoreMedia Workflow Server*, the workflows have to be converted, too. Otherwise, object deserialization errors can occur (see Oracle JDK documentation for details).

CAUTION

Changes at classes that are used in uploaded workflows should happen with great care and intensive testing. The classes *must* be compatible with the uploaded XML workflow definition.

See Section 5.9, “Pitfalls of Implemented Classes” [109] for more hints on this topic.

5.2 Variable Values

Variables are typed. A variable of a certain type can only contain values of its defined type or subclasses of the type.

The existing values are closely related to *CoreMedia CMS* property types and resource objects:

- Boolean
- Blobs
- Contents, Folders and Documents
- Content types
- Dates
- Exceptions
- Groups and Users
- Integers
- Strings
- Timers

All values implement the `java.lang.Comparable` interface. They may contain `null` values and are immutable. So, their `setValue()` methods must never be called from your own code, the result of such an action is unpredictable.

5.3 Programming Actions

Actions are used to automate or semi automate tasks. Two kinds of actions exist:

- Actions running only on server side.
  Server-side actions run completely inside the *CoreMedia Workflow Server*. They may use the *CoreMedia Workflow Server*'s session to the *CoreMedia Content Management Server* to access resources.
- Client actions running partly on the server and on a client.
  ClientActions in a user task run remotely using the Client GUI's session to the *CoreMedia Content Management Server* to access resources. ClientActions in an automated task run in a server internal client environment using the *CoreMedia Workflow Server*'s session to the *CoreMedia Content Management Server* to access resources.

5.3.1 General Rules

Actions can only be used in automated tasks, user tasks, in the predefined `RunActionTimerHandler`, or as final actions. They are performed at different times:

- Entry actions are performed when a user accepts a task.
- Exit actions are performed when a user completes a task.
- Actions in an automated task run when the guard evaluates to true.
- Actions in a timer handler are run if the associated timer expires.
- Final actions run after a process was completed or aborted. Final actions use a different interface, which is not available in the old Workflow API. See the *Unified API Developer Manual* which describes final actions as part of the Unified API.

Actions should run for shortest period that is feasible since they run inside a server transaction and block precious server resources. To avoid problems, stick to the following rules:

- Don't write client actions that require user interaction.
- If you interact with another system and need to wait for a result, for example sending a mail and waiting for a notice of its reception, always use a second task with a guard (see Section 4.1.7.3, “Guards” [53]) following the initial task with your action. The example in Section 5.4.4, “Example Expression” [96] describes an expression which checks whether a mail has been received or not.

Note the following features which are helpful when you program your own actions:

- Actions are Java beans.

- Parameters for the global configuration of the action bean can be defined via the XML workflow definition (see Section 5.3.5, "Access Workflow Variables from the Action" [92]).
- Actions can set a success variable which may be used to control the error handling within the workflow.
- Actions can assign a result to a workflow variable (see Section 5.3.5, "Access Workflow Variables from the Action" [92] for details).

5.3.2 Repeated Execution of Actions

If there are concurrent running transactions in an instance (if you've forked the workflow) and the actions run by these transactions are creating read/write conflicts in the context. They may be seen as transaction serialization errors in the log. To solve a conflict, the *CoreMedia Workflow Server* automatically repeats the conflicting transactions. This means that even already executed actions are repeated, too.

Since there is a complete rollback of the transactions, the actions cannot determine if they are run repeatedly. Try to avoid the conflicts arising from this under all circumstances or you may experience problems with your workflow. Stick to the following rules:

- Write actions that are fault-tolerant and can handle multiple repeated executions.
- Split your critical sections into several tasks to isolate the non-repeatable actions from the actions creating the conflicts.

Note that, even if you follow these rules, an action may be executed repeatedly in the unlikely event of a *CoreMedia Workflow Server* crash. During the next restart, all failed transactions are repeated to reach a consistent state. This may repeat the execution of your action, too.

If an action throws any exception, its task instance will be escalated immediately:

- Side effects on the instances context will become persistent, there is no rollback of the transaction.
- If you are running two actions and the second one fails, the success and result variables of the first action will keep their values.
- Upon a retry, these variables can be used by the first action's guard to avoid repeated execution.

Exceptions within the `RunActionTimerHandler` actions will have no effect other than the handler failing.

5.3.3 Server-Side Actions

CAUTION

Note that the old legacy Workflow API is described here. It is preferable to use the *Unified API* for writing server-side actions. Please consult Section 6.10.3, "Actions" in *Unified API Developer Manual* for details.

Interface to implement

Server-side actions implement the interface `com.coremedia.workflow.WfAction`.

Convenience classes

For convenience you can subclass `com.coremedia.workflow.common.actions.AbstractAction` which already includes implementations of all needed getter and setter methods and which uses a condition as guard (`isExecutable()`). You need to implement the `execute()` method for your own functionality. This method will be called by the *CoreMedia Workflow Server*. In Section 5.3.6, "Example Action" [93] you will find a complete example of a server-side action.

5.3.4 Client-Side Actions

A client-side action consists of a client-side and a server-side implementation, so it's running partly on both sides. Whereas client-side actions run on behalf of the client user, server-side actions run on behalf of the workflow user. While the client part of the action is running, the server side transaction is still active. Client actions always have a timeout after which the action is aborted on the server side, the client side is not affected by this. ClientActions should not require user interaction, if possible, to save precious server resources.

Precisely, a client action consists of three parts:

- A server-side action.
- A client-side event listener.
- A client-side command to execute.

The three parts will be executed in the following way:

- Define the action from the workflow definition using the `class` attribute.

- The *CoreMedia Workflow Server* executes the server-side stub. It creates the parameter list which it includes in the event.
- The event will be received by the client which calls the `handle()` method of the client-side event listener.
- The event listener has to return a callback ID to the server. You can evaluate the event and start some action from the event listener. For example, the event listener belonging to the *Site Manager* will execute the action the server noticed in the event.

CAUTION

Using the *Unified API*, it is not possible to write the server-side parts of new client side actions. You still need the *Workflow API* for this. Note, however, that client-side actions can now be replaced by *Unified API* server-side actions in many cases, because the *Unified API* allows you to act on behalf of a particular user without having to open a separate connection.

Interface to implement

Basically, the server-side stub of a client-side action must implement the interface `com.coremedia.workflow.WfAction`. For convenience and to hide the details how events are created and dispatched, you must subclass `com.coremedia.workflow.common.actions.AbstractClientAction`. This class already includes implementations of all needed getter and setter methods, uses a condition as guard and contains all the event logic. `AbstractClientAction` also implements a default timeout for a client action. The default timeout time is 30 seconds and can be configured using the attribute `timeout` in the workflow definition. In Example 5.1, “Example of the server-side stub for a client-side action” [90] you see a sample action which extends `AbstractClientAction`.

Server Side

The server-side implementation of a client action is a stub which:

- Assembles the argument list and passes it to the client via an event. The `AbstractClientAction` class includes the command and GUI parameters in the argument list. The command is the one used as the parameter in the call of the super() constructor.
- Receives the clients result and creates a `WfActionResult` from it.

Custom clients, that are not event driven, have to be aware that while performing a `Task.accept` or `Task.complete` operation on behalf of connected clients there may be callbacks of the client action for the pending call. The callbacks have to be executed before the server call can return.

```
1:  public class DemoClientAction extends AbstractClientAction
2:  {
3:    public DemoClientAction() {
4:      super("com.customer.example.workflow.action.
                  DemoActionCommand");
5:    }
6:
7:    protected HashMultiMap
8:    processArguments(WfTaskInstance taskInstance,
                                  HashMultiMap map)
9:      throws WfException {
10:     map.put("documentType", "Article");
11:     return map;
12:   }
13:
14:   protected WfActionResult
15:   processResult(WfTaskInstance taskInstance,
                            WfClientActionResult result)
16:     throws WfException {
17:     // result processing...
18:     return new WfActionResult(values, warnings, success);
19:   }
20: }
```

Example 5.1. Example of the server-side stub for a client-side action

Line 1: Use the `AbstractClientAction` instead of the `WfAction` interface.

Line 3 - 5: The constructor of your server stub. The constructor of the super class is called with the command as a parameter which should be executed on client side. This command is automatically included in the event send to the client.

Line 7 - 12: This processes the arguments which will be included in the event. In line 10 the parameter `docmentType` is added to the `HashMultiMap`. This map already contains `WfClientActionListener.GUI` and `WfClientActionListener.COMMAND` as default entries.

Line 14 - 19: This process the result that has been received from the client.

Client Side

The client-side must have a `WfClientActionListener` installed (see Example 5.2, "Example of an action listener" [91]) which handles the incoming action events.

- The callback ID obtained by `WfClientActionEvent.getCallbackId` must be returned in the actions result so that the *CoreMedia Workflow Server* can associate request and callback.
- `WfClientActionEvent.getParameters` returns the call parameters as encoded by the server side stub. For the previous example, the parameters would contain the strings `documentType` and `Article`.
- The event's other methods are reserved for internal use.
- All predefined client actions use a property/value encoding for the action parameters. Everything is encoded as a `java.lang.String`.

Note, that the *Site Manager* has a generic client listener, that tries to find and execute an appropriate `hox.corem.editor.commands.Command`. Have a look at the *Editor Developer Manual* for details.

```
1:  package com.customer.example.workflow.action;
2:
3:  import com.coremedia.workflow.*;
4:  import
     com.coremedia.workflow.common.actions.ClientActionResult;
5:
6:  public class DemoClientActionListener implements
     WfClientActionListener {
7:
8:      public DemoClientActionListener() {
9:      }
10:
11:      public WfClientActionResult
           handle(WfClientActionEvent actionEvent) {
12:        String[] parameters = actionEvent.getParameters();
13:        System.out.println("parameters.length="+
                                        parameters.length);
14:        for (int i=0; i < parameters.length; i++) {
15:          System.out.println("parameters["+i+"]="+
                                          parameters[i]);
16:        }
17:        return new ClientActionResult(actionEvent.
                               getCallbackId());
18:      }
19:  }
```

Example 5.2. Example of an action listener

Line 6: The client listener must implement `WfClientActionListener`.

Line 11: This method must be implemented. It gets the event as a parameter. Here you can implement your functionality evaluating the information from the event.

Line 12 -15: This is only a simple example. The parameters of the event are read in an array and are printed out.

Line 17: An important line: The client listener must return a `ClientActionResult` containing at least the callback ID. It is also possible to return more information to the server. See the *CoreMedia Workflow API* documentation for more details on `ClientActionResult`.

Command for the Site Manager

In Example 5.3, “Command executable on the Site Manager” [91] you see an example command which is executed on the client when an appropriate event is received by the event listener of the *Site Manager*. For this, the action defined in Example 5.1, “Example of the server-side stub for a client-side action” [90] has to be executed.

```
1:  package com.customer.example.workflow.action;
2:
3:  import hox.corem.editor.toolkit.*;
4:
```

```
5:  public class DemoActionCommand implements Command {
6:
7:    public boolean execute(Context context, Target target) {
8:      System.out.println("DemoActionCommand.execute() " +
                                  "context="+context+ " " +
                                  "target="+target);
9:      return true;
10:   }
11:
12:   public boolean isExecutable(Context context, Target target) {
13:     return true;
14:   }
15:
16: }
```

Example 5.3. Command executable on the Site Manager

Line 3: You need to import this package because you are working on the *Site Manager*.

Line 5: The name of the class must be the one called from the server.

Line 7 - 10: This is the method in which you implement your actual action. The example action only prints the content of `context` and `target` and returns true.

Line 12 - 14: This method returns whether the command is executable with the given target and context or not.

5.3.5 Access Workflow Variables from the Action

It is good practice not to hard code the variable names into the action bean, but to use configurable attributes to access the workflow variables. Thus, it is easier to reuse the action in other workflow definitions. Here is how you do this:

- Configure your action bean from the workflow definition by adding an attribute to the `<Action>` element like in Example 5.4, "How to configure an action bean" [92]
- Define a setter method in your action for the configuration like in Example 5.5, "Example of an action" [93].
- Directly access workflow variables using the `WfInstance.getAtomicVariable()` or `WfInstance.getAggregationVariable()` method.

```
1: <Variable name="MyFirstVariable" type="String>
2:   <String value="OnlyATest"/>
3: </Variable>
4: <AutomatedTask name="One" successor="Two">
5:  <Action
   class="com.customer.example.workflow.action.ParameterAction"
```

```
6:          variableToPass="MyFirstVariable"/>
7: </AutomatedTask>
```

Example 5.4. How to configure an action bean

In the example above, you defined a string variable with the name "MyFirstVariable" and the value "OnlyATest". With line 6 you configure the action bean that the method `setVariableToPass()` on an instance of `com.customer.example.workflow.action.ParameterAction` is called with the name of the string variable as a parameter.

```
1: public class ParameterAction extends AbstractAction {
2:  private String text;
3:   ...
4:  public String getVariableToPass() {return variableToPass; }
5:  public void setVariableToPass(String t) {variableToPass = t;}
6:  public WfActionResult execute(WfTaskInstance wfTaskInstance)
     throws WfException {
7:     ...
8:      WfAtomicVariable variable =
           wfTaskInstance.getAtomicVariable(variableToPass));
9:     ...
10:  }
11:}
```

Example 5.5. Example of an action

Line 4 - 5: Here you define the setter and getter methods for the configuration of your action bean.

Line 8: Here you get the workflow variable using the name configured with the `setVariableToPass()` method.

5.3.6 Example Action

The Workflow API described in this manual is an outdated way to write actions. You can find an example action based on the easier and more modern *Unified API* in Section 6.11.3, "Example Code of the Mail Action" in *Unified API Developer Manual*.

5.4 Programming Expressions

Expressions come in two variants:

- generic expressions and
- Boolean expressions.

A generic expression must evaluate to a `java.lang.Comparable` result and can be used for example in a `<Less>` or `<Greater>` expression. A Boolean expression must evaluate to a `Boolean` result value and can be used for example in a `<Condition>` task.

Expressions can be used for many purposes in the workflow:

- Guards for automated and user tasks
- Preconditions and postconditions (assertions) in automated and user task
- Validators for variable assignments in client views
- Conditions for branching tasks
- Guards for actions

5.4.1 General Rules

When you are programming own expressions, respect the following general rules:

- Expressions must not have any side effects.
- Expressions must not hold any state.
- Expressions must be repeatable any number of times.
- All top level expressions used in the workflow configuration must be Boolean expressions.

Depending on their arity, expressions may have a specific number of subexpressions, which are added through the `addExpression()` method. For example, a comparison has an arity of two, as it compares exactly two expressions. A logical expression like `And` or `Or` are n-ary, it must have at least two subexpressions, but may have any number of expressions. In contrast to that, a `Not` must have exactly one subexpression. If a maximum number of expressions is exceeded, a `WfRuntimeException` with the error code `TOO_MANY_SUBEXPRESSIONS` thrown.

5.4.2 Generic Expressions

Interface to implement

For a generic expression you have to implement the interface `com.coremedia.workflow.WfExpression`. Such an expression must return a `java.lang.Comparable` value. If you want to use the result of your expression for further evaluation, you should return a `WfValue` because this is what all built-in expressions operate on.

Convenience classes

For convenience you can subclass from `com.coremedia.workflow.common.expressions.AbstractExpression` and implement the `evaluate()` method, which is called by the *CoreMedia Workflow Server*. See Example 5.7, “Example of a generic expression” [95] for a simple example of an expression.

Define expressions

The following XML fragment shows, how to define your expressions in the workflow definition.

```
.
.
  <Variable name="comment" type="String">
    <String value="TestString"/>
  </Variable>
.
.
  <If name="One">
    <Condition>
      <Less>
        <Expression class="com.coremedia.example.
                           expression.DemoExpression"/>
        <Get variable="comment"/>
      </Less>
    </Condition>
    <Then successor="True"/>
    <Else successor="False"/>
  </If>
.
.
```

Example 5.6. Use a generic expression in the workflow definition

Example generic expression

The following code example shows a simple expression which returns a `StringValue`.

```
public class SampleExpression extends AbstractExpression {

  public String getName() {return "SampleExpression";}
```

```
  public Comparable evaluate(WfInstance instance,
                             Map localVariables) {
    return new StringValue("ConstantValue");
  }
}
```

Example 5.7. Example of a generic expression

5.4.3 Boolean Expressions

Interface to implement

For a Boolean expression you need to implement the interface `WfBooleanExpression`. It extends `WfExpression` and defines an `evaluateExpression()` method with a `Boolean` result.

Convenience classes

For convenience you can subclass from `com.coremedia.workflow.common.expressions.AbstractBooleanExpression` and implement its `evaluateExpression()` method.

The abstract classes `evaluate()` method calls `evaluateExpression()` and builds a `BooleanValue` from the returned value. The next example shows a simple Boolean expression which always returns true - a tautology.

```
public class Tautology extends AbstractBooleanExpression {

  public String getName() {return "Tautology";}

  public boolean evaluateExpression(WfInstance instance,
                                    Map localVariables) {
    return true;
  }
}
```

Example 5.8. Example of a Boolean expression

5.4.4 Example Expression

This chapter describes how to create a Boolean expression and insert it in the workflow definition. Have a look at Example 5.10, “Example Expression” [97] for the example of a simple Boolean expression which always returns "true".

Define the expression in the workflow definition

You can use your expression in the workflow definition via the `<Expression>` tag. See Example 5.9, "Including expressions in the workflow definition" [97] for an expression inserted in an `<If>` tag.

```
<If name="One">
  <Condition>
    <Expression class="com.coremedia.example.expression.
                       DemoExpression"/>
  </Condition>
  <Then successor="True"/>
  <Else successor="False"/>
</If>
```

Example 5.9. Including expressions in the workflow definition

If the expression evaluates to true then the successor is the task named True, otherwise it is the task named False.

Programming the expression

See Example 5.10, "Example Expression" [97] for the important lines of the code. Configuring the expression with variable names from the workflow is not shown in this example but it is similar to the method in the action example. The same is true for accessing the repository.

```
1:  package com.coremedia.examples.workflow.expression;
2:
3:  import java.util.Map;
4:  import com.coremedia.workflow.WfInstance;
5:  import com.coremedia.workflow.common.expressions.
      AbstractBooleanExpression;
6:
7:  public class DemoExpression
      extends AbstractBooleanExpression {
8:
9:    public String getName() {
10:     return "DemoExpression";
11:   }
12:
13:   public String getSymbol() {
14:     return getName();
15:    }
16:
17:   public boolean isInfix() {
18:     return false;
19:   }
20:
21:   public boolean evaluateExpression(WfInstance instance,
                                        Map localVariables) {
22:     return true;
23:   }
24: }
```

Example 5.10. Example Expression

Line 1: The package to which the action belongs.

Lines 3 - 5: All Java classes which are at least necessary for an expression to use.

Line 7: In order to create a Boolean expression you need to implement the interface `WfBooleanExpression`. For convenience you can extend the abstract `AbstractBooleanExpression` class.

Line 9 - 19: If you extend `AbstractBooleanExpression`, you need to implement four methods. Three of them `getName()`, `getSymbol()` and `isInfix()` are used for better reading of the log, if the expression is converted into a string using the `toString()` method.

Line 21 - 23: The fourth method to implement is the most important one, `evaluateExpression(WfInstance instance, Map localVariables)`. This method will be called when the expression is evaluated. Here you can implement the logic of your expression. Using the parameter `instance`, you can access the workflow instance as shown in the action example. The Map `localVariables` gives access to expression local variables, which may be defined with `ForAll` and `Let`.

5.5 Programming Rights Policies

Rights policies protect access to process and task instance operations. They can be performed on the server and client side so a GUI Client component may limit the offered buttons, menus etc. to the actual permitted operations.

The following rights are defined for process instances and can be granted to individual users or groups:

- Read and write variables exported by the processes client view
- Create new process instances
- Start process instances
- Suspend and resume process instances
- Abort process instances

The following rights are defined for task instances and can be granted to individual users or groups:

- Read and write variables exported by the tasks client view
- Reject, accept, cancel and complete a task instance
- Assign, delegate and skip a task instance
- Retry the last transaction of an escalated task instance

The policies are not directly accessible, checks must be performed via `WfInstance.hasPermission()`, which checks the rights of the current session's user.

Customized rights policies must never access any client or server specific classes, as it will be executed on both sides. It may provide a client and a server-specific implementation of an interface, that gives access to client or server specific classes. Logging must be done to the generic logging facility defined by `com.coremedia.workflow.common.Common.`

Interface to implement

Rights policies must implement the interface `WfRightsPolicy`.

Default implementation

If you only want to adapt the default policy to your needs, subclass the default rights policy `AclRightsPolicy` and override the appropriate methods.

Defining the policy in the workflow definition

Defining your own rights policy in the workflow definition is quite simple. You only need to add the `policyClass` attribute to the `<Rights>` tag as shown in Example 5.11, “Integrate own rights policy in the workflow definition” [100]. This class must be available

in the classpath of the *Workflow Server* and *Site Manager*. That means you need a runtime dependency on this JAR file in your Site Manager application module and Workflow Server web application in the workspace.

```
<Workflow>
  <Process name="TestWorkflow" startTask="FirstOne">
    <Rights policyClass="myPackage.MyOwnRightsPolicy">
      <!-- ... more elements and attributes ... -->
    </Rights>
    .
    .
  </Process>
</Workflow>
```

Example 5.11. Integrate own rights policy in the workflow definition

5.5.1 Example Rights Policy

This example describes the implementation of a rights policy. The aim of the policy is to implement a very simple rights policy that can grant rights to the user who started a process instance. The policy should be usable with very large user sets, in an intranet for instance. To this end, the policy computes the members of a group only when necessary. The policy can be used as a replacement of the default `ACLRightsPolicy` in the standard simple publication workflow. It is available bundled with its *Unified API* equivalent in the examples distribution, which also contains the adapted workflow definition `example-publication.xml`. To try the example workflow, deploy the `cap-plugin.jar` from the examples in the `lib` directories of the *Workflow Server* and all clients you want to use, for example in the *Site Manager*.

The new class `OnlyOwnerWfRightsPolicy` will be serializable by means of the interface `WfRightsPolicy`. One field holds the optional id of the group that is granted create rights and one field denotes whether a group was actually set.

```
public class OnlyOwnerWfRightsPolicy implements WfRightsPolicy {
  private static final long
  serialVersionUID = 7389049258655067247L;
  private int groupId;
  private boolean groupIdSet = false;
```

The standard callback for setting the set of rights is unused: the policy grants or denies all rights

```
public void setRights(String[] rights) {}
```

Some methods for managing the policy configuration are needed.

```
public void setGroupId(int groupId) {
    this.groupId = groupId;
    this.groupIdSet = true;
  }
  public int getGroupId() {
```

```
    return groupId;
  }
  public boolean isGroupIdSet() {
    return groupIdSet;
  }
  public void setGroup(String groupAtDomain) throws WfException {
    int pos = groupAtDomain.indexOf('@');
    WfGroup group;
    if (pos < 0) {
      group = WfServer.getDirectoryServiceAdapter().
        getGroup(groupAtDomain, "");
    } else {
      String name = groupAtDomain.substring(0, pos);
      String domain = groupAtDomain.substring(pos+1);
      group = WfServer.getDirectoryServiceAdapter().
        getGroup(name, domain);
    }
    setGroupId(group.getId());
  }
```

Note that the last method is never actually called from Java code. It is called dynamically during the process definition parsing.

Because the policy grants special access to the owner of a process instance, you can make use of a utility method for determining that user.

```
private WfUser getOwner(WfInstance instance) throws WfException
   {
    if (instance instanceof WfTaskInstance) {
      instance = ((WfTaskInstance)instance).getProcessInstance();
    }
    return ((WfProcessInstance)instance).getOwner();
  }
```

Now you can write the methods from the interface `WfRightsPolicy`. Some group-related methods are not shown. They are only called in the context of delegation to a group, which is not an appropriate use case for this class.

```
public boolean hasPermission(WfInstance instance,
    WfDirectoryServiceAdapter adapter, WfUser user,
    String rights)
  throws WfException {
    return hasPermission(instance, adapter, user);
  }
  ...
  public boolean hasPermission(WfInstance instance,
    WfDirectoryServiceAdapter adapter, WfUser user,
    String[] rights)
  throws WfException {
    return hasPermission(instance, adapter, user);
  }
  ...
```

Now have a look at the central method for permission computation. First of all, you must make sure to grant all rights to the internal server user, which performs certain automated actions. The super administrator also needs all rights.

```
private boolean hasPermission(WfInstance instance,
    WfDirectoryServiceAdapter adapter, WfUser user)
  throws WfException {
    if (user.isInternalServerUser()) return true;
```

```
    if (user.getId() == Id.ADMIN) return true;
    if (instance == null) {
```

You are being asked for rights on the definition. This can only be a create operation that needs to be checked.

```
if (!isGroupIdSet()) return false;
      WfGroup group = adapter.getGroup(getGroupId());
      return user.isMember(group);
    } else {
```

You already checked for the admin and for the internal server user, so that the remaining code is simple.

```
WfUser owner = getOwner(instance);
      return owner != null && owner.getId() == user.getId();
    }
  }
```

When computing a worklist, you sometimes need to compute the set of all users. Expensive group operations are only needed in the case of rights on the definition.

```
public WfUser[] getUsers(WfInstance instance,
    WfDirectoryServiceAdapter adapter, String right) throws
     WfException {
    if (instance == null) {
      if (isGroupIdSet()) {
        WfGroup group = adapter.getGroup(groupId);
        return group.getUsers();
      } else {
        return new WfUser[0];
      }
    } else {
      WfUser owner = getOwner(instance);
      WfUser admin = adapter.getUser(Id.ADMIN);
      if (owner == null || owner.getId() == Id.ADMIN) {
        return new WfUser[]{admin};
      } else {
        return new WfUser[]{admin, owner};
      }
    }
  }
  ...
```

Finally, you must provide a marshaller for transferring the rights policy to clients,

```
public WfRightsPolicyMarshaller getMarshaller() {
    return new OnlyOwnerWfRightsPolicyMarshaller();
  }
}
```

The marshaller itself is implemented in a separate class. It is identified by its policy id.

```
public class OnlyOwnerWfRightsPolicyMarshaller
implements WfRightsPolicyMarshaller {
  public String getPolicyID() {
```

```
    return "coremedia:///cap/workflow-rights-policy/OnlyOwner";
  }
```

The main methods affect the marshalling an unmarshalling of the policy group parameter, which has to be encoded as an array of bytes.

```
public byte[] marshal(WfRightsPolicy policy) {
    OnlyOwnerWfRightsPolicy onlyOwner =
      (OnlyOwnerWfRightsPolicy) policy;
    int groupId = onlyOwner.getGroupId();
    return new byte[] {
      (byte)(groupId), (byte)(groupId>>8),
      (byte)(groupId>>16), (byte)(groupId>>24),
      (byte)(onlyOwner.isGroupIdSet() ? 1 : 0)
    };
  }
  public WfRightsPolicy unmarshal(byte[] data) {
    OnlyOwnerWfRightsPolicy result = new OnlyOwnerWfRightsPolicy();
    if (data[4] == 1) {
      result.setGroupId((data[0] & 0x000000ff) +
        (data[1]<<8 & 0x0000ff00) +
        (data[2]<<16 & 0x00ff0000) +
        (data[3]<<24));
    }
    return result;
  }
}
```

This policy has also been implemented using the *Unified API*. For details see the *Unified API Developer Manual*.

5.6 Programming Performer Policies

Performer policies control to which users a task instance should be offered. A performers policy calculates this set of users based on the users which have permission to accept the task instance defined by the rights policy. The performer policy is called by the *CoreMedia Workflow Server*.

A performers policy may optionally support:

- Users who must be excluded from the offer (determined by the `ExcludePerformer` or `ExcludeUser` action).
- Users who may be preferred (determined by the `PreferPerformer` action).
- Groups which may be preferred.
- Users who actively reject the offered task instance.
- A single user who must perform the task (which will force an accept of the instance as soon as the user logs on to the workflow server, determined by the `ForceUser` action)

The `DefaultPerformersPolicy` supports all options.

NOTE

There is no automatic recalculation of the user sets if there are changes in the user management. This may cause the following effects:

- New users or users assigned to new groups won't see any offers already pending.
- Users removed from groups won't see already offered task disappear from their task lists. This is not a security problem, since the rights are checked on every access on the server.

Interface to implement

Own performer policies must implement the interface `com.coremedia.workflow.WfPerformersPolicy`. The important method is `calculateAssignment(WfTaskInstance taskInstance, WfUser[] permittedUsers)` which is called by the *CoreMedia Workflow Server*. It returns a `WfUserAssignment` object (see the *Workflow API* documentation for details).

Default implementation

If you only want to adapt the default performer policy to your needs it would be easier to subclass the default performer policy `DefaultPerformersPolicy` and to override the appropriate methods.

Defining the policy in the workflow definition

In Example 5.12, "Defining a performer policy in the workflow definition" [105] you see how to define your own performer policy in the workflow definition.

```
<Workflow>
    <Process name="PerformerTest" startTask="One">
    .
    .
    <UserTask name="One" final="true">
        <Performers policyClass=
                "com.coremedia.example.DemoPerformersPolicy"/>
        <Rights>
            <Grant group="composer-role"
                   rights="read, accept, complete"/>
        </Rights>
    </UserTask>
    .
    .
    </Process>
</Workflow>
```

Example 5.12. Defining a performer policy in the workflow definition

Customize the performer policy

See Example 5.13, "Invoking a performer policy" [105] for a customization of the default performer policy which performs a very simple task. It calls the default performer policy and cuts off the last user from the result.

```
1:  package com.coremedia.example.policy;
2:
3:  import com.coremedia.workflow.*;
4:  import com.coremedia.workflow.common.policies.
     DefaultPerformersPolicy;
5:
6:  public class DemoPerformersPolicy
     extends DefaultPerformersPolicy {
7:
8:    public String toString() {
9:      return "DemoPerformersPolicy()";
10:   }
11:
12:   public String getName() {
13:     return "DemoPerformersPolicy";
14:   }
15:
16:   public String getDescription() {
17:     return "quite simple policy implementation";
18:   }
19:
20:   public WfUserAssignment
21:   calculateAssignment(WfTaskInstance taskInstance,
         WfUser[] permittedUsers) throws WfException {
22:     WfUserAssignment userAssignment =
23:     super.calculateAssignment(taskInstance, permittedUsers);
24:
25:     WfUser[] users = userAssignment.getUsers();
```

```
26:     WfUser[] result = new WfUser[users.length-1];
27:     if (result.length < 1) {
28:       result = users;
29:     } else {
30:       System.arraycopy(users, 0, result, 0, result.length);
31:     }
32:     return new WfUserAssignment(result, false);
33:   }
34: }
```

Example 5.13. Invoking a performer policy

Line 1 - 4: Your package and the packages to import.

Line 6: You subclass `DefaultPerformersPolicy` for convenience.

Line 12 -14: Return the name of the policy.

Line 16 - 18: Return a description of the policy.

Line 20 - 33: The most important method which is called by the workflow server.

Line 20- 21: On call, the workflow server passes a `WfTaskInstance` and the `WfUsers` to the method. WfUsers contains all users which are allowed to accept the task.

Line 22 -23: At first you call the method `calculateAssignment` method of the super class, because the aim of this example policy is to modify the default result.

Line 25: Prepare the manipulation of the result by getting the `WfUser` from the `WfUserAssignment`.

Line 26: Prepare a new `WfUser` array which should keep the resulting users. Remember, you only want to get rid of the last user, so the length of the array is `users.length-1`.

Line 27 - 29: If the result contains no user, this result is returned.

Line 30: Otherwise, all users but the last are copied from the default result array to the returned array.

Line 32: The result array is returned to the workflow server. The second parameter determines that the selected user is not forced to accept the task.

5.7 Programming Clients

The *CoreMedia Workflow* comes with a workflow client integrated in the *Site Manager* and *Studio*. If you want to implement your own client, for example to trigger external events into the workflows or the query workflow state for reports etc, the *Unified API* provides the `WorkflowRepository`. In order to create a workflow client, use a code like the following:

```
CapConnection connection =
  Cap.connect("http://localhost:40180/ior" +
  "?useworkflow=true", "admin", "admin");
try {
  WorklistRepository r = connection.getWorkflowRepository();
  // ... work on the repository ...
} finally {
  connection.close();
}
```

Example 5.14. Create a workflow client

Remote action handlers

A remote action handler is responsible for executing a user tasks client actions on behalf of the clients user.

- Handlers must implement the interface `RemoteActionHandler`.
- A handler receives the command and parameters to process.
- It has to return an `ActionResult`.

A client action is the result of one of the following client calls to the server:

- `Task.accept()`
- `Task.complete()`
- `Task.retry()`

The client call is blocked at least until all client actions have been handled.

NOTE

Never implement client actions requiring any user interaction by a remote action handler:

- They will block server transactions for an undefined time and will eventually time out.
- They won't work in a synchronous client.

5.8 Spring in the Workflow Server

You can use the Spring framework to make Java beans available to your customized workflow actions and expressions. The workflow server's Spring application context is exposed by the built-in manager named springcontext of type `com.coremedia.workflow.common.util.SpringContextManager`. It can be used by custom actions and expressions that retrieve the Spring application context from the manager.

5.8.1 Using Spring Beans

The Spring context is loaded at startup time and is shut down when the server is shut down. The Spring configuration can refer to the *Workflow Server's Unified API* connection, using the same name ["connection"] as in the *CAE*. An action or expression may implement the interface `com.coremedia.cap.workflow.plugin.CapConnectionAware`. If it does so, the connection is injected before the action is executed or the expression is evaluated for the first time.

In order to use a bean in your action or expression proceed as follows:

1. Use the common Spring ways to add your custom configuration to the workflow server's Spring application context.
2. Let your customized actions or expressions extend `com.coremedia.workflow.common.util.SpringAwareAction` or `com.coremedia.workflow.common.util.SpringAwareExpression` respectively.
3. Get the bean inside your customized code using the `getBean()` method, for example use

```
protected ActionResult execute(Process process) {
InboxFactory inboxes = (InboxFactory) getBean("inboxFactory");
…
}
```

The configured beans may implement the common Spring ways to receive life cycle events from the workflow server's application context. Additionally, the beans may implement the interface `com.coremedia.workflow.common.util.WorkflowServerLifecycleAware`, if they want to initiate asynchronous operations. Such operations may start after the method `workflowServerStart()` is called and must be completed before the method `workflowServerStop()` returns. Only singleton beans receive these callbacks and only if they implement the given interface.

5.9 Pitfalls of Implemented Classes

A workflow definition is stored in the database as a stream of serialized objects. That's why your own workflow beans have to stick to the following rules:

- Avoid incompatible changes to classes which are already in use by a workflow definition.
- Consider using a serial UID for all your classes from the start on.
- Serialize and deserialize the object graph manually (see Oracle JDK Serialization documentation for details). This gives you the most control, but the most work, too.
- Use the `workflowconverter` tool to reparse and rebuild definitions which are not deserializable anymore.
- New versions of a workflow bean *must* be compatible with all uploaded XML definitions.
- New configuration options can be added as long as they are backwards compatible with the old ones.
- Additional objects, such as workflow variables, introduced with a new bean and definition will never be available in any old instance.
- If semantics have to be changed you should consider writing a new bean and keeping the old one.

CAUTION

The semantics must work in any still existing instances of older workflow definitions.

Since the workflow beans of a given definition are shared by all the definitions instances:

- No workflow bean must store any state in a local variable. State is always restricted to an instances context.
- No workflow bean must cache any objects requested from the server or client instances such as ObjectRepository, DirectoryService, *CoreMedia Content Management Server* Session etc. These objects may carry session specific information that is only valid to the current bean invocation.
- Every bean must be reentrant, that means is must be thread safe and never use nested synchronization.

To circumvent some of the mentioned problems, you might want to use the feature to upload a JAR together with a workflow definition. This separates the classes for each workflow definition. But when you update the JAR file for an existing workflow definition, the same problems occur as when loading the classes from the workflow servers classpath.

Additionally, references from the classes *inside* the JAR to classes *outside* of the JAR file are likely to cause problems. It might seem, that packaging all classes that are referenced by the customized workflow classes into one huge JAR file is a solution. But consequently, you would have to package the transitive closure of your workflow classes into that one JAR. That may not be feasible. It's better to document the dependencies of the customized workflow classes and to keep care that they are always fulfilled when running the *Workflow Server*.

6. Reference

In this chapter you will find the XML workflow reference and unabridged code examples from the previous chapters.

6.1 Configuration Reference

In addition to the general configuration possibilities as described in the [Developer Manual] you can configure the workflow system with the following files:

- `workflowclient.properties`
  This file contains the general configuration of a workflow client.
- `capclient.properties`
  This file contains the configuration how the *Workflow Server* connects to the *Content Management Server*
- `sql.properties`
  This file contains the database configuration for the *Workflow Server*. The configured database must match the one of the *Content Management Server*. See the *Content Server Manual* for details.
- `editor.xml`
  Besides the *Site Manager* content client GUI configuration, this file defines the appearance of the *Site Manager*'s workflow window.

Note that *Workflow Server* properties can be overridden in the file `application.properties` or via JNDI. Configuration via JNDI enables you to leave the WAR files untouched and for example define properties in the `context.xml` of the Tomcat installation. For details please consult the [Developer Manual].

6.1.1 Configuration of Workflow Client Properties

The file `workflowclient.properties` defines configuration options for user management for the workflow client, remote action handlers and parameters for the connection to the *CoreMedia Workflow Server*.

6.1.2 Configuration of Workflow Server Properties

All configuration properties are bundled in the Deployment Manual (Chapter 3, *CoreMedia Properties Overview* in *Deployment Manual*). The workflow properties contain general configuration of the *Workflow Server* such as the mapping of LDAP groups to the workflow groups. The following link references the properties that are relevant for the Content Feeder:

- Table 3.33, "Workflow Server Properties" in *Deployment Manual* contains properties for the configuration of the *Workflow Server*.

6.1.3 Managed Properties

In this section, you will find tables with all properties and actions manageable via JMX. The entries below the `JMImplementation` key display information on the JMX implementation which will not be described here.

NOTE

The information contained in the *Statistics* section are not described, because this information can only be interpreted by trained CoreMedia consultants who are familiar with the inner workings of the CoreMedia components.

Workflow Server Properties

Attribute	Type	Description
`AppDesc`	Read-only	Description of the CoreMedia System.
`HostInfo`	Read-only	Installation host of the *Workflow Server*
`JavaClasspath`	Read-only	Classpath used by the current Java installation
`JavaInstDir`	Read-only	Installation directory of the used Java.
`JvmInfo`	Read-only	Information about the used JVM.
`JvmProcessInfo`	Read-only	Java process information, the number of threads, free memory, used memory, total memory.
`LogActions`	Read/Write	Enable ("true") logging of actions.
`LogClientActions`	Read/Write	Enable ("true") logging of client actions.
`LogContentManager`	Read/Write	Enable ("true") logging of the ContentManager.
`LogErrorLog`	Read/Write	Enable ("true") logging of ErrorLog.
`LogExpressions`	Read/Write	Enable ("true") logging of expressions.

Attribute	Type	Description
`LogPersistenceAdapter`	Read/Write	Enable ("true") logging of the PersistenceAdapter.
`LogPolicies`	Read/Write	Enable ("true") logging of policies.
`LogProcessSweeper`	Read/Write	Enable ("true") logging of the ProcessSweeper.
`LogSignals`	Read/Write	Enable ("true") logging of signals.
`LogTimers`	Read/Write	Enable ("true") logging of timers.
`LogTransactions`	Read/Write	Enable ("true") logging of transactions.
`LongActionsMax`	Read/Write	The maximum number of concurrently executed long actions.
`FinalActionsMax`	Read/Write	The maximum number of concurrently executed final actions. The default value is derived from configuration property `workflow.server.final-actions-max`, which is described in Table 3.33, "Workflow Server Properties" in *Deployment Manual*
`FinalActionsRetryEnabled`	Read/Write	If the execution of workflow final actions is retried after a `RetryableActionException` was thrown. Set this to `false` to completely disable retries after these exceptions. The default value is derived from configuration property `workflow.server.retry-final-actions.enabled`, which is described in Table 3.33, "Workflow Server Properties" in *Deployment Manual*.
`OsInfo`	Read-only	Information about the operating system of the *Workflow Server* host.
`SessionReaperTimeout`	Read/Write	The interval in seconds between checks for inactive sessions (see SessionTimeout).
`SessionTimeout`	Read/Write	The time in seconds before an inactive session is closed.

Attribute	Type	Description
`TxIdleTimeout`	Read/Write	The time in seconds before an idle database connection is closed.
`TxMax`	Read/Write	The maximum number of database connections.

Table 6.1. Managed Workflow Server properties

Workflow Server Operations

Operation	Attribute	Description
clearCaches		Clear the caches of the Workflow Server.

Table 6.2. Workflow Server operations properties

6.2 XML Element Reference

The order of the elements in the workflow definition is not relevant except for the Action [118] and the Condition [125] elements. The reason for this is obvious, as you have to control the order of the actions and a condition that is comparing values depends on an ordering, too. Mostly NMTOKEN is used instead of CDATA as the content model for the attributes. This restrictive policy avoids escaping of names.

This chapter describes the workflow definition XML file format. You will find two kinds of items described here:

- Parameter entities (headline printed in ***bold italics***)
  Parameter entities constitute rules for the XML grammar or standard sets of attributes. Parameter entities are reused in various places to shorten the definition of XML elements.
- XML elements (headline printed in **bold**)
  XML elements describe the actual parts of a workflow description.

Action attributes

Grammar:

You will find the attributes of the actions described for each action later in this chapter.

BooleanExpression

Definition: Equal [128] | NotEqual [150] | Greater [138] | GreaterEqual [139] | Less [148] | LessEqual [148] | And [120] | Or [150] | Implies [141] | Not [149] | ForAll [132] | Exists [129] | Let [148] | Get [136] | Read [155] | Length [147] | IsEmpty [144] | NotEmpty [149] | IsFolder [145] | IsDocument [142] | IsDocumentVersion [143] | IsExpired [144] | IsEnabled

The BooleanExpression parameter entity is used to define a subset of all available expressions which evaluate to a Boolean value.

Expression

Definition: Expression [130] | Equal [128] | NotEqual [150] | Greater [138] | GreaterEqual [139] | Less [148] | LessEqual [148] | And [120] | Or [150] | Implies [141] | Not [149] | ForAll [132] | Exists [129] | Let [148] | Get [136] | Read [155] | Length [147] | IsEmpty [144] | NotEmpty [149] | IsFolder [145] | IsDocument [142] | IsDocumentVersion [143] | IsExpired [144] | AddLatestVersion [119] | Value [117]: Blob | Boolean | Content | ContentType | Date | Document | Folder | Group | Integer | String | Timer | User

The Expression parameter entity is used to define all available expressions. You can use the predefined expressions listed above or implement your own expressions, using the Expression [130] element.

FlowControlTask

Definition: Choice [124] | Fork [133] | If [140] | Join [146] | JoinSubprocess [146] | ForkSubprocess [134]| Switch [161]

FlowControlTasks define the flow of control in a workflow process.

This is just an abstract definition, only concrete FlowControl tasks may be used in a valid workflow definition.

Note: A FlowControlTask may not be final.

Task

Definition: AutomatedTask [121] | UserTask [164] | FlowControlTask [117]: Choice | Fork | If | Join | JoinSubprocess | ForkSubprocess | Switch

Tasks define the steps a workflow process must complete. A task is identified by its name. Like a process is a template for concrete process instances, a task is a template for concrete task instances. Tasks refer to each others by the name(s) of their Successor [160](s). Each task must either have at least one successor or be final.

The description of the task is a human readable explanation about what the task does. It may be localized by the editor or used as a key for localization in the *Site Manager*.

Tasks which finish a workflow process are declared final. There has to be at least one task in a process definition which is final. Only UserTasks and AutomatedTasks can be final.

Variables in the task scope define the local state of task instances. This does not restrict the visibility of the variables. A variable in a task may always be referred to from other tasks by prefixing the variable name with the task name and a dot.

There are nine task types:

- An AutomatedTask [121] is executed automatically.
- An UserTask [164] has to be carried out by a user.
- The other task types are used to control the flow of execution of tasks.

Value

Definition: Blob [123] | Boolean [123] | Resource [157] | ContentType [126] | Date [126] | Document [127] | Folder [132] | Group [139] | Integer [142] | String [160] | Section 4.1.11, “Timers” [56] | User [163]

A Value represents one or many values of a variable. A Value element is used to initialize a variable or to be evaluated in expressions (see Example 6.1, “Example of the variable usage” [118]).

```
<Variable name="publicationSuccessful" type="Boolean">
  <Boolean value="false"/>
</Variable>
<AggregationVariable name="success" type="Boolean">
  <Boolean value="true"/>
  <Boolean value="false"/>
</AggregationVariable>
<Condition>
    <Equal>
        <Boolean value="true"/>
        <Get variable="success" index="1"/>
    </Equal>
</Condition>
```

Example 6.1. Example of the variable usage

Boolean

Definition: true | false

Definition of a Boolean XML attribute type.

varies

Definition: Entity for tagging varying parts of the DTD.

Action

- *Grammar:* (Condition, Property)

An action is external code which may be called to customize the processing of the workflow engine (see Section 5.3, "Programming Actions" [86] for implementing own actions).

You can either give the full qualified name of your own action class which must be an implementation of interface `com.coremedia.workflow.WfAction` or an unqualified class name which will be searched for in the package `com.coremedia.workflow.common.actions`.

A predefined Action or one subclassed from `AbstractAction`/`AbstractClientAction` may contain a Condition element which serves as a "guard" for the action code (see the example below). Only if the condition is satisfied, the code is executed, otherwise nothing happens.

The following actions are supplied with the workflow engine by default: ArchiveProcess, ApproveResource, CheckInDocument, CheckOutDocument, CopyResource, CreateDocument, CreateFolder, DeleteResource, DisapproveResource, EnableTimer, ExcludePerformer, ExcludeUser, DisableTimer, MoveResource, OpenDocument, PreferPerformer, PublishResources, RegisterPendingProcess, RenameResource, SaveDocument, UncheckOutDocument, UndeleteResource.

The predefined actions use some of the Action attributes defined above.

Note: The Property [155] child element is valid for the CreateDocument action only.

Attribute	Type	Default	Description
varies			additional parameters according to the implementation of the action class

Table 6.3. Attributes of Action element

```
<UserTask name="TestActionGuard" successor="final">
.
.
  <Action class="EnableTimer" timerVariable="TimeVariable">
    <Condition>
      <Equal>
        <Read variable="document" property="_name"/>
        <String value="Article"/>
      </Equal>
    </Condition>
  </Action>
.
.
</UserTask>
```

Example 6.2. Action with a Guard used in a UserTask

AddLatestVersion

Grammar: ([Expression]*)

An `AddLatestVersion` expression adds the latest version to a document value or to each member of an aggregate containing only documents. If a document already contains version information, the value is handed through. Otherwise, the *Content Management Server* is queried for the latest version of the document and the document version is added.

No attributes.

AggregationVariable

Grammar: ([Value [117]]*)

In contrast to a variable, whose value is one value, an AggregationVariable [119] may have a list of values as its value. See Variable for details.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	The name of the variable

Attribute	Type	Default	Description
type	NMTOKEN	#REQUIRED	The type of the variable, see Value
readOnly	[Boolean [118]]	"false"	Defines whether it is forbidden to modify the variable
static	[Boolean [118]]	"false"	Defines whether the variable is initialized only once

Table 6.4. Attributes of the AggregationVariable element.

```
<Workflow>
   <Process name="AggregationExample" startTask="Start">
     <AggregationVariable name="StringTest" type="String">
        <String value="World"/>
        <String value="Hello"/>
     </AggregationVariable>
     .
     .
   </Process>
<Workflow>
```

Example 6.3. Example of an aggregation variable

And

Grammar: [[Expression]*]

An `And` expression evaluates to the conjunction of its subexpressions, all of which must return Boolean values. The subexpressions are evaluated in a "short-circuit" fashion, that is, they are evaluated top down until the first subexpression evaluates to "false" or all subexpressions have evaluated to "true". This helps to avoid exceptions during the computation, for example when checking the type of a document before accessing a property of the document of that expected type.

```
<Variable name="Comment" type="String"/>
<Assignment>
   <Writes variable="Comment"/>
</Assignment>

<UserTask name="AndTest" successor="theNext">
   <PreCondition>
      <And>
         <Equal>
            <Get variable="OWNER_"/>
            <User value="0"/>
         </Equal>
         <Equal>
```

```
            <Get variable="Comment"/>
            <String value="42"/>
          </Equal>
      </And>
   </Precondition>
   <!-- Code -->
</UserTask>
```

Example 6.4. Example of an And element.

Assign

Grammar: (Expression [116])

Assign transfers a value which is defined by the expression into a variable in the initial client view of the subprocess. For an XML example see Example 6.22, “Example of a ForkSubprocess task” [135].

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	name of the variable in the subprocess

Table 6.5. Attribute of the Assign element

AutomatedTask

- *Grammar:* ([Variable [167] | AggregationVariable [119]]*, Action [118]*, Guard [140]?, PreCondition [152]*, PostCondition [152])

An AutomatedTask [121] is executed automatically by the workflow engine. It performs some automated action on the *Content Management Server* content or on other third-party systems or internal actions. The Action [118] of an automated task are used to customize the processing of the workflow engine. If [140] more than one Action [118] is provided, the actions are executed in the order in which they are specified.

A PreCondition [152] defines requirements which have to be fulfilled before the actions of the automated task are executed. A PostCondition [152] defines requirements which have to be fulfilled after the action has been executed. If [140] more than one PreCondition [152] or PostCondition [152] are provided, then the conditions are evaluated in the order they are defined. The result of such an evaluation operation is equivalent to specifying an 'and' expression with an ordered set of expressions.

A Guard [140] defines an expression, which activates and executes the task as soon as the expression evaluates to true. The expression is evaluated on state changes of process- or task instances in the Workflow [168] Server and content or name changes of referred resources in the *Content Management Server*. Note that changes to other, external entities do not trigger reevaluation of a guard.

A successor must be given if and only if the task is not final.

Note: An Section 4.1.4.3, "Automated Tasks" [45] does not allow you to specify Rights [158], Performer [151], and Client [125]. This is restricted to UserTask [164] elements which interact with the users of the *CoreMedia Workflow Server*.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the task
description	CDATA	#IMPLIED	the textual description of the task
final	[Boolean [118]]	"false"	Defines whether the task is the final task to execute
successor	NMTOKEN	#IMPLIED	Defines the next task to execute after the automated task has been completed

varies

Table 6.6. Attributes of the Automated Task element

```
<Variable name="document" type="Document"/>
<Assignment>
   <Writes variable="document"/>
</Assignment>
<AutomatedTask name="automatic" successor="final">
   <Action class="CheckInDocument" documentVariable="document"/>
</AutomatedTask>
```

Example 6.5. Example of an AutomatedTask

Assignment

- *Grammar:* [[Reads [156] | Writes [168]]*, Validator [166]]

An Assignment element determines that a variable is 'important' to a task or process instance and need to be shown. It can or has to be modified by a user or an external process. Thus, it defines a view on the variables.

With Reads [156] and Writes [168] the variables are specified. The modifications of the variables may be validated by Validators.

Processes have two variants of Assignment specifications, the InitialAssignment which is valid as long as the process instance is not started and the Assignment for all other instance states. This way it is possible to set initial arguments for a process instance which cannot be changed after the instance is started.

No attributes.

```
<Workflow>
   <Process name="ClientExample" startTask="TheFirst">
      <Variable name="Resource" type="Document"/>
      <Variable name="Comment" type="String"/>
      <UserTask name="TheFirst" successor="TheEnd">
         <Assignment>
            <Reads variable="Resource" contentEditable="true"/>
            <Writes variable="Comment"/>
         </Assignment>
         <!-- Code -->
      </UserTask>
      <!-- Code -->
   </Process>
</Workflow>
```

Example 6.6. Example of an Assignment task

Blob

Grammar: EMPTY

The Blob element is used to specify a single constant blob value within expressions or variable initializers.

Attribute	Type	Default	Description
value	CDATA	#IMPLIED	the blob value in bytes
mimeType	CDATA	#REQUIRED	the blob's MimeType

Table 6.7. Attribute of the Blob element

```
<Variable name="Logical" type="Blob">
   <Blob value="Some text..." mimeType="text/plain"/>
</Variable>
```

Example 6.7. Example of a Blob variable

Boolean

Grammar: EMPTY

The Boolean element is used to specify a single constant Boolean value within expressions or variable initializers.

Attribute	Type	Default	Description
value	[Boolean [118]]	#REQUIRED	the Boolean value ["true" or "false"]

Table 6.8. Attribute of the Boolean element

```
<Variable name="Logical" type="Boolean">
   <Boolean value="true"/>
</Variable>
```

Example 6.8. Example of a Boolean variable

Case

Grammar: `(%BooleanExpression;)`

A case extends a condition by defining a successor to be activated if the condition's expression evaluates to true. A 'case' condition may be based on the state of workflow variables, the content of documents from the *Content Management Server* or the external state of third-party products. For an example see Switch [161].

Attribute	Type	Default	Description
name	NMTOKEN	#IMPLIED	The name of the expression.
description	CDATA	#IMPLIED	A textual description of the expression.
successor	NMTOKEN	#REQUIRED	The successor which should be activated if the condition's expression evaluates to true.

Table 6.9. Attributes of the Case element

Choice

Grammar: [[Variable [167] | AggregationVariable [119]]*, Successor [160]+]

A Choice task branches the flow of tasks into two or more successors which must be UserTasks. So it is an implicit choice. One of these successor tasks can be accepted

and executed by a user. As this happens the other Successor [160] tasks are withdrawn from any offer list and reset as if they haven't been started at all.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the task
description	CDATA	#IMPLIED	the textual description of the task

Table 6.10. Attributes of the Choice element.

```
<UserTask name="TheTaskBefore" successor="ChoiceExample">
   <!-- Code -->
</UserTask>
<Choice name="ChoiceExample">
   <Successor name="FirstChoice"/>
   <Successor name="SecondChoice"/>
</Choice>
<UserTask name="FirstChoice" successor="final">
   <!-- Code -->
</UserTask>
<UserTask name="SecondChoice" successor="final">
   <!-- Code -->
</UserTask>
```

Example 6.9. Example of a Choice element

Client

Deprecated. See **Assignment** instead.

Condition

Grammar: (Expression [116])

A condition defines an expression that must evaluate to a Boolean value. It may be based on the state of workflow variables, the content of documents from the *Content Management Server* or the external state of third-party products. A condition is defined based on an expression which may be formed from nested subexpressions.

Attribute	Type	Default	Description
name	NMTOKEN	#IMPLIED	the name of the condition
description	CDATA	#IMPLIED	the textual description of the condition

Table 6.11. Attributes of the Condition element

```
<Variable name="Article" type="Document"/>
<Assignment>
   <Writes variable="Article"/>
</Assignment>

<UserTask name="AndTest" successor="theNext">
  <EntryAction class="CheckOutDocument" documentVariable="Article">
    <Condition>
      <NotEmpty variable="Article"/>
    </Condition>
  </EntryAction>
 <!-- Code -->
</UserTask>
```

Example 6.10. Example of a Condition element. It is checked whether the document variable is null or not.

ContentType

Grammar: EMPTY

The ContentType element is used to specify a single constant content type within expressions or variable initializers.

Attribute	Type	Default	Description
value	NMTOKEN	#REQUIRED	the name of the content type

Table 6.12. Attribute of the ContentType element

```
<Variable name="Type" type="ContentType">
   <ContentType value="Article"/>
</Variable>
```

Example 6.11. Example of a ContentType variable

Date

Grammar: EMPTY

The Date element is used to specify a single constant date value within expressions or variable initializers.

Attribute	Type	Default	Description
value	CDATA	#REQUIRED	the date in the format dd.MM.yyyy hh:mm

Table 6.13. Attribute of the Date element

```
<Variable name="Time" type="Date">
   <Date value="10.11.2002 13:00"/>
</Variable>
```

Example 6.12. Example of a Date variable

Document

Grammar: EMPTY

The Document element is used to specify a single constant document within expressions or variable initializers. It is not useful to define a fixed document ID in the workflow definition. Either `path` or `value` should be specified.

Attribute	Type	Default	Description
path	NMTOKEN	#IMPLIED	The path of a document.
value	NMTOKEN	#IMPLIED	The ID of the document.
version	NMTOKEN	#IMPLIED	The version number of the document.

Table 6.14. Attributes of the Document element.

```
<Variable name="Article" type="Document">
   <Document value="10"/>
</Variable>
```

Example 6.13. Example of a Document variable.

Attribute	Type	Default	Description
value	NMTOKEN	#REQUIRED	the name of the content type

Table 6.15. Attribute of the DocumentType element

Else

Grammar: EMPTY

Else defines the successor of the If [140] task if the condition evaluates to false, see If [140] for details and an XML example.

Attribute	Type	Default	Description
successor	NMTOKEN	#REQUIRED	the name of the successor task for the "else" case

Table 6.16. Attribute of the Else element

EntryAction

Grammar: (Condition [125]?, Property [155]?)

EntryAction and ExitAction [130] elements are identical to Action [118] elements, see Action [118] and Action-Attributes [116] for details.

Attribute	Type	Default	Description
varies			additional parameters according to the implementation of the action class

Table 6.17. Attributes of EntryAction element

```
<Variable name="Article" type="Document"/>
<Assignment>
   <Writes variable="Article"/>
</Assignment>

<UserTask name="AndTest" successor="theNext">
   <EntryAction class="CheckOutDocument"
   documentVariable="Article" gui="false">
      <Condition>
         <NotEmpty variable="Article"/>
      </Condition>
   </EntryAction>
   <!-- Code -->
</UserTask>
```

Example 6.14. Example of an EntryAction which checks out a document

Equal

Grammar: ([Expression [116]], [Expression [116]])

An Equal expression contains exactly two subexpressions, which are both evaluated during the evaluation of the Equal expression. The expression evaluates to "true" if and only if the computed values of the subexpressions are equal.

Although an Equal expression may compare values of any type, this element makes sense only for values like integer, string, date, resource and timer values as defined in the workflow. Note that document references are considered equals only if they refer to the *same* document, that is, the document contents are not considered.

```
<Variable name="Comment" type="String"/>
<Assignment>
   <Writes variable="Comment"/>
</Assignment>

<UserTask name="AndTest" successor="TheNext">
   <Guard>
      <Equal>
         <Get variable="Comment"/>
         <String value="LetMeIn"/>
      </Equal>
   </Guard>
   <!-- Code -->
</UserTask>
```

Example 6.15. Example of an Equal expression

Exists

Grammar: [Expression [116]]

Exists is the counterpart to ForAll [132] and behaves similarly. It evaluates to true if any of the instances of the subexpression evaluate to "true". Evaluation is also short-circuited, that is, it stops as soon as a subexpression instance evaluates to "true".

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of a new variable that iterates over all members of the aggregate
aggregate	NMTOKEN	#REQUIRED	the name of an aggregate variable
index	NMTOKEN	#IMPLIED	the name of a new integer variable that is set to the current index in the aggregate during the iteration

Table 6.18. Attributes of the Exists element

```
<AggregationVariable name="Articles" type="Document"/>
<Assignment>
```

```
   <Writes variable="Articles"/>
</Assignment>

<UserTask name="AndTest" successor="TheNext">
   <Guard>
      <Exists variable="Element" aggregate="Articles">
         <Equal>
            <String value="Sports"/>
            <Read variable="Element" property="Topic"/>
         </Equal>
      </Exists>
   </Guard>
   <!-- Code -->
</UserTask>
```

Example 6.16. Example of an Exists expression which checks if one of the documents in the variable Articles has the entry Sports in Topics

ExitAction

Grammar: (Condition [125]?, Property [155]?)

ExitAction and EntryAction [128] elements are identical to Action [118] elements, see Action [118] for details.

Attribute	Type	Default	Description
varies			additional parameters according to the implementation of the action class

Table 6.19. Attributes of the ExitAction element

```
<Variable name="Article" type="Document"/>
<Assignment>
   <Writes variable="Article"/>
</Assignment>

<UserTask name="AndTest" successor="theNext">
   <EntryAction class="CheckOutDocument"
   documentVariable="Article" gui="false">
      <Condition>
         <NotEmpty variable="Article"/>
      </Condition>
   </EntryAction>
   <!-- Code -->
</UserTask>
```

Example 6.17. Example of an Exit Action which checks whether the document is null or not

Expression

Grammar: ([Expression [116]]*)

You can implement your own expressions (see Section 5.4, "Programming Expressions" [94]). Custom expressions must implement the interface `com.coremedia.workflow.WfExpression` or `com.coremedia.workflow.WfBooleanExpression`.

Attribute	Type	Default	Description
class	NMTOKEN	#REQUIRED	the name of the expression class
varies			

Table 6.20. Attributes of the Expression element

```
<Variable name="comment" type="String">
   <String value="TestString"/>
</Variable>
<If name="One">
   <Condition>
      <Less>
         <Expression
       class="com.coremedia.examples.expression.DemoExpression"/>
         <Get variable="comment"/>
      </Less>
   </Condition>
   <Then successor="True"/>
   <Else successor="False"/>
</If>
```

Example 6.18. Example of an Expression element

FinalAction

The element `FinalAction` defines a final action that is executed after a process was completed or aborted. Its `class` attribute specifies the fully qualified name of a custom final action, which must be an implementation of `com.coremedia.cap.workflow.plugin.FinalAction`. Section 4.4.2, "Predefined FinalAction Classes" [79] lists predefined classes that can also be used.

Attribute	Type	Default	Description
class	NMTOKEN	#REQUIRED	the name of the final action class
varies			additional parameters according to the implementation of the class

Table 6.21. Attributes of the FinalAction element

Folder

Grammar: EMPTY

The Folder element is used to specify a single constant folder within expressions or variable initializers. It is not useful to define a fixed folder ID in the workflow definition. Either `value` or `path` must be selected.

Attribute	Type	Default	Description
value	NMTOKEN	#IMPLIED	The ID of the folder.
path	NMTOKEN	#IMPLIED	The path of the folder.

Table 6.22. Attributes of the Folder element.

```
<Variable name="RootFolder" type="Folder">
   <Folder value="1"/>
</Variable>
```

Example 6.19. Example of a Folder variable

ForAll

Grammar: (Expression [116])

A ForAll expression checks its Boolean subexpression for all members of the value of the "aggregate" AggregationVariable [119] and evaluates to "true" if all instances of the subexpression evaluate to "true". The subexpression can (and should) contain a Get [136] expression with the variable name that evaluates to the n-th value in the aggregate. The logical "and" is short-circuited in the sense that evaluation is done in the order of the aggregate's elements and stops as soon as the subexpression evaluates to "false". The optional index variable evaluates to an IntegerValue representing the index of the current element in the aggregate and can be used, for example to access the member at the same index in another aggregate.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of a new variable that iterates over all members of the aggregate
aggregate	NMTOKEN	#REQUIRED	the name of an aggregation variable

Attribute	Type	Default	Description
index	NMTOKEN	#IMPLIED	the name of a new integer variable that is set to the current index in the aggregate during the iteration

Table 6.23. Attributes of the ForAll element

```
<AggregationVariable name="Articles" type="Document"/>
<Assignment>
   <Writes variable="Articles"/>
</Assignment>

<AutomatedTask name="Approve" successor="TheNext">
   <Action class="ApproveResource" resourceVariable="Articles">
      <ForAll variable="Element" aggregate="Articles">
         <Not>
            <Read variable="Element" property="isCheckedOut_"/>
         </Not>
      </ForAll>
    </Action>
   <!-- Code -->
</AutomatedTask>
```

Example 6.20. Example of a ForAll element which checks if all documents are checked in before approving them

Fork

Grammar: ([Variable [167] | AggregationVariable [119]]*, Successor [160]+)

A Fork task forks the flow of tasks into two or more Successors to perform execution in parallel. All forked tasks must be joined together by a Join [146] task.

```
<!-- Code -->
<Fork name="Parallel" description="Fork tasks">
    <Successor name="FirstParallel"/>
    <Successor name="SecondParallel"/>
</Fork>
<AutomatedTask name="FirstParallel" successor="Together">
    <!-- Code -->
</AutomatedTask>
<UserTask name="SecondParallel" successor="Together">
    <!-- Code -->
</UserTask>
<Join name="Together" successor="Next">
    <Predecessor name="FirstParallel"/>
    <Predecessor name="SecondParallel"/>
```

```
</Join>
<!-- Code -->
```

Example 6.21. Example of a Fork task

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the task
description	CDATA	#IMPLIED	the textual description of the task

Table 6.24. Attributes of the Fork element

ForkSubprocess

- *Grammar:* [(Variable [167] | AggregationVariable [119])*, Parameters [151]]

The ForkSubprocess task starts a separate workflow process, which is referenced by its name, from the current process.

If `detached` is set to true, the forked subprocess has no relationship to its parent process. If set to false, which is the default, a suspend, abort or resume on the parent process suspends, aborts or resumes the forked subprocess, too.

The forked subprocess may be parametrized via Parameters [151] child elements.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the task
description	CDATA	#IMPLIED	the textual description of the task
subprocess	NMTOKEN	#REQUIRED	the name of the sub process to start
`subprocessVariable`	NMTOKEN	#IMPLIED	the name of the sub process to start, defined via a string variable. The name of the string variable is set with `subprocessVariable`. `subprocess` or `subprocessVariable` must be defined.

Attribute	Type	Default	Description
			If both are set, `subprocess` has precedence.
`ownerVariable`	NMTOKEN	#IMPLIED	the owner of the sub process is by default the owner of the parent process. Using `ownerVariable` a user variable can be defined. If this variable contains a valid user id at runtime, this user becomes the owner of the sub process.
successor	NMTOKEN	#REQUIRED	the name of the next task to execute after the subprocess has been started
detached	[Boolean [118]]	"false"	If set to "false", the subprocess may be joined and it is affected by suspend, abort and resume operations on the original process.

Table 6.25. Attributes of the ForkSubprocess element

```
<Workflow>
    <Process name="FirstWF" startTask="Fork">
    <Variable name="Comment" type="String"/>
    <Assignment>
        <Writes variable="Comment"/>
    </Assignment>
    <!-- Code -->
    <ForkSubprocess name="Fork" subprocess="SecondWF"
                            successor="Wait" detached="false">
        <Parameters>
            <Assign variable="SubComment">
                <Get variable="Comment"/>
            </Assign>
        </Parameters>
    </ForkSubprocess>
    <!-- Code -->
    <JoinSubprocess name="Wait" forkTask="SecondWF"
    successor="Final"/>
    <AutomatedTask name="Final" final="true"/>
```

```
    </Process>
</Workflow>

<!-- NEW FILE -->

<Workflow>
    <Process name="SecondWF" startTask="FirstOne">
        <Variable name="SubComment" type="String/>
        <InitialAssignment>
            <Writes variable="SubComment"/>
        </InitialAssignment>
        <!-- Code -->
    </Process>
</Workflow>
```

Example 6.22. Example of a ForkSubprocess task

Get

Grammar: EMPTY

Get evaluates to the value of a variable. The variable can be a workflow variable (normal or aggregate) or an expression-local variable (see Let, ForAll, Exists). If the variable is an AggregationVariable [119], an index can be given either as an integer constant or an integer variable in the index attribute. For aggregation variables the Get expression evaluates to the value at this index in the aggregation, if an index is given, or to the entire aggregate otherwise.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the variable that contains the result value
index	NMTOKEN	#IMPLIED	the optional index into an aggregation variable, given by a variable name or a constant value

Table 6.26. Attributes of the Get element

```
<Variable name="Comment" type="String"/>
<Assignment>
   <Writes variable="Comment"/>
</Assignment>

<If name="IfTask">
   <Condition>
      <Equal>
         <Get variable="Comment"/>
         <String value="42"/>
      </Equal>
   </Condition>
   <Then successor="Task1"/>
```

```
   <Else successor="Task2"/>
</If>
```

Example 6.23. Example of a Get element

Grant

Grammar: EMPTY

Grant authorizes users or groups to perform actions on the process or task instance they are specified in. Grant is only defined for the predefined `ACLRightsPolicy`. If you implement own policies, you may parameterize the policy as you want.

One of 'user', 'group', or 'variable' must be set to specify the subject who is authorized to do actions. If you use 'group' or 'user' the optional 'domain' might be used in addition.

If the attribute 'variable' is set, then the indicated variable is read at runtime. If the variable contains a user, the grant applies to that user. If it contains a group, the grant applies to all direct or indirect members of that group. If it contains a list of users or groups, it applies to all of these.

Rights specified using variables precede user rights, which again precede group rights. Within each category, revokes precede grants.

The 'rights' are a comma-separated list of names for operations, which may be performed. The actions, defined in the `WfRightsPolicy` interface are:

read, write for process and task instances; create, start, suspend, resume, abort for process instances; accept, reject, assign, complete, delegate, cancel, skip, retry for task instances

Attribute	Type	Default	Description
user	NMTOKEN	#IMPLIED	the name of a user or the user ID of a user
group	NMTOKEN	#IMPLIED	the name of a group or the group ID of a group
domain	NMTOKEN	#IMPLIED	The domain of a user or group. May be used if `group` or `user` is chosen.

Attribute	Type	Default	Description
variable	NMTOKEN	#IMPLIED	the name of a variable that stores a user or a group or a list of these
rights	CDATA	#REQUIRED	a comma-separated list of rights as specified above

Table 6.27. Attributes of the Grant element

```
<UserTask name="GrantExample" successor="TheNext">
    <Rights>
        <Grant group="composer-role"
         rights="accept, complete, read"/>
        <Grant user="demo1"
        rights="accept, complete, delegate, read"/>
    </Rights>
        <!-- Code -->
<UserTask>
```

Example 6.24. Example of a Grant element

Greater

Grammar: [[Expression [116]], [Expression [116]]]

A Greater expression contains exactly two subexpressions, which are both evaluated during the evaluation of the Greater expression. The expression evaluates to "true" if and only if the computed value of the first subexpression is greater than the value of the second subexpression.

Although a Greater expression may compare values of any type, this element makes sense only for integer, string, date and timer values as defined in the workflow.

```
<Variable name="Published" type="Date"/>
<Assignment>
   <Writes variable="Published"/>
</Assignment>

<If name="IfTask">
   <Condition>
      <Greater>
         <Get variable="Published"/>
         <Date value="31.12.2000 24:00"/>
      </Greater>
   </Condition>
   <Then successor="NewCentury"/>
   <Else successor="OldCentury"/>
</If>
```

Example 6.25. Example of a Greater expression

GreaterEqual

Grammar: ([Expression [116]], [Expression [116]])

A GreaterEqual expression contains exactly two subexpressions, which are both evaluated during the evaluation of the GreaterEqual expression. The expression evaluates to "true" if and only if the computed value of the first subexpression is greater than or equal to the value of the second subexpression.

Although a GreaterEqual expression may compare values of any type, this element makes sense only for integer, string, date and timer values.

```
<Variable name="Published" type="Date"/>
<Assignment>
   <Writes variable="Published"/>
</Assignment>

<If name="IfTask">
   <Condition>
      <GreaterEqual>
         <Get variable="Published"/>
         <Date value="31.12.2000 24:00"/>
      </GreaterEqual>
   </Condition>
   <Then successor="NewCenturyOrNewYearsEve"/>
   <Else successor="OldCentury"/>
</If>
```

Example 6.26. Example of a GreaterEqual expression

Group

Grammar: EMPTY

The Group element is used to specify a single constant group value within expressions, variable initializers or policies. Either 'value' or 'name' must be specified.

If you delete a group in the user administration, which you have used in the `Group` element of an uploaded workflow definition, its polices will fail.

Attribute	Type	Default	Description
name	NMTOKEN	#IMPLIED	Name of a group.
domain	NMTOKEN	#IMPLIED	Domain of the group. Might be used in addition to `name`.
value	NMTOKEN	#IMPLIED	numeric ID of a group.

Table 6.28. Attributes of the Group element.name

```
<Variable name="Writer" type="Group"/>
   <Group value="10"/>
</Variable>
```

Example 6.27. Example of a Group variable

Guard

Grammar: (Expression [116])

A Guard contains a Boolean expression, that defines a condition which must become true before a task is activated. See UserTask [164], AutomatedTask [121] and Condition [125] for details.

```
<AggregationVariable name="Articles" type="Document"/>
<Assignment>
   <Writes variable="Articles"/>
</Assignment>

<UserTask name="AndTest" successor="TheNext">
   <Guard>
      <Exists variable="Element" aggregate="Articles">
         <Equal>
            <String value="Sports"/>
            <Read variable="Element" property="Topic"/>
         </Equal>
      </Exists>
   </Guard>
   <!-- Code -->
</UserTask>
```

Example 6.28. Example of a Guard

If

Grammar: ([Variable [167] | AggregationVariable [119]]*, Condition [125], Then [161], Else [127])

An If task determines the successor task based on the result of a Condition [125]. A condition may be based on the state of workflow variables, the content of documents from a *Content Management Server* or the external state of third-party products.

If the condition evaluates to true, the successor of the Then [161] element is chosen, else the one of the Else [127] element. See Example 6.29, "Example of an If task" [141].

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the task
description	CDATA	#IMPLIED	the textual description of the task

Table 6.29. Attributes of the If element

```
<Variable name="Comment" type="String"/>
<Assignment>
   <Writes variable="Comment"/>
</Assignment>

<If name="IfTask">
   <Condition>
      <Equal>
         <Get variable="Comment"/>
         <String value="42"/>
      </Equal>
   </Condition>
   <Then successor="Task1"/>
   <Else successor="Task2"/>
</If>
```

Example 6.29. Example of an If task

Implies

Grammar: [[Expression [116]], [Expression [116]]*]

An Implies expression determines whether the first subexpression logically implies all remaining sub expressions. Thus, `<Implies>E1 E2 E3 ...<Implies/>` is equivalent to `<Or><Not>E1</Not> <AND>E2 E3 ...</And></Or>`. For the common case of two subexpressions, an Implies expression evaluates to "true" if and only if the first expression evaluates to "false" (without caring for the result of the second subexpressions) or both expressions evaluate to "true".

```
<Assignment>
    <Writes variable="changeSet" contentEditable="true"/>
    <Validator name="AllCheckedIn"
    description="all-checked-in-validator">
        <ForAll variable="change" aggregate="changeSet">
        <Implies>
            <And>
                <IsDocumentVersion variable="change"/>
                <Equal>
                    <Read variable="change" property="version_"/>
                    <Read variable="change"
                    property="latestVersion_"/>
                </Equal>
            </And>
            <Not>
                <Read variable="change" property="isCheckedOut_"/>
            </Not>
        </Implies>
        </ForAll>
    </Validator>
</Assignment>
```

Example 6.30. Example for an Implies expression

InitialAssignment

- Grammar: [[Reads [156] | Writes [168], Validator [166] Validator [166]]

An InitialAssignment element defines that a variable is 'important' to a process instance during the initial creation of the workflow before the workflow is started. This way it is

possible to set initial arguments for a process instance which cannot be changed after the instance is started.

With Reads [156] and Writes [168] the variables are specified. The variables can or have to be modified by a user or an external process. Thus, the InitialAssignment element defines a view on the variables. The modifications of the variables may be validated by Validators.

```
<Workflow>
    <Process name="InitialClientTest" startTask="TheFirst">
        <Variable name="Comment" type="String"/>
        <Variable name="Articles" type="Document"/>
        <InitalAssignment>
            <Reads variable="Comment"/>
            <Writes variable="Articles"/>
        </InitalAssignment>
        <!-- Code -->
    </Process>
</Workflow>
```

Example 6.31. Example of an InitialAssignment element

InitialClient

Deprecated. See InitialAssignment instead.

Integer

Grammar: EMPTY

The Integer element is used to specify a single constant integer value within expressions or variable initializers.

Attribute	Type	Default	Description
value	NMTOKEN	#REQUIRED	the integer value

Table 6.30. Attribute of the Integer element

```
<Variable name="Number" type="Integer">
   <Integer value="100"/>
</Variable>
```

Example 6.32. Example of an Integer Variable

IsDocument

Grammar: EMPTY

IsDocument queries whether a resource value contained in the variable, which is given as in Get [136], is a document with or without an explicit version.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the document variable
index	NMTOKEN	#IMPLIED	the optional index into an aggregation variable, given by a variable name or a constant value

Table 6.31. Attributes of the IsDocument element

```
<Variable name="Article" type="Resource"/>
<Assignment>
   <Writes variable="Article"/>
</Assignment>

<UserTask name="AndTest" successor="theNext">
 <EntryAction class="CheckOutDocument" documentVariable="Article">
      <Condition>
         <IsDocument variable="Article"/>
      </Condition>
   </EntryAction>
   <!-- Code -->
</UserTask>
```

Example 6.33. Example of an IsDocument expression

IsDocumentVersion

Grammar: EMPTY

IsDocumentVersion queries whether a resource value contained in the variable, which is given as in Get [136], is a document with an explicit version.

This is helpful because document variables may refer simply to a document or to a specific version of that document, so that processing may have to vary depending on the kind of value stored.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the document variable
index	NMTOKEN	#IMPLIED	the optional index into an aggregation variable,

Attribute	Type	Default	Description
			given by a variable name or a constant value

Table 6.32. Attributes of the IsDocumentVersion element

```
<Variable name="Article" type="Document"/>
<Assignment>
   <Writes variable="Article"/>
</Assignment>

<UserTask name="IsTest" successor="theNext">
 <EntryAction class="PublishResource" documentVariable="Article">
      <Condition>
         <IsDocumentVersion variable="Article"/>
      </Condition>
   </EntryAction>
   <!-- Code -->
</UserTask>
```

Example 6.34. Example of an IsDocumentVersion expression

IsEmpty

Grammar: EMPTY

IsEmpty evaluates to true if the value of the specified variable or resource property is "null". For an aggregation variable, length of zero is considered as empty, too. See Length [147] for details. For an XML example see PostCondition [152].

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the document variable
index	NMTOKEN	#IMPLIED	the optional index into an aggregation variable, given by a variable name or a constant value
property	NMTOKEN	#IMPLIED	the optional name of a resource property

Table 6.33. Attributes of the IsEmpty element

IsExpired

Grammar: EMPTY

IsExpired queries whether the timer given by the defined variable has expired.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the timer variable

Table 6.34. Attributes of the IsExpired element

```
<AutomatedTask name="StartTimer" description="SimplyStart"
  successor="Wait">
    <Variable name="waiting" type="Timer">
        <Timer value="100"/>
    </Variable>
    <Action class="EnableTimer" timerVariable="waiting"/>
</AutomatedTask>

<UserTask name="Wait" successor="Next">
    <Guard>
        <IsExpired variable="StartTimer.waiting"/>
    </Guard>
    <!-- Code -->
</UserTask>
```

Example 6.35. Example of an IsExpired expression

IsFolder

Grammar: EMPTY

IsFolder queries whether a resource value contained in the variable given via the `variable` attribute is a folder and not a content item or content version.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the resource variable
index	NMTOKEN	#IMPLIED	the optional index into an aggregation variable, given by a variable name or a constant value

Table 6.35. Attributes of the IsFolder element

```
<Variable name="Location" type="Resource"/>
<!-- Code -->
<AutomatedTask name="CreateDocument" successor="TheNext">
    <PreCondition name="CheckLocation">
        <IsFolder variable="Location"/>
    </PreCondition>
```

```
    <!-- Code -->
</AutomatedTask>
```

Example 6.36. Example of an IsFolder expression

Join

Grammar: ([Variable [167] | AggregationVariable [119]]*, Predecessor [153]+)

A Join task waits for two or more tasks to complete. Joined tasks must have been forked by a Fork [133] task to perform execution in parallel. A Join task waits for all of them to be completed.

The Predecessor elements contained in this element list all tasks that use this Join element as the successor. For an example see Fork [133].

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of this task
description	CDATA	#IMPLIED	the textual description of this task
successor	NMTOKEN	#REQUIRED	the next task to execute after all predecessors have been joined

Table 6.36. Attributes of the Join element

JoinSubprocess

Grammar: (Variable [167] | AggregationVariable [119])*

A JoinSubprocess [146] task waits for a non detached subprocess to complete. For an XML example see ForkSubprocess [134].

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	The name of this task
description	CDATA	#IMPLIED	The textual description of this task
forkTask	NMTOKEN	#REQUIRED	The name of the task that forked the subprocess to wait for

Attribute	Type	Default	Description
successor	NMTOKEN	#REQUIRED	The next task to execute after the subprocess has been joined
processResultVariable	NMTOKEN	#IMPLIED	Name of the variable of the subprocess that contains the result variable.
localResultVariable	NMTOKEN	#IMPLIED	Name of the variable of the current process into that the result value should be stored.

Table 6.37. Attributes of the JoinSubprocess element

Length

Grammar: EMPTY

Length evaluates to the length of the value of the specified variable or resource property and depends on the type. For an aggregation variable it returns the number of elements, for a string variable or string property it returns the length of the string. See also Get [136] and Read [155].

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the variable
index	NMTOKEN	#IMPLIED	the optional index into an aggregation variable, given by a variable name or a constant value
property	NMTOKEN	#IMPLIED	the optional name of a resource property

Table 6.38. Attributes of the Length element

```
<Variable name="Input" type="String">
<Assignment>
    <Writes variable="Input"/>
```

```
</Assignment>
<UserTask name="LengthCheck" successor="TheNext">
    <Guard>
        <Greater>
            <Length variable="Input"/>
            <Integer value="4"/>
        </Greater>
    </Guard>
    <!-- Code -->
</UserTask>
```

Example 6.37. Example of a Length element

Less

Grammar: [[Expression [116]], [Expression [116]]]

A Less expression contains exactly two subexpressions, which are both evaluated during the evaluation of the Less expression. The expression evaluates to "true" if and only if the computed value of the first subexpression is less than the value of the second subexpression.

Although a Less expression may compare values of any type, this element makes sense only for integer, string, date, and timer values as defined in the workflow.

LessEqual

Grammar: [[Expression [116]], [Expression [116]]]

A LessEqual expression contains exactly two subexpressions, which are both evaluated during the evaluation of the LessEqual expression. The expression evaluates to "true" if and only if the computed value of the first subexpression is less than or equal to the value of the second subexpression.

Although a LessEqual expression may compare values of any type, this element makes sense only for integer, string, date and timer values as defined in the workflow. See Less [148] for an XML example.

```
<Variable name="Published" type="Date"/>
<!-- Code -->
<If name="IfTask">
   <Condition>
      <Less>
         <Get variable="Published"/>
         <Date value="31.12.2000 24:00"/>
      </Less>
   </Condition>
   <Then successor="NewCentury"/>
   <Else successor="OldCentury"/>
</If>
```

Example 6.38. Example of a Less expression

Let

Grammar: [[*Expression [116]*], [*Expression [116]*]]

Let binds an expression-local variable to a value determined by the first subexpression. It evaluates to the value of the second subexpression, which can use the expression-local variable. Let is useful to reuse complex subexpressions and store their result in an expression-local variable. Some functions as Length [147] and Read [155] can only be applied to variable values. Using Let they can be applied to any expression (mostly custom expressions), which must return values which must make sense.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the local variable that will be bound to the result of the first subexpression

Table 6.39. Attributes of the Let element

```
<Variable name="Article" type="Document"/>
<Assignment>
    <Writes variable="Article"/>
</Assignment>
<UserTask name="LetTest" successor="Final">
    <Guard>
        <Let variable="Test">
            <Read variable="Article" property="Headline"/>
            <Greater>
                <Integer value="50"/>
                <Length variable="Test"/>
            </Greater>
        </Let>
    </Guard>
    <!-- Code -->
</UserTask>
```

Example 6.39. Example of a Let element which is needed to check whether the headline of an article is longer than 50 characters or not

Not

Grammar: (Expression [116])

A Not expression evaluates its Boolean subexpression and returns the logical negation of the result.

```
<ForAll variable="Element" aggregate="Articles">
    <Not>
         <Read variable="Element" property="isCheckedOut_"/>
    </Not>
</ForAll>
```

Example 6.40. Example of a Not element

NotEmpty

Grammar: EMPTY

NotEmpty is the negation of IsEmpty. See IsEmpty [144] for details.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the document variable
index	NMTOKEN	#IMPLIED	the optional index into an aggregation variable, given by a variable name or a constant value
property	NMTOKEN	#IMPLIED	the optional name of a resource property

Table 6.40. Attributes of the NotEmpty element

NotEqual

Grammar: [[Expression [116]], [Expression [116]]]

A NotEqual expression is the negation of an Equal expression.

```
<Variable name="Comment" type="String"/>
<Assignment>
   <Writes variable="Comment"/>
</Assignment>

<UserTask name="AndTest" successor="TheNext">
   <Guard>
      <NotEqual>
         <Get variable="Comment"/>
         <String value="LetMeIn"/>
      </NotEqual>
   </Guard>
   <!-- Code -->
</UserTask>
```

Example 6.41. Example of a NotEqual expression

Or

Grammar: [[Expression [116]]*]

An Or expression evaluates to the disjunction of its subexpressions, all of which must return Boolean values. The subexpressions are evaluated in a "short-circuit" fashion, that is, they are evaluated from left to right until the first subexpression evaluates to "true" or all subexpressions have evaluated to "false".

```
<UserTask name="AndTest" successor="theNext">
   <PreCondition>
      <Or>
         <Equal>
            <Get variable="OWNER_"/>
            <User value="0"/>
         </Equal>
         <Equal>
            <Get variable="Comment"/>
            <String value="42"/>
         </Equal>
      </Or>
   </Precondition>
   <!-- Code -->
</UserTask>
```

Example 6.42. Example of an Or expression

Parameters

Grammar: (Assign [121]+)

Parameters is used to enclose the elements that define how to parametrize a subprocess. For an XML example see ForkSubprocess [134].

Performers

Grammar: ANY

A Performers element specifies external code that is called to determine which users to offer a task for acceptance. If you do not use this element, the default policy `DefaultPerformersPolicy` is used.

You can either give the fully qualified name of your own Performers class which must be an implementation of `com.coremedia.workflow.WfPerformersPolicy`, an unqualified class name which will be searched for in the package `com.coremedia.workflow.common.policies` or it defaults to a built-in generic implementation `com.coremedia.workflow.common.policies.DefaultPerformersPolicy`.

The default implementation keeps a blacklist of users not permitted to perform a task and a list of preferred users. Upon setting a new preferred user or group the old preference is deleted. For details see the Action class `PreferPerformer`.

Attribute	Type	Default	Description
policyClass	NMTOKEN	#IMPLIED	the class that determines the performers
varies			additional parameters according to the imple-

Attribute	Type	Default	Description
			mentation of the policy class

Table 6.41. Attributes of the Performers element

```
<UserTask name="PerformersTest" successor="TheNext">
    <Performers policyClass="com.coremedia.MyPolicyClass"/>
    <!-- Code -->
</UserTask>
```

Example 6.43. Performers element

PostCondition

Grammar: (*Expression [116]*)

A PostCondition assert a condition that must hold after an (optional) exit action (user task) or action (automated task) has run. See Condition [125] for details.

Attribute	Type	Default	Description
name	NMTOKEN	#IMPLIED	the name of the PostCondition
description	CDATA	#IMPLIED	a textual description of the verified condition

Table 6.42. Attributes of the PostCondition element

```
<Variable name="Article" type="Document">
<UserTask name="PostCondition" successor="TheNext">
   <!-- Code -->
   <Assignment>
       <Writes variable="Article"/>
    </Assignment>
    <!-- Code -->
    <PostCondition name="CheckDocument">
        <Not>
            <IsEmpty variable="Article"/>
        </Not>
    </PostCondition>
</UserTask>
```

Example 6.44. Example of a PostCondition element

PreCondition

Grammar: (*Expression [116]*)

A PreCondition asserts a condition that must hold when the task has been accepted but before an entry action (user task) or action (automated task) has run. It is described by an expression. See Condition [125] for details.

Attribute	Type	Default	Description
name	NMTOKEN	#IMPLIED	the name of the PreCondition
description	CDATA	#IMPLIED	a textual description of the verified condition

Table 6.43. Attributes of the Precondition element

```
<Variable name="Location" type="Folder"/>
<Variable name="DocName" type="String"/>
<Assignment>
    <Writes variable="Location"/>
    <Writes name="DocName"/>
</Assignment>
<AutomatedTask name="CreateDocument" successor="TheNext">
    <PreCondition name="CheckLocation">
        <IsFolder variable="Location"/>
    </PreCondition>
    <Variable name="DocType" type="DocumentType">
        <DocumentType value="Article"/>
    </Variable>
    <Action name="CreateDocument" folderVariable="Location"
                nameVariable="DocName" typeVariable="DocType"/>
</AutomatedTask>
```

Example 6.45. Example of a PreCondition

Predecessor

Grammar: EMPTY

A Predecessor elements defines a predecessor of a Join [146] task by its name. See Fork [133] for an XML example.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of one predecessor task

Table 6.44. Attribute of the Predecessor element

Process

Grammar: (Rights [158]?, (Variable [167] | AggregationVariable [119])*, InitialClient [142]?, Client [125]?, (Task [117])+)

A process is a definition of a workflow process which is identified by its name. It consists of tasks, which reference each other by name. The *startTask* attribute defines the name of the start task. A process is the template for a process instance. To run a process, it has to be instantiated. At that time an actual process instance is created, which carries the process state and completes the workflow steps that are defined by tasks and carried out by task instances.

The description of the process is a human readable explanation about what the process does or a key used for localization.

The *subprocessOnly* attribute defines whether an instance of the process can be created as a top level instance or only as a subprocess instance. The default is false.

The Rights [158] element configures user and group permissions for the process instance operations.

Variables in the process scope define the global state of the workflow process. With InitialClient [142] and Client [125], you define which variables are to be read or written by a user or an external process. The InitialClient [142] element is used for initializing the process before it is started while the Client [125] element is used afterwards when the process is running.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the process
description	CDATA	#IMPLIED	a textual description of what the process does or a localization key.
startTask	NMTOKEN	#REQUIRED	the name of the initial task
subprocessOnly	[Boolean [118]]	"false"	Specify this attribute for processes that cannot run stand-alone.
defaultTimeout	NMTOKEN	#IMPLIED	the maximum number of seconds that an instance of this process is supposed to take

Table 6.45. Attributes of the Process element

```
<Workflow>
   <Process name="Example" description="An example"
    startTask="First">
      <!-- Code -->
   </Process>
</Workflow>
```

Example 6.46. Example of the Process element

Property

Grammar: EMPTY

The Property element defines the properties with which a new document is created.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of a property as defined in the content type
value	CDATA	#REQUIRED	a value of the appropriate type

Table 6.46. Attributes of the Property element

```
<Variable name="Location" type="Folder"/>
<Variable name="DocName" type="String"/>
<Assignment>
    <Writes variable="Location"/>
    <Writes name="DocName"/>
</Assignment>
<AutomatedTask name="CreateDocument" successor="TheNext">
    <Variable name="DocType" type="DocumentType">
        <DocumentType value="Article"/>
    </Variable>
    <Action name="CreateDocument" folderVariable="Location"
              nameVariable="DocName" typeVariable="DocType">
        <Property name="Headline" value="Politics"/>
        <Property name="Creator" value="AutomaticCreator"/>
    </Action>
</AutomatedTask>
```

Example 6.47. Example of a Property element

Read

Grammar: EMPTY

Read evaluates to the contents of the given property of a resource. 'property' can be the name of any implied or schema property of a resource. A blob property will be returned as an XML representation in a string value, a linklist property will be returned as an aggregation variable of documents and an SGML property will be returned as a string. All

other property types will be returned as the appropriate workflow variable value. See Exists [129] for an XML example.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the document variable
index	NMTOKEN	#IMPLIED	the optional index into an aggregation variable, given by a variable name or a constant value
property	NMTOKEN	#IMPLIED	the name of the resource property to read

Table 6.47. Attributes of the Read element

Reads

Grammar: EMPTY

Reads and Writes [168] specify the variables that are 'important' to a task or process instance. For variables that are specified with Reads, it is not possible to modify them. They are just shown in the editor. Accordingly, Writes [168] allows you to modify variables on a workflow client.

The variable attribute specifies the name of the variable. The description is a human readable explanation about how to interpret or modify the variable. It may be localized by the editor.

Resource variables may be declared as `contentEditable`, which means that you can change the content of the resource stored in the variable (if you have the appropriate rights on the resource) but you can not change the resource to which the variable references even if the variable itself is read-only.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	Defines the name of the read variable
description	CDATA	#IMPLIED	Defines the textual description of the meaning of the variable

Attribute	Type	Default	Description
`contentEditable`	[Boolean [118]]	"true"	Defines whether a document referred to by a variable may be edited in the embedded document view

Table 6.48. Attributes of the Reads element

```
<Variable name="Comment" type="String"/>
<Variable name="Article" type="Document"/>
<Assignment>
   <Reads variable="Comment"/>
   <Reads variable="Article" contentEditable="true"/>
</Assignment>
```

Example 6.48. Example of a Reads element

Resource

Grammar: EMPTY

The Resource element is used to specify a single constant resource within expressions or variable initializers. It is not useful to define a fixed resource ID in the workflow. Either `value` or `path` must be selected.

Attribute	Type	Default	Description
value	NMTOKEN	#IMPLIED	The ID of the resource.
path	NMTOKEN	#IMPLIED	The path of the resource.

Table 6.49. Attributes of the Resource element.

```
<Variable name="DocFol" type="Resource">
   <Resource value="12"/>
</Variable>
```

Example 6.49. Example of a Resource variable

Revoke

Grammar: EMPTY

Revoke revokes the operations for users or groups like Section 6.2, "XML Element Reference" [137] grants them (only valid for the default ACL rights policy). See Section 6.2,

"XML Element Reference" [137] for details. Rights specified using variables precede user rights, which again precede group rights. Within each category, revokes precede grants.

Attribute	Type	Default	Description
user	NMTOKEN	#IMPLIED	the name of a user or the user ID of a user
group	NMTOKEN	#IMPLIED	the name of a group or the group ID of a group
domain	NMTOKEN	#IMPLIED	Domain of a group or user. Might be used in addition, if group or user has been chosen.
variable	NMTOKEN	#IMPLIED	the name of a variable that stores a user or a group or a list of these
rights	CDATA	#REQUIRED	a comma-separated list of rights as specified above

Table 6.50. Attributes of the Revoke element.

```
<UserTask name="GrantExample" successor="TheNext">
    <Rights>
        <Grant group="composer-role"
         rights="accept, complete, delegate, read"/>
        <Revoke user="demo1" rights="delegate"/>
    </Rights>
        <!-- Code -->
<UserTask>
```

Example 6.50. Example of a Revoke element

Rights

Grammar: (Section 6.2, "XML Element Reference" [137]*, Revoke [157])

The Rights element defines user and group permissions for the workflow operations.

You can either give the full qualified name of your own Rights class which must be an implementation of `com.coremedia.workflow.WfRightsPolicy`, an unqualified class name which will be searched for in the package `com.coremedia.workflow.common.policies` or it defaults to a built-in generic implementation `com.coremedia.workflow.common.policies.ACLRightsPolicy`.

The default policy `ACLRightsPolicy` defines an access control list like implementation:

- Right can be granted to individual users or group [Section 6.2, "XML Element Reference" [137]].
- Rights can be revoked for individual users or groups [Revoke [157]].
- User defined rights precede group rights.
- Negative rights (revokes) precede positive rights.
- The admin user has all rights (this is the user with id 0).

Specific rights are explicitly granted to the owner of the process and the performer of a task.

The process owner may:

- Read and write variables exported by the processes client view.
- Start the process instance.
- Skip, assign and delegate any user task.
- Retry the last transaction on an aborted task instance (not dependent on the policy).

The task performer may:

- Read and write variables exported by the tasks client view.
- Cancel or complete the accepted task instance.
- Retry the last transaction if the task instance is aborted.

Attribute	Type	Default	Description
policyClass	NMTOKEN	#IMPLIED	the class that determines the policy
varies			additional parameters according to the implementation of the policy class

Table 6.51. Attributes of the Rights element

```
<Workflow>
    <Process name="RightsExample" startTask="First">
        <Rights>
            <Grant group="composer-role"
            rights="create, start, suspend"/>
        </Rights>
        <!-- Code -->
        <UserTask name="First" description="The first Task"
        successor="Next">
            <Rights>
                <Grant user="demo1"
                rights="accept, complete, read"/>
            </Rights>
            <!-- Code -->
        </UserTask>
    <!-- Code -->
    </Process>
</Workflow>
```

Example 6.51. Example of a Rights element

String

Grammar: EMPTY

The String element is used to specify a single constant string value within expressions or variable initializers.

Attribute	Type	Default	Description
value	CDATA	#REQUIRED	the string value

Table 6.52. Attribute of the String element

```
<Variable name="Text" type="String">
   <String value="Hello World"/>
</Variable>
```

Example 6.52. Example of a String variable

Successor

Grammar: EMPTY

A Successor element defines a successor task of a Fork [133] or Choice [124] task by its name. See Fork [133] for an example.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the successor task

Table 6.53. Attribute of the Successor element

Switch

Grammar: (Variable | AggregationVariable)*, (Case)+)>

A Switch task determines the successor based on the result of two or more 'case' conditions. The successor is defined by the first 'case' condition evaluating to true. The conditions are evaluated in sequential order of their definition. A default successor is mandatory if all given conditions evaluate to false.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	The name of the task.
description	CDATA	#IMPLIED	The textual description of the task.
defaultSuccessor	NMTOKEN	#REQUIRED	The default successor task that is chosen if no case condition matches.

Table 6.54. Attributes of the Switch element.

```
<Switch name="SwitchTask" defaultSuccessor="DefaultTask">
 <Case successor="FirstSuccessor">
  <Equal>
   <Get variable="Comment"/>
   <String value="42"/>
  </Equal>
 </Case>
 <Case successor="SecondSuccessor">
  <Equal>
   <Get variable="Comment"/>
   <String value="13"/>
  </Equal>
 </Case>
</Switch>
```

Example 6.53. Example of the Switch element.

Then

Grammar: EMPTY

Then defines the successor of the If [140] task if the condition evaluates to true, see If [140] for details and an example.

Attribute	Type	Default	Description
successor	NMTOKEN	#REQUIRED	the name of the successor task in the "then" case

Table 6.55. Attribute of the Then element

Timer

Grammar: EMPTY

The Timer element is used to specify a single constant timer value within expressions or variable initializers.

Attribute	Type	Default	Description
value	NMTOKEN	#IMPLIED	For relative timers, this attribute specifies the number of seconds until the timer runs out.
relative	[*Boolean [118]*]	"true"	This attribute determines whether the timer should be a relative timer. An absolute timer will not be useful in the workflow definition.

Table 6.56. Attributes of the Timer element

```
<Variable name="Expires" type="Timer">
   <Timer value="100"/>
</Variable>
<Action class="EnableTimer" timerVariable="Expires"/>
```

Example 6.54. Example of a Timer variable

TimerHandler

Grammar: EMPTY

The TimerHandler element is used to assign a timer handler to a timer. The handler must be defined in the same location, that is the process or task definition, where its associ-

ated timer variable is defined. See Section 4.4.3, "Predefined TimerHandler Classes" [80] for a list of predefined timer handlers.

Attribute	Type	Default	Description
class	NMTOKEN	#REQUIRED	Timer handler class that is called.
name	NMTOKEN	#IMPLIED	Name of the timer handler.
timerName	NMTOKEN	#REQUIRED	Name of the timer for which the timer handler is installed.

Table 6.57. Attributes of the TimerHandler element

```
<AutomatedTask name="StartTimer" description="SimplyStart"
   successor="Wait">
    <Variable name="waiting" type="Timer">
        <Timer value="100"/>
    </Variable>
    <Action class="enableTimer" timerVariable="waiting"/>
        <TimerHandler class="RunActionTimerHandler"
           name="TimerHandler"
           timerName="waiting">
            <Action class="Log" info="true"
               message="Entering task with x = "/>
        </TimerHandler>
</AutomatedTask>
```

Example 6.55. Example of a TimerHandler element

User

Grammar: EMPTY

The User element is used to specify a single constant user value within expressions, variable initializers or policies. Either 'value' or 'name' must be specified.

If you delete a user in the user administration, which you have used in the `User` element of an uploaded workflow definition, its polices will fail.

Attribute	Type	Default	Description
value	NMTOKEN	#IMPLIED	The numeric ID of a user.
name	NMTOKEN	#IMPLIED	The name of a user.

Attribute	Type	Default	Description
domain	NMTOKEN	#IMPLIED	The domain of a user. Might be used in addition to `name`.

Table 6.58. Attributes of the User element.

```
<Variable name="Admin" type="User">
   <User value="0"/>
</Variable>
```

Example 6.56. Example of a User variable

UserTask

Grammar: [Rights [158], Performer [151]?, [Variable [167] | AggregationVariable [119]]*, Client [125]*, EntryAction [128]*, ExitAction [130]*, Guard [140]?, PreCondition [152]*, PostCondition [152]]

A UserTask has to be carried out by a participant. The performers policy is external code which is called to determine which users to offer this task for acceptance.

The *`defaultOfferTimeout`* defines the default time in seconds that task instances are offered to users to be accepted. The *`defaultTimeout`* defines the default time in seconds until task instances have to be completed after being accepted. If no timeout time is set, then no timeout is defined at all. A *`defaultPriority`* sets the default priority of task instances. Priorities may be used to distinguish the urgency of task instances. A successor must be given if and only if the task is not final.

The run time of an autocompleted task is determined by the time that the executed actions and the `PreConditions` and `PostConditions` take. It will not be completed by the user but just runs through all included actions. Since EntryActions and ExitActions are executed, the effect is that a user can determine when this execution is supposed to take place and that it takes place on behalf of the user. Consider autocompleted tasks as semi-automatic tasks.

The Rights [158] element configures user and group permissions for the task instance operations.

Client [125] determines which variables are relevant for this task and may be changed.

A user task may perform some automated action (EntryAction [128]) after the task is accepted and after the task has been completed by the user (ExitAction [130]). If [140] more than one EntryAction [128] or ExitAction [130] is provided, then the actions are executed in the order they are specified.

`PreConditions` define requirements which have to be fulfilled before the entry actions of the user task are executed. PostConditions define requirements which have to be fulfilled after all the exit actions have been executed. `PreConditions` and `PostConditions` are evaluated in the order they are specified. The result of such an evaluation operation is equivalent to specifying an 'and' expression with an ordered set of conditions.

A Guard [140] defines an expression, which activates the task, if the expression evaluates to true. The expressions of the condition are rechecked on state changes of process instances or task instances and resources in the *Live Server*.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the task
description	CDATA	#IMPLIED	the textual description of the task
defaultPriority	NMTOKEN	#IMPLIED	the priority of the task
defaultTimeout	NMTOKEN	#IMPLIED	the default timeout in seconds
defaultOffer-Timeout	NMTOKEN	#IMPLIED	the default offer timeout in seconds
successor	NMTOKEN	#IMPLIED	the next task to execute after the user task has been completed
final	[Boolean [118]]	"false"	Defines whether the task is the final task to execute
autoAccepted	[Boolean [118]]	"false"	Defines whether the task is automatically accepted if it was assigned to a single user with the `ForceUser` action. Entry actions of automatically accepted tasks will by default be executed by user `workflow`. Note that even if this attribute is set to "false",

Attribute	Type	Default	Description
			tasks may still be automatically accepted by workflow clients such as the *Site Manager*.
autoCompleted	[Boolean [118]]	"false"	Defines whether the task is autocompleted
varies			additional parameters according to the implementation of the user task class

Table 6.59. Attributes of the UserTask element

```
<UserTask name="UserTaskExample" description="Example UserTask"
  successor="Next">
    <Rights>
        <Grant user="demo1" rights="accept, complete, read"/>
    </Rights>
    <!-- Code -->
</UserTask>
```

Example 6.57. Example of a UserTask task

Validator

Grammar: [Expression [116]]

A validator verifies variable bindings to keep certain rules, which are defined in the Validator element.

By default, the variable bindings are verified only on initial process assignment or task completion. If `validatedOnSave` is set to "true", the verification takes place on every save.

To specify a valid state, you provide an expression to the validator.

Attribute	Type	Default	Description
name	NMTOKEN	#IMPLIED	the name of the validator
description	CDATA	#IMPLIED	the textual description of the condition that is verified

Attribute	Type	Default	Description
validatedOnSave	[Boolean [118]]	"false"	Defines whether the verification should take place on every save
varies			additional parameters according to the implementation of the validator class

Table 6.60. Attributes of the Validator element

```
<Assignment>
    <Writes variable="subject"/>
    <Writes variable="comment"/>
    <Writes variable="changeSet" contentEditable="true"/>
    <Validator name="AllCheckedIn"
     description="all-checked-in-validator">
        <ForAll variable="change" aggregate="changeSet">
            <Implies>
              <And>
                <IsDocumentVersion variable="change"/>
                <Equal>
                  <Read variable="change" property="version_"/>
              <Read variable="change" property="latestVersion_"/>
                </Equal>
              </And>
              <Not>
 <Read variable="change" property="isCheckedOut_"/>
             </Not>
            </Implies>
        </ForAll>
    </Validator>
</Assignment>
```

Example 6.58. Example of a Validator element

Variable

Grammar: [Value [117]]?

Variables carry state for the workflow process. It may be modified from within the workflow engine or by changing client view variables.

A variable is referenced by its name. It has a type which is determined by the Value class given with the type attribute. See Value for details. The value of a variable is defined by one of the elements Boolean, String etc.

If [140] a variable is declared as *readOnly* and the process instance has been started, it is not possible to modify it. If [140] a variable is declared as static, it maintains its state, otherwise it is reinitialized to the defined default every time a task instance is started.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the variable
type	NMTOKEN	#REQUIRED	the type of the variable, see Value
readOnly	[Boolean [118]]	"false"	Defines whether it is forbidden to modify the variable
static	[Boolean [118]]	"false"	Defines whether the variable is initialized only once

Table 6.61. Attributes of the Variable element

```
<Variable name="Comment" type="String">
    <String value="42"/>
</Variable>
```

Example 6.59. Example of a Variable element

Workflow

Grammar: [Process [153]]

You can configure exactly one process per workflow definition, which means one workflow per file. If [140] you wish to define more workflow processes, create their definition in separate files. This might be extended in the future.

```
<Workflow>
   <Process name="WorkflowExample" startTask="First">
      <!-- Code -->
   </Process>
</Workflow>
```

Example 6.60. Example of the Workflow element

Writes

Grammar: EMPTY

In a Client, a Writes element declares that a variable may be viewed and modified. See Reads for details.

Attribute	Type	Default	Description
variable	NMTOKEN	#REQUIRED	the name of the written variable
description	CDATA	#IMPLIED	the textual description of the meaning of the variable
contentEditable	[Boolean [118]]	"true"	Defines whether a document referred to by a variable may be edited in the embedded document view [not enforced by the workflow server]

Table 6.62. Attributes of the Writes element

```
<Variable name="Comment" type="String"/>
<Assignment>
   <Writes variable="Comment"/>
</Assignment>
```

Example 6.61. Example of a Writes element

6.3 Studio Simple Publication Workflow Definition

In this chapter you find the complete workflow definition of the Studio Direct Publication workflow as described in Section 4.3, “Example of Workflow Definition” [59].

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!--
          CoreMedia Simple Publication Workflow for Studio
-->

<Workflow>
  <Process name="StudioSimplePublication" description="studio-simple-publication"
           startTask="AssignUser">

    <Rights>
      <Grant group="administratoren" rights="read, create, start, suspend, resume, abort"/>
      <Grant group="composer-role" rights="read, create, start, suspend, resume, abort"/>
      <Grant group="approver-role" rights="read"/>
      <Grant group="publisher-role" rights="read"/>
    </Rights>

    <Variable name="subject" type="String"/>
    <Variable name="comment" type="String"/>
    <AggregationVariable name="changeSet" type="Resource"/>
    <AggregationVariable name="comments" type="String"/>

    <Variable name="changeSetLockedInStudio" type="Boolean">
      <Boolean value="true"/>
    </Variable>
    <Variable name="publicationSuccessful" type="Boolean">
      <Boolean value="false"/>
    </Variable>
    <AggregationVariable name="publicationResultResources" type="Resource"/>
    <AggregationVariable name="publicationResultCodes" type="Integer"/>
    <AggregationVariable name="publicationResultVersions" type="Integer"/>
    <AggregationVariable name="publicationResultParams" type="String"/>

    <InitialAssignment>
      <Writes variable="subject"/>
      <Writes variable="comment"/>
      <Writes variable="changeSet"/>
      <Writes variable="comments"/>
    </InitialAssignment>

    <Assignment>
      <Reads variable="subject"/>
      <Reads variable="comment"/>
      <Reads variable="changeSet"/>
      <Reads variable="comments"/>
    </Assignment>

    <AutomatedTask name="AssignUser"
                   description="assignuser-task" successor="CheckEmptyChangeSet">
      <Action class="ForceUser" task="Publish" userVariable="OWNER_"/>
      <Action class="ForceUser" task="Compose" userVariable="OWNER_"/>
      <Action class="RegisterPendingProcess" userVariable="OWNER_"7>
    </AutomatedTask>

    <If name="CheckEmptyChangeSet">
      <Condition>
 <IsEmpty variable="changeSet"/>
      </Condition>
      <Then successor="Finish"/>
      <Else successor="Publish"/>
    </If>
```

```
    <UserTask name="Publish"
              description="studio-simple-publication-publish-task"
              successor="CheckPublication" reexecutable="true" autoAccepted="true" autoCompleted="true">
      <Rights>
        <Grant group="administratoren" rights="read, accept, retry"/>
        <Grant group="composer-role" rights="read, accept, retry"/>
      </Rights>

      <Assignment>
        <Reads variable="subject"/>
        <Reads variable="comment"/>
        <Reads description="publish-changeSet" variable="changeSet" contentEditable="false"/>
        <Reads variable="comments"/>
      </Assignment>

      <EntryAction class="ApproveResource" gui="true"
                   resourceVariable="changeSet"
     successVariable="publicationSuccessful"
                   ignoreErrors="true"
                   timeout="180"
                   userVariable="PERFORMER_">
      </EntryAction>

      <EntryAction class="PublishResources" gui="true"
                   resourceVariable="changeSet"
                   resultVariable="publicationResultResources"
                   versionVariable="publicationResultVersions"
                   codeVariable="publicationResultCodes"
                   parameterVariable="publicationResultParams"
                   successVariable="publicationSuccessful" ignoreErrors="false"
                   ignorePublicationErrors="true" timeout="600"
                   userVariable="PERFORMER_"/>
    </UserTask>

    <If name="CheckPublication">
      <Condition>
 <Get variable="publicationSuccessful"/>
      </Condition>
      <Then successor="Finish"/>
      <Else successor="Compose"/>
    </If>

    <UserTask name="Compose"
              description="studio-simple-publication-compose-task"
              successor="CheckEmptyChangeSet" reexecutable="true" autoAccepted="true">
      <Rights>
        <Grant group="administratoren" rights="read, accept, delegate, skip"/>
        <Grant group="composer-role" rights="read, accept, delegate, skip"/>
      </Rights>

      <Assignment>
        <Writes variable="subject"/>
        <Writes variable="comment"/>
        <Writes variable="changeSet" contentEditable="true"/>
        <Writes variable="comments"/>
        <Reads variable="publicationResultCodes"/>
      </Assignment>
    </UserTask>

    <AutomatedTask name="Finish" final="true">
      <Action class="AssignVariable" resultVariable="changeSetLockedInStudio">
        <Boolean value="false"/>
      </Action>
    </AutomatedTask>

    <!-- Finally, make sure finished processes are archived and appear in the list of finished workflows
for
         participating users, i.e. for users for whom the RegisterPendingProcess action was called. -->
    <FinalAction class="ArchiveProcessFinalAction" maxProcessesPerUser="100"/>

  </Process>
</Workflow>
```

Example 6.62. Listing of the direct publication workflow

Glossary

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

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

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

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

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

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

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

Index

A

access variables, 92
AclRightsPolicy, 99
action, 66
actions, 54, 86-87
actionsserver-side, 88
activity diagrams, 33

B

BeanParser, 32, 35

C

case, 124
choice, 45
components, 15
conditions, 53

D

DefaultPerformersPolicy, 104
DTD coremedia-workflow, 118

E

expressions, 52, 94, 96
expressions:boolean, 96
expressions:generic, 95

P

postconditions, 53
process, 38

R

rights, 55-56

S

serialization, 109
serialization error, 29
serialization errors, 84

T

task, 44-45
timer, 56
timer handler, 80

U

upload, 22
upload new workflows, 58

V

validator, 54

W

window, 18
workflow, 56
workflow clients, 107
workflow definition, 32, 59
workflow variables, 52
workflowclient.properties, 112