

# guide

## Workflow Developer Manual

COREMEDIA



**Copyright** CoreMedia AG © 2010

CoreMedia AG

Ludwig-Erhard-Straße 18

20459 Hamburg

### **International**

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

### **Germany**

Alle Rechte vorbehalten. CoreMedia und weitere im Text erwähnte CoreMedia Produkte sowie die entsprechenden Logos sind Marken oder eingetragene Marken der CoreMedia AG 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 AG reproduziert oder vervielfältigt werden. Unberührt hiervon bleiben die gesetzlich erlaubten Nutzungsarten nach dem UrhG.

### **Licenses and Trademarks**

All trademarks acknowledged.

Feb 10, 2011

## Table Of Contents

<b>1 Introduction.....</b>	<b>5</b>
1.1 Audience	6
1.2 Structure Of The Manual	7
1.3 Typographic Conventions	8
1.4 Change Chapter	10
<b>2 Overview And Navigation.....</b>	<b>11</b>
2.1 Content Modeling	13
2.2 Content Management	14
2.2.1 CoreMedia Editor	15
2.2.2 Custom Clients	16
2.2.3 Importer	17
2.2.4 Workflow	18
2.3 Content Delivery	19
<b>3 Overview.....</b>	<b>20</b>
3.1 Component Structure of the CoreMedia Workflow	20
3.2 The Workflow Window	21
<b>4 Predefined Workflows.....</b>	<b>22</b>
4.1 Approval and Publication of Folders and Documents	23
4.2 Predefined Publication Workflows	27
4.3 Features of the Publication Workflows	29
<b>5 Customize Workflow Definitions.....</b>	<b>31</b>
5.1 Defining Workflows	32
5.1.1 The BeanParser	35
5.1.2 Used Elements of Activity Diagrams	37
5.1.3 Processes	39
5.1.4 Tasks	40
Common Features of All Tasks	43
User Tasks	44
Automated Tasks	44
5.1.5 Flow Control	46
5.1.6 Workflow Variables	51
5.1.7 Expressions	52
Conditions	52
Pre- and Postconditions	52
Guards	53
Validators	53
5.1.8 Actions	54
5.1.9 Rights	55
5.1.10 Subworkflows	56
5.1.11 Timers	57
5.2 Upload Workflow Definitions	59
5.3 Example of Workflow Definition	60

5.4 Reference of Predefined Classes	69
5.4.1 Predefined Action Classes	70
5.4.2 Predefined TimerHandler Classes	79
5.4.3 Predefined Editor Classes	81
5.4.4 Predefined Column Classes	83
<b>6 Implementing Extensions.....</b>	<b>85</b>
6.1 Spring in The Workflow Server	86
6.2 Update Workflows	88
6.3 Variable Values	89
6.4 Programming Actions	90
6.4.1 General Rules	91
6.4.2 Repeated Execution of Actions	92
6.4.3 Server-Side Actions	93
6.4.4 Client-Side Actions	94
6.4.5 Access Workflow Variables from the Action	99
6.4.6 Example Action	101
6.5 Programming Expressions	107
6.5.1 General Rules	108
6.5.2 Generic Expressions	109
6.5.3 Boolean Expressions	111
6.5.4 Example Expression	112
6.6 Programming Rights Policies	114
6.6.1 Example Rights Policy	116
6.7 Programming Performer Policies	120
6.8 Programming Clients	123
6.9 Pitfalls of Implemented Classes	124
6.10 Customizing the Workflow GUI	126
6.10.1 Configure the Workflow GUI	127
Example Configuration of the Workflow Window	134
6.10.2 Programming Workflow Startups	137
<b>7 Appendix.....</b>	<b>140</b>
7.1 XML Element Reference	141
7.2 Simple Publication Workflow Definition	191
7.3 Complete Code of the Mail Action	194
<b>8 Support.....</b>	<b>199</b>
<b>Glossary .....</b>	<b>204</b>
<b>Index .....</b>	<b>212</b>

# 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 documents, 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 we are 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](mailto:documentation@coremedia.com).

## 1.1 Audience

This manual is intended 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 administration of the workflow system in the *CoreMedia Workflow Server* section of the Administration Manual. Additional information on the usage of the predefined workflows can be found in the User Manual.

## 1.2 Structure Of The Manual

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

- » In Section 3 "[Overview](#)"<sup>(20)</sup> you will find a short introduction into the GUI and components of the Workflow.
- » In Section 4 "[Predefined Workflows](#)"<sup>(22)</sup> you will find a short survey of the pre-defined workflows delivered with *CoreMedia CMS* and the publication semantics.
- » In Section 5 "[Customize Workflow Definitions](#)"<sup>(31)</sup> you will learn how to develop your own workflow definitions. It explains the syntax of relevant XML files.
- » In Section 6 "[Implementing Extensions](#)"<sup>(85)</sup> you will learn how to implement own extensions of the workflow.
- » In Section 7 "[Appendix](#)"<sup>(140)</sup> you will find a reference of the XML-elements existing for workflow definitions and some code examples and workflow definition examples.

## 1.3 Typographic Conventions

We use 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	Courier new	cm contentserver start
Menu names and entries	Bold, linked with	Open the menu entry <b>Format Normal</b>
Field names CoreMedia Components	Italic	Enter in the field <i>Heading</i> The <i>CoreMedia Component</i>
Entries	In quotation marks	Enter "On"
(Simultaneously) pressed keys	Bracketed in "<>", linked with "+"	Press the keys <Ctrl>+<A>
Emphasis	Italic	It is <i>not</i> saved
Buttons	Bold, with square brackets	Click on the <b>[OK]</b> button
Glossary entry	>>-shaped icon	>> WebDAV
Code lines in code examples which continue in the next line	\	cm contentserver \ start

Table 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.
	Summary: This symbol indicates a summary of the above text.

Table 2: Pictographs

## List of Abbreviations

Find a list of most common abbreviations as we use them in *CoreMedia Technical Documentation* listed below. This list just covers *CoreMedia CMS*-specific words and phrases. For common technical or software-related vocabulary, consult the Glossary section or other sources of information.

Abbreviation	Component
ADS	CoreMedia Active Delivery Server
CAE	CoreMedia Content Application Engine
PADS	CoreMedia ProActive Delivery Server
WAGE	Web Application Generator Extensions

Table 3: List of abbreviations

## New Component Names since Version 5.0

As product and component names underwent a thorough renaming procedure, new and old names might appear simultaneously within this manual. We try to adapt the new nomenclature as consistent as possible, but as old component names shine through in the software itself, we stick to these in case of doubt.

CoreMedia CMS 5.2	Old versions
<i>CoreMedia Social Software Extension</i>	New component.
<i>CoreMedia Analytics Engine</i>	New component.
<i>CoreMedia Differencing Engine</i>	New component.
<i>CoreMedia Editing Services for JSF</i>	New part of the <i>Content Application Engine</i> .
<i>CoreMedia CMS 5.2</i> (The same as <i>CoreMedia CMS 2008</i> )	Replaces <i>Content Application Platform</i> , extended feature range and functionality. <i>CoreMedia Smart Content Infrastructure</i>

Table 4: New component names

## 1.4 Change Chapter

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

[» Changes](#)

Section	Version	Description

## 2 Overview And Navigation

The *CoreMedia CMS* is a powerful and flexible platform for sophisticated high end content applications like high-volume websites or device-independent multi channel services. Its multifaceted possibilities of customization empower you to design content applications with maximum performance and convenience for such different domains as news portals, technical manuals or mobile services

The developer manual series shows you how to customize the *CoreMedia CMS* for your specific needs. This covers several aspects which can be roughly divided into three areas:

- » content modeling
- » content management (possibly workflow driven)
- » content delivery

In order to benefit from this manual you should be familiar with the general *CoreMedia CMS* concepts, especially with the component architecture as described in the Administration Manual and the basic knowledge as described in the User Manual.

*CoreMedia CMS* clients have very different tasks which range from GUI editing to HTML generation. Also the necessary extent of customization differs widely. While the editors are fairly well usable even without any customizing, the CAE is rather a framework which provides basic features like content access and caching. Such differences require specialized APIs which are covered in detail by the continuative manuals of this series.

The following sections give an overview of customizing particular *CoreMedia CMS* clients and developing new clients from scratch. Its purpose is to give you an idea of what is possible, where to start and which continuative manual you should consult for details.

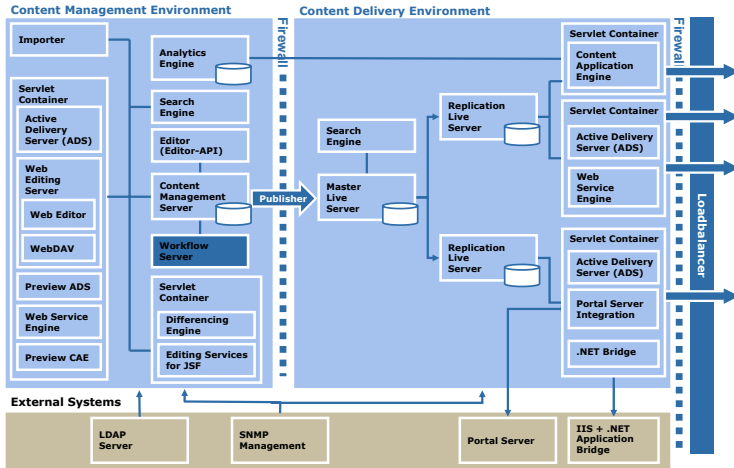


Figure 1: Overview of the CoreMedia system

The next figure shows an overview of the CoreMedia system. The part in dark blue highlights the application area of the *Workflow API Developer Manual*.

## 2.1 Content Modeling

Different content domains deal with different data. While a news portal consists basically of short texts, pictures and navigation structures, mobile services deal with multimedia objects like ring tones, animations and games, each for a variety of devices. Different content has different properties. E.g. an article consists of a headline, the actual text and possibly references to related subjects, whereas a multimedia object needs binary data and maybe a description.

The first step in designing a *CoreMedia CMS* application is to define a content model which represents the particular domain. This content model is the base of the whole application. It concerns content management, delivery and persistence, and thus it affects nearly all components:

- » the basic appearance of the editor,
- » the XML format supported by the importer,
- » the interface for JSP programming,
- » the query facilities,
- » and last but not least relationships among content objects themselves.

Convenience and simplicity of further customizing as well as the efficiency of the application crucially depends on a deliberate content model. Design weaknesses may lead to vast and error prone extra effort.

See the *Administration and Operation Manual* for a comprehensive discussion of content modeling.

## 2.2 Content Management

Content management covers creation of content in the repository and analysis of used content. There are basically two ways to populate the content repository:

- » manual editing
- » importing external content

Out-of-the-box *CoreMedia CMS* supports manual creation of content by a variety of clients which are open for customization. In addition, customized clients could be created from the scratch using different frameworks, described in Section 2.2.2 "[Custom Clients](#)"<sup>[16]</sup>.

- » The *CoreMedia Editor*, a standalone Java application. Configuration and customization of this editor is covered in the *Editor Developer Manual*.
- » The *WebEditing Server*, a servlet which serves the *CoreMedia WebEditor* for content editing. The *WebEditor* offers nearly the same functionality as the *CoreMedia Editor* but needs less installation.
- » The *WebDAV support*, which allows to access the *CoreMedia CMS* by WebDAV-enabled applications (See the Administration Manual for details.)
- » The *Importer* which is used to adopt existing external content, e.g. from legacy systems or content providers. It must be customized concerning the location and the format of the source content. See the *Importer Developer Manual* for details.

For analysis of the content repository and the acceptance of your websites in the Internet (or Intranet) there is *CoreMedia Analytics*. It offers a set of predefined detailed reports, the *Analytics Dashboard*, and easy-to-use reports directly integrated with the preview of your web site, the *In-Site Analytics*. You might extend both report types by your own analytics report. See the *Analytics Developer Manual* for details.

## 2.2.1 CoreMedia Editor

According to the underlying content model the *CoreMedia Editor* generates forms for resource editing. The default forms provide editing facilities for each property of a document (so called property editors), general editing functionality like copy&paste and *CoreMedia CMS* version control features [see User Manual, Basic Knowledge]. You can control the layout and behavior of these forms concerning the following aspects:

- » use special property editors (e.g. a string editor which automatically converts to upper case. CoreMedia provides some alternative property editors to use, but you can also implement your own.)
- » initialize and validate input (e.g. make sure that an integer value is greater than 42)
- » provide complex commands (e.g. create a new document and link it into some navigational structure)

Furthermore you can control

- » localization
- » language specific spell checking
- » rich text formatting
- » filters to hide some objects from the users (e.g. document types they are not supposed to instantiate or folders that do not concern them)
- » renderers to display objects (e.g. the fields in the document table)

Many Editor features can be achieved or controlled simply by configuration, without writing a single line of code. However, understanding all the possibilities requires some knowledge of Java.

See the *Editor Developer Manual* for details.

## 2.2.2 Custom Clients

*CoreMedia CMS* offers different APIs and frameworks for the creation of custom clients.

- » The *Unified API* which enables you to implement custom *CoreMedia CMS* clients from scratch. Taking over session handling and remote communication, it reveals the full functionality of the *Content Server* and the *Workflow Server* in a convenient and intuitive abstraction which allows you to concentrate on business logic. See the *Unified API Developer Manual* for details.
- » The *WebServices Engine*, which allows to access the *CoreMedia CMS* using Web Services. You can use the *Unified API* or the *Content Application Engine* to develop your Web Services. See the *Delivery Developer Manual* for details.
- » The *WAGE API*, which enables you to access the *CoreMedia CMS* using web based clients. The current *WebEditor* is an implementation of the *WAGE API*. Read the *WAGE Developer Manual* for details. You should only use the *WAGE API* for customization of the *WebEditor* or if you need workflow functionality in your clients but not for newly designed stand-alone clients. For this tasks use the more modern *Editing Services for JSF*.
- » *Editing Services for JSF*, a framework which allows to create custom clients to access the *CoreMedia* repository. It is the successor of the *WAGE API* and offers many convincing advantages. *Editing Services for JSF* offers for example UI components and reusable beans. It uses modern technology like JSF and Spring and builds on the *Unified API* and *CAE*. The Portal Server Integration uses *Editing Services for JSF* to add editing functionality to portals. Read the *Content Application Developer Manual* for details.

The *Unified API* replaces the *Scripting API* which you might know from *CoreMedia CMS* 4.2. Old *Scripting* clients are still supported with *CoreMedia CMS*, but for new implementations the usage of *Scripting API* is deprecated. That is why the *Scripting API* is no longer documented in the developer manuals. Consult the *CMS 4.2* manual for maintenance of your legacy *Scripting* clients.

## 2.2.3 Importer

Using the importer in order to import third party content you must basically take care for two aspects:

- » Serve the source content
- » Transform the source content into CoreMedia XML

The importer framework provides an out-of-the-box solution for source content available as files. Alternatively you can implement your own factory, e.g. to serve content from a database, over some network protocol or to generate content on the fly, composed from arbitrary sources.

As your source content is most probably not available in CoreMedia XML format matching your document types, you will have to transform it. The importer supports XSLT transformations (you just have to configure the stylesheet) as well as stream based cross-document transformations for arbitrary (even non-XML) sources.

See the *Importer Developer Manual* for details.

## 2.2.4 Workflow

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 documents, the more difficult it becomes to coordinate tasks and schedules. Computer support can greatly enhance productivity because the users do not have to deal with organizational issues too much.

This goal can be achieved by introducing 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 - *CoreMedia* users are given access to the right resources at the right time via tasks. In contrast to restrictively controlling users, *CoreMedia Workflow* focuses on progress of the overall business processes.

The *CoreMedia CMS* is delivered with some standard workflows which control content publication. They are based on the inherent state model of *CoreMedia CMS* resources (see the User Manual, Basic Knowledge). While the simplest workflow publishes a set of resources immediately, the most complex one ensures that composing the set of resources, approving those resources and the actual publication are executed explicitly by different users.

If these workflows do not fit your needs, you can define your own workflows. A workflow basically consists of a set of tasks which are processed in a specific order (usually linear, but also circles and branches are possible) and a set of variables which represent the state of the workflow. There are automated tasks which are processed automatically at their turn (like publishing some resources) and user tasks which have to be handled explicitly by a user (e.g. composing a set of resources to be published).

The tasks and variables of a workflow are defined in XML. The *CoreMedia CMS* provides some content related standard actions you can choose from to inspire your tasks, but you can also implement your own actions. You can specify performer policies to prevent or enforce users to process a task.

From the technical point of view, processing a task means executing some actions but also changing workflow variables. For user tasks the *CoreMedia Editor* and the *CoreMedia WebEditor* support this in a fashion similar to editing the properties of a resource. And just alike, you might want to use special editing components for restrictions or convenience. Thus introducing custom workflows usually includes some Editor customizing.

See the *Workflow Developer Manual* for details.

## 2.3 Content Delivery

The main goal of a content management system is to deliver the right content, to the right customer in the right time. Important requirements are:

- » Support of high traffic delivery
- » Support of different delivery formats, such as HTML, WAP, PDF
- » Support of personalized content
- » Support of search
- » Integration of data from third-party systems
- » Integration with an existing portal solution
- » Integration with delivery platforms such as Akamai
- » Analysis of usage
- » Integration of user generated content

To meet all these requirements, *CoreMedia CMS* contains the following components:

- » The *Content Application Engine (CAE)* with its components *Http Cache*, *CAE Feeder* and *Proactive Engine* to deliver and search the content.
- » *Content Rules* to deliver content, based on rules not on manual selection.
- » The *CoreMedia Portal Integration* to integrate with portals.
- » The *Analytics Engine* to analyze the usage of the content.
- » The *Social Software Extension* to integrate user generated content.

All these components have a framework character, and you can easily adapt them to your needs.

See the *Content Application Developer Manual* for details of the *CAE*, *Content Rules* and the *Portal Integration*, the *Analytics Developer Manual* for details of the *Analytics Engine* and use the *Social Software Extension* manual to learn more about the integration of user generated content into your applications.

## 3 Overview

### 3.1 Component Structure of the CoreMedia Workflow

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 *CoreMedia Editor*. 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:

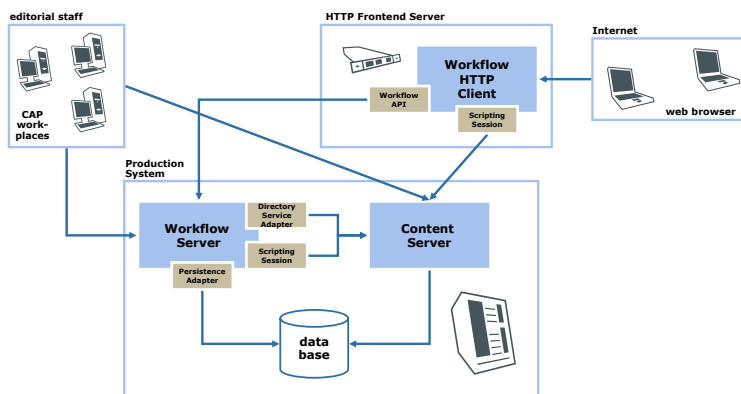


Figure 2: CoreMedia architecture with integrated workflow

## 3.2 The Workflow Window

*CoreMedia CMS* comes with a special user interface for creation and administration of workflows that is integrated into the *CoreMedia Editor*. 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 document selected here opens in the document window.

» **Document window [5]**

Displays a document selected in the detail information window before.

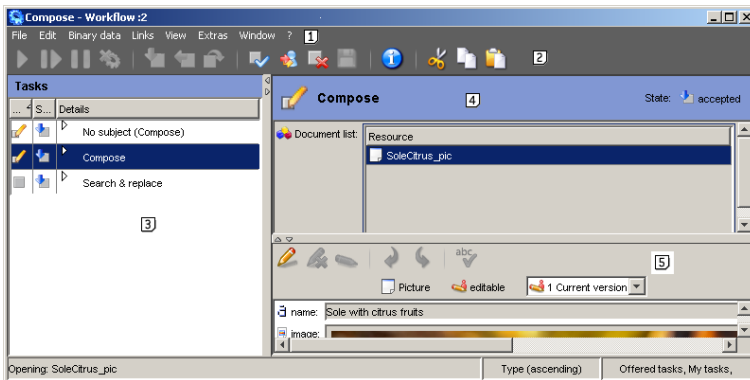


Figure 3: Workflow window

For a detailed description of *task and workflow overview* [3] and *detail information* [4] window see Section 4.3.1 of the User Manual.

## 4 Predefined Workflows

The *CoreMedia Workflow* comes with four predefined workflows which cover the publication of resources and a fifth workflow for a global search and replace functionality. In this chapter you will find a description of the four publication workflows and a description of the publication semantics. The workflow facilities are not restricted to publications, but can be tailored to fit all types of business processes. As many business processes deal with publication issues, we deliver these example workflows.

## 4.1 Approval and Publication of Folders and Documents

A publication synchronizes the state of the *Live Server* with the state of the *Content Management Server*. All actions such as setting up new versions, deleting, moving or renaming files, withdrawing content from the live site require a publication to make the changes appear on the *Live Server*.

»» *What is and what does a publication?*

We make a distinction between the publication of structural and of content changes:

- »» Content-related changes are changes in document versions such as a newly inserted image, modified links, text etc.
- »» Structure-related changes are moving, renaming, withdrawing or deleting of resources. So it becomes possible to publish structural changes separately from latest and approved document versions.

For every publication a number of changes is aggregated in a change set. This change set is normally composed in the course of a publication workflow. The administrator and other users with appropriately configured editors can also execute a direct publication, which provides a simpler, although less flexible means of creating a change set.

### Change Set in Direct Publications

When performing a direct publication, the change set is primarily based on the set of currently selected resources or on the single currently viewed resource. As the set of resources does not give enough information for all possible types of changes, three rules apply:

- »» You cannot publish movements and content changes separately. Whenever applicable, both kinds of changes are included in the change set.
- »» When a document is marked for deletion or for withdrawal, new versions of that document are not published.
- »» If the specific version to be published is not explicitly selected, the last approved resource version is included in the change set.

There are also some automated extension rules for the change set, which modify the set of to-be-published resources itself. These rules can be configured in detail. Ask your Administrator about the current settings.


- »» When new or modified content is published and links to an as yet unpublished resource, the unpublished resource is included in the change set. Depending on the configuration, also recursively linked documents can be included in the change set.
- »» When the deletion of a folder is published, all directly and indirectly contained resources are included in the change set.

- » When the withdrawal of a folder is published, all directly and indirectly contained published resources are included in the change set.
- » When the creation, movement, or renaming of a resource in an unpublished parent folder is published, that folder is included in the change set.

**Preconditions**

Preconditions for a successful publication are:

- » all path information concerning the resource has to be approved too: if the resource is located in a folder never published before, this folder has to be published with the resource. So, add it to the change set or publish the folder before.
- » withdrawals and deletions must be approved before publication.
- » all documents linked to from a document which is going to be published have to be already published or included in the change set. This is because a publication that would cause dead links will not be performed.
- » a document which is going to be deleted must not be linked to from other documents or these documents have to be deleted during the same publication.

 *Preconditions for a successful publication*

<b>Status and action on the <i>Content Management Server</i></b>	<b>Effect on the <i>Live Server</i> on publication</b>
A version of the document does not yet exist on the <i>Live Server</i> . The document is not marked for deletion. You approve the version.	The approved version is copied to the <i>Live Server</i> .
The last approved version of a document already exists on the <i>Live Server</i> . The document is not marked for deletion. You start a new publication without any further preparation.	No effect on the <i>Live Server</i> .
The document is published and is not marked for deletion. It therefore exists on both servers. You rename the document and approve the change.	The document is renamed.
The document is published and is not marked for deletion. It therefore exists on both servers. You move the document and approve the change.	The document is moved.
The document is published. It therefore exists on both servers. No links to this document exists. You mark the document for withdrawal and approve the change.	The document is destroyed on the <i>Live Server</i> .

*Table 5: Publishing documents: actions and effects*

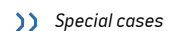
<b>Status and action on the <i>Content Management Server</i></b>	<b>Effect on the <i>Live Server</i> on publication</b>
The document is published. It therefore exists on both servers. No links to this document exists. You mark the document for deletion and approve the change.	The document is destroyed on the <i>Live Server</i> . The document is moved into the recycle bin on the <i>Content Management Server</i> .
The document is published. It therefore exists on both servers. Links to this document from other published documents exist. You mark the document for deletion and approve the change.	The deletion cannot be published, since an invalid link would be created. A message is displayed in the publication window. Remove the link in the other document and publish again.

<b>Status and action on the <i>Content Management Server</i></b>	<b>Effects on the <i>Live Server</i> on publication</b>
The folder is published and is not marked for deletion. It therefore exists on both servers. You rename the folder and approve it.	The folder is renamed.
The folder is published and is not marked for deletion. It therefore exists on both servers. You move the folder and approve the change.	The folder is moved.
The folder is not published and not marked for deletion. You approve the folder.	The folder is created on the <i>Live Server</i> .
The folder is published. You mark it for withdrawal. When queried, you acknowledge the mark for withdrawal of all contained resources. You approve the change.	The folder is destroyed on the <i>Live Server</i> . The withdrawal can only succeed if all resources on the <i>Live Server</i> or <i>Content Management Server</i> that are contained in the folder, and all published resources that link to this folders content, are also contained in the change set.
The folder is published. You mark it for deletion. When queried, you acknowledge the mark for deletion of all contained resources. You approve the change.	The folder is destroyed on the <i>Live Server</i> . The folder is moved to the recycle bin on the <i>Content Management Server</i> . The deletion can only succeed if all resources on the <i>Live Server</i> or <i>Content Management Server</i> that are contained in the folder, and all published resources that link to this folders content, are also contained in the change set.

Table 6: Publishing folders: actions and effects



**Special cases**



Please keep in mind that:

- » Older versions cannot be published.
  - Example:** if a version No. 4 had already been published it is not possible to publish version No. 3 thereafter. To do so, create a version No. 5 from No. 3.
- » During a deletion, a resource that has not been published yet is moved to the recycle bin immediately.

In addition, consult the previous tables for effects of a publication depending on the state of the resource. For all examples it is assumed that you have appropriate rights to perform the action.

### Withdrawing Publications and Deleting Resources

» Delete and withdraw resources

There is only one fundamental difference between withdrawal of publications and deletion: an withdrawal affects only the *Live Server*, whereas the deletion of a resource - folder or document - causes the resource to be moved into the trash folder on the *Content Management Server*.

Before a withdrawal or deletion can be published as described before, a mark for withdrawal or for deletion must be applied using the appropriate menu entries or tool bar buttons. In the case of folders, the contained resources are affected, too. If you have marked a resource for deletion and withdrawal, then the deletion will be executed.

- » When a folder is marked for deletion, all contained published resources are marked for deletion, too. Not published resources are immediately moved into the recycle bin without requiring to start a publication.
- » When a folder is marked for withdrawal, all contained published resources are marked for withdrawal, too.
- » When a mark for withdrawal or deletion of a folder is revoked, this also affects all contained resources with the same mark.
- » If you use direct publication and approve a folder that is marked for withdrawal deletion, that approval is implicitly extended to the contained resources that are also marked for withdrawal or deletion.
- » Disapprovals extend to contained resources in the same way.

## 4.2 Predefined Publication Workflows

The four predefined workflows for the approval and publication of resources are described in the following table. You will find the workflow definitions of these workflows in the file `<InstallDir>/properties/corem/workflows.zip` and you can use them as examples for your own definitions.

### Publication workflows

The following table compares the working steps which are covered by the predefined workflows.

Work-flow name / Step in work-flow	simple publication	2-step publication	2-step approval	3-step publication
1.	A user creates the workflow with all necessary re-sources.	A user creates the workflow with all necessary re-sources.	A user creates the workflow with all necessary re-sources.	A user creates the workflow with all necessary re-sources.
2.	The resources are published (and implicitly approved) in one step, performed by the same user, who needs 'approve' and 'publish' rights.	A second user (needs 'approval' and 'publish' rights) can explicitly approve re-sources.	A second user (needs 'approval' rights) can explicitly approve re-sources.	A second user (needs 'approval' rights) can explicitly approve re-sources.
3.		Publication will be executed when finishing the task after all resources in the change set have been approved.	Workflow ends with approval of all resources in the change set.	After all resources have been approved, a third user can accept the task.
4.		<i>[If not, the work-flow is returned to its 'composer']</i>	<i>[If not, the work-flow is returned to its 'composer']</i>	<i>[If not, the work-flow is returned to its 'composer']</i>

Table 7: Predefined publication workflows

Work-flow name / Step in work-flow	simple publication	2-step publication	2-step approval	3-step publication
5.				If the third user accepts the task, the publication is started automatically.

## 4.3 Features of the Publication Workflows

The predefined publication workflows have some features in common, which are described in the following:

### Users and Groups

In order to execute tasks within workflows, users have to be assigned to special groups. In the predefined publication workflows, these are the following:

1. *composer-role*: to be able to create (and start) a publication workflow and compose a change set
2. *approver-role*: to be able to approve the resources in the change set
3. *publisher-role*: to be able to publish the resources in the change set

Special groups can be defined and linked to the workflow via the `Grant` element in the workflow definition file. Read more about users, groups and administration in the *Administration Manual*.

Note that, when all eligible users for a task reject that task, the task is again offered to all eligible users. So if you are the only user for a *approver-role* group and you start a publication workflow, the second step of the workflow will be escalated. That is because you cannot be the composer and the approver of a resource - and there is no other user than you.

### Basic Steps in a Publication Workflow

After a user has created one or more documents, these documents shall be proofread, approved and published in a workflow:

1. The user (not necessarily the user who did the editing) starts a workflow. If he selects resources at starting time, these resources will be added to the change set and the compose task will be accepted automatically. Otherwise, he has to add the resources to the change set later.
2. The user completes the 'compose' task .
3. The task 'approve' is automatically offered to all appropriate users (members of the *approver-role* group, but not to the composer - even if he is a member of this group). Somebody accepts the task and approves the resources.

The user has the following options:

option A	option B	option C	option D
The user accepts the task, approves the resource(s) and finishes the task. All resources are approved.	The user accepts the task, does not approve all resource(s) and finishes the task	The user rejects the task.	The user accepts the task but delegates it to somebody else.

Table 8: User options.

option A	option B	option C	option D
The task 'Publication' is offered to all members of the group <i>publisher-role</i> .	The change set is sent back to the last editor.	The task is offered all other members of the group approver-role.	The task is automatically accepted by this user.

## 5 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 5.1 "[Defining Workflows](#)"<sup>(32)</sup> 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 5.2 "[Upload Workflow Definitions](#)"<sup>(59)</sup> describes how you can upload your workflow definition to the workflow server.
- » Section 5.3 "[Example of Workflow Definition](#)"<sup>(60)</sup> gives an example on how to define a workflow.
- » In Section 7.1 "[XML Element Reference](#)"<sup>(141)</sup>, all elements of the XML workflow description language are described as a reference.

The BeanParser, that is used to parse the *CoreMedia Workflow* definition allows 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.



## 5.1 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<sup>[33]</sup> 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 document.
- » A user of the chief editor role now has to read the resulting document and judge whether to approve or disapprove it.
- » If the document 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.

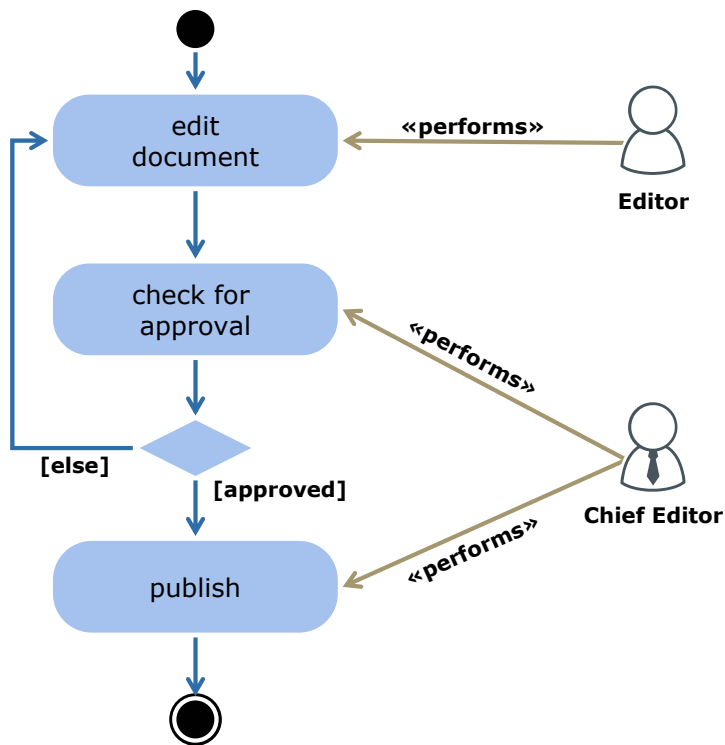


Figure 4: Activity diagram of a simple workflow

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 7.1 "XML Element Reference"<sup>(141)</sup>, the workflow XML reference.

In the *CoreMedia Workflow*, a workflow is defined in a file using XML syntax comparable to the configuration of the *CoreMedia Editor*. A formal description of the syntax of this XML file can be found in the corresponding DTD in `<InstallationDir>/lib/xml/coremedia-workflow.dtd`. 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 9: Workflow elements vs. programming language*

## 5.1.1 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 document types and workflows,
- » configure editor, watchdog, retrieval, feeder and syndicator.

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 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)`, i.e. set the object created in step 3 on the object created in step 1.

### Advanced features:

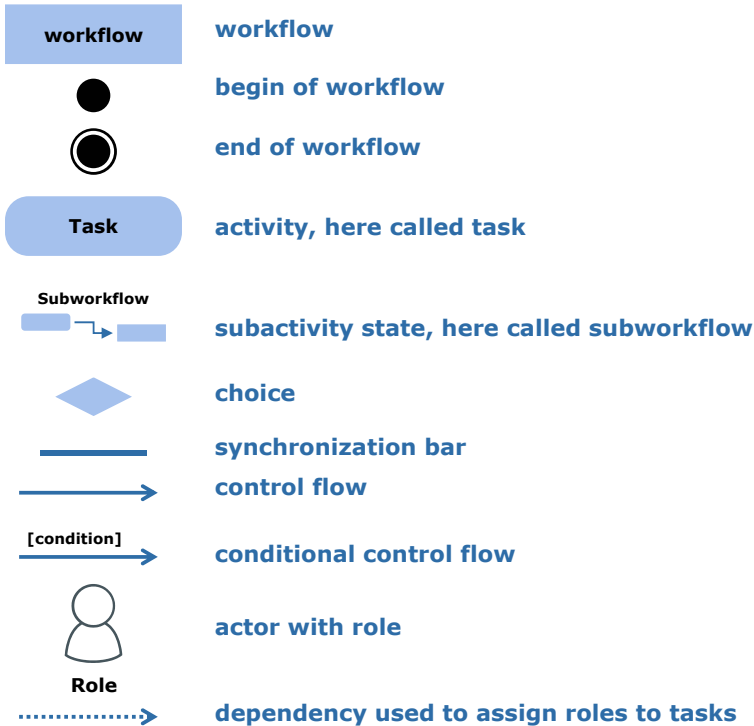
The class attribute has a special meaning as it determines the name of the class to instantiate objects from. For this attribute, no setter methods has to be defined inside the class.

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.

## 5.1.2 Used 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 5: 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*.

» Subactivity 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 follow-up 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. We introduce 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. We use special dependencies to connect roles (see above) with user tasks in order to assign rights.

## 5.1.3 Processes

Each workflow definition describes one process. A process can take several states as shown in Figure 6<sup>[39]</sup>.

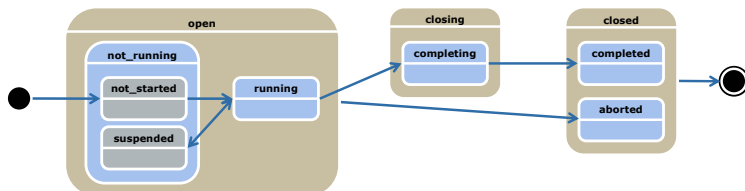


Figure 6: State diagram of a process

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 Client GUI 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`, i.e. 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 e.g. approved resources remain approved.

## 5.1.4 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 7](#)<sup>(40)</sup>.

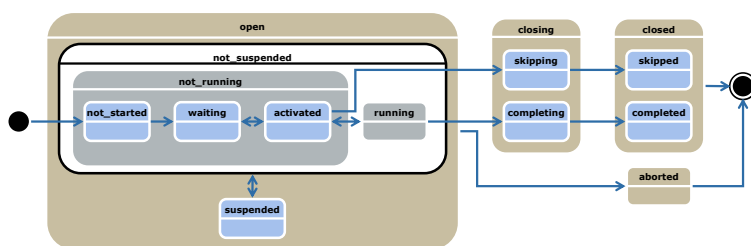


Figure 7: State diagram of a task

Different operations are possible or mandatory during the execution of a task instance to enter or leave the different states. Which operations are allowed to a user is defined by a configurable 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 i.e. if the following conditions have been fulfilled: <ul style="list-style-type: none"> <li>» The control flow of the workflow has reached the task.</li> <li>» The optional guard specified in the task definition evaluates to "true".</li> </ul>

Table 10: Status of Tasks

State	Enter State	Leave State
activated	<p>This state is entered automatically, after the waiting state has been left.</p> <p><b>Assigning a task</b>                      If this state has been entered, you can nominate a user or group for this task via <code>Task.assignTo()</code>. i.e. 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.</p> <p><b>Rejecting a task</b>                      A user can also reject the task via <code>Task.reject()</code>, 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).</p> <p><b>Canceling a task</b>                      The state activated is also entered if a task was accepted by a user and then cancelled by this user via <code>Task.cancel()</code>. 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.</p>	<p>A user task must be accepted by the user in order to leave the state 'activated' via <code>Task.accept()</code>. 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.</p> <p>Another way to leave the state is to skip the task via <code>Task.skip()</code>, switching to the state 'skipping'. A fallback to waiting is possible.</p>
suspended	<p>This state can only be entered via <code>Process.suspend()</code> which suspends the workflow. All task are withdrawn from the task list (GUI specific).</p>	<p>This state can be left via <code>Task.resume()</code>. The workflow will restart at the same task where it was suspended.</p>

State	Enter State	Leave State
running	<p>If an automated task has been activated it automatically leaves the state 'activated' and changes to 'running'.</p> <p>A user task must be accepted by the user via <code>Task . accept ( )</code> in order to enter the state 'running'. The task is then performed by the user and is no more available to other users.</p>	<p>An automated task leaves the state 'running' and enters one of the states 'completed' (via <code>Task . complete ( )</code>) and 'aborted' depending on the success of the actions and pre- and postconditions performed.</p> <p>A user task can leave the state 'running' and enter one of the states 'waiting', 'completed' (via 'completing') and 'aborted'.</p> <p>'Activated' is reached, when the user cancels the task via <code>Task . cancel ( )</code>. All changes made so far by the user are saved, but the task is offered again to all appropriate users.</p> <p>'Completed' is reached, when the task is completed via <code>Task . complete ( )</code> and the exit actions execute successfully and the post conditions evaluate to "true".</p> <p>'Aborted' is reached, when one of the exit actions and postconditions fails.</p>
skipping	Intermediate state.	Intermediate state.
skipped	This state is entered if the task has been skipped by a user via <code>Task . skip ( )</code> . The process continues with the following task.	This state can only be left, when the flow of operation returns to the task. I.e. there is a loop in the workflow definition which returns to the task.
completing	Intermediate state.	Intermediate state.
completed	<p>An automated task enters this state when all actions have been successfully performed and the pre- and postconditions have been evaluated to "true".</p> <p>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".</p>	This state can only be left, when the flow of operation returns to the task. I.e. there is a loop in the workflow definition which returns to the task.

State	Enter State	Leave State
aborted	This state is entered if the process is aborted via <code>Process.abort()</code> .	Final state.
escalated	This state is entered automatically when an error occurs, if e.g. 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 <code>Task.retry()</code> . This will retry the last operation, which has failed: e.g. 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.

## 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 5.1.6 "Workflow Variables"<sup>[51]</sup>.

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).

## 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 *WfPerformer-SPolicy*.

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 *CoreMedia Editor*. While an automated task can change resources (e.g. check out a document), a user task can even open a document view or start a publication with graphical feedback. For a list of possible actions, see Section 7.1 "[XML Element Reference](#)"<sup>[141]</sup> and Section 5.4.1 "[Predefined Action Classes](#)"<sup>[70]</sup>.

Validators (see Section 5.1.7.4 "[Validators](#)"<sup>[53]</sup>) 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 5.1.4.1 "[Common Features of All Tasks](#)"<sup>[43]</sup>).

## Automated Tasks

Automated tasks usually consist of an action sequence, an optional guard and pre- 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 5.1.4.1 "[Common Features of All Tasks](#)"<sup>(43)</sup>.

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 its 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.

## 5.1.5 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.

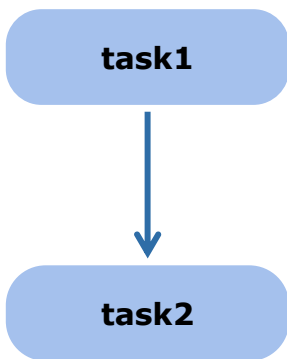


Figure 8: Example of a sequence diagram

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 2: Example listing of a sequence

### Choice

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

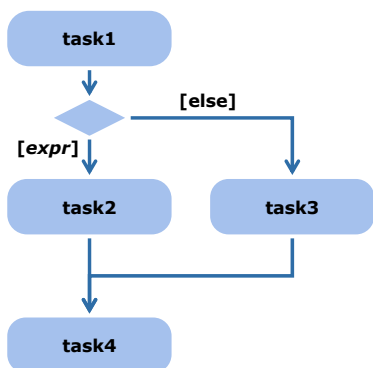


Figure 9: Example of a choice diagram

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 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.

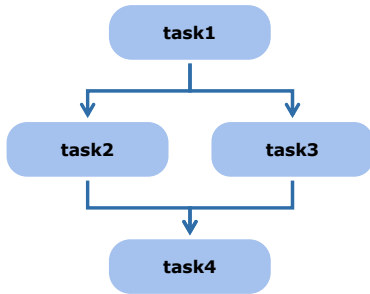


Figure 10: Example of an implicit choice

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: 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.

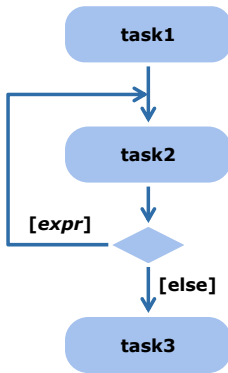


Figure 11: Example of a loop

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 5: Example listing of a loop

### Concurrency/Parallel Execution

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

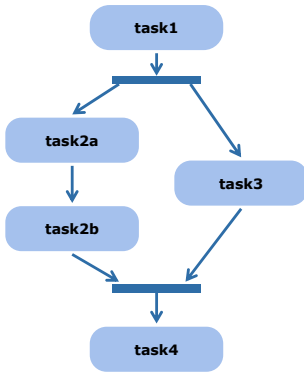


Figure 12: Example of a concurrency diagram

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 6: Example listing of concurrency

## 5.1.6 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 5.1.4.2 "[User Tasks](#)"<sup>(44)</sup>). 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. Of course, its possible to create and configure a custom aggregation editor for the Client GUI.

## 5.1.7 Expressions

Expressions are used to specify conditions in validators, guards, pre- or postconditions 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.

### Conditions

Conditions are used to define how processing should proceed. They are expressions which evaluate to a boolean value. If specified in

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

### Pre- and Postconditions

Pre- 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 pre- and postconditions.

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

## 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 7](#)<sup>(53)</sup> 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`). I.e., the task continues when the document is checked in.

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

*Example 7: Example of a Guard*

## Validators

Validators are boolean expressions that ensure that the variables that may be modified via a client view satisfy certain constraints. E.g., 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".

## 5.1.8 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 or during an automated task:

### 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 pre- and conditions. If pre- or postconditions are violated, the task is aborted.

All actions except for actions in automated tasks are executed with the rights and on behalf of the user who accepted the task. Actions in automated tasks run with the workflow servers "user" account at the Content Management Server.

## 5.1.9 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.
- » A revoke precedes a grant.
- » Concrete rights for a user precede rights for a group the user is member of.
- » Rights for a group precede rights for another group if the other group is a subgroup of the first 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 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.

## 5.1.10 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 5.1.5 "Flow Control"<sup>[46]</sup>) 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.

## 5.1.11 Timers

Timers can be used to define time spans or moments in the execution of a workflow. E.g. 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 5.4.2 "Predefined TimerHandler Classes"<sup>[79]</sup>). The handler *must* be defined in the same location, i.e. the process or task definition, where it's 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 usefull in the Client GUI.

The timer can be started and stopped using the actions `EnableTimer` and `DisableTimer` (see Section 5.4.1 "Predefined Action Classes"<sup>[70]</sup>). 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 9: Example of a self-defined timer which expires after 100 seconds*

## 5.2 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 (see *Administration and Operation Manual* for a detailed description) or the menu item **File|Load** in the workflow window of the *CoreMedia Editor*.

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 *Administration and Operation Manual* for a detailed description).

## 5.3 Example of Workflow Definition

Here the definition of a workflow is shown by means of the 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 120](#)<sup>(191)</sup> you find the complete XML definition of this workflow.

The workflow definition consists of seven blocks:

- >> the general definitions of the workflow
- >> automated task AssignComposer
- >> user task Compose
- >> if task CheckEmptyChangeset
- >> user task Publish
- >> if task CheckPublication
- >> automated task Finish

These seven blocks will be illustrated in detail.

### General definitions

```

1:  <?xml version="1.0" encoding="iso-8859-1"?>
2:
3:  <Workflow>
4:    <Process name="SimplePublication"
5:           description="simple-publication"
6:           startTask="AssignComposer">
7:      <Rights>
8:        <Grant user="admin"
9:              rights="create, start, suspend, resume, abort"/>
10:       <Grant group="composer-role"
11:            rights="create, start, suspend, resume, abort"/>
12:      </Rights>
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:      <Variable name="publicationSuccessful" type="Boolean">
18:        <Boolean value="false"/>
19:      </Variable>
20:      <AggregationVariable name="publicationResultResources"

```

*Example 10: General definitions of the workflow*

```

21:         type="Resource" />
    <AggregationVariable name="publicationResultCodes"
        type="Integer" />
22: <AggregationVariable name="publicationResultVersions"
        type="Integer" />
23: <AggregationVariable name="publicationResultParams"
        type="String" />
24:
25: <InitialAssignment>
26:     <Writes variable="subject" />
27:     <Writes variable="comment" />
28:     <Writes variable="changeSet" />
29:     <Writes variable="comments" />
30: </InitialAssignment>
31:
32: <Assignment>
33:     <Reads variable="subject" />
34:     <Reads variable="comment" />
35:     <Reads variable="changeSet" />
36:     <Reads variable="comments" />
37: </Assignment>
38: .
39: .
40: .
41: </Process>
42: </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 7 - 10:* The rights (see Section 5.1.9 "[Rights](#)"<sup>[55]</sup>) 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 12 - 23:* 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 25 - 30:* 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 32 - 37:* 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 'AssignComposer'

```

1:     <AutomatedTask name="AssignComposer"
2:         description="assigncomposer-task"
3:         successor="Compose">
4:         <Action class="ForceUser" task="Compose"
5:             userVariable="OWNER_" />
6:     </AutomatedTask>

```

*Example 11: Automated task 'Assign Composer'*

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

*Line 1 - 2 + 4:* The automated task is named 'AssignComposer'. The names of tasks are used in the definition of a successor of a task. The task, that is started after task 'AssignComposer' completes, is 'Compose'.

*Line 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` task 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.

### User Task 'Compose'

```

1:     <UserTask name="Compose"
2:         description="simple-publication-compose-task"
3:         successor="CheckEmptyChangeSet">
4:         <Rights>
5:             <Grant user="admin" rights="read,accept,
6:                 delegate,skip"/>
7:             <Grant group="composer-role" rights="read,
8:                 accept,delegate,skip"/>
9:         </Rights>
10:        <Assignment>
11:            <Writes variable="subject"/>

```

*Example 12: User Task Compose*

```

11:         <Writes variable="comment" />
12:         <Writes variable="changeSet" contentEditable="true" />
13:         <Writes variable="comments" />
14:         <Reads variable="publicationResultCodes" />
15:
16:         <Validator name="AllCheckedIn"
17:             description="all-checked-in-validator">
18:             <!-- condition: every document with version in
19:             changeSet is checked-in -->
20:             <ForAll variable="change" aggregate="changeSet">
21:                 <Implies>
22:                     <And>
23:                         <IsDocumentVersion variable="change" />
24:                         <Equal>
25:                             <Read variable="change"
26:                                 property="version_" />
27:                             <Read variable="change"
28:                                 property="latestVersion_" />
29:                         </Equal>
30:                     </And>
31:                 </Implies>
32:             </ForAll>
33:         </Validator>
34:     </Assignment>
35:     <ExitAction class="PreferPerformer" />
36:     <ExitAction class="ForceUser" task="Publish" />
37: </UserTask>

```

This is the second task in the workflow and the first user task. The purpose of the task is to enable the user to collect the documents which should be published at once.

*Line 1 - 3:* 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 Client GUI. The task 'CheckEmptyChangeSet' is started after task 'Compose' has completed.

*Line 4 - 7:* The rights concerning the task are assigned to users and groups. The user *admin* 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.

*Line 9 - 14:* 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 documents, 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 ...>`.

*Line 16 - 30:* A validator is defined and named "AllCheckedIn". The localized version of a corresponding message is shown in the dialog box, which appears when the validator evaluates to "false". The description is looked up in a resource bundle under the key "all-checked-in-validator" and if that fails under "AllCheckedIn" to localize it. The validator is used to check whether all documents which should be published are checked in or not. If not, the user is not allowed to finish the Compose task until all documents are checked in.

*Line 18:* The validator uses an iteration over all elements of the `Aggregation-Variable` `changeSet`. The variable `changeSet` contains all resources which were added to the change set in the Compose task. The current resource is stored in the loop variable "change".

*Line 19:* The `<Implies>` expression ensures that the change set contains only document versions and that every document must be checked in, if its version in the change set is the latest version. If the version in the change set is not the latest version, then the document may be checked in or checked out.

*Line 20:* The expressions contained in the `And` element must all evaluate to "true" so that the whole `<And>` element evaluates to "true".

*Line 21:* The element `<IsDocumentVersion variable="change" />` checks if the current resource stored in "change" is a version of a document. Otherwise it is not possible to publish the document.

*Line 22:* The sub expression results defined in the `Equal` element are checked for equity.

*Line 24:* The value of the property `latestVersion_` of the resource, which is referenced by the loop variable `change`, is read and compared to the value of the property `version_`. Thus it is checked, if the version of the document in the change set equals the latest version of the document.

*Line 27:* The `<Not>` element negates the result of the boolean expression it contains. Here this is the property `isCheckedOut_` of the resource, which is referenced by the loop variable `change`. If the document is checked out, the variable is "true". Thus, the whole `<Not>` expression evaluates to "true", if the document is not checked out.

*Line 33:* An exit action is defined which is automatically carried out right after the task was completed by the user. The action to perform is defined by the class `PreferPerformer`. I.e. if the "Compose" task has to be repeated because of a failing publication, then the "Compose" task should preferably be offered to the previous performer.

*Line 34:* It is possible to define more than one exit action. Here the predefined `ForceUser` action is used, which assigns the current task's performer to the task defined in `task`. Thus, the `Publish` task will be automatically accepted by the user who performed the `Compose` task.

### 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 13: If Task

The third 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 description is necessary, which should be shown in the GUI.

*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'

Example 14: User Task  
"Publish"

```

1: <UserTask name="Publish"
2:   description="simple-publication-publish-task"
3:   successor="CheckPublication" autoCompleted="true">
4:   <Rights>
5:     <Grant user="admin" 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"
13:      variable="changeSet"
14:      contentEditable="false"/>
15:    <Reads variable="comments"/>
16:   </Assignment>
17:   <EntryAction class="ApproveResource" gui="true"
18:     resourceVariable="changeSet"
19:     successVariable="publicationSuccessful"
20:     ignoreErrors="true"
21:     timeout="180">
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"
31:     ignoreErrors="false"
32:     ignorePublicationErrors="true" timeout="600"/>
33: </UserTask>

```

The fourth 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 'Compose' task, that forced the performer of the 'Compose' task as the performer of the 'Publish' 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 user `admin` and the group `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 documents given via the attribute `resourceVariable`.

*Line 24 - 31:* After executing the first entry action, the second one will be performed. Here the documents 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 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 - 2:* 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" />
```

*Example 16: Example of automated task  
CheckPublication*

The last task of the workflow is an automated task which is only needed because the previous `If` task may not be the final task of the workflow. Thus, this `AutomatedTask` contains no action to perform.

*Line 1:* The automated task is named 'Finish'. Because the task should be the last element 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.

## 5.4 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*.

## 5.4.1 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 11: 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	<i>{boolean}</i>	"true"	Defines whether a GUI element will be shown on execution of the action ("true") or not. E.g. executing <code>publishResources</code> with <code>gui="false"</code> will not show the publication window.
ignoreErrors	<i>{boolean}</i>	"false"	If set to "true", this attribute makes sure that the task containing the action will continue normally after an error was encountered.

Table 12: Attributes of client-side actions.

Attribute	Type	Default	Description
timeout	NMTOKEN	"30"	The timeout in seconds for the action. The default timeout is 30 seconds.

### 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 13: Attributes of the ApproveResource action.

### CheckInDocument

Use this action to check-in one or more CoreMedia documents 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 14: Attributes of the CheckInDocument action.

### CheckOutDocument

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

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

Table 15: Attribute of the CheckOutDocument 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

Table 16: Attributes of the CopyResource action.

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

### CreateDocument

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

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

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 document type for which a document should be created

Table 17: 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 18: Attributes of the CreateFolder action.

### DeleteResource

Use this action to mark a resource for deletion.

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

Table 19: 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 20: 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 21: Attributes of the MoveResource action.

### OpenDocument

Use this action to open a document in the editor.

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

Table 22: 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"	whether an unsuccessful publication should be ignored

Table 23: Attributes of the PublishResources action.

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

### 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 24: Attributes of the RenameResource action.

### SaveDocument

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

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

Table 25: Attribute of the SaveDocument action.

### StoreProperties

Use this action to store property values in a document. 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 document

Table 26: Attribute of the StoreProperties action.

### UncheckOutDocument

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

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

Table 27: Attribute of the *UncheckOutDocument* action.

### UndeleteResource

Use this action to remove the deletion from a resource.

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

Table 28: 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*.

#### 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 WfValues 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 17: 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 29: 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 30: 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 31: Attribute of the ExcludePerformer 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 18: 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 32: 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`, `error` or `info` (see Section "Logging" of the Administrator Manual for details on the logging).

Attribute	Description
<code>debug error info</code>	Defines the log level "debug", "error" or "info". Value must be "true".
<code>message</code>	The message which is printed to the log.
<code>facility</code>	Define an own log name for the output. If you use this attribute you can define an own log target for the output in the <code>workflowserver.properties</code> file e.g.: <code>log.action.3.class=FileAction</code> <code>log.action.3.selectors=workflow.&lt;MyLogName&gt;:info</code> <code>log.action.3.initArgs=</code>

Table 33: Attributes of the Log action.

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

Example 19: 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
<code>task</code>	NMTOKEN	#IMPLIED current task	the name of the task for which a preference should be established

Table 34: Attribute of the PreferPerformer 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 35: 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 36: Attribute of the SkipUserTask action.

### InterruptUserTask

Use this action to interrupt an activated user task regardless of its current state and switch the state to `completing` without running any actions. Be careful when using this action, as it may lead to logically inconsistent workflow states.

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

Table 37: Attribute of the InterruptUserTask action.

## 5.4.2 Predefined TimerHandler Classes

Timer handler classes are invoked when a timer expires.

### Example definition of timer handlers

```
<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="InterruptUserTask" 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 20: Example of TimerHandler usage

### AbortTaskTimerHandler

This timer handler aborts the task instance in which it is defined on expiration (see [Figure 13](#)<sup>[79]</sup>).

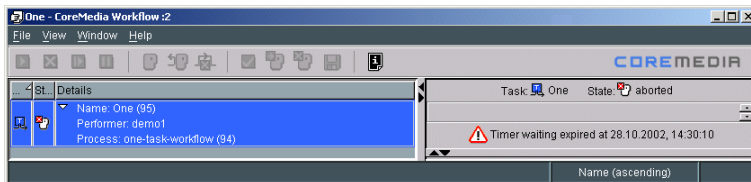


Figure 13: Expired timer with AbortTaskTimerHandler

### AddWarningTimerHandler

This timer handler adds a timer expiration warning to a process or task instance (see [Figure 14](#)<sup>[80]</sup>).

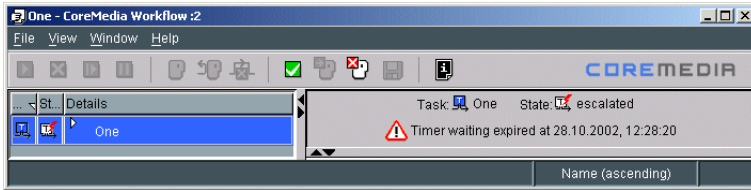


Figure 14: Expired timer with AddWarningTimerHandler

### 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.

## 5.4.3 Predefined Editor Classes

The Client GUI comes with a set of property editors to allow out-of-the-box editing of workflow variables. These predefined property editor classes may be configured and/or replaced by custom property editors.

### Predefined property editor classes

These classes are editing components which will be used for workflow variables shown in the variable view within the element `<Variable>`. These editors are the defaults for the appropriate variables, e.g. a variable of type `Boolean` will be edited using the `JCheckBoxBooleanEditor`.

Class	Description
<code>JCheckBoxBooleanEditor</code>	A check box. "True" is shown as checked, "False" as unchecked.
<code>ComboBoxDocumentTypeEditor</code>	A combo box with the allowed document types.
<code>GroupChooserEditor</code>	A window which shows the available groups.
<code>ResourceChooserEditor</code>	A resource chooser window, like the one in the editor window.
<code>UserChooserEditor</code>	A window which shows the available users.

Table 38: Predefined property editor classes

### Example invocation of own editors

In the workflow definition you have defined a variable as follows:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Workflow>
  <Process name="Test">
    .
    .
    <Variable name="MainUser" type="User"/>
    .
    .
  </Process>
</Workflow>
```

Example 21: Workflow definition of a variable

In the workflow definition above you find the definition of a variable `MainUser` of type `User`. To use your own user editor with this variable you have to insert the following lines in your `editor.xml` file.

```
<Process name="Test">
  <View>
    <Variable name="MainUser"
              editorClass="MyPackage.MyUserChooser" />
  </View>
  .
  .
</Process>
```

*Example 22: Integrate your own variable editor in the editor.xml file*

## 5.4.4 Predefined Column Classes

Column classes define how content is rendered and sorted in table columns of the Client GUI. The classes are shown in [Table 39](#)<sup>(83)</sup>. The workflow column classes are defined in the `editor.xml` file.

Alternatively, a renderer can be set for each column class. The renderer carries out the display itself. For this purpose, an attribute, `renderer="Renderer class"` has to be embedded in the `ColumnDefinition` element. The renderer must be a subclass of the class `hox.corem.editor.toolkit.columnrenderer.LayoutColumnRenderer`. See the API documentation for details.

```
<Workflow>
  <TableDefinition rowHeight="24">
    <ColumnDefinition
      class="hox.corem.editor.workflow.columns.TaskTypeColumn" />
    .
    .
  </TableDefinition>
</Workflow>
```

*Example 23: Integrate the column class into the workflow window*

### Column classes for workflows

Class	Description
<code>hox.corem.editor.workflow.columns.TaskTypeColumn</code>	Using this class, the type of the process or task is shown in the workflow list by its icon.
<code>hox.corem.editor.workflow.columns.TaskStateColumn</code>	Using this class, the state of the task or process is shown in the workflow list by an icon.

*Table 39: Column Classes*

Class	Description
<p>hox.corem.editor.workflow.columns.TaskDataColumn</p>	<p>Using this class, additional information concerning the task or process is shown. If you use this class, the sub element <code>&lt;ComponentFactory&gt;</code> has to be set. The following attributes can be used to define the information to show:</p> <ul style="list-style-type: none"> <li>» <b>class:</b> The class which is used to show the content.                     <ul style="list-style-type: none"> <li><code>hox.corem.editor.workflow.columns.DefaultDataPanelFactory</code> will show the information owned by all tasks or processes.</li> <li><code>hox.corem.editor.workflow.foureyes.FourEyesDataPanelFactory</code> will show the information unique to the processes of the publication workflows.</li> </ul>                     Own classes can be programmed. They must implement <code>WfInstanceComponentFactory</code>.                 </li> <li>» <b>processName:</b> The name of the process, for which the class should be used.</li> <li>» <b>taskName:</b> The name of the task, for which the class should be used. If this attribute is set, <code>processName</code> must also be set. Then, the class should be used for the specified task belonging to the specified process.</li> </ul>

## 6 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 *Editor 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 plug-ins. 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 *CoreMedia Editor* and in the *CoreMedia Active Delivery Server*, 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.

## 6.1 Spring in The Workflow Server

You can use Spring to make Java Beans available to your customized Workflow actions and expressions. The CoreMedia Demo for example uses this mechanism to provide e-mail functionality to a customized action. The Spring context is loaded at startup time and is shut down when the server is shut down. The Spring XML 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.

The Spring XML file can refer to properties defined in `workflowserver.properties`, or in any property file under `config/workflowserver/spring/*.properties`, using spring's property placeholder syntax: `${propertyname}`. The property's value is inserted when the application context is loaded.

If you want the property to be updated when the property file changes, you can use an alternative syntax: `#{propertyname}`. This causes all affected bean property setters to be invoked whenever the property file is reloaded and the value has changed.

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

1. Make a Spring `ApplicationContext` available to the *Workflow Server*. To do so, configure a `SpringContextManager` in the `workflowserver.properties` file:

```
workflow.server.managers.springcontext.class=
com.coremedia.workflow.common.util.SpringContextManager
```

2. Configure the locations of the Spring beans configuration files:

```
workflow.server.managers.springcontext.configurations=
classpath*:framework/spring/*.xml,
config/workflowserver/spring/*.xml
```

3. Configure the beans you want to use in any of the configured configuration files. When using the default configuration files locations, the file `config/workflowserver/spring/beans.xml` is a good place to start.
4. Let your customized actions, expressions or boolean expressions extend `com.coremedia.workflow.common.util.SpringAwareAction`, `com.coremedia.workflow.common.util.SpringAwareExpression` and `com.coremedia.workflow.common.util.SpringAwareBooleanExpression` respectively.
5. 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 interfaces `InitializingBean` and `DisposableBean` in order to receive life cycle events from the context manager. 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.

## 6.2 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 [see *Administration and Operation Manual* for details]. In case of an update of the *CoreMedia Workflow Server*, the workflows have to be converted, too. Otherwise, object deserialization errors can occur [see Suns JDK documentation for details].

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 6.9 "[Pitfalls of Implemented Classes](#)"<sup>[124]</sup> for more hints on this topic.

## 6.3 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:

- >> Booleans
- >> 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

## 6.4 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.

## 6.4.1 General Rules

Actions can only be used in automated tasks, user tasks or in the predefined `RunActionTimerHandler`. 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.

Actions should run for shortest period of time 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 5.1.7.3 "[Guards](#)"<sup>(53)</sup>) following the initial task with your action. The example in Section 6.5.4 "[Example Expression](#)"<sup>(112)</sup> 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 JavaBeans.
- » Parameters for the global configuration of the action bean can be defined via the XML workflow definition (see Section 6.4.5 "[Access Workflow Variables from the Action](#)"<sup>(99)</sup>).
- » 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 6.4.5 "[Access Workflow Variables from the Action](#)"<sup>(99)</sup> for details).

## 6.4.2 Repeated Execution of Actions

If there are concurrent running transactions in an instance (i.e. 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 again 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.

## 6.4.3 Server-Side Actions

### Interface to implement

Server-side actions must 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 6.4.6 "[Example Action](#)"<sup>(101)</sup> you will find a complete example of a server-side action.

## 6.4.4 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. E.g. the event listener belonging to the *CoreMedia Editor* will execute the action the server noticed in the event.

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 implement-

actions of all needed getter and setter methods, uses a condition as guard and contains all of 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 24](#)<sup>[95]</sup> 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 in Section 6.4.4 "[Client-Side Actions](#)"<sup>[94]</sup>.
- » Receives the clients result and creates an `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.coremedia.training.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 24: 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 `documentType` 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 25<sup>\[96\]</sup>](#)) 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 *CoreMedia Editor* 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.coremedia.training.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++) {

```

Example 25: Example of an action listener

```

15:         System.out.println("parameters["+i+"]="+
                               parameters[i]);
16:     }
17:     return new ClientActionResult(actionEvent.
                                   getCallbackId());
18: }
19: }

```

*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 CoreMedia Editor

In [Example 26](#)<sup>[97]</sup> you see an example command which is executed on the client when an appropriate event is received by the event listener of the *CoreMedia Editor*. For this, the action defined in [Example 24](#)<sup>[95]</sup> has to be executed.

```

1: package com.coremedia.training.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 26: Command executable on the CoreMedia Editor*

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

*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.

## 6.4.5 Access Workflow Variables from the Action

It is good practice not to hardcode 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 27](#)<sup>[99]</sup>
- » Define a setter method in your action for the configuration like in [Example 28](#)<sup>[99]</sup>.
- » 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
6:     class="com.coremedia.training.workflow.action.ParameterAction"
7:     variableToPass="MyFirstVariable" />
8: </AutomatedTask>

```

*Example 27: 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.coremedia.training.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)
7:     throws WfException {
8:     ...
9:     WfAtomicVariable variable =
10:       wfTaskInstance.getAtomicVariable(variableToPass);
11:   }

```

*Example 28: 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.

## 6.4.6 Example Action

This chapter describes how to create an action which sends a mail to a receiver. The receiver and the document to send can be passed to the action using the *CoreMedia Editor*. You will learn how to pass variables to the action, how to return variables and how to access the *Content Management Server* and the *CoreMedia Workflow Server*.

For clarity some methods or variable definitions are removed from the following excerpts. You will find the complete code of the action in Section 7.3 "[Complete Code of the Mail Action](#)"<sup>[194]</sup>.

### The Framework

In this chapter you will find the "framework" of your action, e.g. the packages to import or the class to inherit from. See [Example 29](#)<sup>[101]</sup> for the package declaration, imports and class declaration of the action code.

```

1: package com.coremedia.extension.workflow.mail;
2:
3: import javax.mail.*;
4: import javax.mail.event.*;
5: import javax.mail.internet.*;
6: import com.coremedia.workflow.*;
7: import com.coremedia.workflow.common.actions.*;
8: import com.coremedia.workflow.common.values.*;
9: import com.coremedia.cap.content.ContentException;
11:
12: public class SendMail extends AbstractResourceAction {

```

*Example 29: Framework of the action*

*Line 1:* The package to which the action belongs. Choose your own one.

*Lines 3 - 5:* The Java packages which are necessary for mail usage.

*Line 6 - 9:* The packages for the workflow.

*Line 12:* If you implement actions, it is necessary to import the interface `com.coremedia.workflow.WfAction`. The recommended solution is to extend `AbstractResourceAction` which implements `WfAction`.

### Access workflow variables from the action

The example XML workflow definition shows how to configure the action bean to access the mail address of the receiver and how to access the document from the workflow instance. The success of the action will be returned to the `successVariable`. The necessary settings and the definition of the action are shown in the following:

```

1: <Workflow>
2:   <Process name="SendMail" startTask="mail">
3:     <Rights>
4:       <Grant user="admin" rights="create, start, suspend,
5:         resume, abort, read, write"/>
6:       <Grant group="composer-role" rights="create, start,
7:         suspend, resume, abort"/>
8:     </Rights>
9:     <Variable name="receiver" type="String">
10:      <String value="default@test.com"/>
11:    </Variable>
12:    <Variable name="field" type="String"/>
13:    <Variable name="document" type="Resource"/>
14:    <Variable name="delivered" type="Boolean">
15:      <Boolean value="false"/>
16:    </Variable>
17:    <UserTask name="mail" final="true">
18:      <Rights>
19:        <Grant user="admin" rights="read, accept, delegate,
20:          skip"/>
21:        <Grant group="composer-role" rights="read, accept,
22:          delegate, skip"/>
23:      </Rights>
24:      <Client>
25:        <Writes variable="receiver"
26:          contentEditable="true"/>
27:        <Writes variable="field"
28:          contentEditable="true"/>
29:        <Writes variable="document"
30:          contentEditable="true"/>
31:      </Client>
32:      <ExitAction
33:        class="com.coremedia.extension.workflow.mail.SendMail"
34:        receiverVariable="receiver"
35:        documentVariable="document"
36:        fieldVariable="field"
37:        successVariable="delivered" />
38:    </UserTask>
39:  </Process>
40: </Workflow>

```

*Lines 7 - 14:* Here you define the workflow variables from which the action will take the information for the mail. As shown for the variable `receiver`, you can define default values.

*Lines 20 - 24:* Here you define a client view which permits the performer of the task to enter the receiver, the field and the document to send of the mail.

*Line 25:* In this line you will configure the action bean. The action bean to use is defined via the attribute `class`. The names of the variables are set using the attributes in the lines below. Here the component which analyzes the workflow definition (the `BeanParser`) will call the `setReceiverVariable()`, `setDocumentVariable()`, `setFieldVariable()` and `setSuccessVariable()` of the action bean using the variable names as parameters.

You will see right away in [Example 5.4.6c<sup>\[?\]</sup>](#) how to prepare your action to accept these attributes.

```

1: public class SendMail extends AbstractResourceAction {
2:     static final long serialVersionUID=1258062873454333627L;
3:     protected String transportType = "smtp";
4:     protected String host = "smtp.coremedia.com";
5:     protected String user = "testuser";
6:     protected String password = "testpassword";
7:     protected String from = "testuser@coremedia.com";
8:     protected String receiverVariable;
9:     protected String fieldVariable;
10:    protected String documentVariable;
11:    protected String subject = "This is a test mail";
12:
13:    protected String getStringValue(WfInstance instance,
                                   String variableName)
14:        throws WfException {
15:        if(variableName != null) {
16:            try {
17:                WfAtomicVariable variable = instance.
18:                    getAtomicVariable(variableName);
19:                WfValue value = variable.getValue();
20:                if(value instanceof StringValue) {
21:                    String resourceName = ((StringValue)value).
22:                        getString();
23:                    if(resourceName == null) {
24:                        throw new WfException(WfException.
25:                            STRING_HAS_NULL_VALUE,
26:                            new String[] {this.toString(), variableName});
27:                    }
28:                    return resourceName;
29:                }
30:                throw new WfException(WfException.
31:                    VALUE_HAS_WRONG_TYPE,
32:                    new String[] {this.toString(), "StringValue",
33:                        variableName});
34:            }
35:            catch(WfException e) {
36:                return variableName;
37:            }
38:        }
39:        throw new WfException(WfException.
40:            NO_PARAMETER_VARIABLE_SPECIFIED,
41:            this.toString());

```

*Example 31: How to pass variables*

```

36: }
37:
38: protected int getContentId(WfInstance instance,
39:                             String variableName)
40:     throws WfException{
41:     if(variableName != null) {
42:         WfAtomicVariable variable = instance.
43:             getAtomicVariable(variableName);
44:         WfValue value = variable.getValue();
45:         if(value instanceof ResourceValue) {
46:             String contentId = ((ResourceValue)value).
47:                 getContentId();
48:             if(contentId == null) {
49:                 throw new WfException(WfException.
50:                     RESOURCE_ID_IS_NULL,
51:                     new String[] {this.toString(), variableName});
52:             }
53:             return contentId;
54:         }
55:         throw new WfException(WfException.
56:             VALUE_HAS_WRONG_TYPE,
57:             new String[] {this.toString(), "ResourceValue",
58:                 variableName});
59:     }
60:     throw new WfException
61:     (WfException.NO_PARAMETER_VARIABLE_SPECIFIED,
62:     this.toString());
63: }
64: public WfActionResult execute(WfInstance instance)
65:     throws WfException
66:     {
67:     String to = getStringValue(instance, receiverVariable);
68:     String documentId =
69:         getContentId(instance, documentVariable);
70:     String field = getStringValue(instance, fieldVariable);
71:     String body = getXml (documentId, field);
72:     try {
73:         return new WfActionResult(send(host, user, password,
74:             from, to, subject, body));
75:     }
76:     catch(Exception e) {
77:         throw new WfException(WfException.ERROR,
78:             "Cannot send mail", e.getMessage());
79:     }
80: }
81:
82: public void setReceiverVariable(String receiverVariable) {
83:     this.receiverVariable = receiverVariable;
84: }
85:
86: public void setFieldVariable(String fieldVariable) {
87:     this.fieldVariable = fieldVariable;

```

```

77: }
78: public void setDocumentVariable(String documentVariable) {
79:     this.documentVariable = documentVariable;
80: }
81: }

```

*Line 3 - 11:* Make sure, you have defined the variables you want to use in the action.

*Line 13 - 36:* You can use this method to get the value of all atomic string variables. If called, you have to pass the workflow instance and the name of the variable as parameters. The important part is line 17. You use the method `getAtomicVariable` of the interface `WfInstance` to get the atomic variable (in comparison to an aggregation variable) defined via `variableName`. In line 18 you extract the content of the variable using the method `getValue`. If everything went fine, you return the value of the variable in line 25.

*Line 38 - 56:* You can use this method to get the id of a resource. The only difference to the method above is the extraction of the resource id in the lines 45 - 49. If the value of the workflow variable is of type `ResourceValue` you can get the resource id using the `getContentId()` method.

*Lines 57 - 70:* This is the method to be called from the workflow server. In our context of accessing variables the lines 59 - 62 are the interesting ones. Here you get the values of the workflow variables. In line 64 you have to return a `WfActionResult` object instantiated with the boolean result of the mail sending (you find the definition of the `send` method in the complete example in Section 7.3 "[Complete Code of the Mail Action](#)"<sup>[194]</sup>).

*Line 72 - 80:* Here you declare the setter methods of your action Bean for the configuration. The name of the methods must match the names of the attributes of the action in the workflow definition. The setter and getter methods of the success variable are defined in the `AbstractAction` class from which the example action inherits.

### Access the repository

In the following, we will show, how the action can access the session to the *Content Management Server* from the *CoreMediaWorkflow Server* to get the content of a document. See [Example 32](#)<sup>[105]</sup> for the appropriate lines of the action.

```

1: import com.coremedia.workflow.*;
2:
3: // Get the XML content of the document
4: protected String getXml(String id, String field)
5:     throws WfException {
6:     try {
7:         return WfServer.getConnection().getContentRepository().
           getContent(id).getMarkup(field).toString();

```

*Example 32: Access the content repository*

```
8:     }
9:     catch(ContentException e) {
10:         throw new WfException(WfException.ERROR,
11:             "Cannot fetch xml from resource id "+id, e.getMessage());
12:     }
13: }
```

*Line 1:* You have to import the `WfServer` class which you will find in `com.core-media.workflow`. You need it to access the repository.

*Line 4 - 13:* This method gets the content of the document defined via the given id. The important parts are line 6 and 7. Here the method `getConnection()` of the class `WfServer` is used to get a `CapConnection` object. With this object and its `getContentRepository()` method you can access the *CoreMedia Content Management Server* repository using the *Unified API*. With the method `getContent(id)` you retrieve a `Content` object. In the last step you fetch the XML representation from the `Content` using the method `getMarkup(field).toString()` which returns a `java.lang.String` [see the *Unified API* documentation for details]. This string may be inserted into the mail.

## 6.5 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 an `<Condition>` task.

Expressions can be used for many purposes in the workflow:

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

## 6.5.1 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 sub-expressions, 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.

## 6.5.2 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 34](#)<sup>[110]</sup> 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 33: 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 34: Example of a generic expression*

## 6.5.3 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 35: Example of a boolean expression*

## 6.5.4 Example Expression

This chapter describes how to create a boolean expression and insert it in the workflow definition. Have a look at [Example 37<sup>\[112\]</sup>](#) 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 36<sup>\[112\]</sup>](#) 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 36: 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 37<sup>\[112\]</sup>](#) 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() {
```

*Example 37: Example Expression*

```
18:     return false;
19:   }
20:
21:   public boolean evaluateExpression(WfInstance instance,
                                     Map localVariables) {
22:     return true;
23:   }
24: }
```

*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`.

## 6.6 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 `AcIRightsPolicy` and override the appropriate methods.

### Deploy the rights policy

You have to deploy the rights policy to the server and client. Proceed as follows.

### Deploy to the Workflow Server

1. Stop the *Workflow Server* with `cm workflowserver stop`
2. Copy the rights policy `.class`-file with the complete path to `<Workflowserver-Home>/classes` or copy the rights policy as a `jar`-file to `<Workflowserver-Home>/lib`.
3. Start the *Workflow Server* with `cm workflowserver start`
4. Upload the modified Workflow definition with `cm workflowupload -u admin -p <PassWord> -file <WorkflowDefinition>.xml`

### Deploy to the client

If you use the *CoreMedia Editor* via WebStart read section 4.5 "Installing Java Web Start" in the *Administration and Operation Manual*. There you will learn how to deploy *CoreMedia Editor* extensions via WebStart. If you use the *CoreMedia Editor* without WebStart proceed as follows:

1. Stop the *CoreMedia Editor*.
2. Copy the rights policy `.class` -file with the complete path to `<Editor-Home>/classes` or copy the rights policy as a `jar`-file to `<EditorHome>/lib`.
3. Start the *CoreMedia Editor*.

### 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](#)<sup>[?]</sup>.

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

Example 38: Integrate own rights policy in the workflow definition

## 6.6.1 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, e.g., in an intranet. To this end, the policy computes the members of a group only when absolutely 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, e.g., in the *CoreMedia Editor*.

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) {}
```

We need some methods for managing the policy configuration.

```
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, we 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 we 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);
}
...

```

We will now look at the central method for permission computation. First of all, we must make sure to grant all rights to the internal server user, which performs certain automated actions. We'd also like to grant all rights to the super administrator.

```

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) {

```

We 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 {

```

We 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 we 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 we 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*.

## 6.7 Programming Performer Policies

Performer policies control to which users a task instance should be offered. A performer 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 performer policy may optionally support:

- » Users who must be excluded from the offer (determined by the `ExcludePerformer` 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 of the options.

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 39](#)<sup>(121)</sup> 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 39: Defining a performer policy in the workflow definition*

### Customize the performer policy

See [Example 40](#)<sup>(121)</sup> 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 {
```

*Example 40: Invoking a performer policy*

```

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: }

```

*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, we 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.

## 6.8 Programming Clients

The *CoreMedia Workflow* comes with a workflow client integrated in the *CoreMedia Editor*. 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:44441/coremedia/ior" +
        "?useworkflow=true", "admin", "admin");
try {
    WorklistRepository r = connection.getWorkflowRepository();
    // ... work on the repository ...
} finally {
    connection.close();
}
```

Example 41: 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.

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.

## 6.9 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 Suns 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, e.g. 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.

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 ServerSession` etc. These objects may carry session specific information that is only valid to the current bean invocation.
- » Every bean must be reentrant, i.e. 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 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 are likely to cause problems. It might seem, that packaging all classes that are referenced by the customized workflow classes into one huge jar 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.

## 6.10 Customizing the Workflow GUI

The GUI of the *CoreMedia Workflow* is quite flexible. You can simply use the *CoreMedia Editor* configuration file (as default, this file is named `editor.xml`) to configure the GUI or even write your own startup classes or variable editors. Note that variable editors are described in detail in the *Editor Developer Manual*.

The BeanParser, that is used to parse the *CoreMedia Editor* configuration allows 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.



## 6.10.1 Configure the Workflow GUI

In the following section, you will find a description of all workflow-related elements of the `editor.xml` file, which you can use for the configuration. The description of some "universal" elements like `Predicate` which can be used in different contexts is limited to the use in the workflow context.

### <Workflow>

Child elements: <TableDefinition>

Parent elements: <Editor>

```
<Editor>
  <Workflow>
    <TableDefinition>
      .
    </TableDefinition>
  </Workflow>
</Editor>
```

Example 42: Example of the Workflow element

This element is used to define which information should be shown in the columns of the workflow list at the left side of the workflow window.

### <TableDefinition>

Child elements: <ColumnDefinition>\*

Parent elements: <Workflow>

```
<Workflow>
  <TableDefinition>
    <ColumnDefinition
      class="hox.corem.editor.workflow.columns.TaskTypeColumn" />
    .
  </TableDefinition>
</Workflow>
```

Example 43: Example of the TableDefinition element

Within this element of the editor configuration, the columns of the workflow list at the left side of the workflow window are configured.

Attribute	Description
rowHeight	This attribute determines the height of a row in the table. The height is given in pixels.

Table 40: Attribute of the TableDefinition element.

**<ColumnDefinition>**

Child elements: <DisplayMap>\*, <Comparator>?, <Renderer>?, <Varies>

Parent elements: <TableDefinition>

```
<TableDefinition>
  <ColumnDefinition
    class="hox.corem.editor.workflow.columns.TaskTypeColumn">
    .
    .
  </ColumnDefinition>
</TableDefinition>
```

Example 44: Example of the *ColumnDefinition* element

This element is the same as known from the editor configuration. It defines a column in the workflow list. In the context of the workflow GUI a column in the document view is defined which shows information of the workflow instance (WfInstance).

Attribute	Description
name	Name of the column which is shown in the header of the column. It may be localized as described in the <i>Editor Developer Manual</i> .
class	This attribute is used for selecting a class for displaying the column. This determines the field type which can be displayed. Furthermore, the class sorts the contents of the column. You find predefined classes in Section 5.4.4 " <a href="#">Predefined Column Classes</a> " <sup>[83]</sup> .
width	This attribute is used for defining the minimum width of the column in pixels. If the window width is smaller than the total sum of all column widths, a scroll bar appears. Scaling for a larger window is controlled with the attributes <code>weight</code> and <code>resizable</code> . The default value is 100 pixels.
weight	This attribute gives the relative weight of a column in scaling. Rational numbers are entered. The default value for all columns is "1.0".
resizable	This attribute is used for defining whether a column is resized at all. The default setting is "true", i.e. the column is enlarged. Resizing can be switched off with "false".

Table 41: Attributes of the *ColumnDefinition* element.

**<Processes>**

Child elements: <Process>\*, <Predicate>?, <Comparator>?

Parent elements: <Editor>

```
<Editor>
  <Processes>
    <Process name="Publication" />
    .
    </Process>
    .
  </Processes>
</Editor>
```

Example 45: Example of the Processes element

This element is used to configure the property editors, which define the view of the workflow variables and the determine a sort order and filtering criteria for the workflow list. The workflow variables are shown in the upper right part of the workflow window.

**<Comparator>**

Child elements: %varies;

Parent elements: <ColumnDefinition>, <Processes> ...

```
<Processes>
  <Comparator class="MyOwnWorkFlowComparator" />
  .
  .
</Processes>
```

Example 46: Example of the Comparator element

In the context of the workflow this element of the XML file is used for sorting:

- » The entries in the menu **File|New workflow** when used in <Processes> (see the example).
- » The workflow list when used in <ColumnDefinition> of <TableDefinition>.

Attribute	Description
class	Name of the class in which the sorting comparator is defined. The class must contain a public constructor without arguments and must implement an interface depending on the objects to sort (see the API documentation).

Table 42:

**<Predicate>**

Child elements: <DocumentType>\*, %varies;

Parent elements: <Processes>...

```
<Processes>
    <Predicate class="MyWorkflowFilter" />
    .
</Processes>
```

Example 47: Example of the Predicate element used in a Processes element

The predicate for filtering is entered with the <Predicate> element. The provided filter classes are described in the *Editor Developer Manual*. In the workflow context, the element is used for filtering the workflows offered via **File|New workflow...** The default filter restricts the list of workflows to those workflows, from which the user is allowed to create a process instance.

Attribute	Description
class	Name of the class with the predicate for filtering. In the workflow context, there are no predefined filter classes. The default class is GenericProcessPredicate. Own classes can be written and must implement the interface <code>hox.util.Predicate</code> .

Table 43: Attribute of the Predicate element.

**<Process>**

Child elements: <View>?, <Task>\*, <WorkflowStartup>?

Parent elements: <Processes>

```
<Processes>
    <Process name="Publication" />
        <View>
            .
        </View>
    </Process>
    .
</Processes>
```

Example 48: Example of the Process element

This element is used for the configuration of the variable editors for the process (in the `<View>` element) and for each task (in the `<Task>` element). You may specify one `<Process>` element per workflow.

Attribute	Description
name	Name of the process, for which the view should be configured.

Table 44: Attribute of the Process element.

### **<View>**

Child elements: `<Variable>*`, `<AggregationVariable>*`

Parent elements: `<Process>`

```
<Process name="FourEyesProcess">
  .
  .
  <View>
    <Variable name="User"
      editorClass="UserChooserEditor"/>
    .
    .
  </View>
</Process>
```

Example 49: Example of the View element

This element is used to group the elements, which define how the variables of a process are rendered and edited.

### **<Task>**

Child elements: `<Variable>*`, `<AggregationVariable>*`

Parent elements: `<Process>`

```
<Process name="FourEyesProcess">
  .
  .
  <Task name="approve">
    <Variable name="comment" editorClass="StringEditor"/>
    .
    .
  </Task>
</Process>
```

Example 50: Example for the Task element

This element is used to group the elements, which define how the variables of a task are rendered and edited.

Attribute	Description
name	Name of the task for which the definitions should be valid.

Table 45: Attribute of the Task element.

**<WorkflowStartup>**

Child elements:

Parent elements: <Process>

```
<Process name="SimplePublication">
  <WorkflowStartup
    class=
    "hox.corem.editor.workflow.foureyes.FourEyesWorkflowStartup" />
  .
  .
</Process>
```

Example 51: Example of the WorkflowStartup element

This element is used to define a concrete implementation for the configuration hooks, that are executed when a process is started. It is possible to prefill the variable before the user may edit them, skip the initial dialog to set the variables and perform arbitrary actions just before the process is started. If you do not set an own startup class, the default implementation is used, which tries to assign the currently selected resources to appropriate process variables and opens a window in which all variables defined as <Write> in the <InitialClient> tag can be entered.

Attribute	Description
class	This attribute defines the class which manages the startup of the workflow. Own classes must implement the interface <code>hox.corem.editor.workflow.WorkflowStartup</code> . As a default, the class <code>hox.corem.editor.generic.GenericWorkflowStartup</code> is used. It sets the resource variable with the selected resource(s) and opens a window for setting the other workflow variables.

Table 46: Attribute of the WorkflowStartup element.

**<Variable>**

Child elements:

Parent elements: <View>, <Task>

```
<Task name="approve">
  <Variable name="comment" editorClass="StringEditor" />
  .
  .
  .
</Task>
```

Example 52: Example of the Variable element

This element is used to define, which property editor should render and edit a workflow variable. If no element is defined for a variable, the default editor for the variable type is used (see Section 5.4.3 "Predefined Editor Classes"<sup>(81)</sup> for a listing of the editor classes). An editor for variables must implement the subinterface of the PropertyEditor interface, depending on his concrete type.

Attributes	Description
editorClass	Editor class to be used for showing the variable. See Section 5.4.3 "Predefined Editor Classes" <sup>(81)</sup> for a listing of the editor classes.
name	Name of the variable to be shown, as defined in the workflow definition.

Table 47: Attributes of the Variable element.

### <AggregationVariable>

Child elements:

Parent elements: <View>, <Task>

```
<Task name="approve">
  <AggregationVariable name="Resources"
    editorClass="ResourceChooserEditor" />
  .
  .
  .
</Task>
```

Example 53: Example of the AggregationVariable element

This element defines, which property editor should render and edit a workflow aggregation variable. If no element is defined for an aggregation variable and only resources are stored, the default aggregation editor is used.

Note, that there is only a default aggregation editor for aggregations of type resource. A property editor for aggregation variables must implement the `AggregationEditor` interface.

Attributes	Description
editorClass	Editor class to be used for showing the variable. See Section 5.4.3 "Predefined Editor Classes" <sup>(R1)</sup> for a listing of the editor classes.
name	Name of the aggregation variable to be shown, as defined in the workflow definition.

Table 48: Attributes of the `AggregationVariable` element.

## Example Configuration of the Workflow Window

In this chapter, you will find an example for the configuration of the workflow window as it is used for the three-step workflows.

```

1: <Process name="ThreeStepPublication">
2:   <WorkflowStartup class="hox.corem.editor.workflow.foureyes.
3:     FourEyesWorkflowStartup" />
4:   <View>
5:     <AggregationVariable editorClass="hox.corem.editor.
6:       workflow.foureyes.FourEyesAggregationEditor"
7:       name="changeSet" />
8:   </View>
9:   <Task name="Compose">
10:    <AggregationVariable editorClass="hox.corem.editor.
11:      workflow.foureyes.FourEyesAggregationEditor"
12:      name="changeSet" />
13:    <AggregationVariable
14:      editorClass="hox.corem.editor.workflow.variables.
15:        PublicationResultAggregationEditor"
16:      name="publicationResultCodes" />
17:   </Task>
18:   <Task name="Approve">
19:    <AggregationVariable
20:      editorClass="hox.corem.editor.workflow.foureyes.
21:        FourEyesAggregationEditor"
22:      name="changeSet" />
23:    <AggregationVariable
24:      editorClass="hox.corem.editor.workflow.variables.
25:        PublicationResultAggregationEditor"
26:      name="publicationResultCodes" />
27:   </Task>
28:   <Task name="Publish">
29:    <AggregationVariable editorClass="hox.corem.editor.
30:      workflow.foureyes.FourEyesAggregationEditor"

```

Example 54: Example definition of workflow GUI in `editor.xml`

```

31:   name="changeSet" />
32: </Task>
33: </Process>
34: <Workflow>
35:   <TableDefinition rowHeight="25">
36:     <ColumnDefinition
37:       class="hox.corem.editor.workflow.columns.TaskTypeColumn" />
38:     <ColumnDefinition
39:       class="hox.corem.editor.workflow.columns.TaskStateColumn" />
40:     <ColumnDefinition
41:       class="hox.corem.editor.workflow.columns.TaskDataColumn" />
42:     <ComponentFactory class="hox.corem.editor.workflow.
43:       foureyes.FourEyesDataPanelFactory"
44:       processName="ThreeStepPublication" name=" " />
45:   </ColumnDefinition>
46: </TableDefinition>
47: </Workflow>
48: </Editor>

```

*Line 1:* The properties concerning the workflow with the name "ThreeStepPublication" (as it is defined as the workflow's name in the workflow definition file) are configured.

*Line 2 - 3:* The class that is used when a new process is started. Here a specialized class is used, that suppresses the dialog box for initializing the workflow variables normally popping up. In addition, the selected resources are added to the change set.

*Line 4:* Herein, you can define with which editor the variables defined in the <InitialClient> and <Client> sections of the workflow definition will be shown. If no property editor is defined, the default editor for the variable type will be used.

*Line 5 - 7:* In this part, an editor class for viewing the aggregation variable "changeSet" which contains the selected resources is defined. A specialized class for the publication workflows is used, which shows place and content changes concerning the selected resources.

*Line 9 + 18:* Here, the variable view for the `Compose` and `Approve` task is defined.

*Line 10 - 16 and Line 19 - 26:* For both tasks, two editor classes are defined, which show the content of the "changeSet" and "publicationResultCodes" aggregation variables. For the "changeSet" variable the same class as for the workflow variable view is used.

*Line 28:* Here, the variable view for the `Publish` task is defined.

*Line 29 - 31:* As described before, an editor class for the aggregation variable "changeSet" is defined.

*Line 34 - 35:* In this section, the representation of the task and workflow information shown in the workflow list is defined.

*Line 36 - 41:* Three columns should be shown in the workflow list. For each column, a class must be defined which shows the appropriate information. In the first column, the type of the task is shown. In the second column, the state of the task is shown. In the third column additional information concerning the workflow is shown. The class used for it `TaskDataColumn` is more generic than the other two classes. Thus, it has to be configured by another class defined in the `ComponentFactory` element to create the renderers for the column.

*Line 42 - 44:* In this part, the specialized class for showing details about the publication workflows is defined. The attribute `processName` defines the workflow for which the information should be shown. See Section 5.4 "[Reference of Predefined Classes](#)"<sup>[69]</sup> for details on the class.

## 6.10.2 Programming Workflow Startups

The startup of workflows can be customized by using customized startup classes. If you implement a workflow startup class you will use the *Workflow* and *Editor API*. In general, the workflow startup class covers the assignment of variables. Mostly this means that selected resources from the explorer have to be assigned to workflow variables. Then, you have to decide, whether a variable dialog should allow a user to modify the variables before the workflow has been started. Finally, a hook allows custom code to be executed before the process is started. The publication workflow startup classes use this to bring the workflow window to the front and select the process, that has just been started.

### Interface to implement

Startup classes need to implement the `WorkflowStartup` interface (which you will find in the *Editor API*).

### Default implementation

The most simple way to program own startup classes would be the extension of the predefined startup classes `GenericWorkflowStartup` or `AutomaticSelectorWorkflowStartup`. These two classes implement almost the same behavior, but `AutomaticSelectorWorkflowStartup` will automatically select the first task instance in the workflow view, if this task instance is automatically accepted (see the *Editor API* for details).

### Defining the workflow startup class

The workflow startup class has to be integrated with the *CoreMedia Editor* (or your own client implementations, see Section 6.8 "[Programming Clients](#)"<sup>(123)</sup>) in the `editor.xml` definition (see [Example 55](#)<sup>(137)</sup>).

```
<Processes>
  <Process name="parameter">
    <WorkflowStartup
      class="com.coremedia.examples.workflow.startup.DemoStartup" />
    </Process>
  </Processes>
```

*Example 55: Integrate own startup class in editor.xml*

### Example workflow startup class

The following example shows a customized startup class. The example will check all atomic workflow variables if they are capable to store documents. If it finds such a variable it will store the first selected document in this variable. Please be aware that the exception handling in this example is rather poor. The example startup class needs a selected document and needs an atomic variable which is able to store documents. See [Example 56](#)<sup>(138)</sup> for the required packages.

```

package com.coremedia.examples.workflow.startup;

import hox.corem.editor.*;
import hox.corem.editor.generic.*;
import hox.corem.editor.toolkit.*;
import com.coremedia.workflow.*;
import com.coremedia.workflow.common.values.*;

public class DemoStartup extends GenericWorkflowStartup {
    ...
}

```

*Example 56: Basic packages for own startup classes*

The following code example shows how you might implement the initial assignment of the process instance variables.

```

1: public WfVariableAssignment[] getInitialAssignment
2:     (WfProcessInstance processInst
3:     ResourceHolderEnumeration enumeration,
4:     Context context) throws WfException {

5:     WfVariableAssignment[] assignments =
6:         processInstance.getVariableAssignment();
7:     ResourceValue[] resources =
8:         ResourceHolderValueConverter.getValues(enumeration);
9:     ResourceValue[] documents =
10:        filterValues(resources, DocumentValue.class);

11:    for(int i=0; i < assignments.length; i++) {
12:        WfVariableAssignment assignment = assignments[i];
13:        WfVariable variable = processInstance.getVariable
14:            (assignment.getVariableName());
15:        if ("MyVariable".equals(variable.getName())) {
16:            assignment.setValue(documents[0]);
17:        }
18:    }
19:    return assignments;
20: }

```

*Example 57: Assign the document to a workflow variable*

*Line 1 - 4:* The `getInitialAssignment()` method is called by the workflow server. The parameter `processInstance` passes an instance of the workflow to the method, which allows you to access workflow variables, the parameter `enumeration` contains the selected resources, which are passed to the method to put them into the workflow variables.

*Line 5:* The variables of the workflow process definition are stored in the `WfVariableAssignment` array.

*Line 6 - 7:* In order to work with the selected documents, you need to create a ResourceValue from the ResourceHolderEnumeration. For this you can use the static method `getValues()` of the ResourceHolderValueConverter class.

*Line 8:* Now you can use the helper method `filterValues()` which extracts the documents from the selected resources.

*Line 9 - 16:* You iterate over all workflow variables stored in WfVariableAssignment.

*Line 10:* Extracts the first WfVariableAssignment from the workflow variables.

*Line 11 - 12:* Now you can get the first WfVariable from the workflow instance.

*Line 13:* As your workflow definition defines a variable named `MyVariable` of the type `Document`.

*Line 14:* Now you can assign the ResourceValue to the WfVariableAssignment.

*Line 15:* Return the WfVariableAssignment to the editor, which in turn hands it over to the workflow server.

In [Example 58](#)<sup>(139)</sup> you see two more possibilities to customize the workflow startup class. The method `isSkippingVariableDialog` shows how to skip the initial dialog for the assignment of variables. If you want to allow the user to change the variables, return `false`. The method `performInitialActions` allows you to perform actions at start time of the process. The default implementation `GenericWorkflowStartup` is empty.

```
public boolean isSkippingVariableDialog
(WfProcessInstance processInstance)
    throws WfException {
    return true;
} // isSkippingVariableDialog()

public void performInitialActions(WfProcessInstance
processInstance)
    throws WfException {
    Log log = Editor.getLog();
    log.write(Log.LEVEL_DEBUG,
        "DemoStartup.performInitialActions(): done...");
} // performInitialActions()
```

*Example 58: Two more methods for startup classes*

## 7 Appendix

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

## 7.1 XML Element Reference

The order of the elements in the workflow definition is not relevant except for the [Action](#)<sup>[143]</sup> and the [Condition](#)<sup>[150]</sup> 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](#)<sup>[153]</sup> | [NotEqual](#)<sup>[173]</sup> | [Greater](#)<sup>[162]</sup> | [GreaterEqual](#)<sup>[162]</sup> | [Less](#)<sup>[171]</sup> | [LessEqual](#)<sup>[171]</sup> | [And](#)<sup>[145]</sup> | [Or](#)<sup>[173]</sup> | [Implies](#)<sup>[165]</sup> | [Not](#)<sup>[172]</sup> | [ForAll](#)<sup>[156]</sup> | [Exists](#)<sup>[154]</sup> | [Let](#)<sup>[172]</sup> | [Get](#)<sup>[160]</sup> | [Read](#)<sup>[178]</sup> | [Length](#)<sup>[170]</sup> | [IsEmpty](#)<sup>[168]</sup> | [NotEmpty](#)<sup>[173]</sup> | [IsFolder](#)<sup>[168]</sup> | [IsDocument](#)<sup>[166]</sup> | [IsDocumentVersion](#)<sup>[167]</sup> | [IsExpired](#)<sup>[168]</sup> | [IsEnabled](#)

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

#### *Expression*

*Definition:* [Expression](#)<sup>[155]</sup> | [Equal](#)<sup>[153]</sup> | [NotEqual](#)<sup>[173]</sup> | [Greater](#)<sup>[162]</sup> | [GreaterEqual](#)<sup>[162]</sup> | [Less](#)<sup>[171]</sup> | [LessEqual](#)<sup>[171]</sup> | [And](#)<sup>[145]</sup> | [Or](#)<sup>[173]</sup> | [Implies](#)<sup>[165]</sup> | [Not](#)<sup>[172]</sup> | [ForAll](#)<sup>[156]</sup> | [Exists](#)<sup>[154]</sup> | [Let](#)<sup>[172]</sup> | [Get](#)<sup>[160]</sup> | [Read](#)<sup>[178]</sup> | [Length](#)<sup>[170]</sup> | [IsEmpty](#)<sup>[168]</sup> | [NotEmpty](#)<sup>[173]</sup> | [IsFolder](#)<sup>[168]</sup> | [IsDocument](#)<sup>[166]</sup> | [IsDocumentVersion](#)<sup>[167]</sup> | [IsExpired](#)<sup>[168]</sup> | [AddLatestVersion](#)<sup>[144]</sup> | [Value](#)<sup>[142]</sup>: [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<sup>\[155\]</sup>](#) element.

### FlowControlTask

**Definition:** [Choice<sup>\[149\]</sup>](#) | [Fork<sup>\[157\]</sup>](#) | [If<sup>\[164\]</sup>](#) | [Join<sup>\[169\]</sup>](#) | [JoinSubprocess<sup>\[169\]</sup>](#) | [ForkSubprocess<sup>\[158\]</sup>](#) | [Switch<sup>\[183\]</sup>](#)

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<sup>\[146\]</sup>](#) | [UserTask<sup>\[185\]</sup>](#) | [FlowControlTask<sup>\[142\]</sup>](#); [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<sup>\[182\]</sup>](#)(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 *CoreMedia Editor*.

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<sup>\[146\]</sup>](#) is executed automatically.
- » A [UserTask<sup>\[185\]</sup>](#) 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<sup>\[148\]</sup>](#) | [Boolean<sup>\[148\]</sup>](#) | [Content<sup>\[179\]</sup>](#) | [ContentType<sup>\[151\]</sup>](#) | [Date<sup>\[151\]</sup>](#) | [Document<sup>\[151\]</sup>](#) | [Folder<sup>\[156\]</sup>](#) | [Group<sup>\[163\]</sup>](#) | [Integer<sup>\[166\]</sup>](#) | [String<sup>\[182\]</sup>](#) | [Timer<sup>\[184\]</sup>](#) | [User<sup>\[185\]</sup>](#)

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 59](#)<sup>[143]</sup>).

```
<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 59: 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](#)<sup>[150]</sup>?, [Property](#)<sup>[177]</sup> }

An action is external code which may be called to customize the processing of the workflow engine (see Section 6.4 "[Programming Actions](#)"<sup>[90]</sup> for implementing own actions).

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

An predefined Action or one subclassed from `AbstractAction/AbstractClientAction` may contain a [Condition](#)<sup>[150]</sup> 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: ApproveResource, CheckInDocument, CheckOutDocument, CopyResource, CreateDocument, CreateFolder, DeleteResource, DisapproveResource, EnableTimer, ExcludePerformer, DisableTimer, MoveResource, OpenDocument, PreferPerformer, PublishResources, RenameResource, SaveDocument, UncheckOutDocument, UndeleteResource.

The predefined actions use some of the Action-attributes defined in Section 5.4.1 "Predefined Action Classes"<sup>[70]</sup>.

Note: The [Property](#)<sup>[177]</sup> 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 49: 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 60: Action with a Guard used in a UserTask

### AddLatestVersion

Grammar: { [[Expression](#)<sup>[141]</sup>]\* }

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](#)<sup>[142]</sup>]\* }

In contrast to a variable, whose value is one value, an [AggregationVariable](#)<sup>[144]</sup> 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
type	NMTOKEN	#REQUIRED	the type of the variable, see Value
readOnly	[boolean <sup>[143]</sup> ]	"false"	whether it is forbidden to modify the variable
static	[boolean]	"false"	whether the variable is initialized only once

Table 50: 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 61: Example of an aggregation variable

### And

Grammar: { [Expression<sup>[141]</sup>]\* }

An And expression evaluates to the conjunction of its sub-expressions, all of which must return boolean values. The sub-expressions are evaluated in a "short-circuit" fashion, i.e., they are evaluated top down until the first sub-expression evaluates to "false" or all sub-expressions have evaluated to "true". This helps to avoid exceptions during the computation, e.g. when checking the type of a document before accessing a property of the document of that expected type.

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

<UserTask name="AndTest" successor="theNext">
  <PreCondition>
```

Example 62: Example of an And element.

```

<And>
  <Equal>
    <Get variable="OWNER_" />
    <User value="0" />
  </Equal>
  <Equal>
    <Get variable="Comment" />
    <String value="42" />
  </Equal>
</And>
</Precondition>
<!-- Code -->
</UserTask>

```

## Assign

Grammar: { [Expression](#)<sup>[141]</sup> }

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 80](#)<sup>[159]</sup>.

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

Table 51: Attribute of the Assign element

## AutomatedTask

>> Grammar: { ( [Variable](#)<sup>[188]</sup> | [AggregationVariable](#)<sup>[144]</sup> ) \* , [Action](#)<sup>[143]</sup> \* , [Guard](#)<sup>[163]</sup> ? , [PreCondition](#)<sup>[175]</sup> \* , [PostCondition](#)<sup>[175]</sup> }

An [AutomatedTask](#)<sup>[146]</sup> 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 [Actions](#)<sup>[143]</sup> of an automated task are used to customize the processing of the workflow engine. If <sup>[164]</sup> more than one [Action](#)<sup>[143]</sup> is provided, the actions are executed in the order in which they are specified.

A [PreCondition](#)<sup>[175]</sup> defines requirements which have to be fulfilled before the actions of the automated task are executed. A [PostCondition](#)<sup>[175]</sup> defines requirements which have to be fulfilled after the action has been executed. If <sup>[164]</sup> more than one [PreCondition](#)<sup>[175]</sup> or [PostCondition](#)<sup>[175]</sup> 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**<sup>[163]</sup> 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**<sup>[189]</sup>, Server and content or name changes of referred resources in the *Content Management Server*. Note that changes to other, external entities do not trigger re-evaluation of a guard.

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

Note: An **AutomatedTask**<sup>[146]</sup> does not allow to specify **Rights**<sup>[181]</sup>, **Performers**<sup>[174]</sup>, and **Client**<sup>[147]</sup>. This is restricted to **UserTask**<sup>[185]</sup> 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 <sup>[143]</sup> }	"false"	whether the task is the final task to execute
successor	NMTOKEN	#IMPLIED	the next task to execute after the automated task has been completed
varies			

Table 52: Attributes of the Automated Task element

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

Example 63: Example of an AutomatedTask

### Assignment

» Grammar: { ( **Reads**<sup>[179]</sup> | **Writes**<sup>[189]</sup> )\*, **Validator**<sup>[187]</sup> }

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**<sup>[179]</sup> and **Writes**<sup>[189]</sup> 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">
      <Client>
        <Reads variable="Resource" contentEditable="true"/>
        <Writes variable="Comment"/>
      </Client>
      <!-- Code -->
    </UserTask>
    <!-- Code -->
  </Process>
</Workflow>
```

Example 64: Example of a Client 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 53: Attribute of the Blob element

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

Example 65: 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 <sup>[143]</sup> }	#REQUIRED	the boolean value ("true" or "false")

Table 54: Attribute of the Boolean element

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

Example 66: 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<sup>\[183\]</sup>](#).

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 55: Attributes of the Case element

### Choice

Grammar: { ( Variable<sup>[188]</sup> | AggregationVariable<sup>[144]</sup>)\*, Successor<sup>[182]</sup>+ }

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<sup>\[182\]</sup>](#) 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

Table 56: Attributes of the Choice element.

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

```
<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 67: Example of a Choice element

**Client**

Deprecated. See **Assignment** instead.

**Condition**

Grammar: {Expression<sup>[141]</sup>}

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 sub-expressions.

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

Table 57: Attributes of the Condition element

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

<UserTask name="AndTest" successor="theNext">
  <EntryAction class="CheckOutDocument" documentVariable="Article">
    <Condition>
```

Example 68: Example of a Condition element. It is checked whether the document variable is null or not.

```

        <NotEmpty variable="Article"/>
    </Condition>
</EntryAction>
<!-- Code -->
</UserTask>

```

### 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 58: Attribute of the ContentType element

```

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

```

Example 69: 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 59: Attribute of the Date element

```

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

```

Example 70: 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 60: Attributes of the Document element.

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

Example 71: Example of a Document variable.

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

Table 61: Attribute of the DocumentType element

### Else

Grammar: EMPTY

Else defines the successor of the `if(164)` task if the condition evaluates to false, see `if(164)` for details and an XML example.

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

Table 62: Attribute of the Else element

### EntryAction

Grammar: [ `Condition(150)? , Property(177)? ]`

EntryAction and ExitAction<sup>[154]</sup> elements are identical to Action<sup>[143]</sup> elements, see Action<sup>[143]</sup> and Action-attributes<sup>[141]</sup> for details.

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

Table 63: Attributes of EntryAction element

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

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

Example 72: Example of an EntryAction which checks out a document

## Equal

Grammar: { (Expression<sup>[141]</sup>), (Expression<sup>[141]</sup>) }

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 sub-expressions 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, i.e., the document contents are not considered.

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

<UserTask name="AndTest" successor="TheNext">
  <Guard>
    <Equal>
      <Get variable="Comment" />
    </Equal>
  </Guard>
</UserTask>
```

Example 73: Example of an Equal expression

```

        <String value="LetMeIn" />
    </Equal>
</Guard>
<!-- Code -->
</UserTask>

```

**Exists**

Grammar: { **Expression**<sup>[141]</sup> }

Exists is the counterpart to **ForAll**<sup>[156]</sup> and behaves similarly. It evaluates to true if any of the instances of the subexpression evaluate to "true". Evaluation is also short-circuit, i.e. 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 64: Attributes of the Exists element

```

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

<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 74: 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**<sup>[150]</sup>?, **Property**<sup>[177]</sup>? }

ExitAction and EntryAction<sup>[152]</sup> elements are identical to Action<sup>[143]</sup> elements, see Action<sup>[143]</sup> for details.

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

Table 65: Attributes of the ExitAction element

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

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

Example 75: Example of an Exit Action which checks whether the document is null or not

### Expression

Grammar: { (Expression<sup>[141]</sup>)\* }

You can implement your own expressions (see Section 6.5 "Programming Expressions"<sup>[107]</sup>). 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 66: Attributes of the Expression element

```
<Variable name="comment" type="String">
  <String value="TestString" />
</Variable>
<If name="One">
  <Condition>
    <Less>
      <Expression
```

Example 76: Example of an Expression element

```

        class="com.coremedia.examples.expression.DemoExpression" />
        <Get variable="comment" />
    </Less>
</Condition>
<Then successor="True" />
<Else successor="False" />
</If>
    
```

### 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 67: Attributes of the Folder element.

```

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

Example 77: Example of a Folder variable

### ForAll

Grammar: {Expression<sup>[141]</sup>}

A ForAll expression checks its boolean subexpression for all members of the value of the "aggregate" [AggregationVariable<sup>\[144\]</sup>](#) and evaluates to "true" if all instances of the subexpression evaluate to "true". The subexpression can (and should) contain a [Get<sup>\[160\]</sup>](#) expression with the variable name that evaluates to the n-th value in the aggregate.

The logical "and" is short-circuit 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 e.g. 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
index	NMTOKEN	#IMPLIED	the name of a new integer variable that is set to the current index in the aggregate during the iteration

Table 68: Attributes of the ForAll element

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

<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 78: Example of a ForAll element which checks if all documents are checked in before approving them

## Fork

Grammar: { [Variable<sup>[188]</sup> | AggregationVariable<sup>[144]</sup>]\*, Successor<sup>[182]</sup>+ }

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<sup>[169]</sup> 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 79: 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 69: Attributes of the Fork element

**ForkSubprocess**

>> Grammar: { {Variable<sup>[188]</sup> | AggregationVariable<sup>[144]</sup> }\*, Parameters<sup>[174]</sup> }

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 Parameter<sup>[174]</sup> 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

Table 70: Attributes of the ForkSubprocess element

Attribute	Type	Default	Description
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. 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 <sup>[143]}</sup> }	"false"	If set to "false", the subprocess may be joined and it is affected by suspend, abort and resume operations on the original process.

```

<Workflow>
  <Process name="FirstWF" startTask="Fork">
    <Variable name="Comment" type="String"/>
    <Client>
      <Writes variable="Comment"/>
    </Client>
    <!-- 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 -->

```

Example 80: Example of a ForkSubprocess task

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

**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](#)<sup>[144]</sup>, 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 71: Attributes of the Get element

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

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

Example 81: 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', 'userId', 'group' or 'groupId' 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.

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 or userId	NMTOKEN	#IMPLIED	the name of a user or the user ID of a user
group or groupId	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.
rights	CDATA	#REQUIRED	a comma-separated list of rights as speci- fied above

Table 72: 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 82: Example of a Grant element

## Greater

*Grammar:* { ([Expression<sup>\[141\]</sup>](#)), ([Expression<sup>\[141\]</sup>](#)) }

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 sub-expression is greater than the value of the second sub-expression.

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" />
<Client>
  <Writes variable="Published" />
</Client>

<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 83: Example of a Greater expression*

## GreaterEqual

*Grammar:* { ([Expression<sup>\[141\]</sup>](#)), ([Expression<sup>\[141\]</sup>](#)) }

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 sub-expression is greater than or equal to the value of the second sub-expression.

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" />
<Client>
  <Writes variable="Published" />
</Client>

<If name="IfTask">
  <Condition>
    <GreaterEqual>
      <Get variable="Published" />
```

*Example 84: Example of a GreaterEqual expression*

```

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

## 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 73: Attributes of the Group element.name*

```

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

*Example 85: Example of a Group variable*

## Guard

*Grammar:* {[Expression](#)<sup>[141]</sup>}

A Guard contains a boolean expression, that defines a condition which must become true before a task is activated. See [UserTask](#)<sup>[185]</sup>, [AutomatedTask](#)<sup>[146]</sup> and [Condition](#)<sup>[150]</sup> for details.

```

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

*Example 86: Example of a Guard*

```

</Client>

<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>

```

**If**

Grammar: { (Variable<sup>[188]</sup> | AggregationVariable<sup>[144]</sup>)\*, Condition<sup>[150]</sup>, Then<sup>[183]</sup>, Else<sup>[152]</sup> }

An If task determines the successor task based on the result of a Condition<sup>[150]</sup>. 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<sup>[183]</sup> element is chosen, else the one of the Else<sup>[152]</sup> element. See Example 87<sup>[164]</sup>.

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

Table 74: Attributes of the If element

```

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

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

```

Example 87: Example of an If task

## Implies

Grammar: { [Expression<sup>[141]</sup>], {Expression<sup>[141]</sup>}\* }

An Implies expression determines whether the first sub-expression 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 sub-expressions, an Implies expression evaluates to "true" if and only if the first expression evaluates to "false" (without caring for the result of the second sub-expressions) or both expressions evaluate to "true"..

```
<Client>
  <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>
</Client>
```

Example 88: Example for an Implies expression

## InitialAssignment

» Grammar: { [Reads<sup>[179]</sup> | Writes<sup>[189]</sup>]\*, {187} Validator<sup>[187]</sup> }

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<sup>[179]</sup> and Writes<sup>[189]</sup> 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"/>
    <InitialClient>
      <Reads variable="Comment"/>
      <Writes variable="Articles"/>
    </InitialClient>
    <!-- Code -->
  </Process>
</Workflow>
```

Example 89: Example of an InitialClient 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 75: Attribute of the Integer element

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

Example 90: Example of an Integer Variable

**IsDocument**

Grammar: EMPTY

IsDocument queries whether a resource value contained in the variable, which is given as in [Get<sup>\[160\]</sup>](#), 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 76: Attributes of the IsDocument element

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

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

Example 91: Example of an IsDocument expression

### IsDocumentVersion

Grammar: EMPTY

IsDocumentVersion queries whether a resource value contained in the variable, which is given as in [Get<sup>\[160\]</sup>](#), 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, given by a variable name or a constant value

Table 77: Attributes of the IsDocumentVersion element

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

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

Example 92: 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](#)<sup>[170]</sup> for details. For an XML example see [PostCondition](#)<sup>[175]</sup>.

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 78: 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 79: 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 93: 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 document or document 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 80: 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 94: Example of an IsFolder expression

### Join

Grammar: { (Variable<sup>[188]</sup> | AggregationVariable<sup>[144]</sup>)\*, Predecessor<sup>[176]</sup>+ }

A Join task waits for two or more tasks to complete. Joined tasks must have been forked by a Fork<sup>[157]</sup> 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<sup>[157]</sup>.

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 81: Attributes of the Join element

### JoinSubprocess

Grammar: { Variable<sup>[188]</sup> | AggregationVariable<sup>[144]</sup> }

A [JoinSubprocess](#)<sup>[169]</sup> task waits for a non detached subprocess to complete. For an XML example see [ForkSubprocess](#)<sup>[158]</sup>.

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
successor	NMTOKEN	#REQUIRED	The next task to execute after the subprocess has been joined
processResult-Variablec	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 82: 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](#)<sup>[160]</sup> and [Read](#)<sup>[178]</sup>.

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 83: Attributes of the Length element

```

<Variable name="Input" type="String">
<Client>
  <Writes variable="Input" />
</Client>
<UserTask name="LengthCheck" successor="TheNext">
  <Guard>
    <Greater>
      <Length variable="Input" />
      <Integer value="4" />
    </Greater>
  </Guard>
  <!-- Code -->
</UserTask>

```

Example 95: Example of a Length element

## Less

Grammar: { [Expression<sup>\[141\]</sup>](#), [Expression<sup>\[141\]</sup>](#) }

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 sub-expression is less than the value of the second sub-expression.

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<sup>\[141\]</sup>](#), [Expression<sup>\[141\]</sup>](#) }

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 sub-expression is less than or equal to the value of the second sub-expression.

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<sup>\[171\]</sup>](#) 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>
</If>

```

Example 96: Example of a Less expression

```

        </Less>
    </Condition>
    <Then successor="NewCentury" />
    <Else successor="OldCentury" />
</If>

```

### Let

Grammar: { (*Expression*<sup>[141]</sup>), (*Expression*<sup>[141]</sup>) }

Let binds an expression-local variable to a value determined by the first sub-expression. It evaluates to the value of the second sub-expression, which can use the expression-local variable. Let is useful to reuse complex sub-expressions and store their result in an expression-local variable. Some functions as [Length](#)<sup>[170]</sup> and [Read](#)<sup>[178]</sup> 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 sub-expression

Table 84: Attributes of the Let element

```

<Variable name="Article" type="Document" />
<Client>
  <Writes variable="Article" />
</Client>
<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 97: 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*<sup>[141]</sup>)

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 98: Example of a Not element

### NotEmpty

Grammar: EMPTY

NotEmpty is the negation of IsEmpty. See [IsEmpty](#)<sup>[168]</sup> 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 85: Attributes of the NotEmpty element

### NotEqual

Grammar: { [Expression<sup>[141]</sup>], [Expression<sup>[141]</sup>] }

A NotEqual expression is the negation of an Equal expression.

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

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

Example 99: Example of a NotEqual expression

### Or

Grammar: { [Expression<sup>[141]</sup>]\* }

An Or expression evaluates to the disjunction of its sub-expressions, all of which must return boolean values. The sub-expressions are evaluated in a "short-circuit" fashion, i.e., they are evaluated from left to right until the first sub-expression evaluates to "true" or all sub-expressions 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 100: Example of an Or expression

**Parameters**

Grammar:  $\{Assign^{(146)}\}_+$

Parameters is used to enclose the elements that define how to parametrize a subprocess. For an XML example see [ForkSubprocess<sup>\(158\)</sup>](#).

**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 classname which will be searched for in the package `com.coremedia.workflow.common.policies` or have it default to a builtin 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

Table 86: Attributes of the Performers element

Attribute	Type	Default	Description
<a href="#">varies</a> <sup>[143]</sup>			additional parameters according to the implementation of the policy class

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

Example 101: Performers element

### PostCondition

Grammar: [{Expression}](#)<sup>[141]</sup>

A PostCondition assert a condition that must hold after an (optional) exit action (user task) or action (automated task) has run. See [Condition](#)<sup>[150]</sup> 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 87: Attributes of the PostCondition element

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

Example 102: Example of a PostCondition element

### PreCondition

Grammar: [{Expression}](#)<sup>[141]</sup>

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](#)<sup>[150]</sup> 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 88: Attributes of the PreCondition element

```

<Variable name="Location" type="Folder"/>
<Variable name="DocName" type="String"/>
<Client>
  <Writes variable="Location"/>
  <Writes name="DocName"/>
</Client>
<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 103: Example of a PreCondition

### Predecessor

Grammar: EMPTY

A Predecessor element defines a predecessor of a [Join](#)<sup>[169]</sup> task by its name. See [Fork](#)<sup>[157]</sup> for an XML example.

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

Table 89: Attribute of the Predecessor element

### Process

Grammar:  $(Rights^{[181]?, (Variable^{[188]} | AggregationVariable^{[144]})^*, InitialClient^{[165]?, Client^{[147]?, (Task^{[142]})^+})}$

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`<sup>[181]</sup> 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`<sup>[165]</sup> and `Client`<sup>[147]</sup>, you define which variables are to be read or written by a user or an external process. The `InitialClient`<sup>[165]</sup> element is used for initializing the process before it is started while the `Client`<sup>[147]</sup> element is used afterwards when the process is running.

Attribute	Type	Default	Description
<code>name</code>	NMTOKEN	#REQUIRED	the name of the process
<code>description</code>	CDATA	#IMPLIED	a textual description of what the process does or a localization key.
<code>startTask</code>	NMTOKEN	#REQUIRED	the name of the initial task
<code>subprocessOnly</code>	<code>{boolean}</code> <sup>[143]</sup>	"false"	Specify this attribute for processes that cannot run stand-alone.
<code>defaultTimeout</code>	NMTOKEN	#IMPLIED	the maximum number of seconds that an instance of this process is supposed to take

Table 90: Attributes of the `Process` element

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

Example 104: 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 document type
value	CDATA	#REQUIRED	a value of the appropriate type

Table 91: Attributes of the Property element

```
<Variable name="Location" type="Folder"/>
<Variable name="DocName" type="String"/>
<Client>
  <Writes variable="Location"/>
  <Writes name="DocName"/>
</Client>
<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 105: 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 a SGML property will be returned as a string. All other property types will be returned as the appropriate workflow variable value. See [Exists](#)<sup>[154]</sup> 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 92: Attributes of the Read element

## Reads

Grammar: EMPTY

Reads and [Writes](#)<sup>[189]</sup> 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](#)<sup>[189]</sup> allows 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	the name of the read variable
description	CDATA	#IMPLIED	the textual description of the meaning of the variable
contentEditable	{boolean <sup>[143]</sup> }	"true"	whether a document referred to by a variable may be edited in the embedded document view

Table 93: Attributes of the Reads element

```
<Variable name="Comment" type="String" />
<Variable name="Article" type="Document" />
<Client>
  <Reads variable="Comment" />
  <Reads variable="Article" contentEditable="true" />
</Client>
```

Example 106: 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.

Table 94: Attributes of the Resource element.

Attribute	Type	Default	Description
path	NMTOKEN	#IMPLIED	The path of the resource.

```
<Variable name="DocFol" type="Resource">
  <Resource value="12"/>
</Variable>
```

Example 107: Example of a Resource variable

### Revoke

Grammar: EMPTY

Revoke revokes the operations for users or groups like [Grant](#)<sup>[161]</sup> grants them (only valid for the default ACL rights policy). See [Grant](#)<sup>[161]</sup> for details. Revoke precedes Grants.

Attribute	Type	Default	Description
user or userId	NMTOKEN	#IMPLIED	the name of a user or the user ID of a user
group or groupId	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.
rights	CDATA	#REQUIRED	a comma-separated list of rights as specified above

Table 95: Attributes of the Revoke element.

```
<UserTask name="GrantExample" successor="TheNext">
  <Rights>
    <Grant group="composer-role"
```

Example 108: Example of a Revoke element

```

        rights="accept, complete, delegate, read"/>
    <Revoke user="demo1" rights="delegate"/>
  </Rights>
  <!-- Code -->
<UserTask>

```

## Rights

*Grammar:* { [Grant](#)<sup>(161)\*</sup>, [Revoke](#)<sup>(180)\*</sup> }

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 classname which will be searched for in the package `com.coremedia.workflow.common.policies` or have it default 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 ([Grant](#)<sup>(161)</sup>).
- » Rights can be revoked for individual users or groups ([Revoke](#)<sup>(180)</sup>).
- » 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 <sup>[143]</sup>			additional parameters according to the implementation of the policy class

Table 96: 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 109: 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 97: Attribute of the String element

```

<Variable name="Text" type="String">
  <String value="Hello World"/>
</Variable>

```

Example 110: Example of a String variable

### Successor

Grammar: EMPTY

A Successor element defines a successor task of a [Fork](#)<sup>[157]</sup> or [Choice](#)<sup>[149]</sup> task by its name. See [Fork](#)<sup>[157]</sup> for an example.

Attribute	Type	Default	Description
name	NMTOKEN	#REQUIRED	the name of the successor task

Table 98: 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 99: 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 111: Example of the Switch element.

### Then

Grammar: EMPTY

Then defines the successor of the `if(164)` task if the condition evaluates to true, see `if(164)` for details and an example.

Attribute	Type	Default	Description
successor	NMTOKEN	#REQUIRED	the name of the successor task in the "then" case

Table 100: 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	<code>{boolean<sup>(143)}</sup>}</code>	"true"	This attribute determines whether the timer should be a relative timer. An absolute timer will not be useful in the workflow definition.

Table 101: Attributes of the `Timer` element

```
<Variable name="Expires" type="Timer">
  <Timer value="100"/>
</Variable>
<Action class="EnableTimer" timerVariable="Expires"/>
```

Example 112: 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, i.e. the process or task definition, where its associated timer variable is defined. See Section 5.4.2 "[Predefined TimerHandler Classes](#)"<sup>(79)</sup> 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.

Table 102: Attributes of the `TimerHandler` element

Attribute	Type	Default	Description
timerName	NMTOKEN	#REQUIRED	Name of the timer for which the timer handler is installed.

```
<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 113: 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.
domain	NMTOKEN	#IMPLIED	The domain of a user. Might be used in addition to name.

Table 103: Attributes of the User element.

```
<Variable name="Admin" type="User">
  <User value="0"/>
</Variable>
```

Example 114: Example of a User variable

### UserTask

**Grammar:** { [Rights](#)<sup>[181]</sup>, [Performers](#)<sup>[174]</sup>? , { [Variable](#)<sup>[188]</sup> | [AggregationVariable](#)<sup>[144]</sup> }\* , [Client](#)<sup>[147]</sup>\* , [EntryAction](#)<sup>[152]</sup>\* , [ExitAction](#)<sup>[154]</sup>\* , [Guard](#)<sup>[163]</sup>? , [PreCondition](#)<sup>[175]</sup>\* , [PostCondition](#)<sup>[175]</sup>\* }

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 auto-completed task is determined by the time that the executed actions and the Pre- 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 auto-completed tasks as semi-automatic tasks.

The [Rights](#)<sup>[181]</sup> element configures user and group permissions for the task instance operations.

[Client](#)<sup>[147]</sup> determines which variables are relevant for this task and may be changed.

A user task may perform some automated action ([EntryAction](#)<sup>[152]</sup>) after the task is accepted and after the task has been completed by the user ([ExitAction](#)<sup>[154]</sup>). If<sup>[164]</sup> more than one [EntryAction](#)<sup>[152]</sup> or [ExitAction](#)<sup>[154]</sup> 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](#)<sup>[163]</sup> defines an expression, which activates the task, if the expression evaluates to true. The expressions of the condition are re-checked on state changes of process- 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

Table 104: Attributes of the `UserTask` element

Attribute	Type	Default	Description
defaultPriority	NMTOKEN	#IMPLIED	the priority of the task
defaultTimeout	NMTOKEN	#IMPLIED	the default timeout in seconds
defaultOfferTime-out	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 <sup>[143]}</sup> }	"false"	whether the task is the final task to execute
autoCompleted	{boolean <sup>[143]}</sup> }	"false"	whether the task is autocompleted
varies <sup>[143]</sup>			additional parameters according to the implementation of the user task class

```
<UserTask name="UserTaskExample" description="Example UserTask"
  successor="Next">
  <Rights>
    <Grant user="demol" rights="accept, complete, read"/>
  </Rights>
  <!-- Code -->
</UserTask>
```

Example 115: Example of a UserTask task

### Validator

Grammar: {Expression}<sup>[141]</sup>

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

Table 105: Attributes of the Validator element

Attribute	Type	Default	Description
validatedOnSave	{boolean <sup>[143]</sup> }	"false"	whether the verification should take place on every save
varies <sup>[143]</sup>			additional parameters according to the implementation of the validator class

```

<Client>
  <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>
</Client>

```

Example 116: Example of a Validator element

### Variable

Grammar: {Value<sup>[142]</sup>}?

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<sup>[164]</sup> a variable is declared as readOnly and the process instance has been started, it is not possible to modify it. If<sup>[164]</sup> a variable is declared as static, it maintains its state, otherwise it is re-initialized 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 <sup>[143]</sup> }	"false"	whether it is forbidden to modify the variable
static	{boolean <sup>[143]</sup> }	"false"	whether the variable is initialized only once

Table 106: Attributes of the Variable element

```
<Variable name="Comment" type="String">
  <String value="42" />
</Variable>
```

Example 117: Example of a Variable element

### Workflow

Grammar: { Process<sup>[176]</sup> }

You can configure exactly one process per workflow definition, which means one workflow per file. If<sup>[164]</sup> 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 118: 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

Table 107: Attributes of the Writes element

Attribute	Type	Default	Description
description	CDATA	#IMPLIED	the textual description of the meaning of the variable
contentEditable	{boolean <sup>[143]</sup> }	"true"	whether a document referred to by a variable may be edited in the embedded document view (not enforced by the workflow server)

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

*Example 119: Example of a Writes element*

## 7.2 Simple Publication Workflow Definition

In this chapter you find the complete workflow definition of the simple publication workflow as described in Section 5.3 "Example of Workflow Definition"<sup>[60]</sup>.

```
<?xml version="1.0" encoding="iso-8859-1"?>

<Workflow>
  <Process name="SimplePublication"
    description="simple-publication"
    startTask="AssignComposer">

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

    <Variable name="subject" type="String"/>
    <Variable name="comment" type="String"/>
    <AggregationVariable name="changeSet" type="Resource"/>
    <AggregationVariable name="comments" type="String"/>

    <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"/>

    <InitialClient>
      <Writes variable="subject"/>
      <Writes variable="comment"/>
      <Writes variable="changeSet"/>
      <Writes variable="comments"/>
    </InitialClient>

    <Client>
      <Reads variable="subject"/>
      <Reads variable="comment"/>
      <Reads variable="changeSet"/>
      <Reads variable="comments"/>
    </Client>

    <AutomatedTask name="AssignComposer"
      description="assigncomposer-task" successor="Compose">
      <Action class="ForceUser" task="Compose">
```

*Example 120: Listing of the simple publication workflow*

```

    userVariable="OWNER_" />
</AutomatedTask>

<UserTask name="Compose"
    description="simple-publication-compose-task"
    successor="CheckEmptyChangeSet">
  <Rights>
    <Grant user="admin" rights="read, accept, delegate, skip"/>
    <Grant group="composer-role"
        rights="read, accept, delegate, skip"/>
  </Rights>

  <Client>
    <Writes variable="subject" />
    <Writes variable="comment" />
    <Writes variable="changeSet" contentEditable="true"/>
    <Writes variable="comments" />
    <Reads variable="publicationResultCodes" />

    <Validator name="AllCheckedIn"
        description="all-checked-in-validator">
      <!-- condition: every document with version in
        changeSet is checked-in -->
      <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>
  </Client>

  <ExitAction class="PreferPerformer" />
  <ExitAction class="ForceUser" task="Publish" />
</UserTask>

<If name="CheckEmptyChangeSet">
  <Condition>
    <IsEmpty variable="changeSet" />
  </Condition>
  <Then successor="Finish" />
  <Else successor="Publish" />
</If>

<UserTask name="Publish"
    description="simple-publication-publish-task"

```

```

        successor="CheckPublication" autoCompleted="true">
<Rights>
  <Grant user="admin" rights="read, accept, retry"/>
<Grant group="composer-role" rights="read, accept, retry"/>
</Rights>

<Client>
  <Reads variable="subject"/>
  <Reads variable="comment"/>
  <Reads description="publish-changeSet "
            variable="changeSet "
            contentEditable="false"/>
  <Reads variable="comments"/>
</Client>

<EntryAction class="ApproveResource" gui="true"
            resourceVariable="changeSet "
            successVariable="publicationSuccessful"
            ignoreErrors="true"
            timeout="180">
</EntryAction>

<EntryAction class="PublishResources" gui="true"
            resourceVariable="changeSet "
            resultVariable="publicationResultResources"
            versionVariable="publicationResultVersions"
            codeVariable="publicationResultCodes"
            parameterVariable="publicationResultParams"
            successVariable="publicationSuccessful"
            ignoreErrors="false"
            ignorePublicationErrors="true" timeout="600"/>
</UserTask>

<If name="CheckPublication">
  <Condition>
    <Get variable="publicationSuccessful"/>
  </Condition>
  <Then successor="Finish"/>
  <Else successor="Compose"/>
</If>

<AutomatedTask name="Finish" final="true"/>

</Process>
</Workflow>

```

## 7.3 Complete Code of the Mail Action

*Example 121: The sendMail action*

```

package com.coremedia.extension.workflow.mail;

import javax.mail.*;
import javax.mail.event.*;
import javax.mail.internet.*;

import com.coremedia.workflow.*;
import com.coremedia.workflow.common.actions.*;
import com.coremedia.workflow.common.values.*;
import com.coremedia.cap.content.ContentException;

/**
 * An action which sends an EMAIL. Parameters are passed
 * in workflow variables or as constants.
 */
public class SendMail extends AbstractResourceAction {

    static final long serialVersionUID = 125806287345433627L;

    /**
     * All variables which are needed for the mail. Only the
     * receiver and the document to check can be passed to
     * the action.
     */
    protected String transportType = "smtp";
    protected String host = "smtp.coremedia.com";
    protected String user = "testuser";
    protected String password = "testpassword";
    protected String from = "testuser@coremedia.com";
    protected String receiverVariable;
    protected String fieldVariable;
    protected String documentVariable;
    protected String subject = "This is a test mail";

    public SendMail(String name) {
        super(name);
    }

    public SendMail() {
        this("SendMail");
    }

    // Read the receiver variable from the workflow instance
    protected String getStringValue(WfInstance instance,
        String variableName) throws WfException {
        if (variableName != null) {
            try {
                WfAtomicVariable variable =
                    instance.getAtomicVariable(variableName);
                WfValue value = variable.getValue();
                if (value instanceof StringValue) {

```

```

        String resourceName = ((StringValue) value).getString();
        if (resourceName == null) {
            throw new WfException
                (WfException.STRING_HAS_NULL_VALUE, new String[]
                 {this.toString(), variableName});
        }
        return resourceName;
    }
    throw new WfException
        (WfException.VALUE_HAS_WRONG_TYPE, new String[]
         {this.toString(), "StringValue", variableName});
    } catch (WfException e) {
        return variableName;
    }
    }
    throw new WfException
        (WfException.NO_PARAMETER_VARIABLE_SPECIFIED,
         this.toString());
    }
}

// Read the content id of the resource to send.
protected String getContentId(WfInstance instance,
                              String variableName)
    throws WfException {
    if (variableName != null) {
        WfAtomicVariable variable =
            instance.getAtomicVariable(variableName);
        WfValue value = variable.getValue();
        if (value instanceof ResourceValue) {
            String contentId = ((ResourceValue) value).getContentId();
            if (contentId == null) {
                throw new WfException(WfException.RESOURCE_ID_IS_NULL,
                                     new String[]{this.toString(), variableName});
            }
            return contentId;
        }
        throw new WfException(WfException.VALUE_HAS_WRONG_TYPE,
                              new String[]
                               {this.toString(), "ResourceValue", variableName});
    }
    throw new WfException
        (WfException.NO_PARAMETER_VARIABLE_SPECIFIED, this.toString());
    }
}

// Get the XML content of the document
protected String getXml(String id, String field)
    throws WfException {
    try {
        return WfServer.getConnection().getContentRepository().
            getContent(id).getMarkup(field).toString();
    } catch (ContentException e) {
        throw new WfException(WfException.ERROR,
                              "Cannot fetch xml from resource id " + id,
                              new String[0], e);
    }
}

```

```

}

// This method will be called by the workflow
public WfActionResult execute(WfInstance instance)
    throws WfException {
    String to = getStringValue(instance, receiverVariable);
    String documentId = getContentId(instance, documentVariable);
    String field = getStringValue(instance, fieldVariable);
    String body = getXml(documentId, field);
    try {
        return new WfActionResult(send(host, user, password, from,
            to, subject, body));
    } catch (Exception e) {
        throw new WfException(WfException.ERROR,
            "Cannot send mail",
            new String[0], e);
    }
}

protected Message createMessage(Session session, String from,
    String to, String subject, String text)
    throws MessagingException, AddressException {
    MimeMessage message = new MimeMessage(session);
    message.setFrom(new InternetAddress(from));
    message.addRecipient(Message.RecipientType.TO,
        new InternetAddress(to));
    message.setSubject(subject);
    message.setText(text);
    return message;
}

// The method which sends the mail
protected boolean send(String host, String username,
    String password, String from,
    String to, String subject, String text)
    throws MessagingException, AddressException {
    // Get session
    Session session = Session.getDefaultInstance
        (System.getProperties(), null);
    // Define message
    Message message = createMessage(session, from, to, subject,
        text);
    // Send message
    Transport transport = session.getTransport(transportType);
    MessageDelivery delivery = new MessageDelivery();
    transport.addTransportListener(delivery);
    transport.connect(host, username, password);
    message.saveChanges();
    // don't forget this
    transport.sendMessage(message, message.getAllRecipients());
    transport.close();
    transport.removeTransportListener(delivery);
    boolean delivered = delivery.isMailDelivered();
    return delivered;
}

```

```

protected static class MessageDelivery
    implements TransportListener {
    private static final long TIMEOUT = 60000;

    // wait 60 seconds for delivery
    protected synchronized boolean isMailDelivered() {
        long timeout = System.currentTimeMillis() + TIMEOUT;
        while (delivered == null) {
            long now = System.currentTimeMillis();
            if (now >= timeout) {
                break;
            }
            try {
                wait(timeout - now);
            } catch (InterruptedException e) {
            }
        }
        return delivered.booleanValue();
    }

    protected void deliverySuccess(boolean state) {
        synchronized (this) {
            delivered = new Boolean(state);
            notifyAll();
        }
    }

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

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

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

    private Boolean delivered = null;
}

/**
 * Prepare workflow for passing variables to the action
 */
public void setReceiverVariable(String receiverVariable) {
    this.receiverVariable = receiverVariable;
}

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

```

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

## 8 Support

Find a list of different ways to get support for CoreMedia CMS here:

CoreMedia systems are distributed systems that have a rather complex structure. This includes databases, hardware, operating systems, drivers, virtual machines, class libraries, customized code etc. in many different combinations. That's why we need detailed information about the environment for a support case. In order to track down your problem, we need: [»» Support request](#)

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

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

To put it in a nutshell, we need:

[»» Support checklist](#)

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

An essential feature for the CoreMedia system administration is the output log of Java processes and CoreMedia components. They're often the only source of information for error tracking and solving. All protocolling services should run at the highest log level that is possible in the system context. For a fast breakdown, you should be logging at debug level. The location where component log output is written is specified in it's \*.properties or \*.jpfif startup file. » Log files

### Which Log File?

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

### Where do I Find the Log Files?

Log files can be found in the CoreMedia component's installation directory in /var/logs.

Component	Problem	Log files
CoreMedia Editor	general	editor.log contentserver.log workflowserver.log capclient.properties
	check-in/check-out	editor.log contentserver.log workflowserver.log capclient.properties
	publication or preview	contentserver.log (Content Management Server) contentserver.log (Master Live Server) workflowserver.log capclient.properties
	import	importer.log contentserver.log capclient.properties
	workflow	editor.log workflowserver.log contentserver.log capclient.properties

Table 108: Log files check list

Component	Problem	Log files
	spell check	editor.log MS Office version details contentserver.log
	licenses	contentserver.log ( <i>Content Management Server</i> ) contentserver.log ( <i>Master Live Server</i> )
Server and client	communication errors	editor.log contentserver.log ( <i>Content Management Server</i> ) contentserver.log ( <i>Master Live Server</i> ) *.jpfif files
	preview not running	contentserver.log (content server) httpd.log
	website not running	contentserver.log ( <i>Content Management Server</i> ) contentserver.log ( <i>Master Live Server</i> ) contentserver.log ( <i>Replication Live Server</i> ) httpd.log capclient.properties license.zip
Server	not starting	contentserver.log ( <i>Content Management Server</i> ) contentserver.log ( <i>Master Live Server</i> ) contentserver.log ( <i>Replication Live Server</i> ) capclient.properties license.zip

### Email, phone or fax

» *Email, phone and fax:*

The best way to submit your Support queries is via email. In addition to giving system specifications and describing the problem in detail, you can also attach important log and system files to emails.

[support@coremedia.com](mailto:support@coremedia.com)

Our Support employees are available to take your support call weekdays between 9 a.m. and 6 p.m.

Phone: +49. 40 .32 55 87 .777

Fax: +49. 40 .32 55 87 .999

### Online Support

» *Bugtracker, news-groups*

Use our Online Support to submit a support ticket, track your submitted tickets or receive access to our forums. You can access our Online Support at:

<http://support.coremedia.com>

### Bugtracker

Our Bugtracker can be used to submit software errors (bugs) and feature requests directly to CoreMedia developers or to search for bugs that have already been fixed:

<http://jira.coremedia.com>

### Access to Bugtracker and online services

Partners and Support customers will need to register an account with CoreMedia in order to receive access to the Bugtracker. Just send us an email and we will send you your login details:

[support@coremedia.com](mailto:support@coremedia.com)

If you have any other questions or comments, please contact:

CoreMedia AG

Ludwig-Erhard-Strasse 18

20459 Hamburg

Phone: +49.40.32 55 87 .777 Fax: .999

[www.coremedia.com](http://www.coremedia.com)

[support@coremedia.com](mailto:support@coremedia.com)

Our manuals undergo permanent revision, and we are closely tracking progress in development and expertise. To make our manuals valuable tools in development and implementation of the *CoreMedia CMS* and Solutions, do not hesitate to contact us for ideas and suggestions: >> *Documentation*

- >> via email: [documentation@coremedia.com](mailto:documentation@coremedia.com)
- >> via fax: +49.40.325587.999
- >> or comment directly in our online documentation at <http://documentation.coremedia.com>

## Glossary

### Active Delivery Server (ADS)

A component of *CoreMedia CMS*, responsible for the dynamic generation of output pages (e.g. HTML) from documents.

### API

An Application Programming Interface (API) is an open interface of a program with which developers can program extensions and adaptations.

### Article

Typical document type for editorial content.

### Attribute

Applied to a text section or to parts of a table in order to create a special behaviour or layout.

### BLOB

Binary Large Object, a property type for binary objects, e.g. graphics.

### Cacheable

In the JSP tag **coremedia:page** of a template, the attribute **cacheable** can be set to the value **"true"** if the template is deterministic, i.e. if it is only dependent on the ResourceUri accessed and the content of the repository, and is independent of random or time values.

### Category

Documents can usually be allocated to certain regions or areas (e.g. sport or poli-

tics). These regions or areas are called categories here.

### Change Set

Since *CoreMedia CAP 4.1* resource changes in structure and content can be published separately.

These changes to be published are assembled in change sets; once the change set has been completed it can then be published via the workflow.

### Content

The media-neutral, information-containing content of a document which can be prepared for different outputs (e.g. paper or screen), each time with an appropriate layout.

### Content Server

Server on which the documents are edited. Edited documents are published on the Live Server.

### 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 Editor**

Swing component of CoreMedia for editing documents.

**CRM**

Abbreviation for Customer Relationship Management. The aim of applying CRM tools is to improve the relationship of the company to the customer and to organise this relationship in a long-term and profitable way. To this end, customer behaviour data is collected and analysed. Due to the improved transparency, the marketing mix, consisting of communication, distribution and offer policies, can be suited to the customers' needs.

**Dead link**

A link, whose target does not exist.

**Document**

In the *CoreMedia CMS*, documents are specified by properties. Typical document properties are, for example, title, author and text content.

**Document type**

A document type describes the properties of a certain type of documents. Such properties are e.g. title, text content, author, ...

**DOM**

The Document Object Model is a standard interface developed by W3C which enables applications to access parts of a document over a tree-like model.

**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.

**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.

**Folder hierarchy**

Tree-like connection of folders, where the root folder forms the origin of the tree.

**Home page**

Starting page, a web page individually determined by a user (user view) or the entry page to a website for all users (provider view).

**HTML**

Hypertext Markup Language - description language for web pages.

**HTTP**

The HyperText Transfer Protocol (HTTP) serves to transport HTML pages between Web servers and Web browsers.

**Hyperlink**

Link from one web page to another web page.

**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.

**IPTC**

The International Press Telecommunications Council is composed of representatives of large news agencies and publishing houses and develops standards for exchanging press reports.

**Java**

Java is an object-oriented programming language which supports particularly well the development of software in the web environment. The CoreMedia system is implemented in Java.

**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.

**Java Runtime Environment (JRE)**

The part of the Java platform necessary for executing Java programs (i.e. JVM and libraries, but no Java compiler).

**JDBC (Java Database Connectivity)**

The term *JDBC* describes the Java API for access to a wide variety of data sources, usually, but not exclusively, SQL-based. JDBC is part of the Java Standard Edition.

**JDK (Java Development Kit)**

The term *JDK* describes the official Sun Java *software development kit* (SDK).

The primary components of the JDK are:

1. javac, the compiler
2. jar, the archiver
3. javadoc, the documentation generator
4. jdb, the Java debugger

**JRE**

See Java Runtime Environment.

**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.

**JVM**

Abbreviation for Java Virtual Machine. An interpreter which executes the bytecode generated by Java compilers. Core component of the JRE.

**Layout**

Visual design of documents, e.g. in two columns, italic headings, page numbers at the upper outer edge.

**Live Server**

The component of CoreMedia Smart Content Infrastructure which provides published documents for general access.

**Markup**

Marking of parts of a document, structurally (section, paragraph, quote, ...) or with layout (bold, italic, ...).

**MD5**

An algorithm which calculates a 128 Bit long Message Digest (a type of fingerprint) from a text of any length. The algorithm is such that it is extremely difficult (i.e. practically impossible with current technology) to generate two texts with the same Message Digest, or to find a text with a given Message Digest.

**MIME**

With Multipurpose Internet Mail Extensions (MIME), the format of multi-part, multimedia emails and of web documents is standardised.

**Navigation**

Selectively calling up web pages by clicking on hyperlinks.

**NITF**

Abbreviation for News Industry Text Format. An XML format specified by the IPTC, especially suited for press reports.

**Personalisation**

On personalised websites, individual users have the possibility of making settings and adjustments which are saved for later visits.

**Property**

In relation to CoreMedia, properties have two different meanings:

In CoreMedia, documents are described with properties (document fields). There are various types of properties, e.g. strings (such as for the author), Blobs (e.g. for images) and Sgmltext for the textual content. Which properties exist for a document depends on the document type.

In connection with the configuration of CoreMedia components, the system behaviour of a component is determined by properties.

**Publish**

Creates or updates resources on the Live Server.

**pull approach**

Pull approach means that a task is offered to a selected set of users. These users are free to accept the task. In contrast to the pull approach is the push approach which means that a task is assigned to a single user.

**Repository**

The document store of the CoreMedia system. Repository is an abstract term, technically a database lies behind it.

**Resource**

A folder or a document 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.

**role**

In a workflow a task specifies roles which a user must fill to execute the task. Roles are defined as sets of users. A workflow user can take over different roles in a workflow and a role can be filled by different users. Because the workflow engine uses a pull approach for task distribution, a task is typically offered to all members of a role.

**Root folder**

The uppermost folder in the CoreMedia folder hierarchy. Under this folder, CoreMedia users can add further folders and documents.

**SAX**

The Simple API for XML is a standard interface for event-based XML parsers.

**SGML**

The Standard Generalized Markup Language was the Advent of XML. It included all of the main concepts that make up XML but could be very complex. Due to its inherent complexity the development of Software Components is very difficult too. The main problem with SGML is that it is *not* context free. Imagine writing a Parser for that.

The development of SGML started in the 1970ies. One of the main Evangelists was Charles Goldfarb. Early adopters included the US Military, that used SGML for purposes of documentation. Even today Tank and Artillery manuals are delivered on CD-ROM in an SGML format.

**SMTP**

Simple Mail Transfer Protocol, standard protocol to transmit E-Mails

**SQL**

SQL [Structured Query Language] is a standardised query language which contains all language elements necessary for carrying out all operations which occur when using a relational database.

**subworkflow**

Subworkflows are simple workflows, without any special restrictions. Subworkflows are used as sub elements in workflows.

**Surfer**

Colloquial term for a user of the World Wide Web.

**Teaser**

A short piece of text or graphics which contains a link to the actual editorial content.

**Template**

In CoreMedia, JSPs used for displaying documents are known as Templates.

**Thumbnail**

A thumbnail is a ("thumbnail-sized") reduced version of an image used as a preview.

**Transitive Envelope**

A set which is closed in relation to a certain link. In connection with CoreMedia, this always denotes a set of documents of which none links to a document outside this set.

**Unicode**

A character set standard consisting of 16-bit characters. Unicode was developed by the Unicode consortium. Unicode uses two bytes in order to describe a character, allowing the coding of up to 65.536 characters.

**URL**

With a Uniform Resource Locator (URL), content can be referenced in the Web. The best known are the HTTP URLs which link to HTML pages.

**User name**

Every user must log on to the CoreMedia system with a user name. In particular, this is also true for non-human users such as the importers. The rights of the user are

administrated internally with the user name.

**Version history**

A newly created document receives the version number 1. If new versions are created when the document is checked in, these are numbered in chronological order.

**WAP**

The Wireless Application Protocol (WAP) is an open, global specification that empowers mobile users with wireless devices to easily access and interact with information and services instantly.

**Web**

Also: World Wide Web (WWW). An internet service for publication of multimedia documents. In everyday language, the synonym for "The Internet".

**Web browser**

A program for displaying HTML pages, especially Web pages. The best-known browsers are Netscape Navigator and Microsoft Internet Explorer.

**WebDAV**

WebDAV stands for World Wide Web Distributed Authoring and Versioning Protocol. It is an extension of the Hypertext Transfer Protocol (HTTP), which offers a standardised method for the distributed work on different data via the internet. This adds the possibility to the CoreMedia system to easily access CoreMedia resources via external programs. A WebDAV enabled application like Microsoft Word is thus able to open Word documents stored

in the CoreMedia system. For further information, see <http://www.webdav.org>.

**Web Page**

An individual page, as displayed by a Web browser and presented to the user.

**Web server**

A program which provides the HTML pages of a Website.

**Website**

A website consists of several thematically-connected Web pages (e.g. the Internet presence of a company), which are linked to each other with Hyperlinks.

**WML**

WML (Wireless Markup Language) is a markup language based on XML, and is intended for use in specifying content and user interface for narrowband devices, including cellular phones and pagers. WML is designed with the constraints of small narrowband devices in mind.

**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 definition**

A workflow definition includes a set of task definitions and a description of the flow of control between those tasks. To start a workflow, a workflow definition has to be instantiated, resulting in a workflow instance.

**workflow instance**

A workflow instance is the result of an initiated workflow definition and has a state.

**XML**

Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879]. By construction, XML documents are conforming SGML documents.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

**XSL**

Abbreviation for Extensible Stylesheet Language. XSL consists of two parts: XSLT, a transformation language for XML docu-

ments, and XSL Formatting Objects, an XML vocabulary for describing the formatting of documents.

## Index

<b>A</b>		editor.xml	126
AbstractClientAction	95	expressions	52, 107, 112
access repository	101	-example	112
access variables	99	-definition in workflow	112
AclRightsPolicy	114	definition	
action	70	expressions:boolean	111
-SkipUserTask	70	expressions:generic	109
-StoreProperties	70	<b>G</b>	
-AssignVariable	70	getAtomicVariable	99
actions	54, 70, 90, 91, 92	group	29
-predefined ones	70	GUI configuration	126
-where to use	91	<b>I</b>	
-DisapproveResource	70	interface	126
-general rules for implementation	91	<b>M</b>	
-ForceUser	70	mail action	194
-Log	70	-complete code	194
actions:client-side	95	<b>P</b>	
actions:server-side	93	performer policy	120
activity diagrams	32	-customizing	120
<b>B</b>		performers policy	120
BeanParser	31, 126	-definition in workflow	120
-Editor configuration	127	definition	
-workflow DTD	32	postconditions	52, 53
-workflow definition	31, 60	preconditions	52, 53
BeanParser	35	predefined classes	81
<b>C</b>		-property editor classes	81
case	149	predefined workflows	27
choice	46	process	39
client action	95	processes	127
-timeout	95	<b>R</b>	
components	20	repeated execution	92
conditions	52	rights	55, 55
control flow	46	-definition in workflow	114
<b>D</b>		definition	
DefaultPerformersPolicy	120	rights policy	114
DTD coremedia-workflow	143	-definition in workflow	114
<b>E</b>		definition	
		roles:workflow	29
		<b>S</b>	

serialization	124	-UML	32
-pitfalls	124	workflow process	39
serialization errors	88	workflow roles	29
server-side stub	95	workflow startup classes	137
<b>T</b>		workflow variables	51
task	44, 44		
-automated tasks	44		
-definition	44		
tasks	40		
-possible states	40		
-state diagram	40		
timer	57		
TimerHandler	57		
timer handler	79		
-predefined	79		
Typographic conventions	8		
<b>U</b>			
UML	32		
-description	32		
update workflow definitions	88, 124		
upload new workflows	59, 124		
user	29		
user task	44		
<b>V</b>			
validator	53		
<b>W</b>			
WfAction	93		
WfExpression	109		
WfPerformersPolicy	120		
WfRightsPolicy	114		
window	21		
workflow	56		
-subworkflows	56		
workflow clients	123		
-programming	123		
workflowconverter	88, 124		
workflow definition	31, 60		
-example	60		
-create	31		
workflow modeling	32		