



# OpenCms Documentation

---

## Version 5.0

This document contains introductory material about OpenCms. The full 5.0 documentation is only available in the form of OpenCms content modules. Please see chapter [1](#) to learn how to obtain and install these modules.

[www.opencms.org](http://www.opencms.org)

May 5, 2003

# Contents

<b>I. Introduction to OpenCms</b>	<b>1</b>
<b>1. About this documentation</b>	<b>3</b>
1.1. Version of this document . . . . .	3
1.2. The documentation structure of OpenCms 5.0 . . . . .	3
1.3. Where to get the documentation modules . . . . .	4
1.3.1. OpenCms documentation base . . . . .	4
1.3.2. OpenCms JSP basic documentation . . . . .	4
1.3.3. OpenCms JSP taglib documentation . . . . .	4
1.3.4. OpenCms JSP scriptlet documentation . . . . .	4
1.3.5. OpenCms Flexcache documentation . . . . .	5
1.3.6. OpenCms module documentation . . . . .	5
1.3.7. Flex examples set 1 . . . . .	5
1.3.8. Flex examples set 2 . . . . .	5
1.3.9. Flex examples set 3 . . . . .	6
1.3.10. Original JSTL 1.0 standard taglib examples . . . . .	6
1.3.11. Original Tomcat 4.x JSP examples . . . . .	6
1.3.12. Howto: JSP template development . . . . .	6
1.3.13. Howto: Translating the OpenCms workplace . . . . .	7
<b>2. Installation of OpenCms 5.0</b>	<b>9</b>
2.1. Install the Java JDK . . . . .	9
2.2. Install Tomcat . . . . .	9
2.3. Install MySQL . . . . .	10
2.4. Deploy the opencms.war file . . . . .	10
2.5. Install OpenCms using the Setup-Wizard . . . . .	11
2.6. Now your system is ready . . . . .	11
2.7. Security issues . . . . .	11
<b>3. Client Setup</b>	<b>13</b>
3.1. Configuring MS Internet Explorer 5.5 and 6.x Clients . . . . .	13
3.2. Configuring Netscape Navigator 7.x Clients . . . . .	14

<b>4. The OpenCms module mechanism</b>	<b>15</b>
4.1. Introduction to OpenCms modules	15
4.2. How to import a module into OpenCms	15
4.2.1. Restarting the OpenCms server after module import	17
4.3. How to create, administrate, replace and export a module	17
<b>5. OpenCms User Manual</b>	<b>19</b>
5.1. Introduction	19
5.1.1. What is OpenCms?	19
5.1.2. The advantages of OpenCms	19
5.1.3. Configuring the client	19
5.2. Section 1: OpenCms - The demo	21
5.2.1. Editing a website	21
5.2.2. Creating a new page	28
5.3. Section 2: OpenCms - The mechanics	32
5.3.1. The user interface	32
5.3.2. View Modes	32
5.3.3. Working on a project	33
5.3.4. Access permissions	38
5.3.5. The editor	40
5.3.6. Workflow	40
<b>6. Components used by OpenCms</b>	<b>43</b>
6.1. Operating System	43
6.2. Webserver	43
6.3. Servlet runtime engine	43
6.4. Java VM	44
6.5. Database	44
6.6. XML Parser	44
6.7. Mail API	44
6.8. FESI (a Free EcmaScript Interpreter)	44
6.9. JTidy	45
6.10. Jakarta-ORO	45
<b>7. Useful resources</b>	<b>47</b>
7.1. OpenCms online resources	47
7.2. How you can help	47
7.3. How to subscribe to the opencms-dev mailing list	48
<b>II. Archived documentation</b>	<b>49</b>

<b>8. About the archived documentation</b>	<b>51</b>
8.1. Introduction to the archived chapters . . . . .	51
8.2. How you can help . . . . .	51
<b>9. Enterprise JavaBean Integration</b>	<b>53</b>
9.1. Advantages of OpenCms & EJB . . . . .	53
9.2. EJB basics . . . . .	54
9.3. Calling an EJB . . . . .	55
9.4. Refinements . . . . .	56
9.4.1. Configuration . . . . .	56
9.4.2. Error handling . . . . .	57
<b>10. The Idea of Content Definitions</b>	<b>59</b>
10.1. Writing a simple Content Definition . . . . .	60
10.2. Adding access control to a Content Definition . . . . .	62
10.3. Content Definition enhancements . . . . .	64
<b>11. Backoffice Modules</b>	<b>71</b>
11.1. Workplace Classes in general . . . . .	71
11.2. Abstract Backoffice class . . . . .	73
11.3. Changes to the Abstract Backoffice class since OpenCms 4.4 . . . . .	76
11.3.1. Changes to the getContent... methods . . . . .	77
11.3.2. Additional method getSetupUrl . . . . .	78
11.3.3. Filling the template from the CD automatically . . . . .	78
11.3.4. Filling the CD from the template's fields automatically . . . . .	78
11.3.5. Testing for errors . . . . .	79
<b>12. The Mastermodule</b>	<b>81</b>
<b>13. Static Export</b>	<b>89</b>
13.1. Introduction . . . . .	89
13.2. Setting up Static Export . . . . .	89
13.3. How to use static export . . . . .	90
13.4. What have I to do? . . . . .	90
13.5. How does static export work? . . . . .	91
13.6. How to control what is exported? . . . . .	91
13.7. How to export resources with parameters? . . . . .	92
13.8. Advanced properties of the static export: switch standard to dynamic . . . . .	93
13.9. Advanced properties of the static export: Configure Https . . . . .	93
13.10. Advanced properties of the static export: Relative Links in Export . . . . .	94
13.11. Advanced properties of the static export: The Rulesets . . . . .	94

<b>14. The OpenCms synchronization</b>	<b>97</b>
14.1. What is synchronization?	97
14.2. Manage the properties for OpenCms synchronization	97
14.3. How to synchronize	98
14.4. How the files are synchronized	99
<b>15. System architecture</b>	<b>101</b>
15.1. Components	101
15.2. Accessing system resources	102
15.3. Resources	105
15.4. The Virtual File System (VFS) of OpenCms	106
15.5. Class structure	106
<b>16. Core Development</b>	<b>109</b>
16.1. Introduction	109
16.2. How to compile the OpenCms core	109
16.2.1. Before you start	109
16.2.2. Have Ant installed	110
16.2.3. Get the OpenCms Source distribution	110
16.2.4. Get additional classes (optional)	110
16.2.5. Build the source by starting Ant	111
16.2.6. Install your new version	111
16.2.7. Other Ant targets	111
16.2.8. Important	112
16.3. The Projectmechanism	113
16.3.1. Data structure	113
16.3.2. During installation of OpenCms	114
16.3.3. Creating a new project	114
16.3.4. The view of the project	116
16.3.5. Publish a project	116
<b>17. The proprietary XML template mechanism</b>	<b>117</b>
17.1. Introduction to the XML template mechanism	118
17.1.1. The master- frame- and contenttemplate	118
17.1.2. Using methods in XML templates	122
17.1.3. The body element	123
17.1.4. Defining the layout in the frametemplate	126
17.2. XML templates and directories	128
17.2.1. The XML structure of a template	128
17.2.2. Directory structure	129
17.2.3. The subdirectories of the content folder	129
17.3. Details of the proprietary XML template mechanism	131

17.3.1. Using data blocks . . . . .	131
17.3.2. Setting data blocks dynamically . . . . .	132
17.3.3. Using user-defined methods . . . . .	134
17.3.4. The relationship between templates, page-control files, and Java classes . . . . .	135
17.3.5. Process, Method and Element Tags . . . . .	136
17.3.6. Stylesheets . . . . .	150
17.3.7. Javascript Blocks . . . . .	150
17.3.8. Methods of the proprietary XML template mechanism . . . . .	152
17.3.9. Using the element cache for XML templates . . . . .	155
<b>18. Building applications with XML templates</b>	<b>163</b>
18.1. Building a navigation . . . . .	163
18.2. Navigations with XML Templates . . . . .	163
18.3. Frames . . . . .	174
18.3.1. Frames with XML templates . . . . .	174
18.3.2. Frames- and noframes versions of a website with XML templates . .	177
18.4. The CmsObject Class (accessing system resources) . . . . .	179
18.4.1. Accessing user data . . . . .	179
18.4.2. Resource access . . . . .	182
18.4.3. Session management . . . . .	184
18.4.4. A first example using session management . . . . .	185
18.4.5. Session content . . . . .	188
18.4.6. Session history . . . . .	189
18.5. Forms . . . . .	192
18.5.1. A first example using forms . . . . .	192
18.5.2. Presetting values . . . . .	195
18.5.3. Evaluating forms . . . . .	196
18.5.4. Validating Forms . . . . .	200
18.5.5. Using radio buttons and checkboxes . . . . .	201
18.5.6. Presetting the values of select boxes and radio buttons in Java . . .	204
18.5.7. Using session management with forms over multiple pages . . . . .	206
18.5.8. Sending e-mails . . . . .	208
18.5.9. Personalization . . . . .	210





# **Part I.**

## **Introduction to OpenCms**



# 1. About this documentation

## 1.1. Version of this document

This document is Revision: 1.8

It was last changed in the CVS as of Date: 2003/04/02 13:46:13

## 1.2. The documentation structure of OpenCms 5.0

We have finally decided to eat our own dogfood and manage the contents of the documentation with our content management system. So from version 5.0 onward, the main part of the documentation will be maintained in the form of OpenCms interactive documentation modules. These are content modules that you can install and read directly in OpenCms. One of the big advantages is that this gives the reader an interactive documentation with a lot of working example code, because everything can directly be checked out in the running system. It's also easier to write the documentation that way.

The main purpose of this book is now to provide introductory material about OpenCms, especially providing installation instructions, an "end user" introduction and a general system overview.

It will take some time to shift all of the contents of the previous "book only" version to the content module form, so this document will likely be around for some time. However, the new functionality of 5.0 and the following releases are not described here, but only in the content modules. Please refer to chapter 8 for more information about the chapters of the documentation that still have to be shifted to the interactive format.

Of course, if someone wants to contribute his time and do the work to put the interactive documentation modules back into this "static" form, this is very much appreciated. On the other hand we would also appreciate if someone would make interactive documentation modules from the parts of the book that are still described only in this static document. Please send an email to [contributions@opencms.org](mailto:contributions@opencms.org) if you are interested in doing any of this.

## 1.3. Where to get the documentation modules

As of the writing of this document, the following documentation modules are available:

### 1.3.1. OpenCms documentation base

This module provides a set of JSP templates and elements that are used by all other documentation modules provided by Alkacon Software. *This module is required by all other com.alkacon.\* documentation modules listed here and has to be installed first!*

Package name: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.2. OpenCms JSP basic documentation

A basic introduction to JSP in OpenCms: How to create, edit & publish a JSP, options to access the OpenCms system and some technical background information.

Package name: `com.alkacon.documentation.documentation-jsp`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.3. OpenCms JSP taglib documentation

OpenCms includes it's own JSP taglib. This is the reference documentation (with test cases) for each tag of the OpenCms Taglib.

Package name: `com.alkacon.documentation.documentation-taglib`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.4. OpenCms JSP scriptlet documentation

Learn here how to use the same functionality offered by the OpenCms JSP taglib in JSP scriptlet code in your JSPs.

Package name: `com.alkacon.documentation.documentation-scriptlet`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.5. OpenCms Flexcache documentation

The Flexcache can cache JSP output to increase the speed of page delivery. This is the complete documentation of all FlexCache configuration options and parameters to set up your page caching policy using the Flexcache.

Package name: `com.alkacon.documentation.documentation-flexcache`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.6. OpenCms module documentation

Get acquainted with the OpenCms module mechanism that allows you to easily bundle and distribute templates and JSPs.

Package name: `com.alkacon.documentation.documentation-modules`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.7. Flex examples set 1

The first set of tests and examples developed for the OpenCms / JSP Flex integration package. Basic stuff dealing with inclusion, caching, headers and redirects.

Package name: `com.alkacon.documentation.examples-flex-1`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.8. Flex examples set 2

The second set of tests and examples, including demos on how to build a simple template from JSP.

Package name: `com.alkacon.documentation.examples-flex-2`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.9. Flex examples set 3

Further test cases and examples for the OpenCms/Flex JSP integration. These are rather specialized as they deal with encoding and inclusion of subelements.

Package name: `com.alkacon.documentation.examples-flex-3`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.10. Original JSTL 1.0 standard taglib examples

The JSTL will be part of the next JSP/Servlet specification release. These are the original JSTL 1.0 examples imported into OpenCms.

Package name: `com.alkacon.documentation.examples-jstl`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.11. Original Tomcat 4.x JSP examples

The original Tomcat 4.x JSP examples imported into OpenCms. Check these out to see what you can do with JSP in OpenCms.

Package name: `com.alkacon.documentation.examples-tomcat`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.12. Howto: JSP template development

An introduction in the JSP template development with OpenCms. Learn how to create templates for your web site.

Package name: `com.alkacon.documentation.howto-template`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*

### 1.3.13. Howto: Translating the OpenCms workplace

A detailed description how to translate the OpenCms workplace and the online help to another language.

Package name: `com.alkacon.documentation.howto-workplace-translation`  
Depends on: `com.alkacon.documentation`  
Available from: <http://www.alkacon.com/downloads/>  
Provided by: *Alkacon Software*





## 2. Installation of OpenCms 5.0

This chapter provides information on how to install OpenCms using Tomcat and MySQL. All installation parts are described as single steps. After completing each step you are strongly advised to verify the success of your work.

### 2.1. Install the Java JDK

Install Java JDK 1.4 or later (from SUN <http://java.sun.com/products/j2se/>). For details on how to install these components on your operating system, see the documentation that comes with them.

**Important:** This version of OpenCms was tested with Java 1.4 only. Some features regarding file encoding were used that are not available with Java releases before 1.4.

### 2.2. Install Tomcat

OpenCms 5.0 requires a Servlet 2.3 / JSP 1.2 standards compliant container. Tomcat 4 is the reference implementation of this Standard. This release was tested with Tomcat 4.0.x and Tomcat 4.1.x. Older versions of Tomcat (3.x and earlier) do not support this newer standard and are thus not usable for OpenCms 5.0.

Install Tomcat from <http://jakarta.apache.org/tomcat/index.html> into a folder of your choice. This is the CATALINA\_HOME folder. Don't forget to set the environment variables CATALINA\_HOME and JAVA\_HOME.

Test the installation by running Tomcat in standalone mode and check the examples. Note: Tomcat uses port 8080 in standalone mode.

If you wish, you can combine the servlet-engine with a webserver like the Apache Web Server (<http://www.apache.org/httpd.html>). Please see the documentation available with the webserver on how to combine it with your servlet environment.

**Important:** To make sure Tomcat works with the correct charset to read and write files you should add the following parameter to the commandline that is used to start Tomcat:

```
-Dfile.encoding=ISO-8859-1
```

You can also set the environment variable CATALINA\_OPTS to do so:

```
(CATALINA_OPTS=-Dfile.encoding=ISO-8859-1)
```

The default encoding of OpenCms is ISO-8859-1, but you can also set another encoding supported by your Java VM, e.g. UTF-8. In case you do not care about the encoding, the default is most likely what you need.

**Setting the file encoding on Windows systems:** To avoid problems regarding the encoding, run Tomcat in standalone mode and edit the properties of the shortcut "Start Tomcat" in the start menu of Windows. Add the parameter `-Dfile.encoding=ISO-8859-1` to the command line behind the call of the java executable.

Here's an example of how the start command of Tomcat should look like after the modification:

```
c:\j2sdk1.4.1\bin\java.exe -jar -Dfile.encoding=ISO-8859-1  
-Duser.dir="C:\tomcat4" "C:\tomcat4\bin\bootstrap.jar" start
```

## 2.3. Install MySQL

Install MySQL from <http://www.mysql.com/downloads/index.html> (see the MySQL documentation on <http://www.mysql.com/documentation/index.html>). This release was tested with MySQL 3.23.x and 4.0.x.

Note: On Windows-based systems MySQL has to be installed on the C: drive and should be registered as service using `(your MySQL path)/mysql/bin/mysql -install`.

Start the MySQL server by running the service (WIN32) or executing `(your MySQL path)/mysql/bin/mysqld` (UNIX).

Check that MySQL is running before you continue by starting the MySQL monitor (execute `mysql` in your MySQL bin folder). The database works correctly if a MySQL prompt appears after calling the monitor. Quit the MySQL monitor by typing `exit` and go to the next step.

## 2.4. Deploy the opencms.war file

Copy the `opencms.war` file from the binary distribution ZIP file to `CATALINA_HOME/web-apps/`. Replace `CATALINA_HOME` with the real path to your Tomcat installation.

Start (or restart) Tomcat. Tomcat will now deploy the web application OpenCms.

**Important:** OpenCms requires that its `*.war` file is unpacked. OpenCms can not be deployed as war file only. Make sure Tomcat does unpack the war file and creates the `CATALINA_HOME/webapps/opencms/` directory, placing the OpenCms files in this directory.

The default configuration for your Servlet containers / environment could be to not unpack the deployed \*.war file. If this is the case you must unpack the `opencms.war` file manually. Use an unzip tool for this, \*.war files are just \*.zip files with a different extension. The OpenCms setup wizard will display a warning and not allow you to continue if you did not unpack the \*.war file.

## 2.5. Install OpenCms using the Setup-Wizard

Start the Setup-Wizard by pointing your webbrowser to `http://localhost:8080/opencms/ocsetup`. Depending on your configuration, you have to replace `localhost` with your servername. The port 8080 is only used if you start Tomcat in standalone mode.

Follow the instructions of the OpenCms Setup-Wizard, using the "Standard" setup. It will set up the OpenCms database and import all workplace resources into the system. For normal installations with MySQL and Tomcat running on the same server all default settings will fit your needs.

**Important:** In case you want to import content from older OpenCms versions (5.0 beta 1 or 4.x), you must turn on the "directory translation" feature to make sure your pages still work. You can later change this setting (and all other settings selected during setup) by editing the file `CATALINA_HOME/webapps/opencms/WEB-INF/config/opencms.properties`.

## 2.6. Now your system is ready

Now your system is ready to use. You can login with username: `Admin` and password: `admin`. Please change this password as soon as possible. The login URL of OpenCms in a default configuration is:

`http://localhost:8080/opencms/opencms/system/login/`

## 2.7. Security issues

After you have installed OpenCms you should have a look at the security settings.

First change the `Admin` password of OpenCms by calling the user preferences (the "hammer" icon on the main screen of the workplace). Then you should add a password to the MySQL database. Enter the following commands at the MySQL command line.

```
use mysql;
insert into user values ('localhost', 'opencmsuser', password('XXXXX'));
```

```
'N','N','N','N','N','N','N','N','N','N','N','N','N','N','N');  
insert into db values ('localhost', 'opencms', 'opencmsuser', 'Y','Y',  
  'Y','Y','Y','Y','Y','Y','Y','Y');  
flush privileges;
```

Make sure you replace `opencmsuser` and `opencms` with the name of your user and database in case you have changed them on the setup wizard.

Don't forget to add the new user and password to all connect strings of the database in your `opencms.properties` file. Only the new user can now connect to the OpenCms tables. For more information see the MySQL documentation.

## 3. Client Setup

**All configurations issues on this page only apply when you set up a workstation client computer to access the backoffice part of OpenCms.**

The backoffice is the part where you can edit pages, create new pages, manage users etc. In OpenCms this is called "the Workplace" and it makes use of dynamic HTML and some plugins to provide editing functionality that can not be achieved using standard HTML.

**The HTML generated by OpenCms as output for the final website is 100% controlled by you and no extra setup or plugin is needed for clients to access the OpenCms generated sites.**

For the development of OpenCms, we so far are focused on the implementation of the server components. To save time during the development we decided to use existing components for specific client functions. This means that ActiveX components were used for the user editors to offer the end user a rich set of functionality. This, however, currently restricts the WYSIWYG editor to Microsoft Internet Explorer. For Netscape Navigator, a simple Textarea is used as source code editor for both WYSIWYG and source code mode.

It would be possible to replace the ActiveX controls with Java applets or other technology. Recently some Java applets have become available in the open source space that could be fitted into OpenCms. We would appreciate any contribution of such an alternative editor. If you are interested in doing this, please send an email to *contributions@opencms.org*

### 3.1. Configuring MS Internet Explorer 5.5 and 6.x Clients

First step is the installation of the necessary controls. The following components are needed by OpenCms:

1. For the WYSIWYG editor, the "Dynamic HTML Edit Control" is used. This control is part of all MS IE installations since version 5.0, which means that it is already installed if your IE version is not older than 5.0.
2. The source code editor is a component developed by **AY Software** and it's called LeEdit OCX Control. You can download the shareware version of this control from the site:  
<http://www.aysoft.com/ledit.htm>. This control must be installed on all clients

that need access to the code editor functionality. If you do not install this component, OpenCms will provide a html textarea for the source code input, which is less convenient but also usable in general.

**Note:** The next step is optional and only required for the full functionality of the source code editor. If you don't need the advanced functions like search, replace or undo in the source code editing mode and want to work with a html textarea, you can ignore the following ActiveX settings.

The second step is configuring the ActiveX settings for the source code editor to work properly. Open IEs "Internet options." Then do the following:

1. On the tab "Security", select "Trusted site zones" from the drop-down menu and click on "Add Sites" to add the URL (e.g. `http://opencms.mycompany.com` - ask your system administrator for the exact URL) of the zone's OpenCms server. Deactivate the radio button "Require server verification (https:)" for all sites in this zone."
2. On the tab "Security", select "Trusted site zones" from the drop-down menu and click on "Settings". All ActiveX control elements must be set to "Enable." A note on security: It is safe to use ActiveX controls with these settings since their use is allowed only for the "Trusted sites", and ActiveX remains disabled for all other web sites.

This setup must be repeated for all clients / workstations that use the OpenCms workplace. Cookies and JavaScript must be enabled on these machines.

## 3.2. Configuring Netscape Navigator 7.x Clients

In Netscape Navigator OpenCms switches automatically to an html text area in which the end user can edit the content as html source code. A WYSIWYG editor is currently not available for Netscape Navigator Clients.

Cookies and JavaScript must be enabled to use the workplace.

# 4. The OpenCms module mechanism

## 4.1. Introduction to OpenCms modules

OpenCms offers a module mechanism which provides an easy and convenient way to bundle and distribute templates and other functionality. A module usually consists of a set of templates, images, Java classes or libraries and other resources that depend on each other and thus should be distributed together.

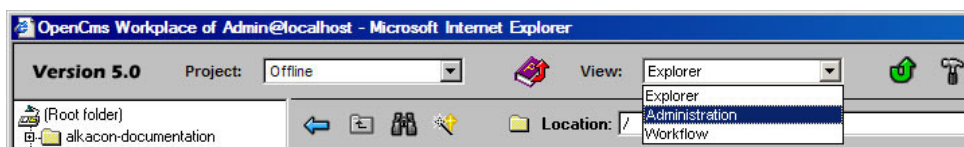
This chapter provides an overview about the module installation process. The full module documentation is available only in the interactive part of the OpenCms documentation. This interactive documentation is of course packaged in modules, so to install the interactive documentation you first need to know how to install modules in OpenCms. This is why we have this chapter also here in the book part of the documentation.

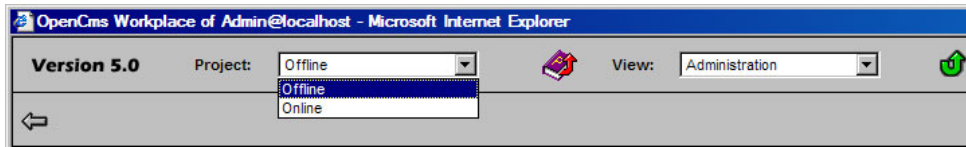
Modules are managed in the OpenCms workplace in the Administration view under the option “Module management”. You can create, import, export or delete modules there. A module in a running system is a set of resources (folders and files) in the OpenCms VFS. If you export a module, all resources that belong to the module are written to a single ZIP file, including system settings required by OpenCms for the module to run.

You can import a module ZIP created this way in another OpenCms system with the module import option. The module ZIP file will then be unpacked and its content will be copied to the specified location in the OpenCms VFS. Information about all installed modules is stored in the file `/WEB-INF/config/registry.xml`, located in the OpenCms servers web application context.

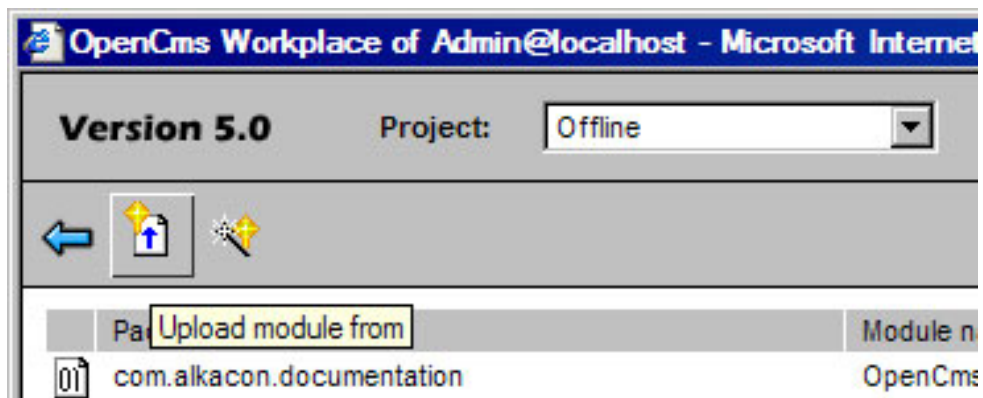
## 4.2. How to import a module into OpenCms

- Select *Administration* in the *View* selection of the workplace.

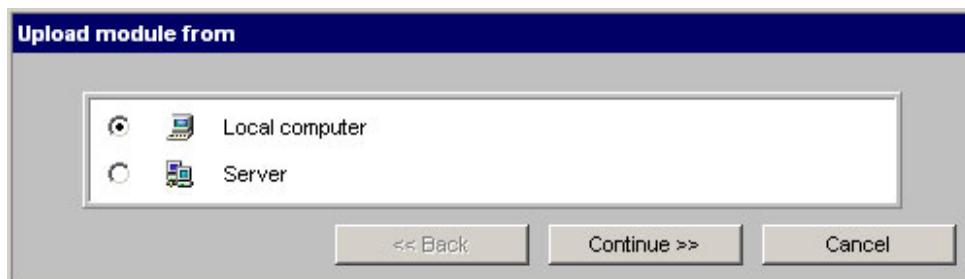




- Select any “offline” project (e.g. the default *Offline* project) in the *Project* selection:
- Open the *Module management*.
- Click the *Upload module from* icon:



- The module import wizard starts. On the first page of the wizard you can choose from where you want to upload the module’s zip file:



- *Local computer*: use this option to upload the module anywhere from your local filesystem (using http upload). On the next wizard page a file search dialog is shown and you can select the module’s zip file in the local file system of your computer.
- *Server*: use this option to upload the module from the `WEB-INF/export/modules/` directory of your OpenCms web application on the server. On the next wizard page you get a selection with the package names of all modules found in `WEB-INF/export/modules/` to choose from.



- Finally, click *Continue*. The import starts, reporting which files and folders are imported.
- You might get an error or exception message during the import that tells you that you should do a restart afterwards. If so, do what you are told, and restart the OpenCms server after the last step. Be aware that you might have to restart the OpenCms server even if no such message is shown, see below for more details.
- After you have left the import dialog the wizard ends and you should find the imported module in the list of all modules.

### 4.2.1. Restarting the OpenCms server after module import

A server restart *might* be required after you imported a new (or updated) module.

Some modules contain Java archives (JARs), class files or other resources that are automatically copied to the WEB-INF/classes/ or WEB-INF/lib/ folder of the OpenCms web application during the module import. Such modules sometimes require a restart of the OpenCms Servlet container so that the Java Classloader can load these new classes or resources.

The individual module documentation should contain a note if a module requires a server restart after installation. You will **not** always get an exception message during import if the module requires a server restart.

In case you upload several modules, one server restart is usually enough after uploading all modules.

## 4.3. How to create, administrate, replace and export a module

For a detailed description about the

- creation of a new module
- administration of an existing module
- replacement of an existing module with a new version (i.e. updating)
- distribution of a module with the export function

please import the OpenCms interactive documentation module

`com.alkacon.documentation.documentation-modules`.

See chapter 1 on how and where to obtain this module.



# 5. OpenCms User Manual

## 5.1. Introduction

### 5.1.1. What is OpenCms?

OpenCms 5 is a Content Management System that is based on Open Source Software. Complex Intranet and Internet websites can be quickly and cost-effectively created, maintained and managed.

OpenCms enables you to create complete websites offline that are published when you're satisfied with the results. An offline project enables several users with different permissions to work on the offline version as a team. OpenCms enables you to easily coordinate users that are writing, designing or managing content. It also enables you to manage the project's workflows.

Once it has been completed and approved, the offline version is published by the project manager. Subsequent modifications and maintenance activities are performed offline. The site is updated by replacing the online files with the files that were modified offline.

### 5.1.2. The advantages of OpenCms

OpenCms is platform independent. The software used to create the website is installed on a web server and is accessed via the Internet from any location such as your home. Users use their web Browser to access OpenCms via the Internet. The combination of their user name and password ensures that their work environment remains protected. All of the web server's software components are based on Java technology.

### 5.1.3. Configuring the client

Netscape Navigator or Microsoft Internet Explorer must be installed on the client. You should use IE 5.5 or higher or Netscape Navigator 7.0 or higher. Configuration details for each of the Browsers can be found in chapter 3.

We divided the user's guide, which explains how to use the system's most important functions, into two sections: Section 1 consists of a demo that is intended to give you a practical

introduction to OpenCms; Section 2 describes the individual functions of OpenCms more detailed.

**NOTE:** Our explanations are based on the assumption that OpenCms has been successfully installed and is running on your web server or local computer.

## 5.2. Section 1: OpenCms - The demo

The following demo is based on a real life scenario and was designed to get you up and running as quickly as possible. The module `org.opencms.welcome` must be present in your OpenCms system (it is installed per default by the OpenCms setup).

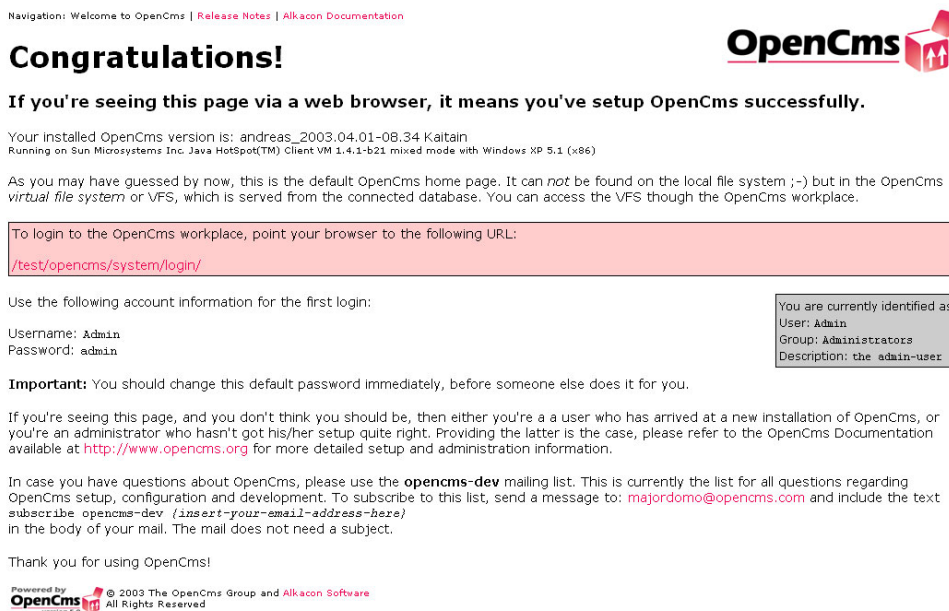
Open your Internet browser and enter

```
http://localhost:8080/opencms/opencms/
```

to access the OpenCms welcome site.

The OpenCms site welcome page is displayed (Figure 5.1). The navigator on top contains the links to all of the pages. Click on "Release notes".

The release notes page is displayed. The first section of the demo consists of modifying this page in OpenCms by following the instructions described below.



Navigation: Welcome to OpenCms | [Release Notes](#) | [Alkacon Documentation](#)

# Congratulations!

**If you're seeing this page via a web browser, it means you've setup OpenCms successfully.**

Your installed OpenCms version is: andreas\_2003.04.01-08.34 Kaitain  
Running on Sun Microsystems Inc. Java HotSpot(TM) Client VM 1.4.1-b21 mixed mode with Windows XP 5.1 (x86)

As you may have guessed by now, this is the default OpenCms home page. It can *not* be found on the local file system ;- ) but in the OpenCms *virtual file system* or VFS, which is served from the connected database. You can access the VFS through the OpenCms workplace.

To login to the OpenCms workplace, point your browser to the following URL:

```
/test/opencms/system/login/
```

Use the following account information for the first login:

Username: `admin`  
Password: `admin`

**Important:** You should change this default password immediately, before someone else does it for you.

If you're seeing this page, and you don't think you should be, then either you're a user who has arrived at a new installation of OpenCms, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the OpenCms Documentation available at <http://www.opencms.org> for more detailed setup and administration information.

In case you have questions about OpenCms, please use the **opencms-dev** mailing list. This is currently the list for all questions regarding OpenCms setup, configuration and development. To subscribe to this list, send a message to: [majordomo@opencms.com](mailto:majordomo@opencms.com) and include the text `subscribe opencms-dev` (*insert-your-email-address-here*) in the body of your mail. The mail does not need a subject.

Thank you for using OpenCms!


Powered by  © 2003 The OpenCms Group and Alkacon Software  
All Rights Reserved

Figure 5.1.: Welcome page

### 5.2.1. Editing a website

Activate the "Workplace" to edit a page in OpenCms. The "Workplace" is the OpenCms user interface that is used to change, delete and create new pages. The "Workplace" is entirely based on HTML, which is why you don't have to install additional software to run it from your Internet browser.

Enter the following address in your browser to start the "Workplace:"

```
http://localhost:8080/opencms/opencms/system/login/
```

## Logging into the system

In the login dialog box (Figure 5.2), enter "Admin" as the user name and "admin" as the password.



Figure 5.2.: "Login" dialog box

Logging in automatically opens a new browser window and closes the old one (you will be prompted to confirm this action). You are now in the "Workplace," in the so-called "Explorer" view (Figure 5.3). This is where the actual editing starts.

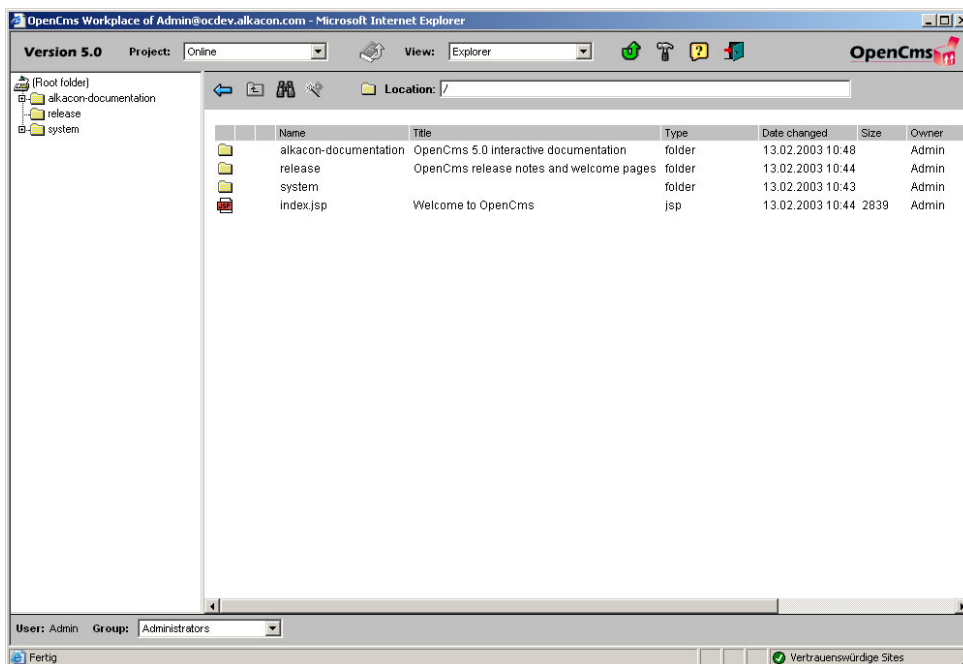


Figure 5.3.: Explorer view

**NOTE:** OpenCms is a project based application, which means that you cannot modify the pages online. You first have to create an editing (offline) project. The default "Offline" project can also be used for editing pages.

## Creating a new project

Switch to the administration window by clicking on “View” on the menu bar and selecting “Administration” from the drop-down menu (Figure 5.4).

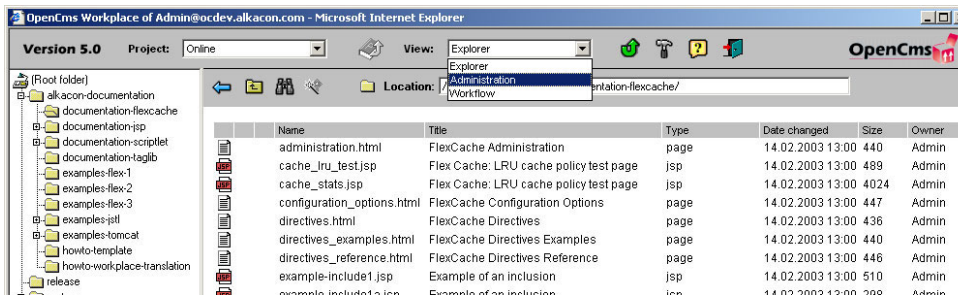


Figure 5.4.: Drop down menu “View”

The Administration view is opened. The following buttons are displayed: (Figure 5.5):

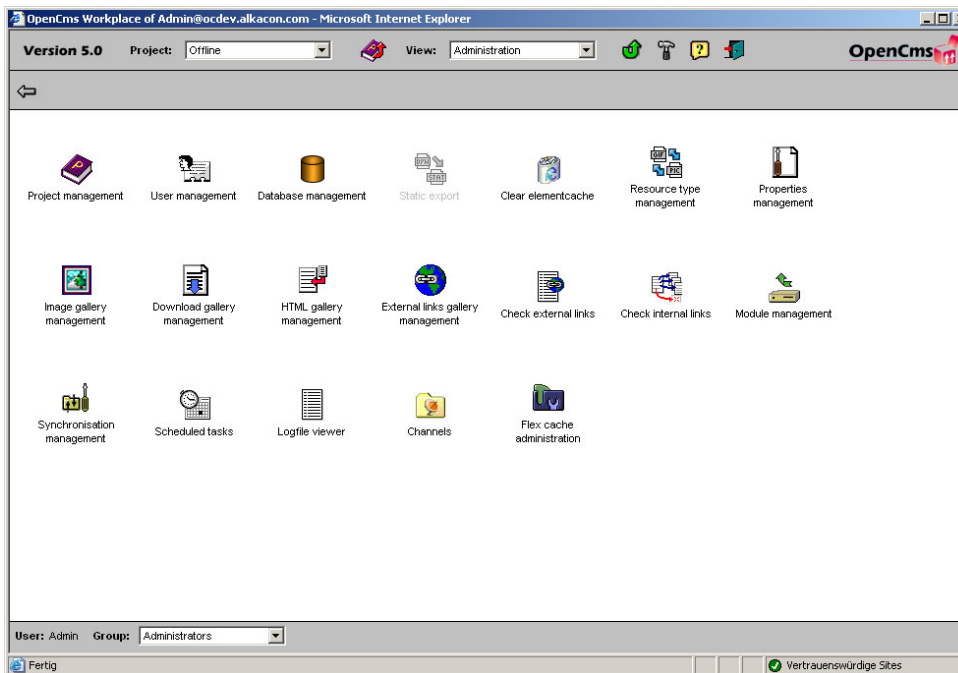


Figure 5.5.: “Administration” view

Click on “Project Management.” A new page is opened that contains the following buttons (Figure 5.6):

Click on “New Project” to create a new project.

The dialog window “Create a new project” is opened (Figure 5.7). Enter the project data:

- Project name: Nice try

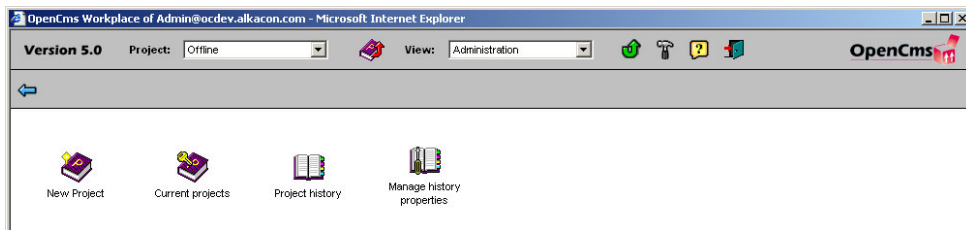


Figure 5.6.: Project Management buttons

- Description: Create test project
- Folders: /release/
- Channels: [leave empty]
- User group: Users
- Manager group: Project manager

Click on the "OK" button.

**NOTE:** Clicking on the folder icon next to the "Directories" field opens a second window that contains a list of the existing directories. Here you can select other directories for your new project. The window closes automatically.

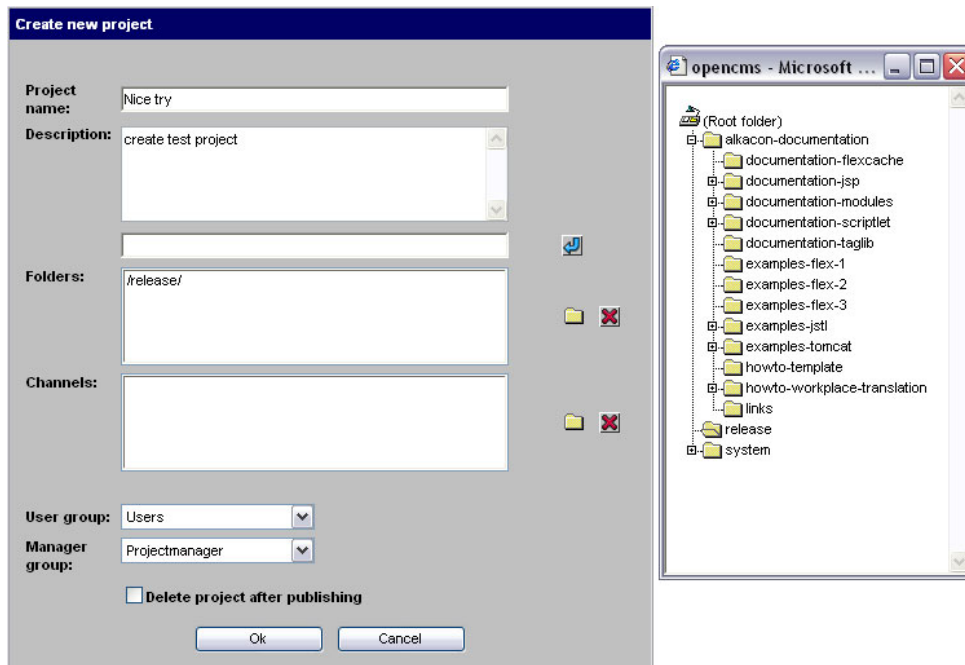


Figure 5.7.: Dialog window "Create a new project"



Switch back to the Explorer view by selecting it from the “View” drop-down menu. Select the “/release/” directory (the same one you selected when creating your project) from the drop-down menu. The Explorer view is opened. Open your “Nice try” project by selecting it from the Project drop-down menu on the menu bar (Figure 5.8).

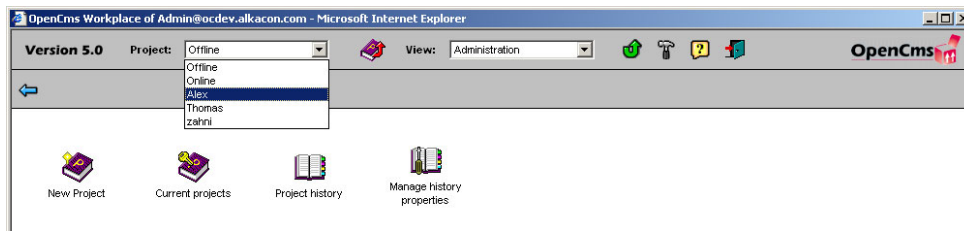




Figure 5.8.: ”Project” view

The view for your project is open. You can now edit any one of its pages.

**NOTE:** You have surely noticed that all of the folders in the folder tree are inactive except for “/release/”, “/system/galleries/download/”, “/system/galleries/pics/”, “/system/galleries/externallinks/”, “/system/galleries/htmlgalleries/” and “/system/bodies/”. Once the project is selected, OpenCms only allows you to modify files in your project directories.

## Editing pages

A page must be locked for all other users before it can be opened. Click on the icon next to the text ”installation.html” with the left mouse button to display the context-sensitive menu (Figure 5.9). Click on the menu item ”Lock.”

An open lock is displayed next to your locked file. Should the  icon not appear, check your browser settings (see above) or refresh the view with the button .

Click on the icon next to the text ”installation.html” with the left mouse button again to display the context-sensitive menu. Click on ”Edit page” to open the HTML editor.

## Working with the HTML editor

You are now in the OpenCms HTML editor (Figure 5.10). Here you will find a number of editing icons that should be familiar from other applications, such as Open Office or Word.

In the text editor, which is just below the icons, the text that you saw at the release notes page is displayed. Make changes to the text and its format (e.g. bold, italics, underline).

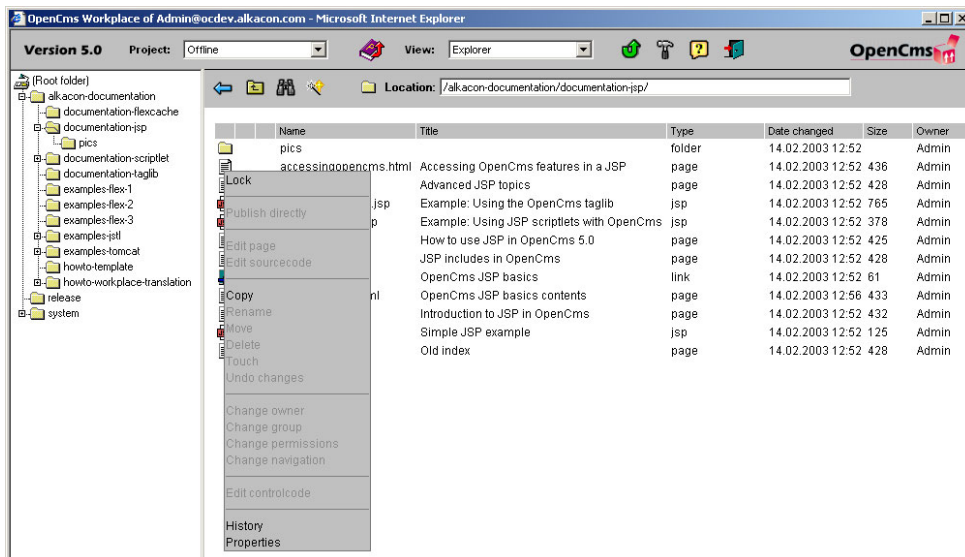


Figure 5.9.: Context-sensitive menu

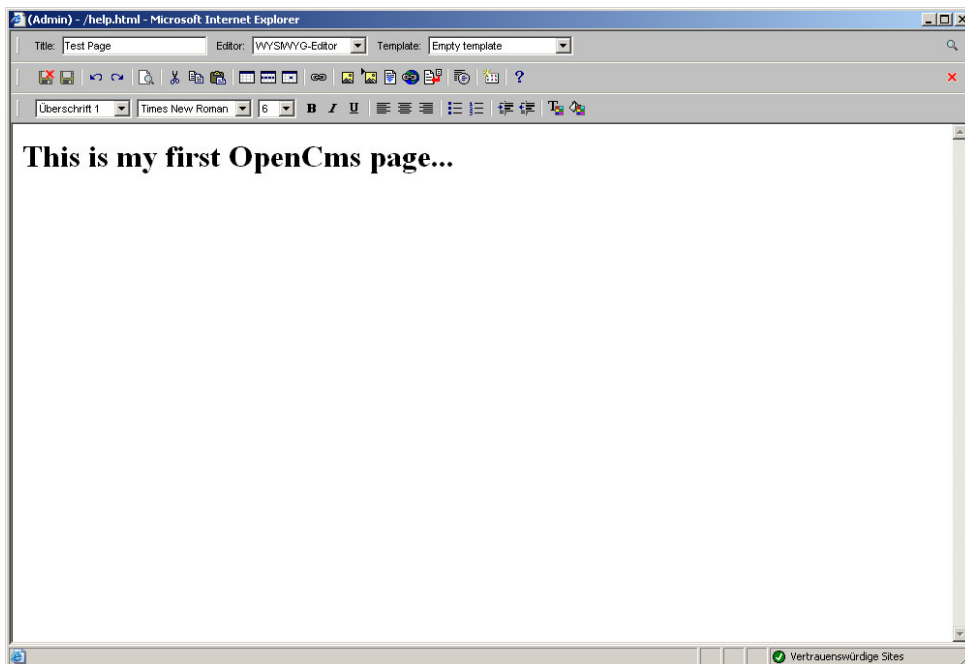


Figure 5.10.: OpenCms HTML editor

**By the way:** The HTML editor view you are currently in is a WYSIWYG view (“**W**hat you see is what you get”); this means that pages are displayed here exactly as they will be seen later on the Internet.

The “Template” drop-down menu is on the right part of the menu bar (Figure 5.11). Select one of the pre-defined page structures and layouts. You will see the changes by clicking

on the "preview" button in the upper right corner of the editor.

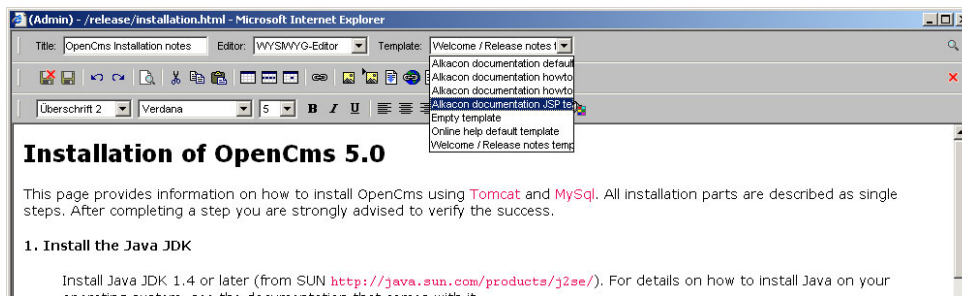



Figure 5.11.: Drop-down menu "Template"

In the upper menu bar is the "Editor" drop-down menu, which contains a selection of Editor views. You are currently in the WYSIWYG view. Select the "Sourcecode editor" view. The view is opened and displays the same content as source code. Close the view by clicking on the button "Save & Exit" , which is at the far left of the menu bar.

The Explorer view is displayed.

## Completing the editing phase

The page's details are currently displayed in red, which means that the page has been edited. Additionally there is a little red flag, which means that this page was locked and changed in the current project. Click on the page icon of "installation.html" with the left mouse to "unlock" it. Click on the menu item "Unlock" in the context-sensitive menu to unlock the page. The lock disappears (Figure 5.12).

## Publishing a project

Open the "Administration" view. Click on the "Project management" button and after that on "Current projects." The project you just edited is displayed in the overview. Click on the project's icon to access the context-sensitive menu. Select the menu item "Publish" (Figure 5.13). Click on "OK."

**Note:** You can also publish the current project by clicking on the icon between the project selector and the view selector on top of the OpenCms workplace.

Start the demo application again:

<http://localhost:8080/opencms/opencms/release/installation.html>.

The test page is displayed with your modifications (Figure 5.14).

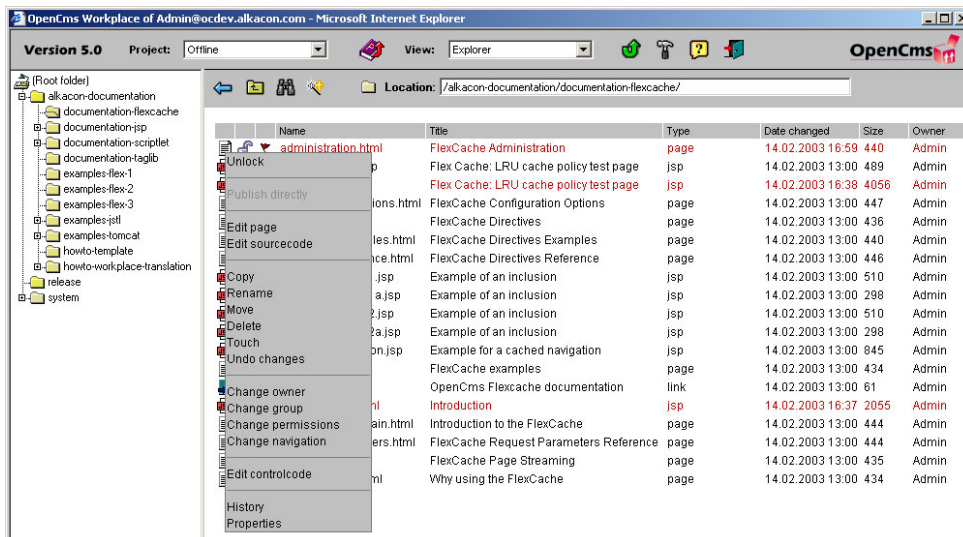


Figure 5.12.: Unlock an edited page

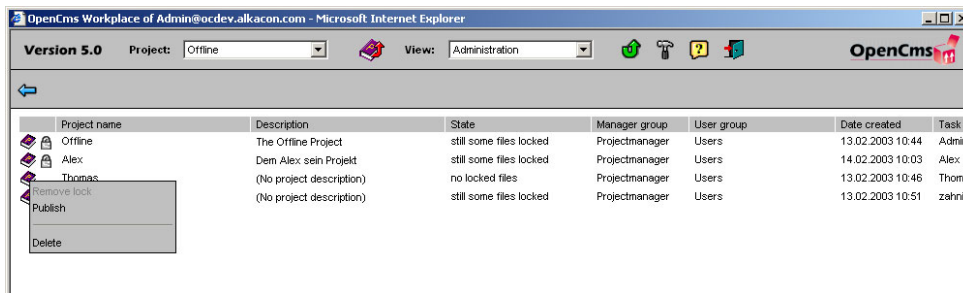


Figure 5.13.: Publishing a project

## 5.2.2. Creating a new page

You can add new pages and folders for your web page from the Explorer view. Create a new project or select the “Nice try” project by following the steps described above. Go to the “/release/” directory by clicking on the appropriate folder in the directory tree with the left mouse button.

Click on the “new” icon . The dialog window “New” is opened (Figure 5.15).

Click on the radio button “Page.”

Click on the “Continue” button. Enter the following data in the dialog window “Create a new page” (Figure 5.16):

- Name: help
- Title: Test page

Navigation: [5.0 final release notes](#) | [5.0 rc 1 notes](#) | [5.0 rc 2 notes](#) | [5.0 beta 2 notes](#) | [Installation](#) | [Welcome to OpenCms](#)

## My first modification



### Installation of OpenCms 5.0

This page provides information on how to install OpenCms using [Tomcat](#) and [MySql](#). All installation parts are described as single steps. After completing each step you are strongly advised to verify the success.

#### 1. Install the Java JDK

Install Java JDK 1.4 or later (from SUN <http://java.sun.com/products/j2se/>). For details on how to install Java on your operating system, see the documentation that comes with it.

**Important** : This version of OpenCms was tested with Java 1.4 only. Some features regarding file encoding were used that are not available with Java releases before 1.4.

#### 2. Install Tomcat

OpenCms 5.0 requires a Servlet 2.3 / JSP 1.2 standards compliant container. Tomcat 4 is the reference implementation of this Standard. This release was tested with Tomcat 4.0.x and Tomcat 4.1.x. Older versions of Tomcat (3.x and earlier) do not support this newer standard and are thus not usable for OpenCms 5.0.

Install Tomcat from <http://jakarta.apache.org/tomcat/index.html> into a folder of your choice. This is the CATALINA\_HOME folder. Don't forget to set the environment variables CATALINA\_HOME and JAVA\_HOME.

Test the installation by running Tomcat in standalone mode and check the examples. Note: Tomcat uses port 8080 in standalone mode. If you wish, you can combine the servlet-engine with a webserver like the Apache Web Server <http://www.apache.org/httpd.html>. Please see the documentation available with the webserver on how to combine it with your servlet environment.

**Important**: To make sure Tomcat works with the correct charset to read and write files you should add the following parameter to the commandline that is used to start Tomcat: `-Dfile.encoding=ISO-8859-1`. You can set the environment variable CATALINA\_OPTS to do so (`CATALINA_OPTS=-Dfile.encoding=ISO-8859-1`). The default encoding of OpenCms is `ISO-8859-1`, but you can also set another encoding supported by your Java VM, e.g. `UTF-8`. In case you do not care about the encoding, the default is most likely what you need.

#### 3. Install MySQL

Figure 5.14.: Modified installation page

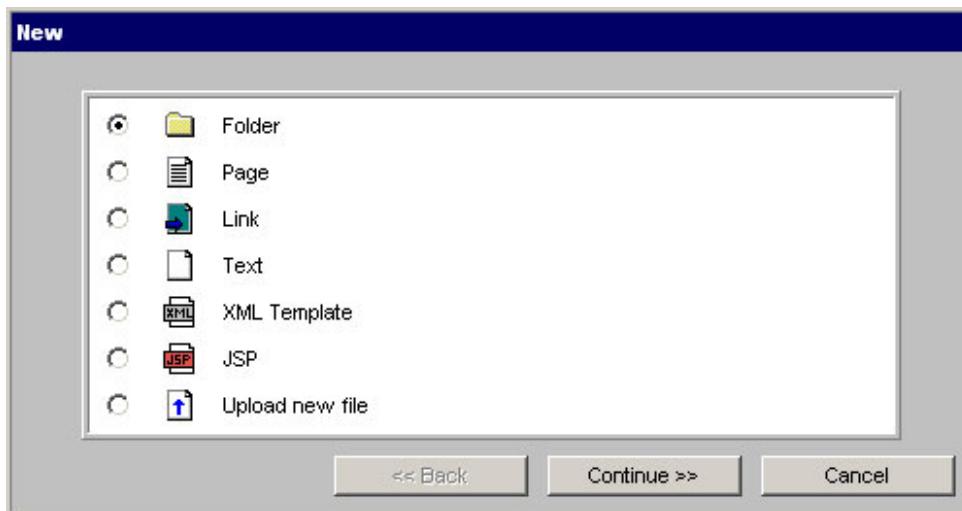


Figure 5.15.: Dialog window “New”

- Template: Welcome / Release notes template
- Keywords: Here you can enter some keywords
- Description: Here you can enter a description
- (Checkbox) add to navigation: check (default)
- Text in Navigation: Help

- Insert after: at the last position

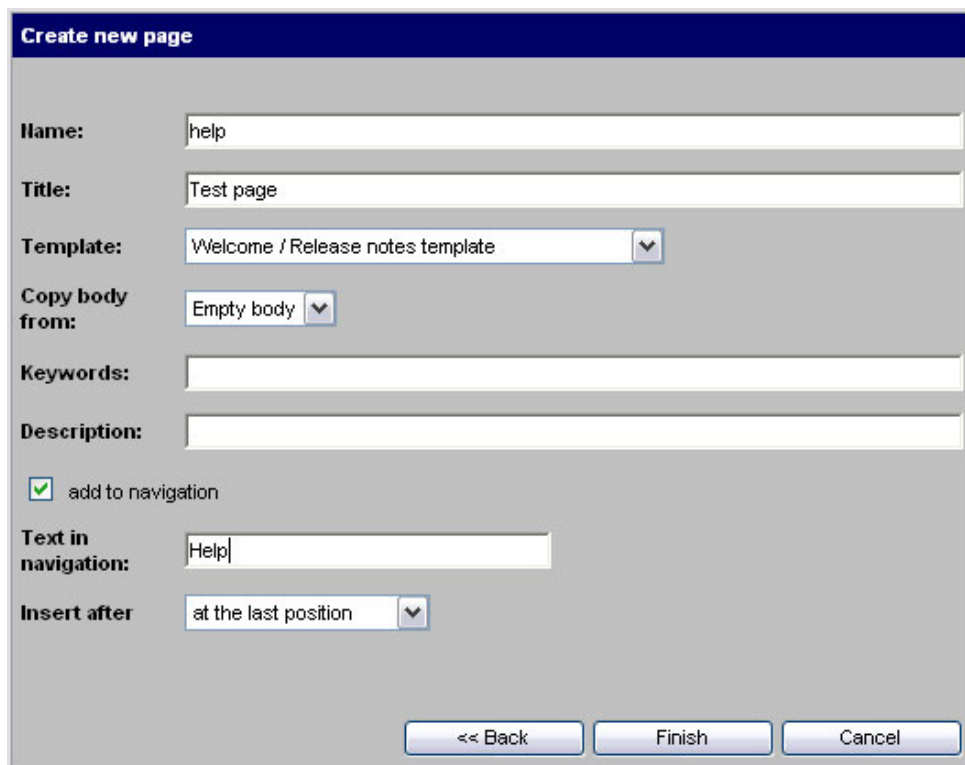
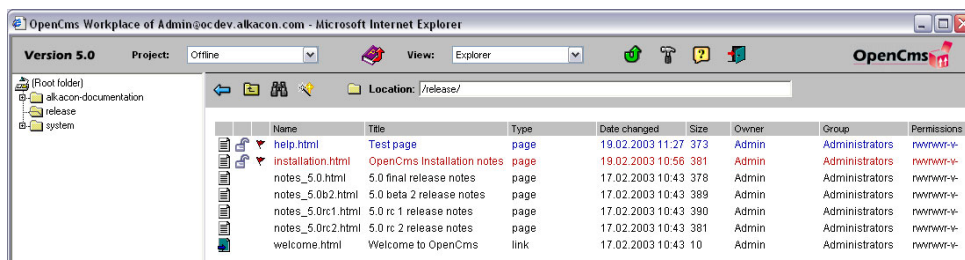


Figure 5.16.: Dialog window “Create a new page”

**NOTE:** The “Text in navigation” is the name that will later be displayed as the link in the online navigator.

Click on “Finish”. This takes you back to the Explorer view and displays the details of your new page in blue with a lock (Figure 5.17). You can either continue editing the page (see above) or use the context-sensitive menu to unlock it for other users.



Name	Title	Type	Date changed	Size	Owner	Group	Permissions
help.html	Test page	page	19.02.2003 11:27	373	Admin	Administrators	rwxrwxr-x
installation.html	OpenCms Installation notes	page	19.02.2003 10:56	381	Admin	Administrators	rwxrwxr-x
notes_5_0.html	5.0 final release notes	page	17.02.2003 10:43	378	Admin	Administrators	rwxrwxr-x
notes_5_0b2.html	5.0 beta 2 release notes	page	17.02.2003 10:43	389	Admin	Administrators	rwxrwxr-x
notes_5_0rc1.html	5.0 rc 1 release notes	page	17.02.2003 10:43	390	Admin	Administrators	rwxrwxr-x
notes_5_0rc2.html	5.0 rc 2 release notes	page	17.02.2003 10:43	381	Admin	Administrators	rwxrwxr-x
welcome.html	Welcome to OpenCms	link	17.02.2003 10:43	10	Admin	Administrators	rwxrwxr-x

Figure 5.17.: Explorer view with new page

Publish the project as explained under “Publishing a project” and open the installation page:

<http://localhost:8080/opencms/opencms/release/installation.html>

The installation page is displayed with the new link to your page “help.html” in the top navigation. Click on the link to view your new page (Figure 5.18).



Figure 5.18.: New help page listed in navigation

This is the end of our short demo. Thank you for your attention. Additional and more detailed information about OpenCms is described in section 2 of the guide.

## 5.3. Section 2: OpenCms - The mechanics

### 5.3.1. The user interface

The user interface is customized by clicking on the “Settings” tab (the “hammer” button



on the menu bar):

- The additional file information that is to be displayed in the Explorer view is defined on the “Explorer” tab.
- The Standard view and messaging options for task management are defined on the “Tasks” tab.
- The user’s default startup settings and permissions for new files are defined on the “Startup Options” tab.
- A user’s group and password are changed on the “User Data” tab.

### 5.3.2. View Modes

The user screen contains all of the components, grouped in views, that enable users to create, maintain and manage web pages. In the “Explorer” view, users manage directories and files and edit HTML pages. A project’s workflow is managed in the “Tasks” view. Users, groups, projects, database connections, images, documents etc. are managed in the “Administration” view. The user screen (default startup view) is started via a login dialog and is displayed as a file explorer. The directory tree structure, the file view and the menu bar make it easy for users to access files and functions (Figure 5.19) :

Users can switch between the online version of the web page and the individual offline projects by selecting one of the options in the “Project” drop-down menu.

Users can switch between the “Explorer”, “Tasks” and “Administration” views by selecting one of the options in the “View” drop-down menu.

The files and directories are managed via a context-sensitive menu that is activated by clicking on the file or directory icon in the file overview.

The content of the website as it will be seen online is displayed in the browser screen (Figure 5.20). The website’s files and directories cannot be modified in the online version. The browser screen, which displays the page as it would be seen by someone visiting the live site, is activated by clicking on the file name in the “Explorer” view:

- The Template of an OpenCms-generated browser page contains the static and dynamic navigation components as well as other dynamic components such as logos or text fields that can be freely defined. The template determines the page’s layout.



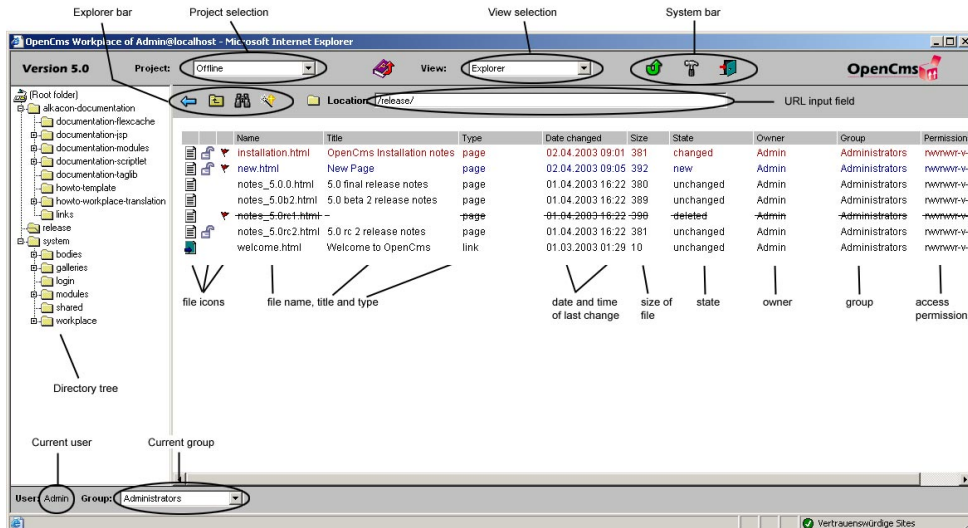


Figure 5.19.: User screen

- The content is displayed in the Body. Content is freely defined. Its layout is based on previously defined format templates. The format templates ensure that the content is uniformly displayed.

### 5.3.3. Working on a project

An offline project is created by a user that belongs to the "Project manager" group. The project can be created for an existing website that needs updating or a brand new website.

When you create an offline project for an existing website a view on the offline resources for this website is created. When the modified pages are published, the changes are copied to the online project. Additionally the changed resources are stored in backup tables for traceability purposes.

By default, an offline project's files and directories can only be modified by the project members. Responsibilities are clearly defined by assigning specific users and groups to specific files and directories. It is also possible to process part of the web site in different projects at the same time. Locking and unlocking functions ensure that access security remains intact for larger user groups.

The modifications that are made to the files are written to a backup table after publishing. The information about all versions can be accessed via the "History" option in the context-sensitive menu (Figure 5.21).

To create new content, the user creates a new file or directory by clicking on the "New" button on the menu bar. This opens a dialog window in which the user selects the element to create: "Folder," "Page," "Link", "Text", "XMLTemplate", "JSP" or "Upload new file."

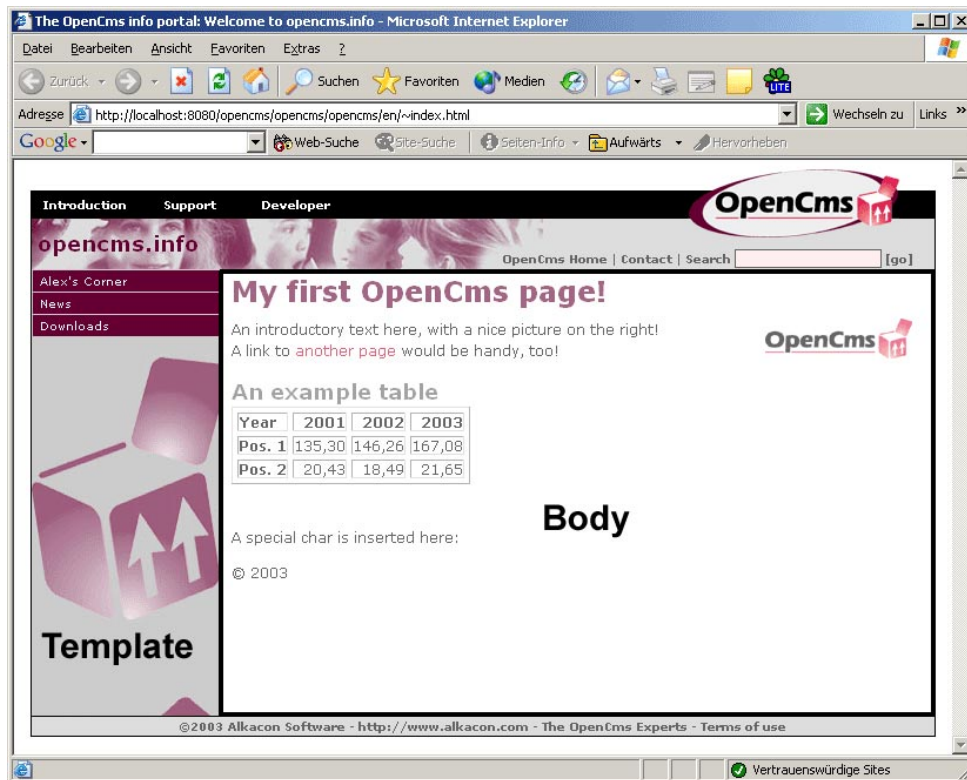


Figure 5.20.: Browser screen

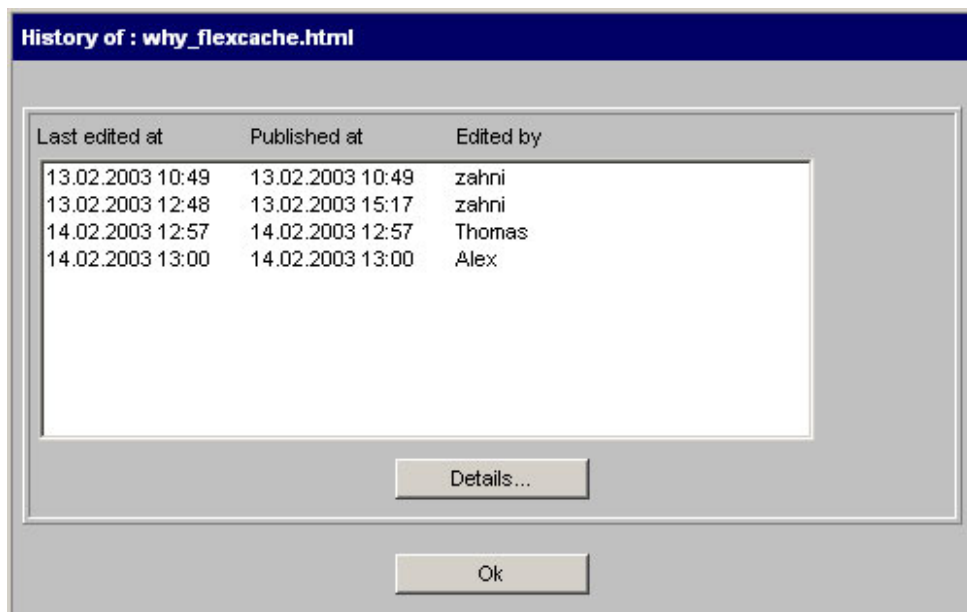


Figure 5.21.: Project history

The name, title and navigation text as well as the position of the file or directory in the

navigation, are defined in separate dialog boxes. Selecting the option “Upload new file” enables users to load files that are stored on their PC’s local hard disks into an OpenCms directory. Users access their assigned directories and files via the file “Explorer” view of OpenCms.

Clicking on the directory name (e.g. “OpenCms”) in the directory tree structure or in the “Explorer” view displays the directory’s contents. Clicking on an HTML file name (e.g. “index.html”) opens a preview of the file in the Internet browser. Clicking on a directory or file icon (text file, image etc.) in the file overview displays a context-sensitive menu. The menu contains the following functions that are accessible (active/inactive) based on the file’s or directory’s status and the user’s access permissions:

- **Lock or Unlock:** Directories or files can be locked throughout the editing phase to prevent them from being accessed or modified by several users at the same time. A file or directory is unlocked when the user has finished modifying it.
- **Publish directly:** A single resource can be published directly. This item is enabled when a changed file is unlocked.
- **Edit page:** Clicking opens a WYSIWYG editor in which the file can be modified. At this stage, no HTML knowledge is required to modify a page.
- **Edit sourcecode:** Clicking opens a text editor in which the file can be modified. This requires at least basic HTML knowledge to modify a page.
- **Edit control code:** Clicking opens a text editor in which control codes can be edited. **Warning:** Extensive knowledge of the OpenCms template mechanism is required to make modifications of this type.
- **History:** The history file displays all of the file’s previous versions. The history file enables users to see who made which modifications when. When the file is locked a button for restoring a version is shown in the “Detail” of the version.
- The respective access attributes for the individual files and directories are set on the system side by using the functions “**Change owner**”, “**Change group**” and “**Change permissions**”.
- **Change navigation:** Here the navigation text and position of a resource can be changed.
- **Properties:** To every resource properties can be attached as key / value pairs, e.g. the title or the description is stored as a resource property. Properties can be added, modified or deleted.
- Standard functions such as “**Copy**”, “**Rename**”, “**Move**” and “**Delete**” can be selected from the context menus. “**Touch**” changes the timestamp of a resource, which marks it as changed, and it will be published when the project is published.

- **Undelete resources:** When a resource that already exists in the online project is deleted it is marked by crossing out its entry in the file list. Now it is possible to undelete these resources. When undeleting a folder all subresources in this folder that were marked as deleted are undeleted, too. You must have write access to undelete a resource.

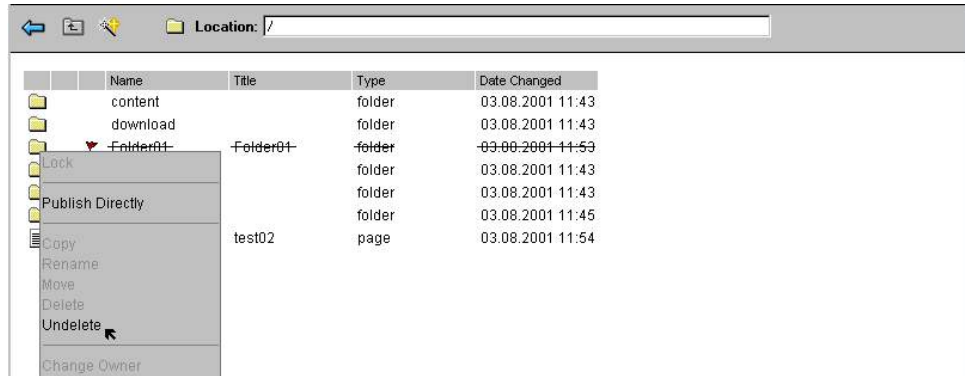


Figure 5.22.: Undelete a resource

After the resource is undeleted its state is set to changed and the resource is locked by the current user.

- **Undo changes**

Changes of a resource can be undone. The resource must be marked as changed and it must be locked. You must have write access to the resource. When the changes of a folder are undone all changes of subresources are undone, too. This feature copies the information from the online project, so all changes that were not published yet are lost and new resources are deleted.

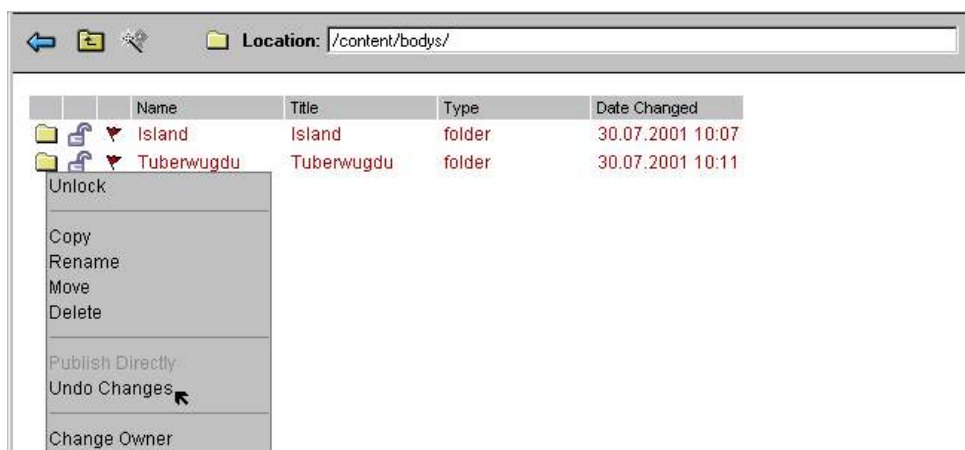


Figure 5.23.: Undo changes of a resource

- **Restore a version from history**

Older versions of a file can be restored from the history. The file must be locked. This feature currently works only for files, not for folders. The function is only enabled for locked files. It is implemented in the detail view of the history.

You have to choose the version from the list of versions and click on the detail button. To restore the version click on the “Restore version”-button. You must have write access to the file if you want to restore a version.

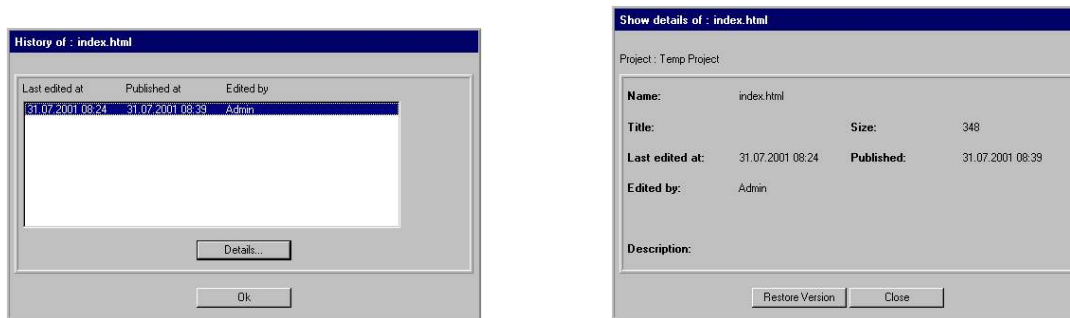


Figure 5.24.: Restore a version

- **Publish a resource**

Resources can be published directly. They must be unlocked to enable the function in the context menu. Only members of the projectmanager and the administrator groups are allowed to publish resources of a project.

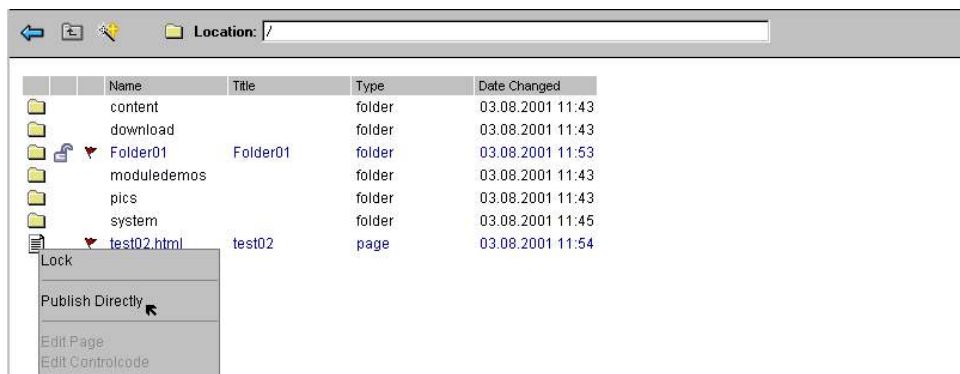


Figure 5.25.: Publish a resource

When a resource is published directly

- a temporary project is created
- the resource is copied to the new project
- the projectid of the resource and all its subresources is set to the new project

– the new project is published and deleted.

- **Copy a resource to the project**

A resource that does not belong to the current offline project is colored grey. You can copy this resource to the current offline project with this new feature in the context menu. Only the projectmanager and the administrator are allowed to copy resources to the project.

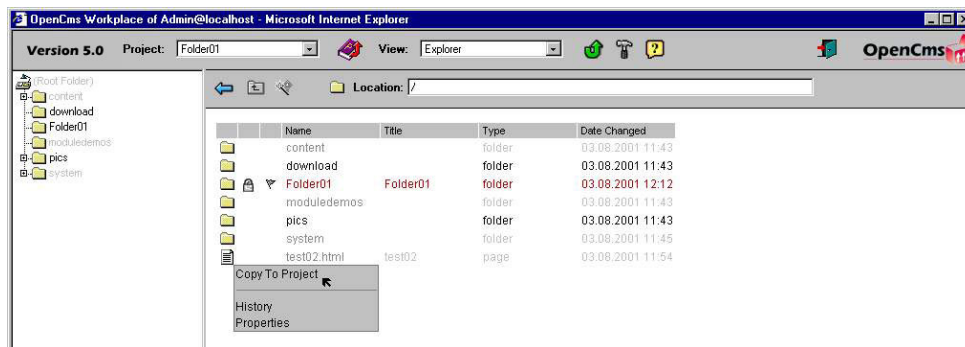


Figure 5.26.: Copy a resource to project

You can also copy the parent resource of an already existing resource to the project, but it is not possible to copy the root folder to an existing project. For this you must create a new project.

### 5.3.4. Access permissions

On the user side, different access permissions determine which actions users can perform and which components they see, i.e. users see only those files and directories that are relevant to them. Doing this provides users with a well structured overview. Each user belongs to at least one user group, but can belong to several (Figure 5.27).

The following groups are pre-defined in OpenCms:

- **Administrators:** An administrator has full access permissions to all of the files and directories. Administrators create and manage users and user groups.
- **Projectmanager:** A project manager creates new projects and coordinates their workflow. A project manager's access permissions are restricted to the projects he/she creates. A project manager can, however, access all of the files and directories within his/her projects. A project manager can also publish projects, i.e. put them online.

- **Users:** A user can create new files and directories within the project he/she is assigned to. As the owner, the user has full access permissions to the files and directories he/she creates.
- **Guest:** This group consists of all other users, such as visitors.

Based on the requirements, additional groups such as editors, designers, testers etc. can be created.

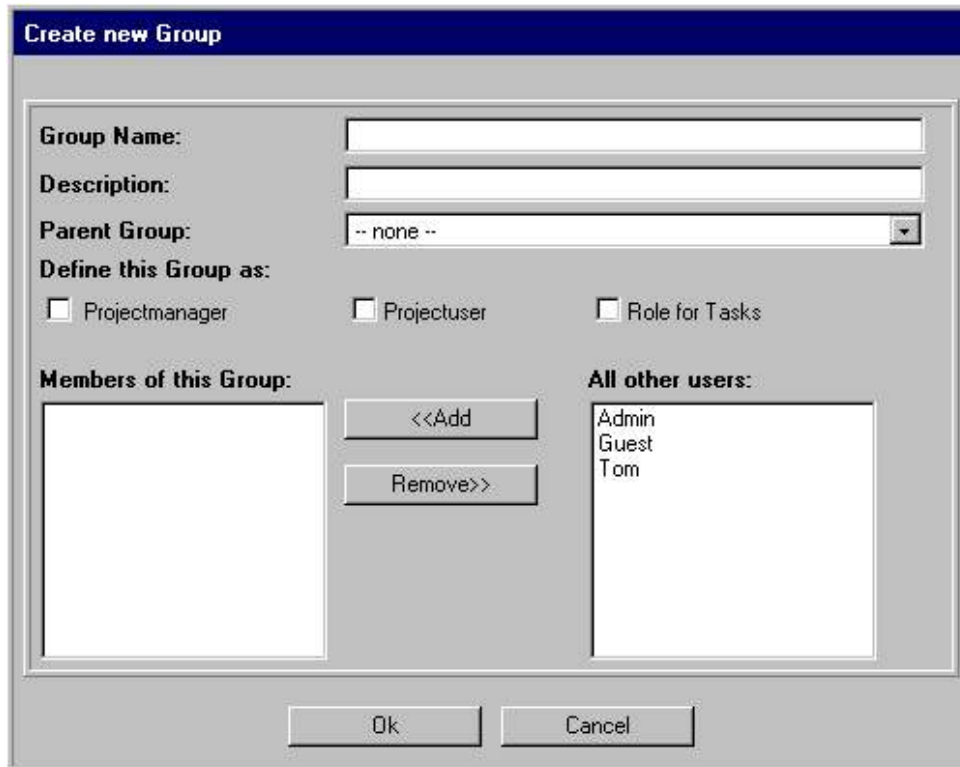


Figure 5.27.: Create a new group

On the system side, each file and each directory is assigned to one user, which by default, is the user that created the file or directory, making him/her its owner. Each file and each directory is also assigned to one user group that has specific access permissions. This enables the users in this group to access the files and directories that are relevant to them. File and/or directory attributes, such as the owner of a file or directory, or the group assigned to the files, can only be modified by the administrator and/or the file and/or directory owner. All access permissions are maintained via the “Change owner”, “Change group” and “Change access permissions” options in the context menu. File and directory access permissions are classified by “Owner permissions”, “Group permissions” and “Other permissions.” A user’s access permission can be restricted to the files and directories that he/she owns. The following abbreviations are used to set a user’s permissions: r = read, w = write, v = visible, i = internal.

### 5.3.5. The editor

The editor is used to create content in the body, which is structured by a template that ensures that the layout for the navigator, the content and the individual pages - basically the layout for the whole site - is uniform (Figure 5.28):

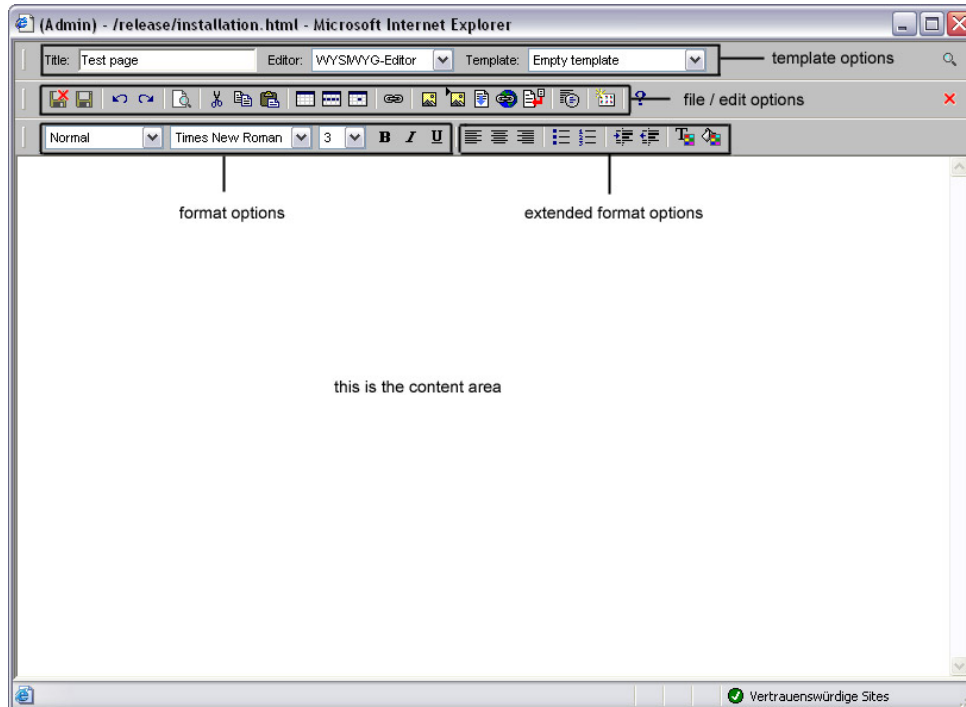


Figure 5.28.: The HTML editor

The functionality of the OpenCms editor is very similar to that of a standard WYSIWYG HTML editor with templates, tables and preview functions, such as Netscape Composer. It is also possible to switch between sourcecode and WYSIWYG mode. A page's layout is determined by a template. Complex layouts are created by cascading several templates. The editor in OpenCms is only used to edit the body of an HTML page.

### 5.3.6. Workflow

The project manager creates a new task and defines its role by clicking on the "New" button in the "Workflow" view. A role consists of several users that have the skills to perform a specific task such as editing, designing graphics, writing HTML code, etc. A preferred user is selected for each task. The name, the definition, the due date and the priority of the task as well as various messaging options are also defined. Depending on the selected messaging options, an e-mail is automatically sent to either the preferred user or all users that are assigned to a role as soon as a new task has been added to the project. The task is active once the appropriate user has accepted it. Optionally, the



project manager receives an e-mail when the task has been accepted, rejected, forwarded or completed. Users can view the tasks that have been assigned to them as well as those they have forwarded to others by filtering on specific criteria in the drop-down menu “Filter” (Figure 5.29). Different icons and colors are also used to define a project’s status. A task that is still within the due date is shown in black. If the due date is exceeded, the task is shown in red. A completed task is shown in gray. A task’s priority is defined by icons (low, normal and high). The task descriptions provide a clear overview of a user’s task account.

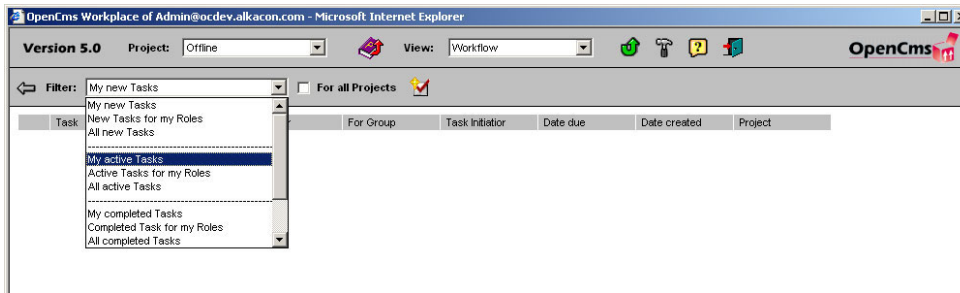


Figure 5.29.: ”workflow” view

Clicking on the name of a new task displays its details and history. Each stage of the project is tracked and ensures that the workflow remains transparent.



## 6. Components used by OpenCms

This chapter describes all external parts required to run OpenCms. If you want to prepare an installation, please collect all components listed here. If you download a major release of OpenCms (e.g. version 5.x) the MySQL connector, Xerces, JavaMail, Fesi, Jakarta-ORO and JTidy will be shipped with the distribution.

### 6.1. Operating System

OpenCms is implemented in 100% Java. Since Java should be platform independent, OpenCms should run on all operating systems supporting Java. Currently, we have installations running on Red Hat Linux 7.2 (<http://www.redhat.com>). Other Linux or Unix brands should work without problems. We also have installations on Windows NT 4.0, Windows 2000 and Windows XP.

### 6.2. Webserver

The reference webserver for OpenCms is, of course, the excellent Apache webserver (<http://www.apache.org>). Apache is the clear market leader with more than 50% market share, it is used by IBM as webserver for their operating systems, and it is also Open Source and free of charge. We have been using OpenCms with Apache since version 1.3.6 on Linux and Windows. For Windows systems, the Microsoft IIS (<http://www.microsoft.com/iis>) has also been successfully tested with OpenCms. Other web servers should also be working without problems.

### 6.3. Servlet runtime engine

The Java Servlet API (JSDK) 2.3 and a servlet engine are needed by the webserver to start up Java programs on http requests. The servlet standard has been developed by SUN, see <http://java.sun.com/products/servlet/index.html>. We are most often using TOMCAT as a Java Servlets and Java Server Pages reference implementation from the Apache Foundation. Please use Tomcat 4.x or later. See <http://jakarta.apache.org/>

For a Microsoft IIS installation, we have successfully used Jrun from Allaire (<http://www.allaire.com/Products/Jrun/>).

Since servlets are a standard API which is part of the J2EE specification, other servlet engines should be working without problems.

## 6.4. Java VM

To run servlets, a Java Virtual Machine (VM) is needed. For development we are currently using a version 1.4 compliant JDK, like the one provided by SUN (<http://java.sun.com/j2se/1.4/>) or IBM (<http://www.ibm.com/java/jdk/index.html>).

For running systems we recommend using a Java VM version 1.4 or higher.

## 6.5. Database

The OpenCms architecture allows to use any SQL capable database that offers a JDBC connector. These include enterprise databases like e.g. Oracle (<http://www.oracle.com>), but also Open Source databases like MySQL (<http://www.mysql.com>). The reference installation uses MySQL. For development we are currently using version 3.23 or version 4.0 of MySQL and version 2.0.14 of the MySQL Connector/J driver.

## 6.6. XML Parser

OpenCms makes heavy use of XML to store content and template data. For XML parsing, we make use of the excellent Xerces developed by the Apache Foundation (<http://xml.apache.org/xerces-j/index.html>). We recommend using version 1.4.4.

## 6.7. Mail API

The integrated task management is able to send mails when task events occur. To enable this feature you will need the JavaMail 1.2 API available at <http://www.javasoft.com/products/javamail/index.html> and the activation Java-Bean (<http://www.javasoft.com/beans/glasgow/jaf.html>).

## 6.8. FESI (a Free EcmaScript Interpreter)

OpenCms uses FESI as Interpreter for EcmaScript. This interpreter is used by OpenCms if you start the cmsshell in ExmaScript mode. (<http://home.worldcom.ch/jmlugrin/fesi>)

## 6.9. JTidy

OpenCms uses JTidy release 7 to cleanup and parse HTML-fragments produced by the HTML-Control. (<http://sourceforge.net/projects/jtidy>)

## 6.10. Jakarta-ORO

OpenCms uses regular expressions for linkreplacements in case of static export. Therefor jakarta-oro-2.0.6.jar is used. (<http://jakarta.apache.org/oro/index.html>)



# 7. Useful resources

## 7.1. OpenCms online resources

- Use the OpenCms mailinglist archive to search the `opencms-dev` mailing list archives. In case you have trouble setting up OpenCms, or in case you think you have found a bug, make sure you check the archive first to see if your issue has already been addressed. The archive is available at:  
<http://www.opencms.org/opencms/en/development/maillinglist-archive.html>
- Please use our OpenCms bugzilla to report bugs that you have found in OpenCms. Before doing so, please check if the bug that you have found is certainly a new bug:  
<http://www.opencms.org/bugzilla/>
- The CVS web interface allows you to browse through the OpenCms sources:  
<http://www.opencms.org/cvs>

## 7.2. How you can help

### **Please report bugs**

You might find that something does not work as it should. If so, you should provide a bug report. Please use the OpenCms Bugzilla for all of your bug reports. You might have to create a Bugzilla account first. Please use the Bugzilla if possible and not the "opencms-dev" mailing list to report bugs. If you encounter setup related problems, please post them on the mailing list first, because in 99.5% of all cases setup issues are not bugs in OpenCms but related to your local environment configuration.

### **Test out the new functionality**

The easiest way to participate in the development process would be to help testing the JSP integration, the interactive documentation and other new functionality. Please use the `opencms-dev` mailing list for discussions on the development process, about this documentation or about OpenCms in general. Note: This mailing list requires subscription prior to posting (see below). Please use the "opencms-dev" mailing list for questions and comments regarding OpenCms or the documentation and not Bugzilla.

### Provide examples and demos

Another way to help extending OpenCms is to provide examples and demos for the use of JSP pages in OpenCms. If you contribute these, we might include these in this documentation in a later release. Or we could make your examples available as a separate module download. The best way to distribute your new demo or example would be to post it to the `opencms-dev` mailing list or to `contributions@opencms.org`.

## 7.3. How to subscribe to the `opencms-dev` mailing list

The main place for the latest OpenCms related information is the mailing list `opencms-dev@opencms.com`. This list is for all general and development related news, messages and questions regarding OpenCms. Traffic on the list is currently about 10 messages a day (as of February 2003).

To subscribe to the list, go to the URL:

`http://www.opencms.org/opencms/en/development/maillinglist.html`

and just enter your email address and a password in the subscription form. You have the possibility to change your subscription details later (set options like digest and delivery modes, get a reminder of your password, or unsubscribe from `opencms-dev`) with your email address and chosen password.

Before posting a question to the list, you should check out the `opencms-dev` mailing list archive if this question has already been answered.



## **Part II.**

### **Archived documentation**



# 8. About the archived documentation

## 8.1. Introduction to the archived chapters

We are currently in the process to shift the book documentation into interactive documentation modules. During this process, the content of the book documentation is generally revised and updated. Please refer to chapter 1 for further details on this process.

Not all sections of the book documentation are already available as interactive documentation modules. This “archived documentation” part contains chapters that we think should be made interactive modules in the future.

OpenCms 5.0 supports more than one template mechanism, and we recommend to use standard JSP templates instead of proprietary XML templates. However, most of the examples in the archived part are still only dealing with XML templates. Thus it would make little sense to transform the documentation “as is” into an interactive module, because it has to be thoroughly revised. Moreover, the interactive documentation already available contains a large set of alternative examples, so just translating and updating everything here seems to be of little value.

Nevertheless the chapter contents of the archived documentation have been revised for OpenCms 5.0, so for example most system path references should be up to date now, and we also fixed typos and updated screen shots.

## 8.2. How you can help

In case you are interested to help in the process of transforming the archived chapters of the documentation into interactive modules, please send an email to *contributions@opencms.org*.



# 9. Enterprise JavaBean Integration

## 9.1. Advantages of OpenCms & EJB

Running OpenCms in an application server environment provides facilities for making use of distributed object architectures, particularly with regard to Enterprise JavaBeans technology. Using these techniques, processes behind the web site may be structured and distributed in a component oriented way. Server sided presentation logic and business logic can be developed strictly separated, according to the four-tier architecture described in the J2EE Application Model: OpenCms takes care of the presentation of the data, using the integrated template engine and master templates for displaying it in a general layout, while the generation of the content data is relocated to the EJB (figure 9.1).

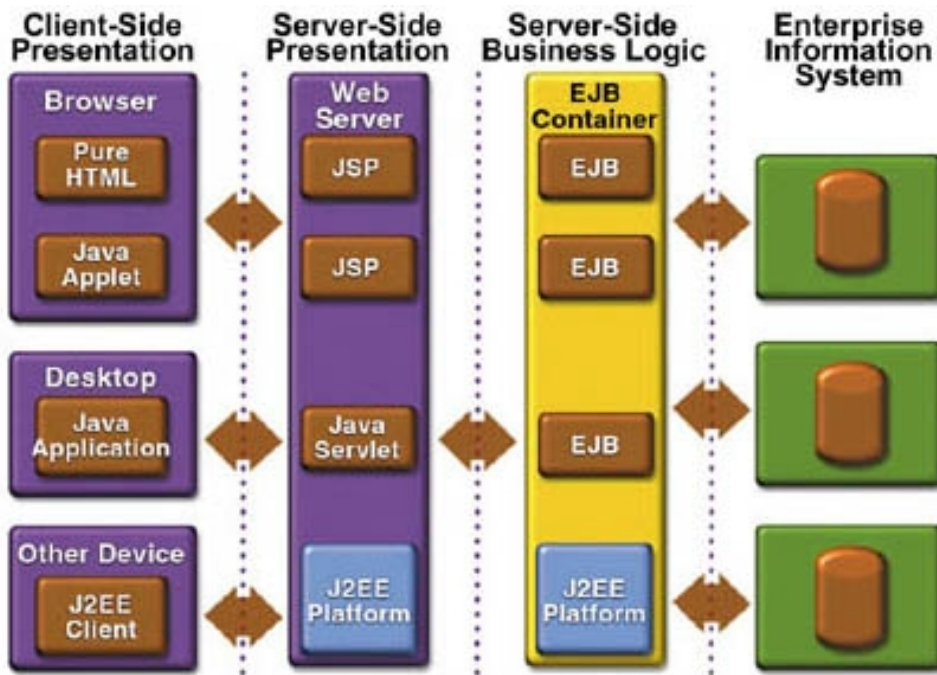


Figure 9.1.: OpenCms and EJB.

Implementing EJB components implicates a lot of advantages that enhance the ease of handling with business data. This includes:

- **Concurrency:** The EJB server handles any concurrencies between EJB. The developer doesn't have to care about threads or synchronization.
- **Transactions:** All set of tasks that have to be executed together will be managed by the EJB server
- **Persistence:** The EJB server ensures, that any object is actually containing the most recent data from any external database.
- **Distributed Objects:** Clients do not have to know the exact location of Beans it is using. This is transparent to the client. The EJB server takes care of all communication and remote method invocations (RMI).
- **Naming:** The EJB server provides a service for looking up distributed objects. This enables the client to request remote objects.
- **Security:** Authentication, access control and secure communications are ensured by the server.

In the environment of OpenCms, EJBs may be used for different purposes, such as plausibility checks of user input, database access or connection to host systems. Even reuse of existing shared libraries is possible by "*wrapping*" them with EJB components.

## 9.2. EJB basics

There are two different types of Enterprise Beans: Entity and Session Beans. Entity Beans model real-life objects usually represented by some records in a database. Session beans are used to map processes or tasks. They usually operate on the data modelled by Entity Beans. There is a further distinction between stateful Session beans, if the Bean stores data over more than one session and stateless Session beans, if the Bean only operates with the current parameters. To implement an Enterprise Java Bean at least three components are required:

- **Remote interface:** This interface contains all business methods that may be called remotely (i.e. all methods visible from OpenCms). It has to extend EJBObject.
- **Home interface:** This one keeps all methods for finding, deleting or creating new EJB objects. It must contain the create() Method for building new objects and extends EJBHome.
- **Bean class:** This class implements all methods defined in the two interfaces above. Note! Though the methods match the ones defined in the interfaces, these are not implemented in this class. For defining a entity bean, an additional component is required:

- **Primary key:** This is a class implementing a pointer to the database.

Since OpenCms only calls, but not implements EJB modules, only the two interfaces become important at this point: They are essential for looking up and calling an EJB. Two sample home and remote interfaces may look as follows (these examples are taken from the WebLogic *"examples" package*):

```
public interface TraderHome extends EJBHome {
    Trader create() throws CreateException, RemoteException;
}
public interface Trader extends EJBObject {
    public TradeResult buy (String stockSymbol, int shares)
        throws RemoteException;
    public TradeResult sell (String stockSymbol, int shares)
        throws RemoteException;
}
```

### 9.3. Calling an EJB

Calling an Enterprise JavaBean from OpenCms is quite easy and only requires four steps:

1. Get the *"initial context"*. Every Bean has a initial context object defining the Bean's environment containing the URL for the naming service, the user and his password.
2. Get the EJB home. This is the implementation of the home interface and can be used to get the EJB object.
3. Get the EJB object
4. Call any required EJB method on this object.

In the Java code, this may look like:

```
import javax.naming.Context;
import javax.naming.InitialContext;
Context jndiContext = getInitialContext();
TraderHome home = (TraderHome)jndiContext.lookup(C_JNDI_TRADER);
Trader t = home.create();
TradeResult res = t.buy("FFN", 100);
```

Instead of the `buy()` method every other method defined in the remote interface could be called. For getting the `initialContext`, some additional steps must be executed. Note, that the name of the context factory is application server dependend. This example again shows the code for the WebLogic server:

```
protected Context getInitialContext() throws Exception {
    // Get an InitialContext
    String factory = "weblogic.jndi.WLInitialContextFactory";
    String url = "t3://localhost:7001";
    Properties h = new Properties();
    h.put(Context.INITIAL_CONTEXT_FACTORY, factory);
    h.put(Context.PROVIDER_URL, url);
    // the following lines may be omitted,
    // if no user authentication is required
    h.put(Context.SECURITY_PRINCIPAL, "system");
    h.put(Context.SECURITY_CREDENTIALS, "weblogic");
    return new InitialContext(h);
}
```

In Order to compile an OpenCms template class including EJB calls, some additional import statements are required for making the EJB components known to the class:

- **javax.naming.\***. This will be needed to lookup an EJB component using the `initialContext`. This package can be found in Sun's Java 2 SDK, Enterprise Edition (J2EE).
- **Home interface** for getting the EJB object
- **Remote interface**, since the EJB object will implement this.

## 9.4. Refinements

### 9.4.1. Configuration

It might be useful not defining the application server URL, user name and password hard-coded in the Java class. Instead of this, these values should be in an external properties file. You can use the `opencms.properties` for this, which can be accessed using the method `getConfigurations` on the `cms` object.

Alternatively, a class can use it's own property file residing in the server's file system (similar to the `opencms.properties`). The class

```
source.org.apache.java.util.ExtendedProperties
```

can be used for easily accessing this file.



### 9.4.2. Error handling

It is highly recommended to add an error and exception handling to an EJB call. This avoids any server configuration error or EJB method exception from being reflected to the calling OpenCms class. Otherwise each non-caught exception inside the EJB component will probably cause a strange result on the generated web page or even crash the calling template class (the end user will see an "Internal Server Error" in his browser). Debugging will be very hard in this case because often the bug cannot be located in the EJB immediately. Therefore the exception handling should include a debugging output in the OpenCms log file by using the static `OpenCms.log()` method.

The error handling should consider the following cases:

- Application server not reachable
- Wrong user/password
- EJB component not found (wrong JNDI name)
- Method on EJB object not found (probably using wrong remote interface)
- EJB-Method throws an exception

In most of these cases, further generation of the current page will be impossible since important data expected by the EJB is missing. Then, a redirect to an error page can be initiated to inform the user, that the server could not be processed.



## 10. The Idea of Content Definitions

When writing a template class that produces the output for a dynamically created element (e.g. a news element) there is always the problem where to get the actual content from. For the news example, let's consider that the news content is taken from a database table.

The straight forward way to access this data would be to implement the JDBC-Database connection in the template class, but this approach has several disadvantages:

- If several template classes have to access the news database, the DB access code has to be written several times.
- A change in the DB structure would require changes in several Java classes.
- If the news content has to be read from a different kind of database (or maybe the VFS), the DB access in several template classes has to be changed.

To avoid these problems, the idea is to encapsulate the content and the access to it in a separate class, the so called Content Definition.

The Content Definition is an object that contains all fields of the content itself (in the news example, this would be headline, author, teaser and the article). This object provides a general interface for manipulating the data fields of an entry. It has *get()* and *set()* methods to access the data in the Content Definition object and a *write()* and *delete()* method to handle the data in the data source. The Content Definition object is used by the template classes to access the required data in a general way. Therefore, the data source can be modified without changing the template classes at all. In addition, the Content Definition is also used to maintain the content data with an appropriate Backoffice module (see chapter 11 about Backoffice modules).

The way the Content Definition encapsulates the data is very similar to the way a Java Bean is defined. Because of this, the Content Definition can be written as a Java Bean as well, enabling its use in other components of the system.

In a certain way, the CmsObject can be seen as the Content Definition of the OpenCms itself, it contains methods to access the data of the system, independent of the means how the data is actually stored.

## 10.1. Writing a simple Content Definition

Because of the nature of a Content Definition, there are several different ways how to implement it, especially the access to the data source always depends on the kind of data source and cannot be generalized. But the interface that is visible for the programmer using this Content Definition should be the same for different kind of data and data sources. This section will describe the purpose and signature of the methods to access the data encapsulated in the Content Definition. This description is the proposal how to implement a Content Definition. There are other possibilities how to write a Content Definition, but it is highly recommended to follow the proposals of this document.

A Content Definition object always contains the data of one single entry of the content type, for the news example already mentioned, this would be one single news entry, either representing a row in a database table or a file in the VFS. A news entry could be defined containing the following content values:

- ID
- Title
- Text
- Author

The simple Content Definition provides five ways to access the data:

- Constructors to read a data entry from the data source or to add a new entry to it.
- Set and get Methods to access the single data fields.
- A *write()* method to update an entry in the data source.
- A *delete()* method to delete an entry in the data source.
- Static methods that return groups of Content Definition objects.

To get a specified news entry, a new Content Definition object has to be created. This is done by a constructor that takes an argument that can be used to non-ambiguously select an entry from the data source. In case of a database record this is usually the primary key of the table (in most cases an integer value).

Note: if the Content Definition should be used to manipulate the data in the OpenCms Backoffice (from the OpenCms workplace) two constructors have to exist. One that takes the *CmsObject* as a parameter and another one that take the *CmsObject* and an Integer object as parameters. For this reason it is necessary to have a numeric value as a primary key in your database table (see section 10.3 for details about adding OpenCms Backoffice functionality).

```
/**
 * NewsContentDefinition constructor.
 * @param id The id of the news entry to be read.
 */
public NewsContentDefinition(CmsObject cms, Integer id) {
    // read data from data source and
    // initialize the member variables here
}
```

Creating a new entry is done with a constructor as well. In the next example, the constructor gets all the data of a news entry and creates a new Content Definition object with this data:

```
/**
 * NewsContentDefinition constructor.
 * @param title The title of the news entry.
 * @param text The text of the news entry.
 * @param author The author of the news entry.
 */
public NewsContentDefinition(String title, String text, String author)
{
    // the object's members are initialized here
}
```

This will only create a new Content Definition object, but no entry in the data source. To add the new entry in the data source, it must be written by using the *write()* method of the Content Definition. Of course the constructor `NewsContentDefinition(CmsObject cms)` can be used as well. The data can then be set afterwards by using the corresponding set-methods.

On this object, the data stored in the Content Definition can be accessed by using the different set and get methods, for example the title field of a news object could be modified or read like this:

```
/**
 * sets the title of the CD
 */
public void setTitle(String title) {
    m_title=title;
}
/**
 * gets the title of the CD
```

```
*/  
public String getTitle() {  
    return m_title;  
}
```

Set and get methods should be implemented for all data fields of the Content Definition.

When updating the content of a data field, this data is only updated in the Content Definition itself. Therefore it is necessary to update the data source as well. This is done with the *write()* method defined in the Content Definition which will update all data fields of the current Content Definition object in the data source.

The *write()* and *delete()* methods to access the data in the data source have the following signature:

```
public void write(CmsObject cms);  
public void delete(CmsObject cms);
```

Both methods take the *CmsObject* as a parameter. This object is used to access system-resources in OpenCms and will be used in this methods to check access rights or get information about the current user that can also be stored in the data source (for example the information who has last edited a record).

As stated before, a Content Definition object always stores the data of a single entry of the content type. It is often required to get a list of Content Definition objects, e.g. a list of the 10 latest news entries. This can be achieved by adding static methods to the Content Definition which return multiple Content Definition objects (preferable stored in a Vector, Hashtable or an array). The number and kind of methods that are implemented this way depends on the requirements of the stored content and the access to it.

## 10.2. Adding access control to a Content Definition

A Content Definition can provide a mechanism to control the access to the data to prevent unauthorized people from manipulating it. In the news example every news entry could save access flags to control who has the rights to read or write a news entry. To use this functionality, information about the owner, group and access flags for a news entry have to be stored together with the data for every entry. Therefore three new fields in the data source and the ContentDefinition are necessary. The OpenCms provides an abstract class *A\_CmsContentDefinition* in the `com.opencms.defaults` package where some of this functionality is already implemented. This class has the member variable *m\_accessFlags* which stores the access rights for owner group and others in an integer value. The access information can be set with the method *setAccessFlags(int accessFlags)*. The access right handling uses the bit-representation of the int-value. For example:

```

access rights: r w v   r w v   r w v   i
                1 1 1   1 1 1   0 0 0   1
            value: 1 2 4   8 16         -> 512   = 575

```

Thus, you have to set the access rights to 575 to get the above settings. Notice: for certain reasons the bit representation of the integers is just the other way round as normal!

The class has also some methods to check the access rights for a user. To check if an entry can be read or write the methods *isReadable()* and *isWritable()* are used. This is the default implementation in the abstract class *A\_CmsContentDefinition* of these methods:

```

/**
 * returns true if the CD is readable for the current user
 * @returns true
 */
public boolean isReadable() {
    return true;
}

/**
 * returns true if the CD is writeable for the current user
 * @returns true
 */
public boolean isWritable() {
    return true;
}

```

As you can see the default implementation of these methods always return true. So if you want to implement access control in your Content Definition you have to override these methods and change their default behaviour. An easy way to do this is to use the already implemented methods *hasReadAccess(CmsObject cms)* and *hasWriteAccess(CmsObject cms)* of the class *A\_CmsContentDefinition*. These methods implement a standard OpenCms access control by using the already mentioned fields for user, group and access flags of the ContentDefinition. So you just have to call these methods in the *isReadable()* and *isWritable()* methods:

```

/**
 * returns true if the CD is readable for the current user
 * @returns true
 */
public boolean isReadable() {
    return hasReadAccess(m_cms);
}

```

```
/**
 * returns true if the CD is writeable for the current user
 * @returns true
 */
public boolean isWriteable() {
    return hasWriteAccess(m_cms);
}
```

The variable `m_cms` passed to the methods is a member variable of the Content Definition containing the *CmsObject*. This object should be initialized when creating the Content Definition object with the current *CmsObject*. Because a Content Definition object is usually never reused in requests of different users it is save to store the *CmsObject* as a member of the Content Definition. The methods *isReadable()* and *isWritable()* should be used in the constructors and *write()* and *delete()* methods to check if the current user has the rights to access the resource. If you want to implement a different kind of access control you can do so by calling your own methods inside the *isReadable()* and *isWritable()* methods. If you want to check the access flags you can use the constants defined in the *I\_CmsConstants* interface. There are constants like `C_ACCESS_OWNER_READ`, `C_ACCESS_GROUP_READ`, `C_ACCESS_PUBLIC_READ` for the read flag for user, group and others. The constants for write access are named respectively. Take a look at the methods *hasReadAccess(CmsObject)* and *hasWriteAccess(CmsObject)* in the class *A\_CmsContentDefinition* to see how this constants can be used.

### 10.3. Content Definition enhancements

So far, it was shown how to write a simple Content Definition to access some encapsulated data. In most cases it is necessary to have a Backoffice module to maintain the data of this Content Definition. To simplify the creation of such a Backoffice (see the chapter 11 about Backoffice modules), the Content Definition must be enhanced to offer several methods that support the implementation of it. To do so, the Content Definition must extend the abstract class *com.opencms.defaults.A\_CmsContentDefinition* and override some of the methods defined in this class. Here is an excerpt of this class, showing how these methods are defined:

```
public abstract class A_CmsContentDefinition implements I_CmsContent{
    ...
    /**
     * gets the getXXX methods
     * @returns a Vector containing the methods
     */
}
```



```
public static Vector getFieldMethods() {
    return new Vector();
}

/**
 * gets the headlines of the columns
 * @returns a Vector with the fields
 */
public static Vector getFieldNames() {
    return new Vector();
}

/**
 * Gets the filter methods.
 * You have to override this method in your content definition.
 * @returns a Vector of FilterMethod objects containing the methods,
 * names and default parameters
 */
public static Vector getFilterMethods() {
    return new Vector();
}

/**
 * gets a unique id of a content definition object
 */
public abstract String getUniqueId();
}
```

The five methods shown in this code snippet are used to provide some data required by the Backoffice module for generating the output. Lets consider the news example: a Backoffice tool would be necessary to add, edit or delete news entries. Therefore a list of available news entries must be shown where a single entry can be selected. As there can be a huge number of entries, this list can be configured with various filters.

Since this kind of display is very common for many different kinds of content types (news, guestbooks etc.) OpenCms provides some general classes for generating Backoffice modules. Those classes depend on information given by Content Definition, like which filters are available or which labels the columns in the list should have. This information is provided by the methods that override those of the abstract class:

- *getFilterMethods()*: Returns a vector of available filters for a Content Definition. The data for each filter is stored in an instance of FilterMethod. These instances carry the data: filtername, filtermethod and default-parameters for that filter. A filter method returns a group of Content Definition objects that are a subset of all available entries, filtered by a specified criteria.

- *getFieldNames()*: This method returns the labels of the columns for the list of Content Definition entries generated by the Backoffice classes.
- *getFieldMethods()*: The methods of the Content Definition that return the values of the fields defined in the method above.
- *getUniqueId()*: Returns the unique id of the Content Definition object. This id is used to identify a single Content Definition object.
- *isLockable()*: Returns a boolean value indicating if the Content Definition object should be lockable in the Backoffice. When implementing locking the user-id of the user that has locked a resource will be stored in the Content Definition and the data source. Therefore an additional member variable and an additional field in the data source are necessary. Also a method *getLockstate()* and *setLockstate()* has to be implemented in the Content Definition to access the member variable of the Content Definition. The Backoffice rely upon the implementation of a *getLockstate()* and *setLockstate()* method when the *isLockable()* method returns true.

Despite of these methods a Content Definition that supports the OpenCms Backoffice functionality has also to provide a constructor that takes the *CmsObject* and an Integer value as parameters. For a class *NewsContentDefinition* the constructor would look like this:

```
/**
 * Constructor
 * @param id the unique id of the entry to read
 */
public NewsContentDefinition (CmsObject cms, Integer id) {
    // read data from data source and
    // initialize the member variables here
}
```

This constructor is used by the Backoffice classes to create a Content Definition object when the user edits or deletes an object. So if this constructor is missing, the Backoffice class will fail to generate a Content Definition object.

The general Backoffice classes are able to generate lists of the data provided by the Content Definitions. Since those lists are created dynamically, the used filters and data fields to be shown must be defined by the Content Definition in a general way.

The three methods *getFieldNames()*, *getFieldMethods()* and *getFilterMethods()* are used by the Backoffice classes to generate the lists of entries. Now we will take a look how an implementation of these methods can look like. The method *getFieldNames()* returns a Vector of Strings that contains the labels of the column headings of the created list. For our news example this method could be implemented this way:

```

public static Vector getFieldNames() {
    Vector columns = new Vector();
    columns.addElement("id");
    columns.addElement("title");
    columns.addElement("author");
    return columns;
}

```

This would result in a creation of a list with three columns for the id, title and author in the Backoffice.

Note: the Strings stored in the Vector are the names of the labels in the language-file of the module. So the exact text that will be shown is defined in a language-file and can easily be changed without changing the code in the class. Also you can provide language-files for multiple languages to have different output when changing to another language in the workplace (at the moment only german and english is supported by the workplace). The labels in our language-file could look like this:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<LANGUAGE>
  <mypackage_NewsBackoffice>
    <label>
      <id>ID</id>
      <title>Title</title>
      <author>Author</author>
    </label>
  </mypackage_NewsBackoffice>
  ...
</LANGUAGE>

```

The labels have to be defined inside a tag named after the Backoffice class that creates the lists (fully qualified names with underscores as separator). In this example the tag is named `<mypackage_NewsBackoffice>`. So the corresponding class would have the name *mypackage.NewsBackoffice* (see chapter 11 about how to implement Backoffice classes). The content of the columns is generated by invoking the methods that are returned by the method *getFieldMethods()*. In our news example this method could be implemented this way:

```

public static Vector getFieldMethods() {
    Vector methods = new Vector();
    Class cd = NewsContentDefinition.class;
    try {

```

```
        methods.addElement(cd.getMethod("getTitle", null));
        methods.addElement(cd.getMethod("getText", null));
        methods.addElement(cd.getMethod("getAuthor", null));
    } catch(NoSuchMethodException exc) {
        // ignore the exception
    }
    return methods;
}
```

The *getFieldMethods* method returns a *Vector* of *java.lang.reflect.Method* objects. These objects are used by the *Backoffice* class to invoke the get methods that return the values for the columns. The caught *NoSuchMethodException* will only be thrown if one of the method's names is not typed correctly.

As stated before a Content Definition should provide filter-methods that show a subset of the available data. For a Content Definition of a news-module you could have for example filters that show the entries sorted by the title or a filter that shows all entries that have an id greater than a given number. The *Backoffice* class can generate the list of entries, if it knows about these different filter-methods and what parameters they have. This information about the filter-methods is given by the method *getFilterMethods()*. This method returns a *Vector* of objects of type *CmsFilterMethod*:

```
public static Vector getFilterMethods(CmsObject cms) {
    Vector filter = new Vector();
    String titleFilter = null;
    String idFilter = null;
    try {
        CmsXmlLanguageFile lang = new CmsXmlLanguageFile(cms);
        titleFilter = lang.getLanguageValue(
            "mypackage_CmsNewsBackoffice.label.titlefilter");
        idFilter = lang.getLanguageValue(
            "mypackage_CmsNewsBackoffice.label.idfilter");
    } catch(Exception e) {
        // handle exception that might be thrown
        // when accessing the language file
    }
    filter.addElement(
        new CmsFilterMethod(
            titleFilter,
            NewsContentDefinition.class.getMethod(
```

```

        "newsByTitle",
        new Class[] {CmsObject.class}
    ),
    new Object[0]
)
);
filter.addElement(
    new CmsFilterMethod(
        idFilter,
        NewsContentDefinition.class.getMethod(
            "newsById",
            new Class[] {CmsObject.class, String.class}
        ),
        new Object[0],
        "1"
    )
);
return filter;
}

```

Inside the *getFilterMethods()* two objects of type *CmsFilterMethod* are created and added to a *Vector* which is then returned. The class *CmsFilterMethod* has two constructors:

```

public CmsFilterMethod(String filterName, Method filterMethod,
    Object[] filterParameters)

public CmsFilterMethod(String filterName, Method filterMethod,
    Object[] filterParameters, String defaultFilterParam)

```

The parameters of the constructors have the following meanings:

**filtername** - the description of the filter as it will appear in a select-box in the Backoffice.

This description can either be defined in a language-file and then read from there like in the example above, or directly coded inside the class. The first approach has the advantage that it can be changed later without having to change the Java code.

**filterMethod** - an instance of *java.lang.reflect.Method* that describes the filtermethod of the Content Definition. You have to give the name and the type of the parameters as an array of class objects to create this object.

**filterParameters** - this is an array of parameters that will be passed to the method when it is invoked by the Backoffice class. The parameter of type *CmsObject* has

to be omitted if it is the first parameter of the filtermethod, it will then be passed automatically to the method. These parameters are useful if you have a filtermethod that can be used for different filters available in the Backoffice. You can pass a parameter to control the output and so create multiple filter-entries with one method. For example you could have an filter that gives back all news entries belonging to a certain news-category. The filtermethod could have a parameter indicating which category to search for and by passing a String parameter to the method you could select the category. Normally you will pass an empty array of objects here if your filtermethod doesn't take a parameter to "customize" its output.

**defaultFilterParam** - the filtermethod can have a user-parameter that is typed in an input-field in the Backoffice and then passed to the method. This parameter is a default value for this parameter, that will be used if the Backoffice list is generated for the first time when the user couldn't have provided any value for the parameter.

**Note:** the Backoffice class checks whether the filtermethod has a user-parameter by comparing the actual number of parameters of the method as provided by the *java.lang.reflect.Method* object with the number of parameters that should be passed to the method. These are the parameters given by the parameter **filterParameters** and additional the *CmsObject* (if it is the first parameter of the method). If the method has more than these parameters, it is assumed that a user-parameter is used to provide this parameter.

The filtermethods in the Content Definition for our example would have the following signature:

```
public Vector newsByTitle (CmsObject cms) { .... }  
public Vector newsById (CmsObject cms, String id) { ... }
```

# 11. Backoffice Modules

When writing a module for OpenCms (e.g. a News Module), it is often required to edit the external data that is used to create the output with this module. This functionality is called the Backoffice of this module. A Backoffice can either be an integrated component of a module or an additional module that will use the functionality and Content Definition of an already existing module (like the News Module).

Backoffice modules are integrated into the OpenCms administration view, but it is also possible to add them as a separate view into the system. Since Backoffice modules are part of the workplace, this chapter will first show the differences between a "normal" template class and a workplace template class and then point out the special needs that are required to use the predefined Backoffice super class.

## 11.1. Workplace Classes in general

In principle, workplace template classes are written the same way as normal template classes, they contain a `getContent` method that is used to generate the output of this class which is done with the same three steps:

- Read the template file.
- Set the dynamically generated out
- Generate the HTML

Since the workplace classes need some special extensions to ease their production, the workplace classes do not extend the `CmsXmlTemplate` class, but the `CmsWorkplaceDefault` class, so that a workplace class would start like this:

```
public class CmsAdministration extends CmsWorkplaceDefault implements  
I_CmsConstants ....
```

The `CmsWorkplaceDefault` class provides some other additional predefined user methods that can be used in the templates used for creating the workplace. Those methods can be used the same way as the default methods in the normal template classes. The following methods are provided by the `CmsWorkplaceDefault` class (Table 11.1).

Method	Description
<code>commonPicsUrl()</code>	This method generates an URL for the common template pics folder.
<code>helpUrl()</code>	User method to generate an URL for a help file. The system help file path and the currently selected language will be considered. The path to the help file folder can be set in the <code>workplace.ini</code>
<code>picsUrl()</code>	User method to generate an URL for the system pics folder.
<code>userName()</code>	User method to get the name of the user.

Table 11.1.: Methods

The other difference to the normal template classes is the kind of template file to be used. Instead of using a template file of the type `CmsXmlTemplateFile`, a special workplace template file named `CmsXmlWpTemplateFile` is used in the `getContent` method. So a `getContent` method for a workplace class would start like this:

```
public byte[] getContent(CmsObject cms, String templateFile,
    String elementName, Hashtable parameters,
    String templateSelector) throws CmsException {
    CmsXmlWpTemplateFile xmlTemplateDocument =
        new CmsXmlWpTemplateFile(cms,templateFile);
    ....
}
```

This is done because the `CmsXmlWpTemplateFile` is able to handle several additional tags that are not required in normal templates but used to generate the workplace templates more easier. To use this kind of template file object, the appropriate XML template must be defined slightly different than normal. All XML templates so far were defined this way:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmltemplate>
  <template>
    ...
  </template>
</xmltemplate>
```

XML templates to be used with the `CmsXmlWpTemplateFile` do not use the `<xmltemplate>` tag, but the `<workplace>` tag. Besides this difference, those templates are written the same way than the already known normal templates



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<workplace>
  <template>
  ...
  </template>
</workplace>
```

The reason to use a different kind of templates and template objects is the enhanced functionality that is needed for writing workplace templates and classes, like multi language support and the additional tags. For further information of their use, see their usage in the existing workplace classes and templates. The only tag that is needed to explained in detail is the `<label>` tag.

To enable multilanguage support, a workplace template must not contain any text output coded directly into the template, otherwise it would not be possible to change the language for those templates. Therefore, the `<label>` tag is used to add the required information to the template. An example for using it is shown in the following code fragment:

```
...
<tr>
<td class="head">]]><LABEL value="label.group" /><![CDATA[</td>
</tr>
...
```

This tag will be replaced with an corresponding text string depending on the language used on the workplace. The actual language output is defined in a language file of the module, see the documentation of the module mechanism how to do this. In this example, the language information for this value would stored like this:

```
<label><group>Group</group></label>
```

It is highly recommended to use this mechanism to add text output to the workplace templates and not to add the text directly to the template.

## 11.2. Abstract Backoffice class

It is now possible to write a workplace template with a corresponding class that can be used as a Backoffice module for a given module - but in many cases this would be an inconvenient way. Many module - like a news module or guestbook - require a very similar Backoffice functionality. The single data objects of those models must either be edited, created or deleted. This can be done with a Backoffice tool that allows two different views:



Title	Text	Author
Titel 10	Das ist Text10	Autor 2
Titel 9	Das ist Text9	Autor 1
Titel 8	Das ist Text8	Autor 3
Titel 7	Das ist Text7	Autor 3
Titel 6	Das ist Text6	Autor 1
Titel 5	Das ist Text5	Autor 2
Titel 4	Das ist Text4	Autor 1
Titel 3	Das ist Text3	Autor 2
Titel 2	Das ist Text2	Autor 1
Titel 1	Das ist Text1	Autor 1

Figure 11.1.: List view in the Abstract Backoffice

- A list of all available data objects which can be configured with various display filters. Data objects can be deleted in this view.
- An input form to edit existing or adding new data objects.

To speed up the development of new Backoffice modules, OpenCms provides a predefined abstract class (and templates) that enables the functionality of the data list. It contains a *getContent* method to produce the output of the data list template based on the information provided by a Content Definition. This class can be easily extended and modified for each individual Backoffice module. Only the input form - which is always depending on the data fields of a module - must be implemented separately. The list of all available data objects that is generated with the predefined classes looks like figure 11.1.

To use the abstract class, it has to be extended by an implementation class that extends several configuration methods, providing the necessary data about the Content Definition to be used and the URLs of the forms required for adding, editing or deleting data. The following methods have to be overwritten in the implementation class to archive this:

- *getContentDefinitionClass()*: Returns the Content Definition class used by this implementation
- *getCreateUrl*: Returns the URL of the dialog to add a new data entry.
- *getEditUrl*: Returns the URL of the dialog to edit an existing data entry.
- *getDeleteUrl*: Returns the URL of the dialog to delete an existing data entry.
- *getSetupUrl*: Return the URL of the setup dialog

The returned URLs of those methods direct to the dialogs (i.e. templates and classes) that have to be produced in the process of creating the Backoffice Module. Those dialogs have to be adapted to the requirements of the data fields of the Content Definition which should be possible to be edited. When those dialogs are called, they require a special information, which data object should be edited or deleted. This is done by adding the URL-Parameter `?id=value` to the URL. The value given is either the unique id of the data object (which can be accessed by calling the `getUniqueId()` method of the used Content Definition) or the keyword "new" to add a new data object. In most cases the `getCreateUrl()` and `getEditUrl()` will return the same value, i.e. the same dialog will be shown when adding a new data entry or editing an existing one, the second option would require that the input fields of the form would be preset with the existing values.

A sample implementation of the News Backoffice has the following implementation of those abstract classes:

```
public class NewsBackoffice extends A_CmsBackoffice {
    public Class getContentDefinitionClass() {
        return NewsContentDefinition.class;    }
    public String getCreateUrl(CmsObject cms) {
        return "NewsEditBackoffice.html";
    }
    public String getDeleteUrl(CmsObject cms) {
        return "NewsEditBackoffice.html";
    }
    public String getEditUrl(CmsObject cms) {
        return "NewsEditBackoffice.html";
    }
}
```

Using the predefined classes, the list of data objects of a given content definition can be created without writing new template files. They are required for the create, edit and delete dialogs. Since those dialogs depend on the data to be modified, there is no general rule how to write the templates and the belonging classes. Those dialogs are made the same way than a normal form in the OpenCms (see the chapters about creating forms in OpenCms). There are some formalities to comply with when writing those forms:

- The form class must extend the class `CmsWorkplaceDefault`.
- The template file object must be an instance of `CmsXmlWpTemplateFile`.
- The form template must be of the XML type `<workplace>`.
- The information, which data object has to be edited is given as the URL-Parameter `?id=value` .

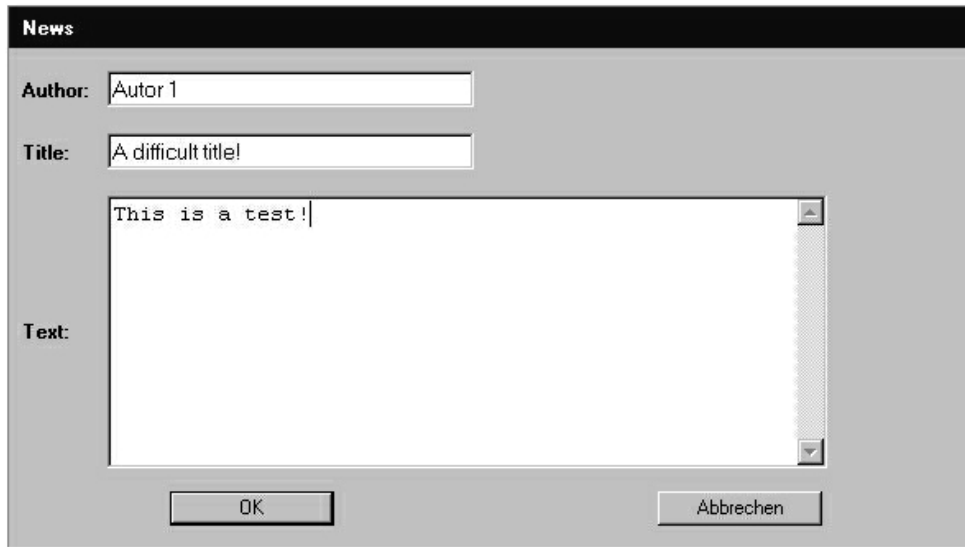


Figure 11.2.: Abstract Backoffice class Pic 2.

- The form class must use the given Content Definition to access the data object.

It is important that the template classes for those dialogs are able to check the entered data for correct inputs before storing the values in the Content Definition. For example, it should be checked if a given date is correct or an email address is formed after the correct syntax.

The dialog to edit an existing news entry could look like figure 11.2.

The Java class to process this dialog has to do the following steps: First to read all given parameters from the input fields, then to check if all data values are non empty, to create a new Content Definition object with the given data and finally to store it in the data storage.

### 11.3. Changes to the Abstract Backoffice class since OpenCms 4.4

OpenCms uses the abstract backoffice class *A\_CmsBackOffice* from the package *com.opencms.defaults* to make the writing of new backoffice modules more easy. This part

describes the changes to the existing mechanism and how these changes can be used.

### 11.3.1. Changes to the `getContent...` methods

The abstract `backoffice` class defines three methods for processing the input-form of the `backoffice`:

- `getContentEdit`: form to modify an existing dataset
- `getContentNew`: form to create a new dataset
- `getContentDelete`: form to delete a dataset

The parameters of these `getContent...` methods are now the following:

```
public String getContentEdit(CmsObject cms,  
                             CmsXmlWpTemplateFile templateFile,  
                             A_CmsContentDefinition cd,  
                             String elementName,  
                             Enumeration keys,  
                             Hashtable parameters,  
                             String templateSelector)  
    throws CmsException
```

The modifications to the existing version are highlighted.

- `A_CmsContentDefinition cd`: gives the correct instance of the Content Definition to the method. This method is either a new (empty) CD or a CD which includes an existing dataset. This way you can always access the correct CD, without fetching it with your own programm code.
- `Enumeration keys`: Inherits the names of all data which comes from the form (URL and form-paramter). It can be used to determine which parameters had been on a certain page, when working with forms over multiple pages.
- return-value `String`: The method returns a `String`. An empty `String` signals that no error has occurred while processing the form. If an error has occurred, the `String` contains the error-message which will be displayed in the template.

member variable	get-method	name of datablock	name of inputfield
m_text	public String getText()	<process>text</process>	text

Table 11.2.: name conventions

### 11.3.2. Additional method `getSetupUrl`

Now it is possible to add an additional setup page to the module, which can be called from the backoffice. This does the method

```
public String getSetupUrl(CmsObject cms, String tagcontent,
A_CmsXmlContent doc, Object userObject) throws Exception
```

This method has to be overwritten from your own class, which extends the class *A\_Cms-Backoffice*, if this function is wanted. The method returns the URL of the setup page of the module. This page can be put at any place within the module directory.

If a setup page is defined in this way, a button to access this page is shown in the list view of the module.

### 11.3.3. Filling the template from the CD automatically

The main target of the changes of abstract backoffice class is to decrease the sumpturay of creating an own backoffice class. In order to let the abstract backoffice do as many tasks as possible, some conventions concerning names of variables, methods, datablocks and fields have to be known. Table 11.2 shows an example for these naming conventions.

With the help of this naming scheme, the form is automatically filled with the right values.

The methods *getXYZ* have to return a String to enable the automatically assignment of the data to the fields. In the case that the method *getXYZ* accesses a value that is no String, another method *getXYZString* has to be written, which returns the value as a String. The returned value is then inserted into the datablock *XYZ*.

Member variables have always to be initialized, with 'null' in case of an Object, and '-1' in case of an integer.

### 11.3.4. Filling the CD from the template's fields automatically

Just like filling the datablocks before, the assignment of the data from the templateform to the CD takes place automatically. The naming scheme for this is the same as before (Table 11.3).

member variable	set-method	name of datablock	name of inputfield
m_text	public String setText()	<process>text</process>	text

Table 11.3.: name conventions

**special case: file upload**

To make it possible to read out uploaded files automatically, some further naming conventions has to be followed (Table 11.4).

This means that two values are always stored, the name of the file which was uploaded as well as the content of the file. The name of the inputfield in the template is used as a key for the name within the Content Definition. The according method name to set the content is derived from that.

**11.3.5. Testing for errors**

A method in the Content Definition tests the values which have been inserted. This method is called automatically before the data is stored to the database.

```
public void check(boolean finalcheck) throws CmsPlausibilizationException
```

All data has to be tested within this method and a CmsPlausibilizationException has to be thrown in case of an error. The exception contains a vector of error codes, which is passed to the constructor of the exception.

```
public CmsPlausibilizationException(Vector error)
```

The certain errorcodes within the vector are Strings which are build after the following scheme:

member variable	set-method	name of datablock	name of inputfield
m_upload	public void setUpload(String filename)	<process>text </process>	Upload
m_uploadcontent	public void setUploadContent(byte content)	—	—

Table 11.4.: name conventions

'fieldname\_error'

'Fieldname' is the inputfield in which the error has occurred and error is the kind of error (e.g. 'empty'). Error messages according to these error codes are searched within the backoffice. There is a mechanism with 3 stages, which serves as a naming scheme for the datablocks inside the template. The following datablocks are searched one after another and issued in case one was found. This way it is possible to have different graduations of the error messages.

- `<errfieldname_error>errorcode<errfieldname_error>`: This datablock defines an error text which is issued when the error code 'fieldname\_error' appears.
- `<errfieldname>errorcode<errfieldname>` This datablock defines an error for the inputfield 'fieldname'. It is used if the datablock '`<errfieldname_error>errorcode<errfieldname_error>`' could not be found.
- `<errerror>errorcode<errerror>`: This datablock defines an error text for a kind of error. It is used if the datablock '`<errfieldname>errorcode<errfieldname>`' could not be found.

If no error text can be found for the error code, the error code is given directly to the template.



## 12. The Mastermodule

The Mastermodule is a generic framework for the implementation of OpenCms modules. It consists of generic classes (especially the generic Content Definition class *com.opencms.defaults.master.CmsMasterContent*) and database tables that are designed to suit the needs of common modules.

In the past, when implementing your own modules the design and access to the database tables used by a module had to be designed and implemented new for every module. From now on this functionality is already implemented in the mastermodule and can be used in your own modules.

The database tables used by the mastermodule contain fields for common information like the title, the creation date, the date of last modification etc. and also fields for text, numbers, dates and binary data that can be freely used by your modules.

The mastermodule also provides functionality like the integration of Content Definition entries in the projectmechanism and the use of channels (they work similar to folders) to organize Content Definition entries. An entry can be associated with one or more channels. These channels can be used to filter the data and show only entries which belong to a certain channel.

Beside this the MasterContentDefinition provides constructors to create new empty objects or to read an entry from the database. And also the *write()* and *delete()* methods that write changes to the database or delete an entry are already implemented. In the old way of writing Content Definitions these methods had to be written new for every module.

Another feature that is already included is the access-control for Content Definition entries: each entry belongs to a user and group and also has its own access rights like other resources in the vfs. If someone has not the appropriate rights he will not be able to edit a Content Definition entry.

Here is a short summary of the improvements coming with the Mastermodule:

- generic database design with fields for text, numbers, dates and binary data
- access to database (constructors, *write()*, *delete()*) already implemented
- entries are fully integrated in the projectmechanism
- manages access-rights for the entries

- entries can be organized through channels
- simplified backoffice implementation

Also the implementation of backoffice functionality has been simplified. A new class *com.opencms.defaults.A\_CmsChannelBackoffice* provides support for the managing of channels and media files (binary data) in the backoffice. The template *extended\_backoffice* in the folder */system/workplace/templates/* can be used for creating backoffice edit templates. It includes a template for upload of media files (binary data, accessible in the Content Definition via the method *getMedia()*) and selection of channels in the backoffice. To use the backoffice list view with integration of the projectmechanism you have to override the method *isExtendedList()* to return true in your backoffice class. There are also new methods added in the backoffice class that return special urls:

- *getHistoryUrl()* returns the url for the history view
- *getPublishUrl()* returns the url for publishing
- *getUndeleteUrl()* returns the url for undeletion

To write a Content Definition class it must be inherited from the *CmsMasterContent* class and implement the following methods:

- *getSubId()* (returns a unique SUB\_ID to identify entries of a module)
- *get()* and *set()* methods (to access the module data)
- filtermethods for backoffice filters
- *getFilterMethods()* (returns the filtermethods used in the backoffice)
- *getFieldNames()* (returns the column headings of the backoffice list)
- *publishProject()* (called when the data is published)
- *moduleParameterWasUpdated()* (event-method that reads module parameters)

The access to the database is encapsulated in a class *CmsDbAccess* that resides in a subpackage of the module. You can write multiple *CmsDbAccess* classes for support of different databases. The type of the database can then be controlled by a moduleparameter (e.g. *dbType*) and an object of the appropriate *CmsDbAccess* class can be created.

The following sections will describe the purpose of the different methods and show some examples how to implement them. The methods *getFilterMethods()* and *getFieldNames()* have already been introduced in chapter 10 (page 59) about Content Definitions and can be looked up there.

The method *getSubId()* is declared as an abstract method in the class *CmsMasterContent* and should return a unique id for the module. Because all data of modules using the Mastermodule is inserted in one database table it is necessary to have a unique identifier for entries of a module.:

```
/** unique SUB-ID for this Content Definition */
protected static final int C_SUB_ID = 1;

/**
 * Returns the sub-id of this Content Definition.
 */
public int getSubId() {
    return C_SUB_ID;
}
```

The *get()* and *set()* methods provide access to the module data. The object *m\_dataSet* of type *CmsMasterDataSet* is defined in the *CmsMasterContent* class and contains the member variables that actually hold the data:

```
/**
 * Returns the name
 */
public String getName() {
    return m_dataSet.m_dataSmall[0]; }

/**
 * Sets the name
 */
public void setName(String newName) {
    m_dataSet.m_dataSmall[0] = newName;
}
```

As you can see the methods access an array *m\_dataSmall* which is contained in the *m\_dataSet* object. The Mastermodule provides String variables in the arrays *m\_dataSmall*, *m\_dataMedium* and *m\_dataBig* which are mapped to database text fields of different size and also the arrays *m\_dataDate* and *m\_dataInt* for date and int fields. Currently the Mastermodule tables includes these free usable fields:

- 40 fields DATA\_SMALL\_0 ... DATA\_SMALL\_39 for short Strings
- 10 fields DATA\_MEDIUM\_0 ... DATA\_MEDIUM\_9 for longer Strings

- 10 fields DATA\_BIG\_0 ... DATA\_BIG\_9 for extra long Strings
- 10 fields DATA\_INT\_0 ... DATA\_INT\_9 for integer values
- 5 fields DATA\_DATE\_0 ... DATA\_DATE\_4 for date values

The main decision to make when implementing the *get()* and *set()* methods is what information should be stored in which database field and under what name should it be accessible.

Filtermethods in the Content Definition class delegate the database access to the *CmsDbAccess* class of the module. The filtermethod must have a parameter of type *CmsObject* and can have additional parameters to be used in the database query. The method has to return a Vector of Content Definition objects:

```
/**
 * Filter method.
 * @param cms - the CmsObject to get access to the cms ressources.
 * @return a Vector of Content Definition.
 */
public static Vector filterAll(CmsObject cms) throws CmsException {
    return ((CmsDbAccess)getDbAccessObject(C_SUB_ID)).filterAll
        (cms, CmsContentDefinition.class, C_SUB_ID);
}
```

The actual database query will be executed in a filtermethod in the *CmsDbAccess* class:

```
public Vector filterAll(CmsObject cms, Class contentDefinitionClass, int subId)
{
try {
// get a connection from the connection pool
Connection con = getConnection(cms);

// prepare the statement filter_all
PreparedStatement stmt = sqlPrepare(cms, con, "filter_all");

// set the value for the sub_id in the statement
stmt.setInt(1, subId);

// execute the query and get the result
ResultSet res = stmt.executeQuery();

// build a Vector of Content Definition objects from the result
```

```

// boolean parameter viewonly set to true to return only entries
// which are allowed to be viewed by the current user (v-right)
return createVectorOfCd(res, contentDefinitionClass, cms, true);
} catch(SQLException exc) {
exc.printStackTrace(System.err);
throw new CmsException("Error accessing database.",
CmsException.C_SQL_ERROR, exc);
} finally {
sqlClose(con, stmt, res);
}
}
}

```

The `filterMethod` first gets a connection of the connection pool and then uses a prepared statement to execute the query. The SQL-statements are stored in a property file `query.properties` which is located in the folder of the `CmsDbAccess` class:

```

filter_all : \
    select ${column_names_master} \
    from $CMS_MODULE_MASTER \
    where sub_id = ? \
    order by title;

```

The query uses two variables to specify the columns and the name of the table. By using a variable for the column names you can be sure that you always use the correct names. Because OpenCms uses different tables for the online and offline data, the variable for the tablename will be replaced with `CMS_MODULE_ONLINE_MASTER` or `CMS_MODULE_MASTER` depending on the actual project your are in. If you would decide to use explicit table names you had to use different statements for the online and offline tables.

The mechanism of the database access in the mastermodule can be seen in figure [12.1](#).

The method `publishProject()` must be implemented to publish the module data. You can take this example implementation and only have to replace the name of the Content Definition class:

```

public static void publishProject(CmsObject cms, Boolean enableHistory,
    Integer projectId, Integer versionId, Long publishingDate,
    Vector changedResources, Vector changedModuleData)
    throws CmsException {

    // publish the ressources for this module
    CmsMasterContent.publishProject(

```

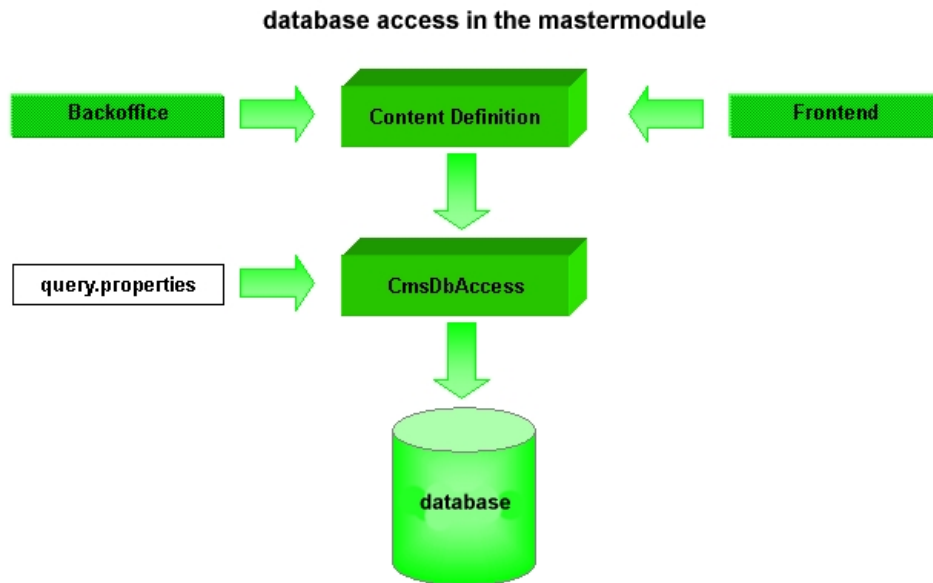


Figure 12.1.: database access in the mastermodule

```

    cms,
    enableHistory.booleanValue(),
    projectId.intValue(),
    versionId.intValue(),
    publishingDate.longValue(),
    C_SUB_ID,
    YourContentDefinition.class.getName(),
    changedResources,
    changedModuleData
  );
}

```

To make sure that this method gets called when the project is published you also have to provide the name of the Content Definition class in the *registry.xml* file of OpenCms inside the section for your module:

```

<modules>
  <module>
    <name>module.package</name>
    ...
    <publishclass>
      <name>module.package.YourContentDefinition</name>
    </publishclass>
  </module>
</modules>

```

```

    ...
    </module>
</modules>

```

The method *moduleParameterWasUpdated()* is used to read the module parameters from the OpenCms *registry.xml* file. The method is automatically invoked on the eventhandling class of the module (this should be your Content Definition class) when the parameters are changed via the module administration of OpenCms. The mastermodule uses several parameters to provide the database urls for online, offline and backup tables and the type of the database:

```

/**
 * Reads the module parameters from the registry.
 * Automatically invoked when the parameters are changed
 * in the module administration.
 * @param cms - CmsObject to access system resources.
 */
public static void moduleParameterWasUpdated(CmsObject cms)
    throws CmsException {
    // read all poolnames from the module parameters
    String connect = OpenCms.getRegistry().getModuleParameterString(
        "module.package", "onlinePool", "jdbc:opencmspool:mysqlonline");
    String connectOffline = OpenCms.getRegistry().getModuleParameterString(
        "module.package", "offlinePool", "jdbc:opencmspool:mysql");
    String connectBackup = OpenCms.getRegistry().getModuleParameterString(
        "module.package", "backupPool", "jdbc:opencmspool:mysqlbackup");
    // read the dbtype from the module properties
    String dbType = OpenCms.getRegistry().getModuleParameterString(
        "module.package", "dbType", "genericsql");
    // read the root channel from the module properties
    String rootChannel = OpenCms.getRegistry().getModuleParameterString(
        "module.package", "rootChannel", "/");

    if("genericsql".equalsIgnoreCase(dbType)){
        CmsDbAccess dbAccess =
        new CmsDbAccess(connectOffline, connect, connectBackup);
        CmsMasterContent.registerDbAccessObject(C_SUB_ID, dbAccess);
        dbAccess.setRootChannel(rootChannel);
    } else {
        throw new CmsException("No database selected");
    }
}

```

The method reads the parameters *onlinePool*, *offlinePool* and *backupPool* that define the jdbc urls for access to the online, offline and backup tables of the module. The third String parameter in the method call to *getModuleParameterString()* is a default value that will be returned by the method if the parameter is not specified in the *registry.xml* file. If you omit this parameter the method will return null if the parameter cannot be found. The parameter *dbType* is used to control which database will be used. Depending on this parameter the appropriate *CmsDbAccess* object will be created with the urls for the online, offline and backup tables. Another module parameter is the *rootChannel* parameter. It is used to define a parent channel for all channels accessible for this module. For example if the root channel is set to */news/* all subchannels of */news/* will be available in the backoffice to put a news entry in it. This way you have a separation of channels of different modules. The value of the *rootChannel* must be set in the *CmsDbAccess* object. If you don't set it, it will be set to the default value, the root folder (*"/*). So if you don't need channels in your module you can simply forget about this parameter.



# 13. Static Export

## 13.1. Introduction

OpenCms has the feature "Static Export". With this feature it is possible to export the static resources to a docroot in the server filesystem. During this export all links to this static resources and back to dynamic resources will be adjusted automatically. In this scenario it is possible that a traditional web server like Apache (<http://httpd.apache.org>) serves all static resources and OpenCms delivers all dynamic resources (like forms or personalized sites). This splits up the work to each software that can handle the specific request in a better way.

## 13.2. Setting up Static Export

The static export is already preset for a tomcat standalone installation, so you don't have to modify anything if you use tomcat in standalone-mode and can jump to the next section.

In the file `opencms.properties` (located in `webapps/opencms/WEB-INF/config/`) you can enable static export:

```
staticexport.enabled = true
```

Note: If you enable the static export your publishing cycle will slow down because of the additional work of exporting the resources to the server filesystem.

After this you have to define the export path on your server. This path should be the http-server's docroot or a subdirectory of it. If you define a relative path it is relative to your webapplication.

```
staticexport.path = export/
```

Now you have to define some `url_prefixes` so the static export can adjust all links to the correct location (http-server or OpenCms).

```
url_prefix_export = /${WEB_APP_NAME}/export  
url_prefix_http   = /${WEB_APP_NAME}/opencms
```

The string `$WEB_APP_NAME` will be replaced by the current name of the webapplication. In normal installations this would be `opencms`.

Now you can define resource(s) the static export should start with. This can be a single or multiple file(s) or folder(s). This property is only used if you start the static export manually (via admin view).

```
staticexport.start = /
```

All the other properties for static export are advanced properties. In normal installations you will never need to adjust them on your own. If you are interested in them you should have a look into the comments of the `opencms.properties` file.

### 13.3. How to use static export

To get used to the static export you should create two pages like `/index.html` and `/page1.html`. You should add some content to this pages and create a link from each page to the other with the HTML edit control. Now you can publish your project. If static export is enabled all changed resources will be exported to the server filesystem into the folder you have specified. The two links will be adjusted, so they will refer to each other even after the export. You can browse the two resources by pointing your explorer or navigator to the following location:

```
http://yourserver.com:8080/opencms/export/index.html
```

This is the static copy of the resource you can reach with:

```
http://yourserver.com:8080/opencms/opencms/index.html
```

### 13.4. What have I to do?

OpenCms needs your help to adjust all links between resources. Therefore you have to mark all links in the templates (mastertemplates, frametemplates, contenttemplates and elements) with the tag `<link>`. In this link tag you define the link to another resource within the OpenCms system (No scheme, server name, port or webapplication name is allowed). You can add url parameters to the link, if you need:

```
href=""]><link>/index.html</link><![CDATA["  
href=""]><link>/news.html?newsid=7</link><![CDATA["  
src=""]><link>/pics/logo.gif</link><![CDATA["
```

An editor can use the HTML edit control. OpenCms will take care about links added with the control. Therefore it uses the JTidy library to find all "a href" and "img src" tags and inserts the needed OpenCms link tag.

A module developer has to call the method `getLinkSubstitution` to get adjusted links:

```
String adjustedLink = cms.getLinkSubstitution(String link);
```

You need to call this method for dynamic creation of pages. E.g. a navigation element that creates a list of links dynamically should call `getLinkSubstitution` for each navigation entry.

## 13.5. How does static export work?

After the publishing to the online project is done OpenCms starts to export all static resources. It starts with all changed resource(s) in that project. After the export of this resource(s) it will follow all links defined in the link tags or substituted with the `getLinkSubstitution` method. These resources will be exported, too. This cycle will end if all resources are exported.

A next run of the static export will only export the changed resources. A resource is marked as changed, if one of the resources this resource depends on was changed. OpenCms will find the "normal" dependencies like `mastertemplate`, `frametemplate`, `contenttemplate` and `elements`. Additional dependencies like dependencies for dynamic navigation templates or displaying content definitions you have to register yourself. How to register these dependencies you have learned in the chapter about using the element cache.

Note: You can find the names of all exported resources in the `opencms.log` file. So you can check which resources are exported.

## 13.6. How to control what is exported?

Only resources that are readable as a `Guest` user are exported. You can control the export behaviour of these resources by adding properties to that resource. The property `"export"` controls if and how the resource will be exported. The following values are possible:

- `false`:  
The resource will never be touched by the export (not exported and no link adjustment)
- `dynamic`:  
The resource is dynamic. It will not be exported. All requests will be delivered by OpenCms. A link to this resource will be adjusted to a link into the OpenCms system.
- `https`:  
The resource is dynamic. A link to this resource will be adjusted to a link into the OpenCms system. Additionally the scheme will be set to `https`. You need to configure OpenCms and the web server to handle `https` requests.

- `true`, or no property:  
This is a static resource. The resource will be exported to the docroot of the web server.
- `https_enabled`:  
This is a static resource. The resource will be exported to the docroot of the web server. The links to this resource from a https page will not be set to http. Used to show pics on a https page.
- `dynamic_https_enabled`:  
Like `dynamic` but the links to this resource from a https page will not be set to http.

You can control the name of an exported resource with the property `"exportname"`. This is useful for modules which reside in the folder

```
/system/modules/YOURMODULENAME/YOURMODULEFILES.
```

If you want to reach the resource with a shorter and nicer URL you can define the `"exportname"` property. You should add it to the folder `YOURMODULENAME` and set it to a nice short path (e.g. `/YOURSHORTMODULENAME/`). After the export was done, you can reach your module with the URL:

```
/opencms/export/YOURSHORTMODULENAME/YOURMODULEFILES
```

instead of

```
/opencms/export/system/modules/YOURMODULENAME/YOURMODULEFILES.
```

Note: You can control single resources or complete folders with all sub resources by adding the property `"export"` and `"exportname"`.

## 13.7. How to export resources with parameters?

OpenCms will even export static resource with parameters:

```
/news.html?newsid=7&channelid=25 -> /news_1.html  
/news.html?newsid=44&channelid=3 -> /news_2.html  
...
```

The parameters will be substituted to a consecutive number. There is no relationship between the consecutive number and the URL parameters.

## 13.8. Advanced properties of the static export: switch standard to dynamic

As above mentioned there are properties for the static export you may wish to use as a advanced user.

The static export is controlled by the resource property export. Normaly a resource with no property set (including the parent folders) is handeld as if the property is set to true. This feature is good when most of the side is static and can be exported, but when everthing is dynamic you sure want OpenCms to export only files with the property export=true and let everything withouth the export proptery stay in OpenCms. Therefore you can set the default value for the export property to 'dynamic'.

```
staticexport.default.export=dynamic
```

Only 'true'(standard) and 'dynamic' are allowed. There are two set of linkrules defined in the opencms.properties. In the dynamic mode the second ruleset is used. This rule-set includes automatical export of some filetypees like css, js, gif, jpg,... If you want additional filetypees automaticaly exported, just add the corresponding lines to the rule-set.dynamic\_exportrules and the ruleset.dynamic\_externrules:

```
s#(*\..ppt$)#${url_prefix_export}$1#,\
s#(*\..ppt$)#$1#,\
```

Note that all files will be exported that include the '.ppt' at the end of there name.

## 13.9. Advanced properties of the static export: Configure Https

If you want to use the ssl protocol you can configure the static export to set the scheme as prefix ahead of the link if the Page has the https property and to set the http scheme ahead of links that reference from a https page to a http page. All this is done automaticaly by OpenCms. You just have to set the property "export" to https on the folder where your ssl pages are. Then you configure the prefixes in the opencms.properties:

```
url_prefix_https=https://server.de/${WEB_APP_NAME}/opencms
url_prefix_servername=http://server.de
```

The servername is used in addition to the export resp. the http prefix for links from https to http pages. The https prefix is set ahead of links to the https pages. Note that only link tags will be processed. If you write a template, don't forget to use it consequently.

## 13.10. Advanced properties of the static export: Relative Links in Export

If you just need a static site with no dynamic content you might want to use the exported pages without OpenCms. Then it could be necessary to have all links in the export relative.

```
relativelinks_in_export=true
```

This feature only works with the standard rulesets and it needs some extra performance.

## 13.11. Advanced properties of the static export: The Rulesets

In the majority of cases you don't need to modify the linkrules. First you can configure which ruleset is used for which situation. OpenCms needs four rulesets. Three for the link replacement for a link on a page that is a) shown in the online project, b) shown in the offline project and c) exported to the filesystem. The fourth rule is used for OpenCms to know under which name a exported page should be saved in the filesystem. Mostly the export and the online rules are the same.

```
linkrules.true.export=exportrules
linkrules.true.online=exportrules
linkrules.true.offline=offlinerules
linkrules.true.extern=externrules
linkrules.dynamic.export=dynamic_exportrules
linkrules.dynamic.online=dynamic_exportrules
linkrules.dynamic.offline=dynamic_offlinerules
linkrules.dynamic.extern=dynamic_externrules
```

Which definition is used depends on the value of the `staticexport.default.export` parameter: `true` or `dynamic`. The value of the linkrules must appear as the name of a ruleset in the `opencms.properties`.

A ruleset is a commaseperated list of substitution rules in perl5 standard. In General when a link tag is resolved or the `getLinkSubstitution()` method is called in a module one rule after the other is executed till a rule matches. Then the result is returned. When it comes the the `*dynamic rules*` entry the dynamic rules OpenCms has generated with the resource property `'export'`. In this case all dynamic rules are executed till one matches. If nothing matches the original is returned. You can use the `${WEB_APP_NAME}` variable for the

webapplication name and the four prefix variables defined above (`${url_prefix_export}`, `${url_prefix_http}`,...). They will be replaced before using the regular expression. In addition to this it is possible to define the place where the dynamic generated rules should be used instead of a rule use the expression `*dynamicRules*` (including the `*`'s). OpenCms replaces this with the dynamic generated rules. There are two types of dynamic rules. The first one is generated with the resourceproperty "exportname". For each resource with this property a rule will be generated that replaces the absolute path of this resource with the value of the property. It is principally used to get nice short folder names on the disc. The second kind of dynamic rules are generated with the property "export". The value can be 'dynamic' or 'https'. The system generates rules so that the links to these resources go back to the opencms system from the static content (with protocol https in the second case). There is a third value possible: 'false' means the resource is not exported and it will not be created for getting further links on it like the 'dynamic' ones. New possible values are 'true', 'https\_enabled' and 'dynamic\_https\_enabled'. True is needed for the case when default is set to dynamic. https\_enabled works like true but it will not set the http scheme ahead the link. dynamic\_https\_enabled is the same for dynamic resources.

The dynamic rules are only for export, online and extern rules. Don't use them in the offline ruleset. The parameter replacement is done in the dynamic rules. So it works together with the exportname rule. If you have a rule before the dynamic rules that is triggered the parameter replacement will not happen.





# 14. The OpenCms synchronization

## 14.1. What is synchronization?

The synchronization tool is used to synchronize files in the Virtual File System of OpenCms and their corresponding files in the Server File System .

It's a feature for development only. It speeds up the development cycle because you can modify the file on your Server File System with your favourite application and update the file in the file system of OpenCms.

## 14.2. Manage the properties for OpenCms synchronization

The synchronization requires some entries in the *registry.xml*. You will find the registry in your configuration path of OpenCms (*config/*). You need to enable and modify the following section:

```
<syncpath>c:/work41/sync</syncpath>
<syncproject>_syncProject</syncproject>
<syncresource>
  <res1>/content/</res1>
  <res2>/download/</res2>
  <res3>/pics/</res3>
</syncresource>
```

The *<syncpath>* is the path on your server where you want to save the synchronized files. The *<syncproject>* is the name of your project in OpenCms whose files you want to synchronize. In the section *<syncresource>* you put the names of the resources you want to be synchronized. Put each resourcename between a tag named *<resx></resx>* where the *x* is a serial number.

To make it easier for you to manage these entries there is a management tool in the backoffice. Only administrators are allowed to manage the synchronization properties.

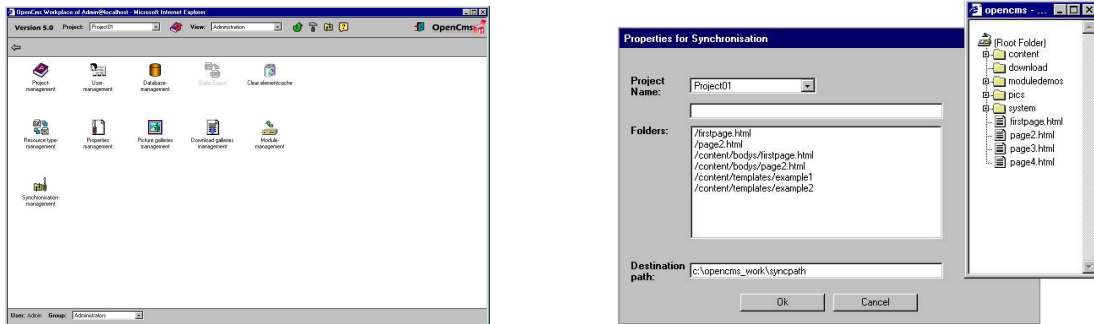


Figure 14.1.: The synchronization management icon and dialog

A click on the icon will open the dialog for synchronization management. If the entries for synchronization already exist in the *registry.xml*, they will be shown in the dialog.

You can choose any project from the project list. The list contains your accessible projects except the online project. So there must be at least one offline project. When you have chosen a project the folder tree, that you get with the folder button, shows the resources in this project. By clicking on a resource you can add it to the resource list for synchronization. You can also add resources by filling in the resource name into the field above the resource list and clicking the arrow button. To remove resources from the list you use the delete button.

The third entry that is needed is the destination path on your server.

When you choose all entries you can update the registry by clicking on 'OK'. There will be error messages if you left the project, the resource list or the destination path empty or if a resource can not be read from the chosen project.

### 14.3. How to synchronize

If there are the required entries in the *registry.xml* the button for synchronization will appear in the head section between the icons for the preferences and the help (Figure 14.2).

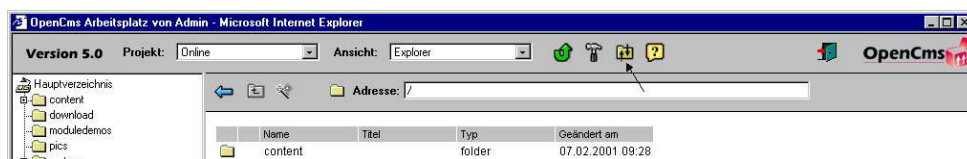


Figure 14.2.: The button for synchronization

A click on the button will start the synchronization. If the project for synchronization, which you specified in the registry, does not exist it will be created and the resources that should be synchronized will be copied from the online project to the synchronization project.

The synchronization will not work if there is more than one project with the name of the synchronization project or if the resource to be synchronized does not exist in the project.

## 14.4. How the files are synchronized

First the resource and all its subresources are read from the Virtual File System and checked for changes.

- The resources that does not exist on the Server File System will be created.
- If the resource is marked as deleted the corresponding resource will be deleted from the Server File System.
- Files are compared with the entries in the *synchronize.list*.

The *synchronize.list* is created after the synchronization and contains the name of each synchronized file and the date of its last modification in the Virtual File System and the Server File System.

- If the date of the last modification of the file on the Virtual File System is after the corresponding date in the *synchronize.list* the modification date of the file on the server is checked.
  - If the date of the file on the server has not changed after the last synchronization, the file will be updated with the content of the file on the Virtual File System.
  - Has the file be changed on the Server File System, too, it will be copied to a backupfile (the backupfile is marked with a leading \$) and updated with the content of the file on the Virtual File System.
- If the file has not be changed on the Virtual File System but on the Server File System, the file on the Virtual File System will be updated with the content of the file on the Server File System.

After the resources on the Virtual File System were checked, the filelist of the resource on the Server File System is read. The Virtual File System is checked if it contains all of the files and folders in this list. Files and folders that does not exist in the Virtual File System are created.



# 15. System architecture

## 15.1. Components

OpenCms is a client server application that can be used in HTTP-based environments such as the Internet. OpenCms is a classic web application with a 3-tier-architecture (figure 15.1).

The presentation layer consists of a web browser (Internet Explorer or Netscape Navigator) that is used to display and navigate through the HTML user interface. You need a state of the art browser to use the workplace of OpenCms. If you use an older version of the browser you will not be able to execute the existing DHTML functions (JavaScript code) or attach external style sheets. Both are necessary for a correct usage of the application.

The logic layer lies on the web server, which is extended by a runtime environment for Java servlets. The web server can be the Apache web server (Version 1.3.6 or higher), the configuration of which must be completed by the so called "Jserv module" (the Servlet Engine, Version 1.0 or higher), in order to execute servlets. All Java classes are installed in the servlet environment on the server. The OpenCms servlet (one single class) provides the interface to the presentation layer as well as the interface to the database layer. It uses the HTTP protocol to establish the communication between the client and the application on the server (in this case OpenCms). The database is accessed through the JDBC interface. The database contains tables for resource, user and property data.

The following figure shows the most important components of OpenCms (Figure 15.2).

**CmsObject:** All resources are accessed via the CmsObject. This is the interface to the OpenCms system for the Module Developer. The CmsObject is initialized with the user data.

**Resource Broker:** The resource broker checks requests for resources and performs them if the user has the appropriate access permissions. It's configuration depends on the database that is used.

**Launcher Manager:** The Launcher Manager determines the launcher that will generate the output with the requested content based on the type and/or content of the selected file. If the template launcher has been selected, the Template Mechanism is started.

**Template Mechanism:** The Template Mechanism creates HTML pages based on the content in its structured form. Content definitions enable you to access content that

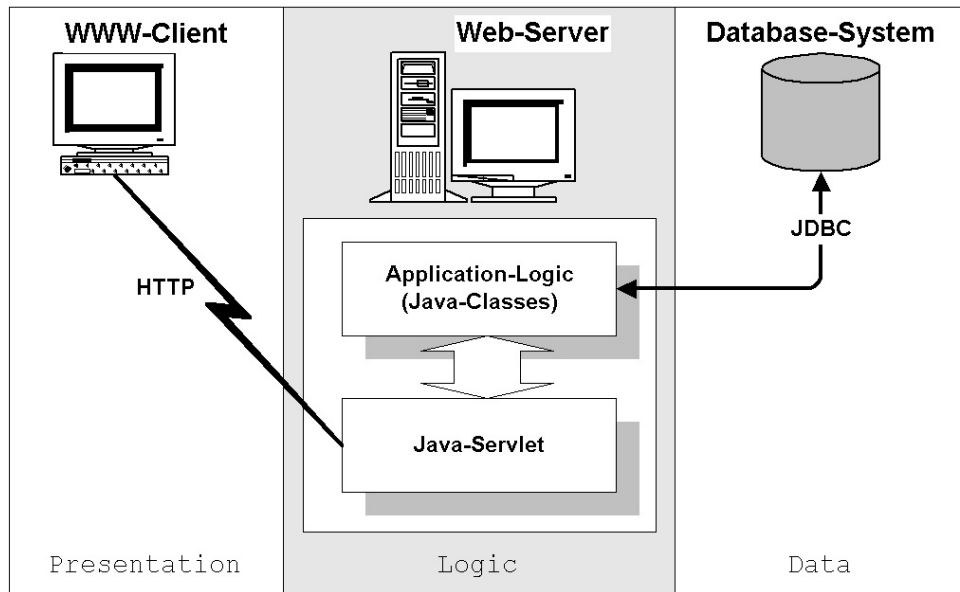


Figure 15.1.: Example of a 3-tier-architecture using OpenCms

originates from different sources (e.g. Virtual File System (VFS), database, file system ...).

## 15.2. Accessing system resources

The general procedure of requesting an OpenCms web page is shown in the next figure. First, a request is sent from the web browser. The URL contains the host, the servlet path and the URI. The web server starts the servlet engine because the URI is within the servlet path, enabling the servlet engine to pass it to OpenCms (Figure 15.3).

System resources are accessed via an instance of the class *CmsObject*, which provides the public interface that is used to access the system and create the corresponding context. From the *msObject*, requests are forwarded to the non-public parts of OpenCms. Thus, an instance of the class *CmsObject* is created and initialized with the current user first. The *CmsObject* passes the resource request to the resource broker. The *resource broker* is the interface to the database. It performs the request via the database access module and returns the resource if the user has the right to access it. The access module establishes the connection between the logic layer (application) and the database layer. If a resource is requested, the *resource broker* forwards the request to the access module. The access module generates an SQL statement to get the data from the database. It then generates a *CmsRessourcObject* from the database data and returns it to the resource broker. The *resource broker* can now check the access permissions of the file and return the file to the *CmsObject* providing the user has the appropriate access permissions.

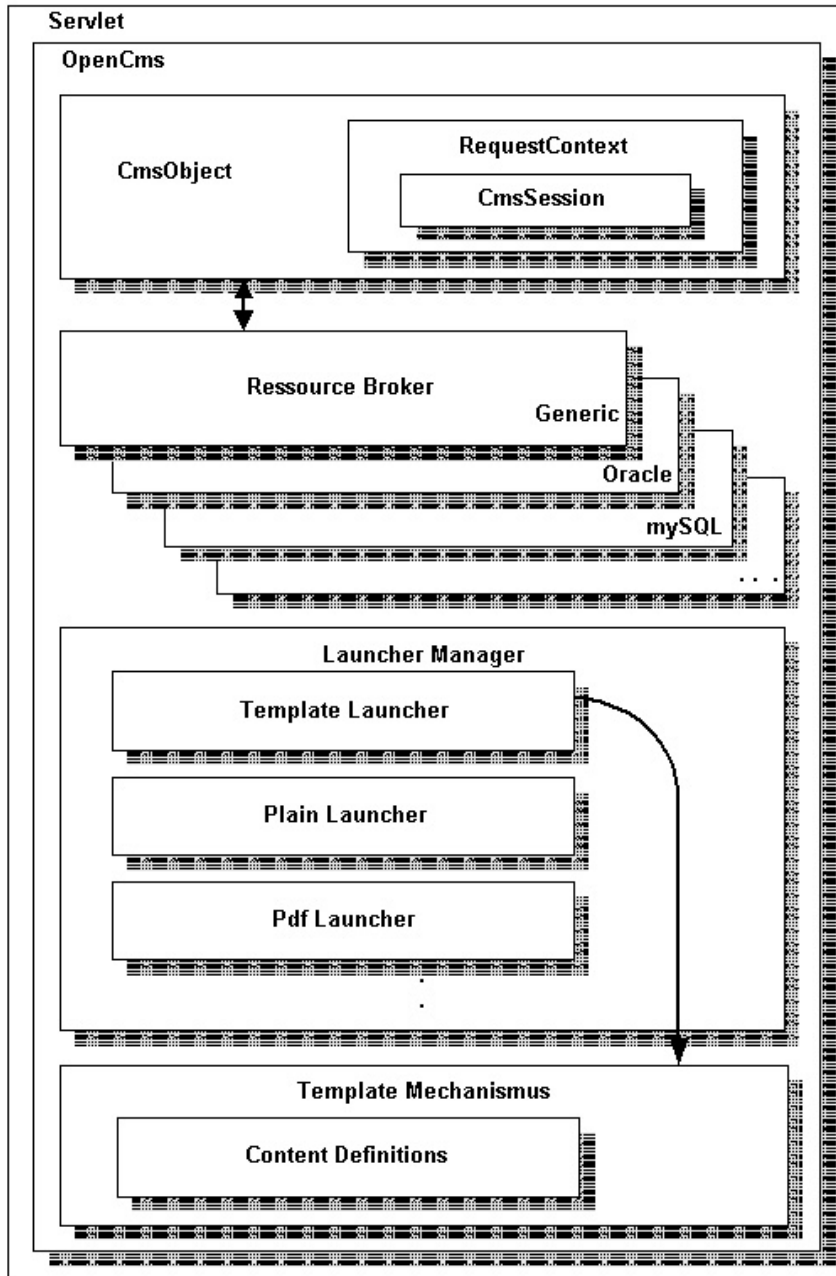


Figure 15.2.: Compoments

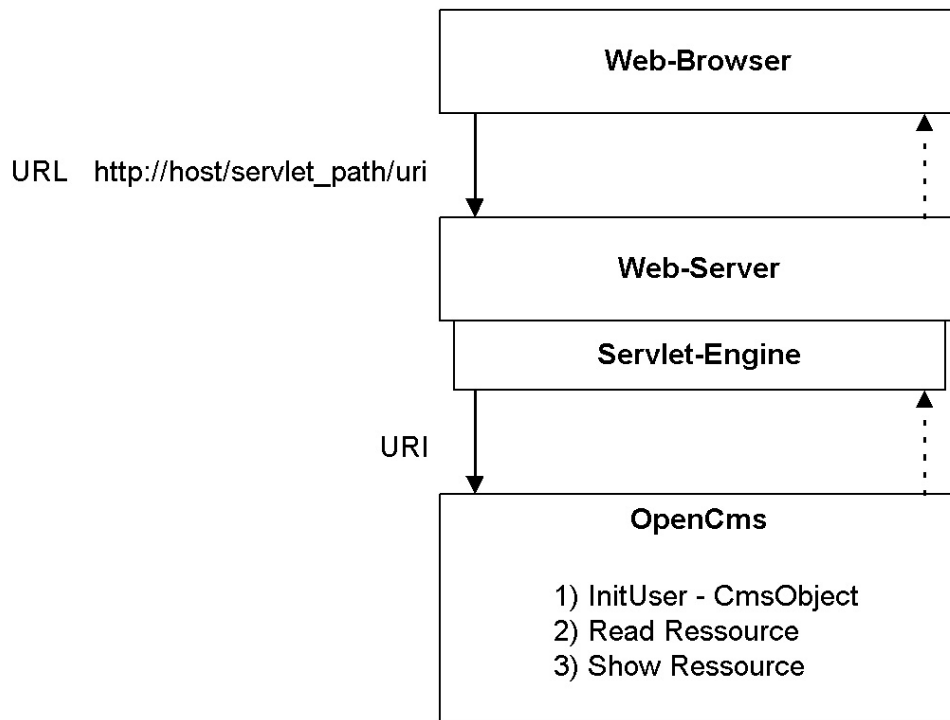


Figure 15.3.: Accessing system resources

Because OpenCms works with different kinds of databases, the *resource broker* and the access module are configured according to the used database (Oracle, mySQL, etc.). The following figure shows the structure used to access resources (figure 15.4).

If the operation is successful (i.e. the user has permission to read the file), the resource is displayed (sent to the user's web browser).

This is done by the *Launcher Manager*. The *Launcher Manager* starts a certain launcher according to the type of the requested resource. For pure text files, images etc. the plain launcher is used, which forwards the content of the resource directly. The *template launcher* is used to start the Template Mechanism when a template is requested. The *Template Mechanism* creates HTML pages based on the content in its structured form.

The different launchers and the *Template Mechanism* will be explained in detail in the following chapters.



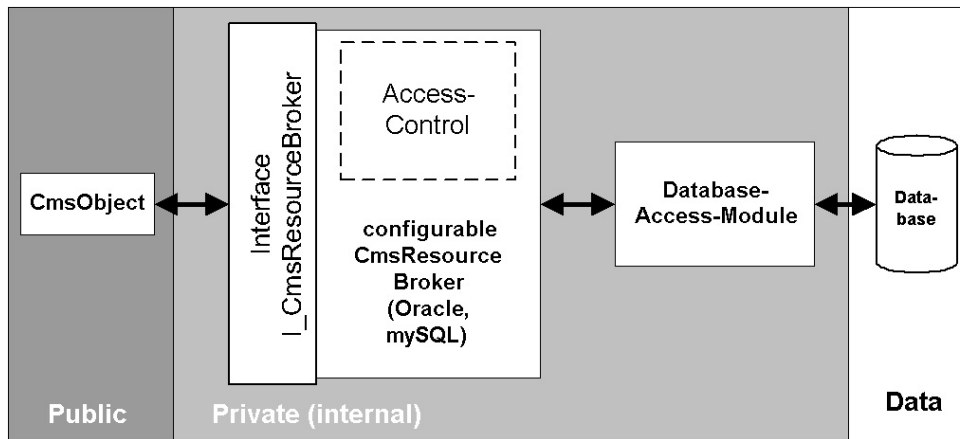


Figure 15.4.: Accessing resources in OpenCms

### 15.3. Resources

All of the Objects that are administrated in OpenCms are called *resources*. At the moment, there are eight different types of resources. With the exception of the resource type *folder*, all other types are resources files that have content that can be displayed in a browser. The resource type is used to provide every file with a resources processing instance - called *Launcher* - which starts the processing process and then displays the data. The following table shows the existing relationships. Table 15.3

Name	ID	Description	Launcher
folder	0	Folder	none
plain	1	Files that are not processed by OpenCms (text files, JavaScript libraries etc.)	Dump launcher
XMLTemplate	2	XML coded template file.	XML launcher
page	3	Representation of a whole site ("clickable file" or "page file").	XML launcher
binary	4	Binary files (like zip archives and PDF documents, but not graphics) that can be uploaded.	Dump launcher
image	5	Binary graphic files (like GIF/JPEG)	Dump launcher
script	6	Designed for JavaScript on the server-side	Dump launcher
newspage	7	Representation of a news article, special case of a page file	XML launcher
JSP	8	Java Server Page	JSP loader

Table 15.1.: Resources in OpenCms

## 15.4. The Virtual File System (VFS) of OpenCms

All of the files and folders of the different projects that are displayed in the Explorer view that is part of the OpenCms Workplace, are not stored in the normal file system. OpenCms has a *Virtual File System* that holds the files and folders in a database, and an integrated access permission administration. The access permissions in OpenCms are similar to those of the UNIX file system. However, by using a *VFS* and a database, OpenCms is independent from the operating system that is installed on the server. In this way OpenCms can run on a Windows 95/98 system, which itself has no access permission administration at all. Here is a short overview of the folders and the files in the standard OpenCms *Virtual File System*: Table [15.4](#)

Please note that the VFS structure was modified in the 5.0 release of OpenCms. Previous versions had a layout different from the one described above.

## 15.5. Class structure

The functionality of OpenCms is implemented in many different Java classes. The classes are divided into different packages to structure the code. Currently, there are 10 packages that cover a certain part of the system's functionality. Table [15.3](#)

Directory	Files located in the directory
<i>/system</i>	Parent directory of the directories that contain workplace templates and configuration files for the system.
<i>/system/bodies</i>	Body templates are stored in this directory or a subdirectory thereof. They are stored in a directory with the same name as the directory in which the web site is located.
<i>/system/modules</i>	OpenCms modules are automatically placed below this directory.
<i>/system/modules/default</i>	Location of the default module which contains a basic empty template and some resource files.
<i>/system/modules/default/templates</i>	The default master templates that define the general layout of a web site.
<i>/system/galleries</i>	Basis location of all galleries that are available in the editor.
<i>/system/galleries/download</i>	Gallery for files that can be downloaded.
<i>/system/galleries/pics</i>	Gallery for pictures that are used in web sites.
<i>/system/login</i>	The OpenCms workplace login screen is located here.
<i>/system/workplace</i>	The OpenCms workplace application files are located in this subdirectory.
<i>/system/workplace/action</i>	Executable files like login, copy etc.
<i>/system/workplace/templates</i>	The templates that are used to create the workplace.
<i>/system/workplace/scripts</i>	Javascript files used in the workplace are placed here.
<i>/system/workplace/administration</i>	Subdirectories for every Item in the Administration view.
<i>/system/workplace/locales</i>	The workplace locales, i.e. language files reside here. They are stored in a separate subdirectory for every language.
<i>/system/workplace/resources</i>	Workplace static resources are placed in this directory. These are the style sheets used and the images for system elements like buttons etc.
<i>/system/workplace/help</i>	The help system.

Table 15.2.: VFS of OpenCms

<i>com.opencms.core</i>	Classes of the system's core. The package also contains the OpenCms servlet.
<i>com.opencms.file</i>	Definitions of all of the objects.
<i>com.opencms.file.genericSql</i>	Resource broker for generic SQL databases.
<i>com.opencms.file.mySql</i>	Special resource broker for mySQL.
<i>com.opencms.file.oraclepsql</i>	Special resource broker for ORACLE.
<i>com.opencms.file.utils</i>	File package.
<i>com.opencms.launcher</i>	All of the launchers that are used to display files or templates and a Launcher Manager.
<i>com.opencms.template</i>	All of the template classes that are used to create documents dynamically and a Template Manager.
<i>com.opencms.util</i>	Some general functionality (encoder, decoder etc.)
<i>com.opencms.workplace</i>	All of the components (e.g. buttons, context-sensitive menus etc.) and functions (e.g. create project) of the user interface (OpenCms Workplace).

Table 15.3.: Class structure

# 16. Core Development

## 16.1. Introduction

In this section you can learn something about the interna of OpenCms. This is interesting for people who want to contribute to the project at corelevel. Currently there are two subsections. One about how to build OpenCms from scratch by your own using ant. The other about interna of the project mechanism used by OpenCms since version 4.4.

## 16.2. How to compile the OpenCms core

OpenCms provides a full source distribution that you can use to build the OpenCms core. This is only needed if you want to add your own core extensions. To develop the usual kind of website functionality, you should use the OpenCms module mechanism which is much easier to start with and also much better documented. Before even considering starting to work on the core, you definitely should have written some OpenCms modules to understand the separation between a module and a core extension.

The OpenCms core comes with the best possible documentation: The source code itself. This is really not something for the novice Java developer. However, if you have some experience in Java, Java Servlets, JDBC, XML in general and the Apache Xerces API you might take a look. As said before, you should also have already a firm understanding of the OpenCms module API.

Since this is rather deep stuff more for experts than for beginners, I will not explain every detail of the process.

### 16.2.1. Before you start

In case you really intend to contribute to the OpenCms core development, please make sure you use the latest version from the CVS source tree. The source distribution ZIP file provided for download is the code of the latest major release, not the current development status of OpenCms. The ZIP provides a good starting point, but contributions should be based on the latest development version. The way to do it is to first get the source distribution (because it also contains the libraries), see if you can build this, and then

simply checkout the latest `/opencms` directory from the CVS, replacing the `/opencms` directory from the source distribution (see below).

Ok, now on to the real work:

### 16.2.2. Have Ant installed

Apache Ant is a Java based build tool. In theory it is kind of like make without make's wrinkles. You need Ant version 1.4 or later to build the OpenCms core. Ant is part of the Jakarta Apache Project (<http://jakarta.apache.org/>) and can be downloaded here: <http://jakarta.apache.org/ant/>. Please check the Ant documentation to make sure you understand the basic principles behind Ant.

Ant installation is described in the manual (<http://jakarta.apache.org/ant/manual/>). It requires that you have set up your path to Java correctly. Make sure Ant runs before proceeding.

### 16.2.3. Get the OpenCms Source distribution

Download the latest OpenCms source distribution. It contains all classes necessary to build the OpenCms core. You can also checkout an OpenCms version from our CVS tree, but make sure you have downloaded the source distribution before that because the necessary libraries are not included in the CVS.

The source distribution comes in a ZIP file. Unpack this in your working directory, lets say this is called `/work`. You will then end up with the following structure in your work directory:

```
/work
  /opencms          <= Start of the OpenCms source tree
    /etc
    /src
    /web
    build.xml       <= This is needed by Ant
    ...
  /ExternalComponents <= Contains the necessary libraries
    activation.jar
    fesi.jar
    ...
```

### 16.2.4. Get additional classes (optional)

In case you want to use the Oracle database, you need the Oracle JDBC driver which you can download from the Oracle Technology Network (<http://otn.oracle.com>). You

have to register there to get the driver. The file you need is the JDBC driver called `classes12.zip`. Place the file in the directory `/ExternalComponents` of the OpenCms source tree (see above). If you don't have this file, the OpenCms Oracle Driver (package `com.opencms.file.oraclepsql`) is not compiled, which is ok if you don't run on Oracle.

### 16.2.5. Build the source by starting Ant

Ok this is the easy part. Call up a commandline, move to the `/opencms` directory (where the file `build.xml` resides) of the OpenCms source tree. In your commandline, enter the following:

```
ant all
```

That's it! This will build a complete OpenCms distribution. Your work directory will look like this after Ant is finished:

```
/work
  /opencms          <= Unchanged
  /ExternalComponents <= Unchanged
  /build
    /classes        <= Will contain the compiled OpenCms classes
    /opencms        <= This is the standard .war directory layout
      /WEB-INF
        /lib
          /oclib
            /META-INF
  /zip              <= The OpenCms distribution file will be placed
  /pdf              here
```

The final result of the compilation will be a ZIP file which will be placed in the `/zip` directory. This ZIP is exactly the same layout as the OpenCms binary distributions, so it will contain the `opencms.war` archive.

### 16.2.6. Install your new version

Now that you have your new OpenCms binary distribution, you simply need to follow the installation guide for your server setup. Please make sure you don't mess up any existing installation. Best have a separate machine that is used only for testing and development.

### 16.2.7. Other Ant targets

There are several more targets in the ANT that might be useful for you. Here's a short overview:

- Ant target: `war`  
This creates the `opencms.war`, but does not create the binary distribution ZIP file.
- Ant target: `srcdist`  
This generates the source distribution ZIP file. Like the one you can download.
- Ant target: `tomcat.dist`  
This is useful if you really are in developing core extensions. Provided that OpenCms runs on your machine using Tomcat (and that you have all environment variables set up, esp. `TOMCAT_HOME` or `CATALINA_HOME`), calling this target will have Ant updating the OpenCms classes on your machine. This is done by replacing the files in the Tomcat `webapps/opencms` directory. If you have renamed `opencms.war`, or if your Tomcat is not installed the usual way, this probably will not work.
- Ant target: `setup`  
This is an extension to `tomcat.dist`: It depends on `tomcat.dist`, but will also update the OpenCms database, provided you run MySQL on your development machine. This is usefull if you do changes to the OpenCms Workplace template files. CAUTION: This drops your complete database without further notice. Don't use until you know what you do.
- Ant target: `tex`  
This generates the OpenCms documentation from the TeX sources. It's outside the scope of this document to explain what setup you need to do that.

To get a complete actual list of all targets please call `ant` with this parameter:  
`ant -projecthelp`.

### 16.2.8. Important

In case you replace files in the Tomcat OpenCms `webapps` directory without doing a new setup of the database, you will get a message on login saying something like:

```
The version of workplace-templates and workplace-classes are different.  
Please update one.
```

Since the workplace templates make very close usage of the core, this should definitely be fixed on production systems. To remove the message, make a new setup of the OpenCms database using the installation wizard from your newly generated binary distribution file. If you know what you do, you can ignore this message on development machines.



## 16.3. The Projectmechanism

OpenCms contains a projectmechanism where all resources exist at least twice - once for the "Online" project and once for all offline projects. If you have enabled versioning, the historic versions of the resources are also saved in a separate backup table space.

The projects that can be created with the project management are views on a shared set of offline resources. If a resource is published, it is copied from the offline space to the online space and from the online space to the backup (history) space.

### 16.3.1. Data structure

There are several tables for the online resources, the offline resources and the backup resources. This minimizes the quantity of data in the tables and accelerates queries on the data. Another advantage is the possibility to store online, offline and backup resources on several databases. Each of these three groups has its own database connection pool.

For the online resources there are the tables

- CMS\_ONLINE\_RESOURCES
- CMS\_ONLINE\_FILES
- CMS\_ONLINE\_PROPERTIES
- CMS\_ONLINE\_PROPERTYDEF

The tables of the online resources are

- CMS\_RESOURCES
- CMS\_FILES
- CMS\_PROPERTIES
- CMS\_PROPERTYDEF

The backup resources are stored in the tables

- CMS\_BACKUP\_RESOURCES
- CMS\_BACKUP\_FILES
- CMS\_BACKUP\_PROPERTIES
- CMS\_BACKUP\_PROPERTYDEF

Each version of a backup resource has a version number. The information of the project that was published is stored in the backup tables

- CMS\_BACKUP\_PROJECT
- CMS\_BACKUP\_PROJECTRESOURCES.

There is another new table named CMS\_PROJECTRESOURCES. This table contains the ID of a project and the names of the resources that were selected when the project was created. The entries describe the view of the project on the offline resources.

### 16.3.2. During installation of OpenCms

When you setup OpenCms the first time to install the workplace some data is filled in by default. With regard to the new projectmechanism

- the online project is created as permanent project
- the root folder is created in the online tables
- the name of the root folder is inserted into the table CMS\_PROJECTRESOURCES for the online project
- the setup project for the import of the workplace is created as temporary project
- the root folder is created in the offline tables
- the name of the root folder is inserted into the table CMS\_PROJECTRESOURCES for the setup project

Because the setup project is created by default the setupscript does not create a project for the import of the workplace.

When the setup project is published the workplace will be copied to the online tables and to the backup tables. The temporary setup project will be deleted but the resources still exist in the offline tables and can be accessed by creating a new project.

### 16.3.3. Creating a new project

A project can be created for permanent or temporary use. A temporary project will be deleted after it was published.

When creating a new project only the names of the selected resources are stored in the new table CMS\_PROJECTRESOURCES and no resources are copied. So creating a new

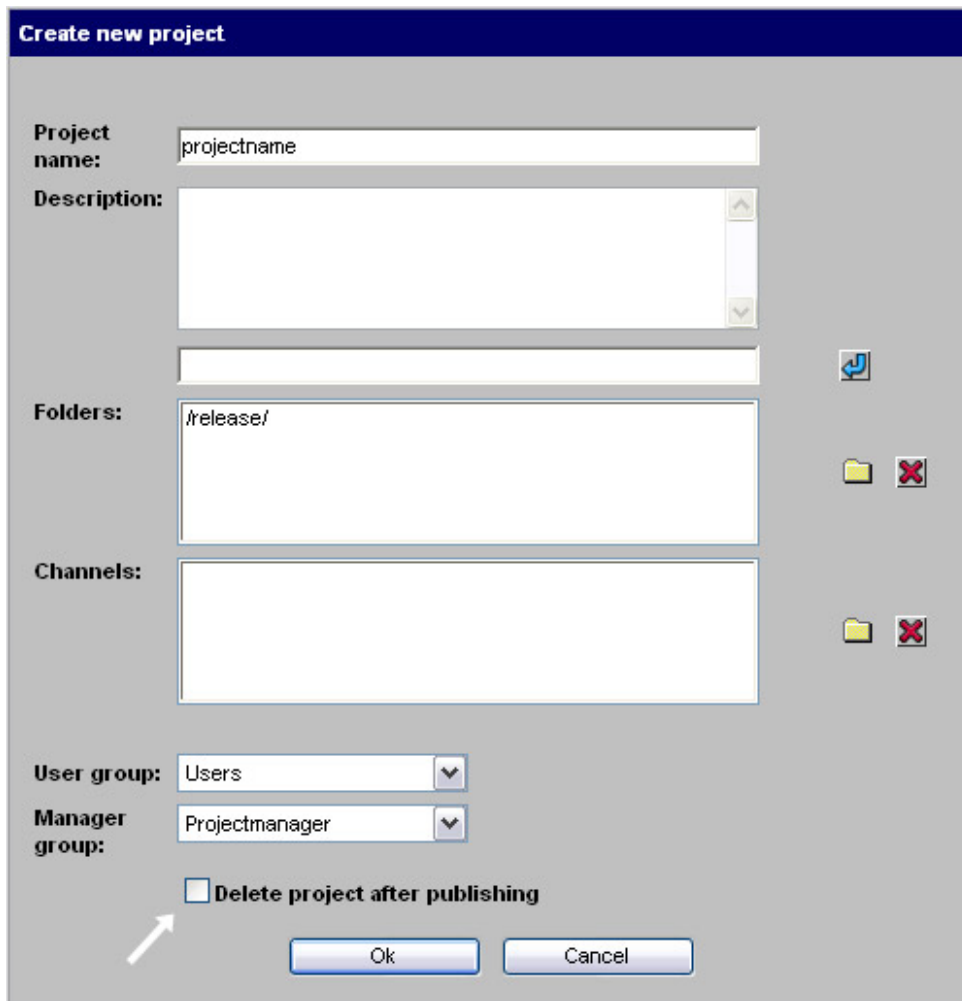


Figure 16.1.: Creating a new project

project is much faster than before. The new project contains the selected resources and always the folders `/system/galleries/pics/`, `/system/galleries/download/`, `/system/galleries/externallinks/`, `/system/galleries/htmlgalleries/` and the folders in `/system/bodies/` that correspond with the selected folders.

The project is now a view on the offline resources. If two projects reference the same resource the changes on this resource will be visible to both projects. If a folder is referenced in both projects and a new resource is created in this folder in one project, the new resource is accessible in the other offline project, too.

To avoid that resources that are in work in one project are published unintentional by another project it is dissuaded from creating projects that reference the same resources.

### 16.3.4. The view of the project

The explorer shows all offline resources in an offline project. The resources that do not belong to the project are grey. The resources that are referenced by more than one project are displayed in each of this projects in the same manner: locked resources with the lock sign, new resources are colored blue, changed resources are red, deleted resources are crossed out.

If a resource is locked in a project the projectid of this resource is changed to the current projectid. This is necessary to guarantee that only the resources that are locked by this project can be unlocked before they are published.

If you create, delete or change a resource in one project the changes are immediatly visible to all projects that reference the resource or one of its subfolders. This behaviour is fundamental different to the behaviour of the former projectmechanism where each project has its own version of the resource.

The new, changed or deleted resource gets a little flag. The red flag means the resources was changed in the current project, the grey one means the resource was changed in another project.

### 16.3.5. Publish a project

Before a project is published all resources that are locked in this project are unlocked. Resources that are locked in another project will be still locked even if they are referenced by the project that is published.

Only the files and folders in this project that are marked as new, deleted or changed and that are unlocked will be published. Resources that were changed and unlocked by another project are not published.

When a resource is published a copy of this resource is stored in the backup tables. The versions of a resource in these backup tables are shown in the history of the resource. The state of the published resource is set to unchanged.

The current information of the published project is stored in a backup table, too. If the project was created as permanent project you can go on working in the same project after it is published. You need not to create a new project as it was required in the former projectmechanism. A project that was created as temporary project is deleted and is not accessible any longer.

# 17. The proprietary XML template mechanism

**Since Version 5 of OpenCms, we suggest to build templates and all sub-elements based on Java Server Pages (JSP).**

The first versions OpenCms supported only a proprietary template mechanism based on XML. When the OpenCms project was started there where no open source JSP implementations available. Since this has changed very much today, OpenCms now supports standard JSP for building templates and interactive functionality.

The full documentation for the JSP integration is available as an interactive module. Please refer to the introduction chapter of this document for more information on how to obtain this part of the documentation.

Using templates based on JSP offers you some important advantages:

- Experienced Java programmers can start immediately to implement their own templates without having to learn a new template language.
- Java Server Pages make it easy to extent OpenCms by implementing reusable standard components such as Java Beans.
- OpenCms offers a package with Java Beans plus a taglib to access the most common OpenCms functions such as navigation entries etc. pp.
- HTML producers can develop JSP templates by using the OpenCms JSP Taglib and the Java Standard Tag Library (JSTL).
- The FlexCache is a highly efficient caching mechanism to cache JSP templates/elements.
- JSP templates/elements are localized by using standard Java resource bundles.

The proprietary XML template mechanism is still a part of OpenCms for backward compatibility reasons. We recommend the use of JSP templates for new projects.

## 17.1. Introduction to the XML template mechanism

### 17.1.1. The master- frame- and contenttemplate

In this chapter, we start with showing the XML template mechanism by some examples that should be modified by you. This way you should get used to the system and see that most templates can easily be modified from the interface point of view. Thus, it is no problem if you don't understand all tags and functions that are used at the beginning, they will all be explained later in this document in detail. Right at the beginning, you should try to find out how they work and behave by working with them.

We will now create a first template set of a very simple kind. Our first template set will not include any special sub templates and will not define a special layout, but it should give a little insight in the template mechanism.

The first example will show how to create a template set that displays the string *Hello, world!*. A template set consists of three templates: *mastertemplate*, *frametemplate* and *contenttemplate*

The *frametemplate* is the template that defines the general design of a page, it defines a "layout framework" for the other elements included in the page. In a classic website, the *frametemplate* usually would consist of the navigation included as one ore more elements and the common HTML-layout for the website.

The *contenttemplate* is included as an element in the *frametemplate*. It defines the layout of the content part of the HTML-page. It contains one or more body-Elements which hold the content generated with the HTML editor and optional other user-defined elements. Figure 17.1 shows what parts of a page are defined by the *frametemplate* and the *contenttemplate*.

The *mastertemplate* only defines which *frametemplate* and which *contenttemplate* should be used by the page. We will create a template set of

*/system/modules/org.opencms.default/templates/template1* (the *mastertemplate*),

*/system/modules/org.opencms.default/frametemplates/frametemplate1*

and */system/modules/org.opencms.default/contenttemplates/contenttemplate1*.

To create a template you must first create a new text file in the project's

*/system/modules/org.opencms.default/templates/*

directory. Go to the

*/system/modules/org.opencms.default/templates/*

directory in the Explorer view of the new project and start the wizard that creates the new files by clicking on the icon with the magic wand. Select *Text* for the type of the file from the New dialog box (figure 17.2).

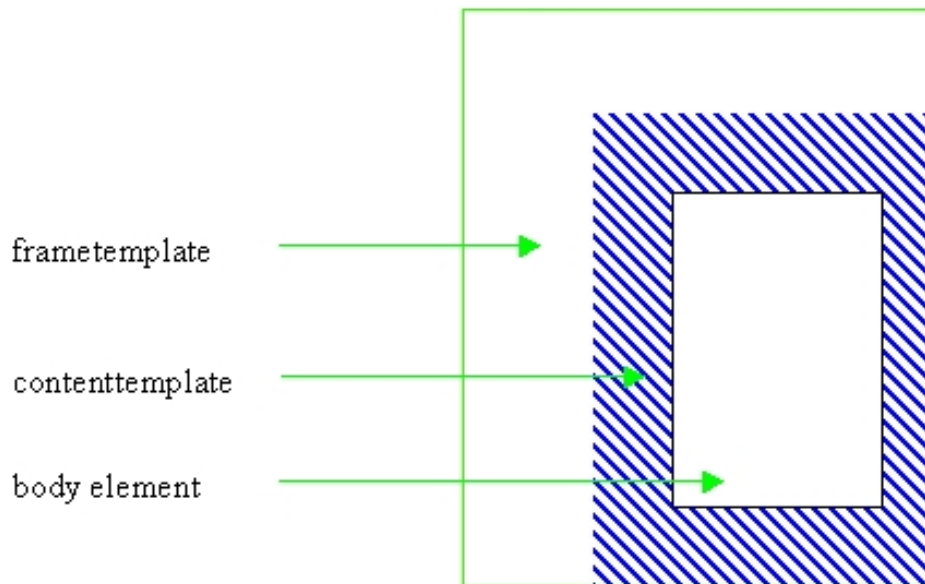


Figure 17.1.: frametemplate and contenttemplate

Enter the name and title for the new template in the Create a new File dialog box and click on *Finish* to end the creation of the new text file (figure 17.3).

Select *Edit Code* from the file's context menu to edit the new file. The context menu is accessed by clicking on the new file's icon (figure 17.4). This will start the HTML editor. A template file is simply an XML file that contains XML and HTML tags. The mastertemplate contains the definition of the frame- and the contenttemplate. To accomplish this the template must contain the following text:

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

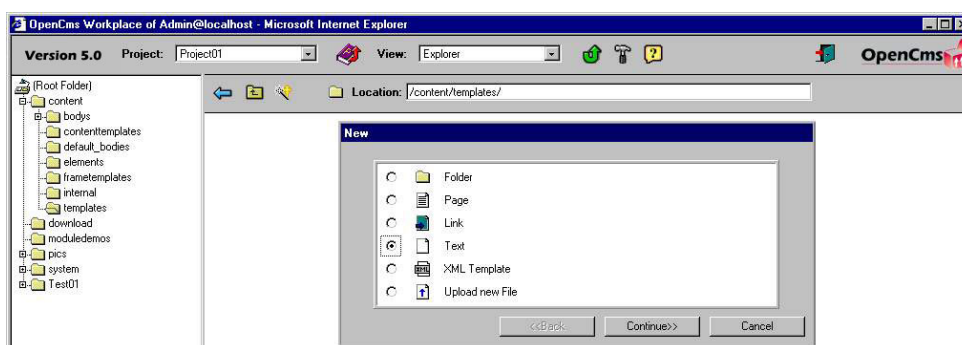


Figure 17.2.: Select Text for the type of the new file

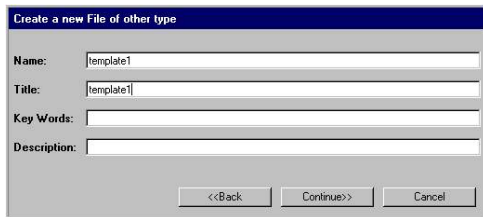


Figure 17.3.: Enter name and title

```
<ELEMENTDEF name="contenttemplate">
  <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
  <TEMPLATE>../contenttemplates/contenttemplate1</TEMPLATE>
</ELEMENTDEF>
<ELEMENTDEF name="frametemplate">
  <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
  <TEMPLATE>../frametemplates/frametemplate1</TEMPLATE>
</ELEMENTDEF>

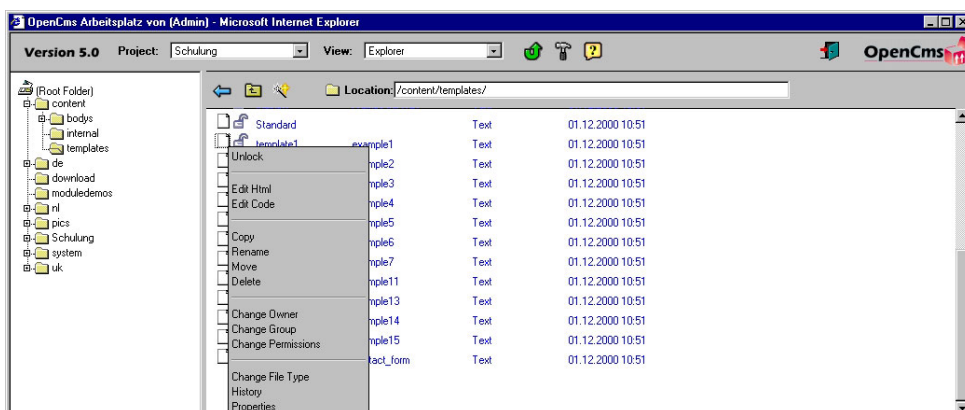
<TEMPLATE>
  <ELEMENT name="frametemplate"/>
</TEMPLATE>

</XMLTEMPLATE>
```

You can either type in this text or copy and paste it into the editor. Exit the editor and save the file by clicking on the icon to the left that shows a floppy disk and a cross. Now create the `frametemplate1` in the directory

`/system/modules/org.opencms.default/frametemplates/`

with the text:

Figure 17.4.: select *edit code* from the context menu



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
  <TEMPLATE><![CDATA[
    <html>
      <head>
        <title>The first templates</title>
      </head>
      <BODY >
        ]]><ELEMENT name="contenttemplate"/><![CDATA[
      </BODY>
    </html>]]>
  </TEMPLATE>
</XMLTEMPLATE>
```

At last create the `contenttemplate1` in `/system/modules/org.opencms.default/contenttemplates/` with this content:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
  <TEMPLATE><![CDATA[
    Hello World! <br>
  ]]><ELEMENT name="body"/>
</TEMPLATE>
</XMLTEMPLATE>
```

The next step is to create a new page that uses the template set you just created. As mentioned before, the page-control file is the one that has to be clicked on to view a page in the Browser. This file has to be created now. To do so, change to the root folder and use the wizard to create a new file. Select *Page* as the new file type. Enter the web page's name and title. The name is the name of the file and the title is the title that should be shown when the page is requested. Select the master template for the new page from the Template drop-down list, which displays the titles of the templates that can be found in the

`/system/modules/org.opencms.default/templates/`

directory (figure 17.5). The other fields in this dialog will be discussed later. For the time being, they can retain their default values. Click on the *Finish* button to create the new page.

You can now view the page by clicking on it. A new browser window will be opened in which the new page is displayed. The page shows the *Hello, world!* text from the `contenttemplate`. The title of the page is the title that was defined in the `frametemplate`.

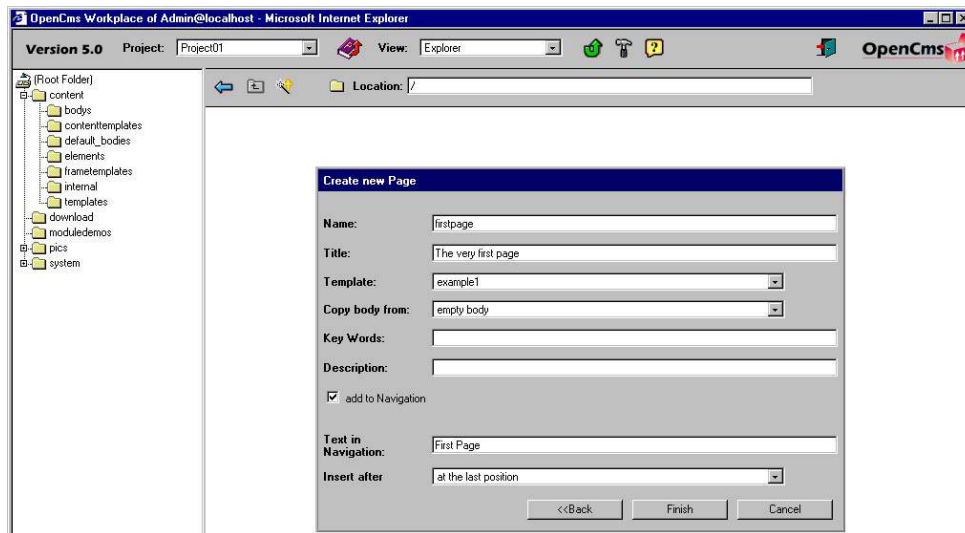


Figure 17.5.: The Template drop-down list displays the titles of the templates that are stored in the */system/modules/org.opencms.default/templates/* directory

### 17.1.2. Using methods in XML templates

We will now create a new frametemplate that uses a method to set the title in the template. This way the title will be set dynamically and is not the same for all pages created with our template. To create the new frametemplate you have to create a new text file (frametemplate2) in the

*/system/modules/org.opencms.default/frametemplates/*

directory. Follow the steps to create a new template that are described in the first example, and insert the following text in your frametemplate:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
  <TEMPLATE><![CDATA[
    <html>
      <head>
        <title>]]><method name="getTitle"/><![CDATA[</title>
      </head>
      <BODY >
        ]]><ELEMENT name="contenttemplate"/><![CDATA[
      </BODY>
    </html>]]>
  </TEMPLATE>
</XMLTEMPLATE>
```

Now you also need a new mastertemplate which uses the new frametemplate2 and the old contenttemplate. Create it in

`/system/modules/org.opencms.default/templates/`

as described in the first example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
  <ELEMENTDEF name="contenttemplate">
    <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
    <TEMPLATE>../contenttemplates/contenttemplate1</TEMPLATE>
  </ELEMENTDEF>
  <ELEMENTDEF name="frametemplate">
    <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
    <TEMPLATE>../frametemplates/frametemplate2</TEMPLATE>
  </ELEMENTDEF>
<TEMPLATE> <ELEMENT name="frametemplate"/> </TEMPLATE>
</XMLTEMPLATE>
```

You can now create a new page with this mastertemplate and see that the title of the generated HTML page is exactly the one you typed in when creating the page. There are a number of standard methods that can be used when creating a template. The methods are defined in the `com.opencms.template.CmsXmlTemplate` class which is assigned to the mastertemplate per default in the page-control file. As you can see, the usage of standard methods is quite simple.

```
<TITLE>]]><method name="getTitle"/><![CDATA[</TITLE>
```

The method-tag is a special tag which allows to call functions from a template. The return value of the method (in this case the correct title of the page) is inserted into the page when the template is processed. Other standard methods will be discussed later in this document.

### 17.1.3. The body element

To achieve a separation of content and layout of a website, the content is normally inserted in a body element. A body element is a subtemplate, that holds editable content, which will be shown at the place where this element is located in the contenttemplate. Create a new page in the root folder and select the title of the new template from the drop-down list. For example, if you have defined the title of the template as `template2`, this text should appear in the drop-down list. Enter a title and name for your new page, and select *Finish* to create the page (figure 17.6).

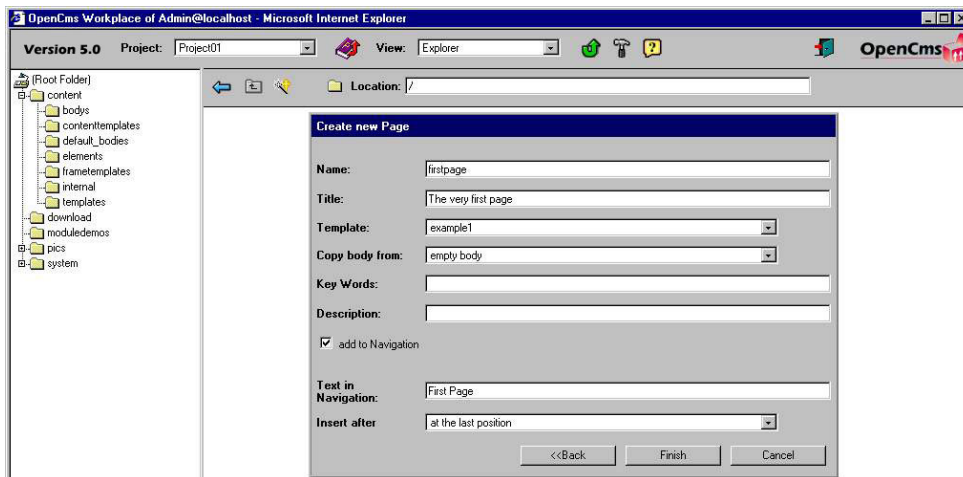


Figure 17.6.: Select the title of the new template when creating the page

Before we take a look at the new page, we will use the HTML editor to add text to the body element. To edit the page with the HTML editor select *Edit Page* from the new page's context menu (figure 17.7).

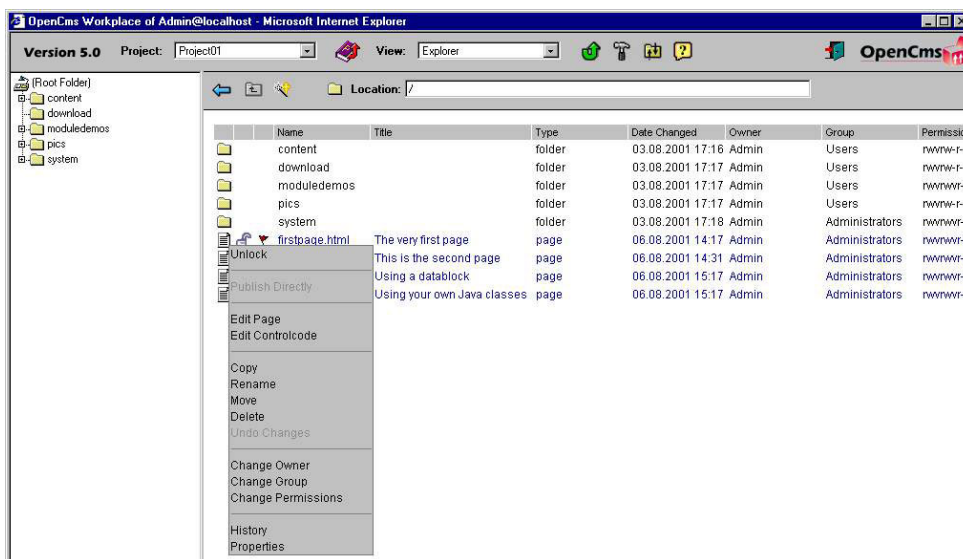


Figure 17.7.: Select *Edit Page* from the context menu to edit the page in the HTML editor

The editor is a WYSIWYG editor that provides standard HTML file editing functionality. You can select the font style and size, insert pictures, and so on. Entering text in the editor inserts it in the new page's body element (figure 17.8).

After you have entered your text, close the editor by clicking on the *Save and Exit* icon to the left. Have a look at the new page by clicking on its name. If you entered the text in the above example, the page will look like figure 17.9.

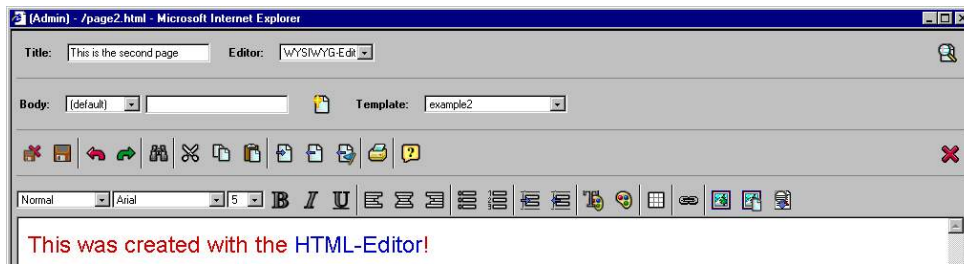


Figure 17.8.: Edit the body with the HTML editor

As you can see, the first part of the text (*Hello, world!*) still comes from the content-template, but the second part, which was created with the help of the WYSIWYG editor, comes from the body element. You can see in the contenttemplate, that the body template is inserted via the ELEMENT-tag:

```
...
Hello, world!<br>
]]><ELEMENT name="body"/>
...
```

Here, the element with the name *body* is inserted after the *Hello, world!* text. The assignment of the element *body* to the body template in the */system/bodies/* directory is made in the page-control file.



Figure 17.9.: The output of the second page

### 17.1.4. Defining the layout in the frametemplate

We will now go a little step further and come closer to the recommended layout of a XML template generated website. Normally, a website is more or less designed like in figure 17.10.

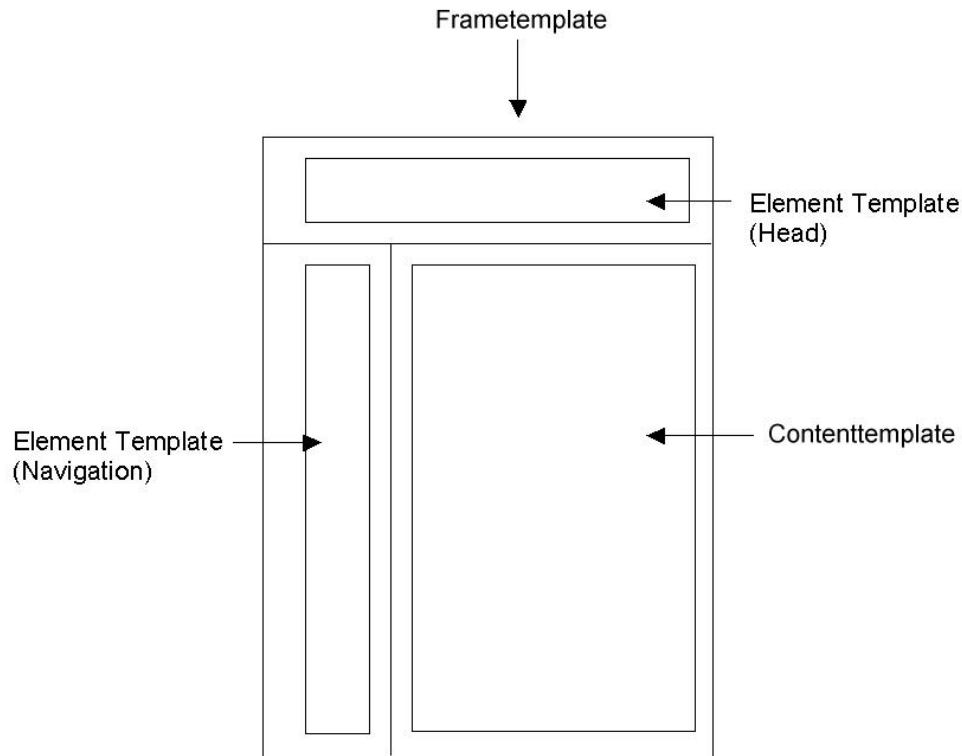


Figure 17.10.: Frametemplate and Elements

A head-section for logos and things like that, an element for the navigation and a content area for the content. This layout can for example be realized with a table defined in the frametemplate and three elements which are inserted. To create a page with this layout, you should first create the frametemplate3 with the table in the

*/system/modules/org.opencms.default/frametemplates/*

directory. The creation can be done similar to the way we did it before. You can then add the following code to the frametemplate3:

```
<XMLTEMPLATE>
<TEMPLATE><![CDATA [
  <HTML>
    <HEAD>
      <TITLE>]]><method name="getTitle"/><![CDATA[</TITLE>
    </HEAD>
```

```

<BODY>
  <TABLE border width="100%" height="100%">
    <TR height="30%">
      <TH colspan=2 width="100%" align="center">
        The Head section
      </TH>
    </TR>
    <TR height="70%">
      <TD width="20%" align="center" valign="top">
        Navigation
      </TD>
      <TD width="80%" align="center">
        ]]><ELEMENT name="contenttemplate"/><![CDATA[
      </TD>
    </TR>
  </TABLE>
</BODY>
</HTML>]]>
</TEMPLATE>
</XMLTEMPLATE>

```

You also need a new mastertemplate in

*/system/modules/org.opencms.default/templates/*

where you use the new frametemplate and an empty contenttemplate:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
  <ELEMENTDEF name="contenttemplate">
    <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
    <TEMPLATE>../contenttemplates/empty</TEMPLATE>
  </ELEMENTDEF>
  <ELEMENTDEF name="frametemplate">
    <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
    <TEMPLATE>../frametemplates/frametemplate3</TEMPLATE>
  </ELEMENTDEF>
<TEMPLATE> <ELEMENT name="frametemplate"/> </TEMPLATE>
</XMLTEMPLATE>

```

As you can see, the text for the head-section and the navigation-section is just inserted directly in the frametemplate, but the content is inserted as an element. The empty contenttemplate just inserts the body element. Please create a new page which uses the

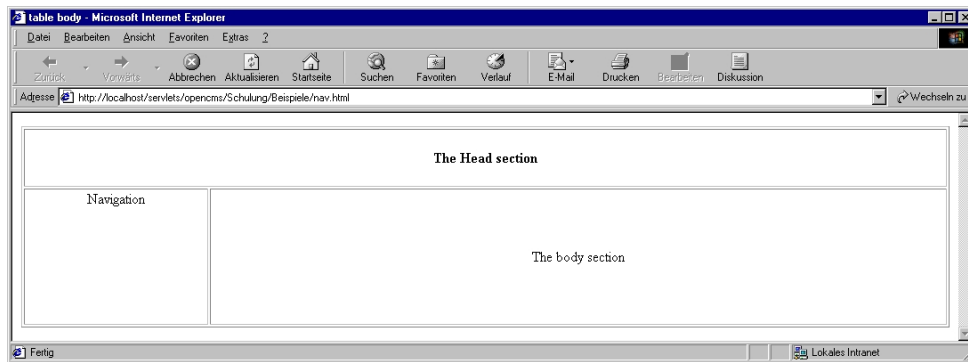


Figure 17.11.: Master and subelementes

new mastertemplate with the wizard and edit the body with the WYSIWYG editor so that you can see some output. Afterwards the output could look like in figure 17.11.

This example should give a little insight in how to handle standard HTML features as e.g. tables and the XML tags together. You can now try to change the layout of the page and edit the frametemplate and contenttemplate.

## 17.2. XML templates and directories

### 17.2.1. The XML structure of a template

XML is a markup language, which is just like HTML defined by SGML and is used to store content in a structured way. The main difference, is that one is able to define his own tags. By that, it is possible to achieve a separation between content and layout. While HTML allows to define how something should be displayed XML allows to define items that belong logically together. For example, it is possible to define blocks of surnames, addresses etc.. XML is a powerfull markup language, but you don't need to know much about it for understanding the XML template mechanism. The XML template mechanism uses XML to deposit HTML fragments single and structured. These fragments can then be fetched specifically and used dynamically. From the technical side of view, there is no difference between the master and the body (sub) templates. Both are XML-files with the same syntax. The difference is more a logical one. The following section explains the structure of a template file. Below is the general framework of a template file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<!-- user-defined data blocks or elementdefintions -->
<TEMPLATE>
  <![CDATA[
```



```
<!-- HTML-code ->
]]> <XML-TAGS> [![CDATA[
<!-- the HTML-code continues here... ->
]]>
</TEMPLATE>
<!-- user-defined data blocks or elementdefinitions ->
</XMLTEMPLATE>
```

The first line determines the type of the document and must be placed at the very beginning of the document. Leading spaces are not allowed. The whole template is enclosed between `<XMLTEMPLATE>` tags. Inside this tag the template is defined between the start `<TEMPLATE>` tag and the end `</TEMPLATE>` tag. HTML code can be placed in the template between the CDATA tags, which starts with `<![CDATA[` and ends with `]]>`. Everything between CDATA tags is ignored by the XML parser and directly streamed to the output. XML tags that are used in the template must be placed outside of the CDATA tags. Examples for XML tags that can be used have been shown in the first examples. You will primarily place process, element, or method tags in your document. Outside the template tags and inside the *xmltemplate* tags you can place data blocks or element definitions.

### 17.2.2. Directory structure

The templates, body templates and element templates etc. are stored in different standard directories of the VFS. If you create a new file of the type *Page* within a project, a page-control file is created in the current directory. Also a body-template is created with the same name as the page in the `/system/bodies/` directory under the same directory structure. For example if you create a page in the folder `/examples` the body-template will be located in `/system/bodies/examples`. If you click on the page-control file and choose *Edit Page* from the context menu, you actually edit the body in the `/system/bodies/` directory with the WYSIWYG editor. You can check this out by looking at both files with the source-code editor after you have changed something, but you should never change the files in the body directory directly.

### 17.2.3. The subdirectories of the content folder

#### - templates:

In the directory

```
/system/modules/org.opencms.default/templates/
```

are the definitions for the different combinations of frame- and contenttemplates. These files have always the same structure as you have seen in the examples so far. The only

things that can be changed are the names of the two templates (only names not the path!). Never add any other html or xml stuff here.

**- frametemplates:**

In the frametemplates the design of the page is build. e.g. it can contain the navigation and an head element. There is always the area where the contenttemplate is included which was defined in the mastertemplate.

**- contenttemplates:**

The contenttemplates defines the content part of the page. It includes the body element and maybe other elements.

**- bodys:**

This folder is for internal use only. It contains the above mentioned body elements.

**- default\_bodies:**

In this folder you can store some bodies which you use often. When you create a new page you can choose one of these bodies to be copied in your new pagebody.

**- internal:**

This directory is not used any more. It is still there for compatibility reasons and its job is now taken over by the elements folder.

**- elements:**

Here you can store your dynamic elements. An element is dynamic if it doesn't use the standard class CmsXmlTemplate. For Example the navigation element.

## 17.3. Details of the proprietary XML template mechanism

### 17.3.1. Using data blocks

Up to here, you got an overview over the template mechanism, but many things which have already been used have not been explained. We will now go one step back and take a look at some earlier examples and the basic structures of the template mechanism.

In this example you will learn how to use data blocks in a template. A data block is a named block of text that can contain XML and HTML code. We will change the earlier example by putting the *Hello, world!* text into a data block with the name *message*. To create the new contenttemplate, create a new text file named contenttemplate2 in the `/system/modules/org.opencms.default/contenttemplates/` directory with the following text:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
  <MESSAGE><![CDATA[Hello world!]]></MESSAGE>
  <TEMPLATE><![CDATA[
    ]]><PROCESS>message</PROCESS><![CDATA[ <br>
    ]]><ELEMENT name="body"/>
  </TEMPLATE>
</XMLTEMPLATE>
```

Note the changes that have been made in the template. The *Hello, world!* text is now enclosed between `<MESSAGE>` and `</MESSAGE>`, which are the start and end tags for the data block. You can select an arbitrary name for the data block, which must be defined outside the template tag and inside the `xmltemplate` tag. In the body section of the HTML code there is now a `<PROCESS>` tag with the text *message* enclosed in it. This process tag will insert the data block with the name *message* in the document. The output of this example is exactly the same as the one of the first example, assuming you haven't added text to the body in the HTML editor.

You can of course edit the body if you want to insert additional text or pictures in the document. Data blocks can be used to insert blocks of XML or HTML code in your document. The content of these blocks can also be defined dynamically in Java classes. The next example will show how to do this.

### 17.3.2. Setting data blocks dynamically

We do already now how data blocks are used in templates. In this example we will define the content of the data block in a Java method. That means we will write our own class



Figure 17.12.: The output of this example

that controls the output of the template and places the text in the data block. To place something in a data block that can dynamically be changed we will use the `<PROCESS>` tag inside of the data block. This functionality will be enclosed in an element which we will place inside the contenttemplate. Note that functionality which requires Java programming should always be placed inside an element and not the frametemplate or contenttemplate itself, because these two templates are always associated with the default `com.opencms.file.CmsXmlTemplate` class and not a userdefined class. Here is the text that the contenttemplate should contain:

```
<?xml version=1.0 encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE>
  <ELEMENT name="example4"/>
  <ELEMENT name="body"/>
</TEMPLATE>
<ELEMENTDEF name="example4">
  <CLASS>CmsExample4</CLASS>
  <TEMPLATE>../elements/example4</TEMPLATE>
</ELEMENTDEF>
</XMLTEMPLATE>
```

As you can see we have included an additional element `example4` inside the contenttemplate with the name `example4`. Inside the `<ELEMENTDEF>` tag we define which Java class and template is used to create the output of this element. Create a file `contenttemplate4` with the above content and make a new mastertemplate which uses this contenttemplate. The next step is to create the element template. Create a new text file `example4` in the folder

```
/system/modules/org.opencms.default/elements/
```

with the following content:

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

```
<MESSAGE>
  <![CDATA[Hello, World ]]>
  <PROCESS>greeting</PROCESS>
</MESSAGE>

<TEMPLATE>
  <PROCESS>message</PROCESS>
</TEMPLATE>

</XMLTEMPLATE>
```

The template contains a data block `<MESSAGE>` with a `<PROCESS>` tag inside it. The `<PROCESS>` tag is used to place the processed input of data blocks in a template. The text `<PROCESS>greeting</PROCESS>` in the template will be replaced by the processed value of the data block with the name `greeting`. The content of this data block is not defined yet, it will be defined dynamically inside the Java class. Next create a new page and select the newly created mastertemplate as the template for this page. The next step is to write the Java class that controls the output of our new template. The new class will be derived from the `com.opencms.template.CmsXmlTemplate` class. This class provides the basic functionality and methods to control template files. In the new class we will override the `getContent()` method. This method creates the output of the template and is invoked automatically by the system.

This is the source code for our new class:

```
import com.opencms.template.*;
import com.opencms.file.*;
import com.opencms.core.*;
import java.util.*;
public class CmsExample4 extends CmsXmlTemplate {
    public byte[] getContent(CmsObject cms,
        String templateFile, String elementName,
        Hashtable parameters,String templateSelector)
        throws CmsException {
        CmsXmlTemplateFile templateDocument =
            getOwnTemplateFile(cms, templateFile, elementName, parameters,
                templateSelector);
        templateDocument.setData("greeting","from Java!");
        return startProcessing(cms, templateDocument,
            elementName, parameters, templateFile);
    }
}
```



Figure 17.13.: Output generated using the new Java class

The first method we use is *getOwnTemplateFile()*. This method parses the XML template and creates an object of *com.opencms.template.CmsXmlTemplateFile* which contains the templatefile as a DOM-object and has methods to manipulate the data blocks of the template. This method call is a standard call that will always be used inside the *getContent()* method to access the template file. The next method we use is the *setData()* method. This method sets a data block in the template with a given String value. The first parameter of method *getData()* takes the name of the data block and the second its value. As you can see we fill a data block named *greeting* with the value "from Java". This means that now if this data block is used in the template it will have the value "from Java". The final method call in the method *getContent()* invokes the method *startProcessing()* which results in the processing and generating of the output of the whole template. The return value of this method is the output of the template and will be returned by the *getContent()* method. The calls to method *getOwnTemplateFile()* and *startProcessing()* are used in almost every case when writing a *getContent()* method because they provide a rather basic functionality that allows manipulating and processing of the template.

You can compile the class and place the generated *Example4.class* file in the subdirectory *WEB-INF/classes/* of your webapp. To load the new class you have to restart the webapplication. If everything works fine you should see something like in figure 17.13.

### 17.3.3. Using user-defined methods

In the last example we saw one way of creating dynamic content in a template. Another way is to use user-defined methods in the template. You have already seen how to use the *getTitle()* method in a template in the second example. If you want to write your own methods, you can derive a class from the existing template class *com.opencms.template.CmsXmlTemplate* and add your own methods.

We will write a method that returns the string *Hello, World from Java!* and give it the name *getHello()*. The return value of the method will be placed inside the template where we call the method with the *<METHOD>* tag. You should create another content-template which includes an element *example5* and a new mastertemplate which uses this contenttemplate. You have to create a new element template *example5*, or if you want to

use the `contenttemplate` and `mastertemplate` of the previous example you can also just replace the element of the previous example with the following content:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE>
    <METHOD name="getHello"/>
</TEMPLATE>
</XMLTEMPLATE>
```

A new Java class *CmsExample5* defines the method *getHello()*:

```
import com.opencms.template.*;
import com.opencms.file.*;
import com.opencms.core.*;
import java.util.*;
public class CmsExample5 extends CmsXmlTemplate {
    public Object getHello(CmsObject cms, String tagcontent,
        A_CmsXmlContent doc, Object userObject) {
        return "Hello, World from Java!";
    }
}
```

Note that the signature of user-defined methods is always the same and cannot be changed. To use the class, compile it and place it in the subdirectory *WEB-INF/classes/* of your webapp. Create a new page that uses the new `mastertemplate`.

The output of the new template will be the same as that of the previous example: it displays the string *Hello, World from Java!* on the screen. The only difference is the procedure we followed to accomplish the task of inserting something in the document. In the previous example we set a datablock with the *setData()* method and now we used the a new user-defined method *getHello()*.

#### 17.3.4. The relationship between templates, page-control files, and Java classes

For every new page that is created with an XML template the system automatically creates a *page-control* file. This page-control file determines the `mastertemplate` that is used by the new page and contains the element definition of the body element. Examples for page-control files have already been shown before. Below is the page-control file for an earlier example:

```
<?xml version= "1.0" encoding="ISO-8859-1"?>
<page>
  <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
  <MASTERTEMPLATE>/system/modules/org.opencms.default/tem-
plates/template5</MASTERTEMPLATE>
  <ELEMENTDEF name="body">
    <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
    <TEMPLATE>/system/bodies/page5.html</TEMPLATE>
  </ELEMENTDEF>
</page>
```

The name of the mastertemplate file is enclosed between the `<MASTERTEMPLATE>` tags. The `<CLASS>` tag above it determines the Java class that controls the mastertemplate. This class is the generic `com.opencms.template.CmsXmlTemplate` class and is always used to control the mastertemplate. The `<ELEMENTDEF>` tag defines the class and template used for the body element. The body element is the one that contains the HTML-code generated with the WYSIWYG editor. An element definition must be provided for every subelement. The body element can be placed inside a template with the element tag as follows:

```
<ELEMENT name="body"/>
```

You can place other subtemplates in your document in the same way you placed the body template in it, by inserting a `<ELEMENT name="aName"/>` tag and element definition in your document. The name for the new element can be freely defined. The controlling Java class for the body template is always the class `com.opencms.template.CmsXmlTemplate` because a body element does not have any user-defined functionality.

### 17.3.5. Process, Method and Element Tags

The *process*, *method* and *element* tags enable you to insert dynamic content in your template. They have already been used in the first examples. Their syntax and details of their usage will be discussed in the following sections.

#### The Process Tag

The *process* tag can be used to insert the content of data blocks in your document. The data blocks can be defined in the document or set by Java classes. We will explain how a process tag works by taking a look at the example of section 17.3.1 on page 131. Here is the content of the contenttemplate for this example:

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



```

<XMLTEMPLATE>
<MESSAGE>
  <![CDATA[Hello, World ]]>
  <PROCESS>greeting</PROCESS>
</MESSAGE>

<TEMPLATE>
  <PROCESS>message</PROCESS>
</TEMPLATE>
</XMLTEMPLATE>

```

There are two process tags in this template: the first inside the data block *message* and the second inside the template tag. The text inside the process tag defines the data block that will be processed. In this example two data blocks will be processed, one named *greeting* and one named *message*. The *message* data block is defined in the template document. The data block *greeting* is not defined in the document, but set in the Java class that is associated with the template. The *setData()* method was used to accomplish this:

```
templateDocument.setData("greeting", "from Java!");
```

Calling the method *setData* here has the same effect as adding a datablock of the form:

```
<greeting><![CDATA[from Java!]]></greeting>
```

into the template. The difference is that it is set dynamically by a Java method and not statically inside the template.

To show that it is possible to set the data block dynamically to different values inside a Java class, the next example inserts a randomly selected link into a template. The Java program randomly selects one of two data block values for a link's target attribute.

Create a new contenttemplate which includes a new element with the following content:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<LINK1>www.opencms.org</LINK1>
<LINK2>www.yahoo.de</LINK2>
<TEMPLATE>
<![CDATA [
<a href="http://"]><PROCESS>link</PROCESS><![CDATA [">
Where will we go?</a>
]]>
</TEMPLATE>
</XMLTEMPLATE>

```

The template defines a hypertext link where the value of the target attribute is defined by a data block named *link*. The value of the data block is set inside the template with the `<PROCESS>` tag. The value of the data block *link* will be set randomly in a Java class. This is the source code for the new class:

```
import com.opencms.template.*;
import com.opencms.file.*;
import com.opencms.core.*;
import java.util.*;

public class CmsExample6 extends CmsXmlTemplate {

    public byte[] getContent(CmsObject cms,
        String templateFile, String elementName,
        Hashtable parameters, String templateSelector)
        throws CmsException {
        String linkTarget;
        linkTarget = "LINK"
            + String.valueOf((int)(Math.random()*2+1));
        CmsXmlTemplateFile templateDocument =
            getOwnTemplateFile(cms, templateFile, elementName,
                parameters, templateSelector);
        templateDocument.setData("link",
            templateDocument.getDataValue(linkTarget));
        return startProcessing(cms, templateDocument, elementName,
            parameters, templateFile);
    }

    public CmsCacheDirectives getCacheDirectives(CmsObject cms,
        String templateFile, String elementName,
        Hashtable parameters, String templateSelector) {
        return new CmsCacheDirectives(false);
    }
}
```

The data block that is used for the link's target is randomly selected in the line:

```
linkTarget = "LINK"
    + String.valueOf((int)(Math.random()*2+1));
```

This statement produces a string *"Link1"* or *"Link2"* based on the number that is randomly created by the expression  $(\text{Math.random}()*2+1)$ . The value of the data block *link* is defined in the following expression:

```
templateDocument.setData("link",
templateDocument.getDataValue(linkTarget));
```

The data block named *link* is defined with the method *setData()* and the content of the data block is the value of the data block that will be fetched with *getDataValue(linkTarget)*. Because the variable *linkTarget* has either the value "Link1" or "Link2" the link will be defined with the content of data block "Link1" or "Link2". To ensure that the link is refreshed every time the page is requested, the class overrides the method *getCacheDirectives()*. This method controls the way XML template elements are cached. The implementation in the superclass *com.opencms.template.CmsXmlTemplate* uses the caching mechanism and would prevent the element from being refreshed every time it is requested. The details of the element caching mechanism are explained in section 17.3.9 (page 155). At this point it is only necessary to know that if an element should be dynamically created on every request the method *getCacheDirectives()* has to be implemented in the template class in the way we did in this example.

The next example will show how to create tables or lists using the *process* tag. You can use the *process* tag to set the data entries in tables in Java classes. The table will be dynamically built in the Java class by producing a string by concatenating table rows. The layout of the table is defined inside the template. The element template could look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE>
<![CDATA[ <H1>Animal list</H1>
<table border=1>
]]>
    <process>tablehead</process>
    <process>list</process>
<![CDATA[
</table>
]]>
</TEMPLATE>
<tablehead>
<![CDATA[
    <tr>
        <td>Nr</td>
        <td>animal</td>
        <td>owner</td>
    </tr>
]]>
</tablehead>
```

```
<row>
<![CDATA[
  <tr>
    <td>]]><process>number</process><![CDATA[</td>
    <td>]]><process>name</process><![CDATA[</td>
    <td> ]]]><process>owner</process><![CDATA[</td>
  </tr>
]]>
</row>
</XMLTEMPLATE>
```

The template defines a table and two data blocks. The data block *tablehead* defines a static heading for every column and will be inserted in the table by the *process* tag. The table content will be created by a Java class and will be set as the data block *list*. The layout of the table's rows is defined by the data block *row*. The Java class will set the value for *number*, *name* and *owner* for every row and concatenate the values of the resulting rows and insert them in the document as the data block *list*. Here is the source code of the Java class:

```
import com.opencms.file.*;
import com.opencms.template.*;
import java.util.*;

public class CmsExample7 extends CmsXmlTemplate {
public byte[] getContent(CmsObject cms, String templateFile,
    String elementName, Hashtable parameters,
    String templateSelector) throws CmsException {
String[] owners =
    "Martin","Thomas","Andreas","Bill","Michael","Doris";
String[] animals =
    "cat","dog","mouse","rat","bird","snake";
CmsXmlTemplateFile template =
    getOwnTemplateFile(cms,templateFile,elementName,
    parameters,templateSelector);
String list = "";
for (int i=0; i < animals.length; i++) {
    template.setData("number", i+"");
    template.setData("name",animals[i]);
    template.setData("owner",owners[i]);
    String row = template.getProcessedDataValue("row");
    list += row;
}
}
```

```

        template.setData("list",list);
        return startProcessing(cms, template, elementName,
            parameters, templateSelector);
    }
}

```

The Java class overrides the method *getContent()* that creates the template's output. Two string arrays are defined in this method that hold the entries for the owners and the animals that will be inserted in the table. The rows are generated by setting the data blocks *number*, *name* and *owner* and fetching the processed data block *row* (this means the process tags have been replaced by the corresponding values of the data blocks). The result is concatenated in a string and set as the data block *list*. The final call to *startProcessing()* starts the generation of the template's output. The last example can be modified to produce a list instead of a table. The Java class can be left unchanged and reused to create the list. This shows that it is possible to change the layout in the template without having to change the Java class. This is the template that will generate a list instead of a table:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE>
<![CDATA[ <H1>Animal list 2</H1>
<ul>
    ]]>
    <process>list</process>
    <![CDATA[
</ul>
]]>
</TEMPLATE>
<row>
<![CDATA[
    <li>
        ]]><process>name</process><![CDATA[,
        ]]><process>owner</process><![CDATA[
    </li>
]]>
</row>
</XMLTEMPLATE>

```

The template places the data block *list* in an unordered list. The row's layout has been changed to list entries. The name and the owner are set in the Java class and the list is generated in the same way that it was in the last example. Note that the data block

number is not used by the template, but still set in the Java class. Although this is unneeded, it is harmless because the data block is set without being processed.

## The Method Tag

The method tag is used to insert the output of a method in a document. You can define your own method inside a class derived from *com.opencms.template.CmsXmlTemplate* or use the standard methods in this class. In section 17.3.3 (page 134) we created the new method *getHello()* to show how method tags are used. In general, a method is used in a template by inserting a `<METHOD>` tag. A method can take parameters that are passed to the method by specifying them in the method tag.

The syntax for the method tag:

- `<method name="test"/>`
- `<method name="test2">parameter(s)</method>`

The name attribute of the `<METHOD>` tag specifies the method to call and is case sensitive. A parameter can be passed to the method as a string value enclosed by the *method* tag. For methods that don't need any parameters, the end tag `</METHOD>` can be omitted and the tag can be closed by entering `/>`. When parameters are passed as a string to a method, the method tag has to be closed with the corresponding end tag (`</METHOD>`). The previous examples that generated a table and a list using a process tag can easily be modified to use a method instead. The next example will show how to generate a table with a new method *getTable()*. Here is the element template:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE> <![CDATA[ <H1>Animal list</H1>
<table border=1>
]]>
    <process>tablehead</process>
    <method name="getTable"/>
<![CDATA[
</table>
]]>
</TEMPLATE>
<tablehead>
<![CDATA[
    <tr>
    <td>Nr</td>
    <td>animal</td>
    <td>owner</td>
```

```

</tr>
]]>
</thead>
<row>
<![CDATA[
    <tr>
        <td>]]><process>number</process><![CDATA[</td>
        <td>]]><process>name</process><![CDATA[</td>
        <td> ]]]><process>owner</process><![CDATA[</td>
    </tr>
]]>
</row>
</XMLTEMPLATE>

```

To insert the table in the document using a method rather than a process tag, we modified the line

```
<process>list</process>
```

to

```
<method name="getTable"/>
```

The table will be generated by the method. This is the Java source code for the class needed for this example:

```

import com.opencms.template.*;
import com.opencms.file.*;
import com.opencms.core.*;
import java.util.*;

public class CmsExample9 extends CmsXmlTemplate {
    public Object getTable(CmsObject cms, String tagcontent,
        A_CmsXmlContent doc, Object userObject)
        throws CmsException {
        String[] owners =
            "Elmar","Randolf","Sandra","Geoffrey","Claudia","Doris";
        String[] animals =
            "spider","eagle","lion","cheetah","scorpion","snake";
        CmsXmlTemplateFile template = (CmsXmlTemplateFile)doc;
        String list = "";
        for (int i=0; i < animals.length; i++) {

```

```
        template.setData("number",i+"");
        template.setData("name",animals[i]);
        template.setData("owner",owners[i]);
        String row = template.getProcessedDataValue("row");
            list += row;
    }
    return list;
}
}
```

The names of the owners and the animals have been changed to emphasize that this table is not the same as the one that was produced by using the process tag. The code that produces the table's content is very similar to the one using a process tag for insertion of the data. The main difference is that inside a user-method you don't have to call the methods *getOwnTemplateFile()* and *startProcessing()* like in the *getContent()* method in the previous example. A user method can access the template file via the parameter *doc* which is of type *com.opencms.template.A\_CmsXmlContent* and can be casted to *com.opencms.template.CmsXmlTemplateFile*.

To introduce another example of how methods can be used, we will take a look at a method that takes a color name as parameter and returns the hexadecimal color value that specifies the RGB value of the color. The method will be used in the template as follows:

```
method name="color">red</method>
```

The parameter will be used in the method to return the appropriate hexadecimal number that represents the color. This number will be inserted in the template to define the color that is used. This is the source code for the new Java class including the method:

```
import com.opencms.core.*;
import com.opencms.file.*;
import com.opencms.template.*;
import java.util.*;

public class CmsExample10 extends CmsXmlTemplate {
    static Hashtable colors = new Hashtable();
    static {
        colors.put("black", "000000");
        colors.put("maroon", "800000");
        colors.put("green", "008000");
        colors.put("olive", "808000");
        colors.put("navy", "000080");
        colors.put("purple", "800080");
    }
}
```



```
        colors.put("teal", "008080");
        colors.put("gray", "0808080");
        colors.put("silver", "C0C0C0");
        colors.put("red", "FF0000");
        colors.put("lime", "00FF00");
        colors.put("yellow", "FFFF00");
        colors.put("blue", "0000FF");
        colors.put("fuchsia", "FF00FF");
        colors.put("aqua", "00FFFF");
        colors.put("white", "FFFFFF");
    }
    public Object color(CmsObject cms, String tagcontent,
        A_CmsXmlContent doc, Object userObject)
        throws CmsException {
        String color = null;
        if (tagcontent == null || tagcontent.equals("")) {
            color = (String)colors.get("black");
        }else {
            color = (String)colors.get(tagcontent);
            if (color == null) {
                color = (String)colors.get("black");
            }
        }
        return "#" + color;
    }
}
```

As you can see, a hash table is used to store the names of the colors and the corresponding hexadecimal values. The method checks if a tagcontent was passed to the method (The tagcontent is the string between the start and end method tag). If a tagcontent is passed, the method tries to get the appropriate color value. If the name of a color is passed that is not contained in the hash table or if no tagcontent is passed, the color black is used. Of course this example is of little practical value because you can use these colornames directly in an HTML page, but this should only be considered as an example how a method works and not a serious suggestion what to do with a method.

## Element Tag and Element Definition

The element tag is used to insert subelements (subtemplates) in a template. The element definition defines the template and Java class that are used for the element. The element tag must have the following syntax:

```
<ELEMENT name="aName"/>
```

where *aName* must be replaced with the name of the element. A valid element definition has the following structure:

```
<ELEMENTDEF name="aName">
<TEMPLATE>absolutePathToTheTemplateFile</TEMPLATE>
<CLASS>nameOfTheControllingClass</CLASS>
<PARAMETER name="param1">value1</PARAMETER>
<PARAMETER name="param2">value2</PARAMETER>
</ELEMENTDEF>
```

In the element definition, the template file and the controlling Java class of the element are specified in the tags `<TEMPLATE>` and `<CLASS>`. The `<PARAMETER>` tags are optional and used to pass parameters to the controlling Java class. These parameters are stored in a hash table together with the parameters that are appended to a URL. Note that the element's name followed by a dot will be added as a prefix to the parameter's name, so if you specify a parameter *param1* in the element definition it will be accessible under the name *elementName.param1*.

The element definition is defined either in the document in which the element is inserted, or in the page-control file. If the element definition is missing, the element definition of the element *body* will be used. The element definition can also be overridden in the page-control file. If an element definition is specified in both the file in which the element is inserted and in the page-control file, the element definition stored in the page-control file takes precedence. This means that the page-control file can be used to override a default element definition. The next example will show the effect of overriding element definitions. First we need to create a template that contains elements that use templates from the previous examples. This is the contenttemplate for the new example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE>
<![CDATA[
<TABLE border width="100%" height="100%">
  <TR height="30%">
    <TD width="20%"><IMAGE src="
      ]]><method name="getServletPath"/><![CDATA[
      pics/opencmslogo.gif" align="center">
    </TD>
    <TD width="80%" align="center"
      ]]><element name="hello"/><![CDATA[
```

```

        </TD>
    </TR>
    <TR height="70%">
        <TD width="20%" align="center" valign="top">
            ]]><element name="random"/><![CDATA[
        </TD>
        <TD width="80%" align="center">
            ]]><element name="body"/><![CDATA[
        </TD>
    </TR>
</TABLE>
]]>
</TEMPLATE>
<ELEMENTDEF name="hello">
    <TEMPLATE>/system/bodies/page4.html</TEMPLATE>
    <CLASS>CmsExample4</CLASS>
</ELEMENTDEF>
<ELEMENTDEF name="random">
    <TEMPLATE>/system/bodies/page6.html</TEMPLATE>
    <CLASS>CmsExample6</CLASS>
</ELEMENTDEF>
</XMLTEMPLATE>

```

The template defines a table with two rows and two columns. A picture of the OpenCms logo is placed in the first column of the first row. An element called *hello* is placed in the second column of the first row. The corresponding element definition specifies this element as the one we used in section 17.3.2 (page 132): it displays the string *Hello, World from Java!* on the screen. The first column of the second row contains an element called *random*. This element is the random link that we created on page 137. The body element is inserted in the second column of the second row.

The effect of overriding an element definition will be shown in the next example. An element definition is overridden by adding another element definition in the page-control file. This is the content of the original page-control file:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<page>
    <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
    <MASTERTEMPLATE>/system/modules/org.opencms.default/tem-
plates/template8</MASTERTEMPLATE>
    <ELEMENTDEF name="body">
    <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>

```

```
<TEMPLATE>/system/bodies/page_11.html</TEMPLATE>
</ELEMENTDEF>
</page>
```

To change the content of the *hello* element you can add a definition for the *hello* element to the page-control file:

```
<?xml version= "1.0"?>
<page>
  <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
  <MASTERTEMPLATE>/system/modules/org.opencms.default/tem-
plates/template8</MASTERTEMPLATE>
  <ELEMENTDEF name="body">
    <CLASS>com.opencms.template.CmsXmlTemplate</CLASS>
    <TEMPLATE>/system/bodies/page_11.html</TEMPLATE>
  </ELEMENTDEF>
  <ELEMENTDEF name="hello">
    <TEMPLATE>/system/bodies/page6.html</TEMPLATE>
    <CLASS>CmsExample6</CLASS>
  </ELEMENTDEF>
</page>
```

Because this element definition will be used instead of the one that is defined in the contenttemplate, the random link will be displayed instead of the *Hello, World from Java!* text.

If an element definition is missing, the element definition of the body element in the page-control file is used. This definition will also be used for missing parts of an element definition. If the element's class or template is not specified, it will be taken from the body element definition in the page-control file. You can see this when deleting the definition for the *hello* element in the contenttemplate. Doing this will display the content of the body element instead of the *Hello, World from Java!* text.

The next example will show how parameters that are specified in an element definition are used. The previous example has been modified and parameters added to the definition of the element *hello*. The Java class and the element's template have been modified to insert the parameters. The parameters are used to specify the element's foreground and background colors. This is the new definition of the *hello* element:

```
<ELEMENTDEF name="hello">
<TEMPLATE>/system/bodies/page_14_param.html</TEMPLATE>
<CLASS>CmsExample14</CLASS>
<PARAMETER name="foreground">blue</PARAMETER>
<PARAMETER name="background">green</PARAMETER>
</ELEMENTDEF>
```

This element definition is located in the `contenttemplate`. Two parameters, *foreground* and *background*, define the colors of the element. These color parameters are evaluated in the Java class that controls the element's template. This is the source code for this class:

```
import com.opencms.template.*;
import com.opencms.file.*;
import com.opencms.core.*;
import java.util.*;
public class CmsExample14 extends CmsXmlTemplate {
public byte[] getContent(CmsObject cms,
    String templateFile, String elementName,
    Hashtable parameters,String templateSelector)
    throws CmsException {
    CmsXmlTemplateFile templateDocument =
    getOwnTemplateFile(cms, templateFile, elementName,
        parameters,templateSelector);
    templateDocument.setData("greeting", "from Java!");
    templateDocument.setData("fgcolor",
        (String)parameters.get("hello.foreground"));
    templateDocument.setData("bgcolor",
        (String)parameters.get("hello.background"));
    return startProcessing(cms, templateDocument, elementName,
        parameters, templateFile);
    }
}
```

The parameters are stored in a hash table that will be passed to the classes `getContent()` method. To access an element's parameters you have to put the element name in front of the parameter name followed by a dot. The class sets two data blocks *fgcolor* and *bgcolor*, which are inserted in the template with the process tag.

Parameters can be passed to templates by appending them to a page's URL. However, the URL always calls the mastertemplate even if the data is intended for a subtemplate. This is rectified by specifying the template that is to be called. Below are two examples:

1. Special parameter (e.g. *datafor*) that specifies the subtemplate:  
`http://www...de?datafor=elem1&param1=hello`
2. Renaming a parameter and giving it a prefix that denotes the parameter's target:  
`http://www...de?elem1.param1=hello`

Variant 2 has the advantage of being able to specify more as one subtemplate as the target. The parameters are stored in the parameter hash table in the form:

*ElementName.parameterName*

Both of the variants are implemented so that the best method can be selected on a case-by-case basis.

### 17.3.6. Stylesheets

Stylesheets can be used in XML templates just like in standard HTML. There is only one point concerning stylesheets which is worth to be mentioned here in the documentation. This is that stylesheet information can be used in the WYSIWYG editor as well. This way WYSIWYG-behaviour can be realized even when using stylesheets. There is only one change within the frametemplate necessary to achieve this behaviour. The special tag `<stylesheet>` has to be inserted and the path to the stylesheet-file has to be determined. Here is an example of a frametemplate which includes stylesheet information and allows the WYSIWYG editor to use them as well.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<stylesheet>system/modules/.../resources/style.css</stylesheet>
<TEMPLATE>
<![CDATA[
<html>
<head>
  <title>]]><method name="getTitle"/><![CDATA[</title>
  <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-1">
  <link rel="stylesheet" href="]]><method
    name="getServletPath"/>
  <![CDATA[system/modules/.../resources/style.css">
</head>
<body ...
```

As you can see, the file *style.css* is referenced twice. The first reference is inside the tag `<stylesheet>`. This is the specification which makes the stylesheet information useable within the WYSIWYG editor. The second reference is just a common way a stylesheet file can be used in HTML. This specification makes the stylesheet information useable in the HTML-page.

### 17.3.7. Javascript Blocks

With XML templates, Javascript code is easily inserted in templates, in particular in the frametemplate. Although one shouldn't rely on Javascript when creating a website, it

is useful to know how it can be inserted in documents. Javascript blocks are inserted in the frametemplate by defining a template with the key name *script*. This template can be inserted as an element in the header section of the frametemplate. The element definition of the Javascript element just specifies the templateselector to use (the name of the template). The XML templates use the body element's class and template file if they are not specified in the element definition. The next example template will show how to insert Javascript code in a frametemplate:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE>
<![CDATA[
<HTML>
<HEAD>
<TITLE>]]><method name="getTitle"/><![CDATA[</TITLE>
]]><element name="javascript"/><![CDATA[
</HEAD>
<BODY>
]]><element name="body"/><![CDATA[
</BODY>
</HTML>
]]>
</TEMPLATE>
<ELEMENTDEF name="javascript">
<TEMPLATESELECTOR>script</TEMPLATESELECTOR>
</ELEMENTDEF>
</XMLTEMPLATE>
```

The Javascript code is defined in one file together with the body template as a template with the special name "script" as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE>
<!-- body template -->
</TEMPLATE>
<TEMPLATE name="script">
<!-- Javascript code ... -->
</TEMPLATE>
</XMLTEMPLATE>
```

### 17.3.8. Methods of the proprietary XML template mechanism

The class *com.opencms.template.CmsXmlTemplate* defines a number of basic methods that can be used when creating a template. The following section describes the functionality they provide and how these methods are implemented.

All methods of the XML template mechanism have a common signature (sequence and types of the method's parameters). The methods common signature is:

```
public Object methodName(CmsObject cms, String tagcontent,
    A_CmsXmlContent doc, Object userObject)
    throws CmsException {
    // statements;
}
```

The parameters that are passed to the method provide access to essential XML template resources:

- cms (type *com.opencms.file.CmsObject*)  
provides access to the user properties as well as to the files stored in the OpenCms system and their properties. The API that is provided by the *CmsObject* will be described later in this document.
- tagcontent (type *java.lang.String*)  
contains the string that is enclosed in the method tag. This string is the parameter that can be passed to the method inside a template. If you have a method that doesn't need a parameter, the tagcontent can be ignored.
- doc (type *com.opencms.template.A\_CmsXmlContent*)  
represents the template file. In the case of a normal *XmlTemplate* this variable is a reference to a *com.opencms.template.CmsXmlTemplateFile*. This reference provides access to the template's elements and data blocks.

There are two other types of template files that are derived from the abstract *com.opencms.template.A\_CmsXmlContent* class. They were designed to be used with special types of templates, for example templates that are used to build the workplace or HTML forms.

These classes are *com.opencms.workplace.CmsXmlWpTemplateFile* and *com.opencms.defaults.CmsXmlFormTemplateFile*.

- userObject (type *java.lang.Object*)  
this parameter contains the parameters of the request in form of an object of type *java.util.Hashtable*.



Method	Description
<i>getTitle()</i>	Is used to refer to the title of a page in a template. The title, defined as a property of a page when it is being created, can easily be changed by selecting Properties from the context menu of the page's file.
<i>getKeywords()</i>	Returns the keywords that were entered in the create new page dialog. This method can be used inside a template to put the keywords in a meta-tag to support search-engines.
<i>getDescription()</i>	Returns the description of the page that was entered in the create new page dialog.
<i>getProperty()</i>	Takes a property name as parameter and returns the value of this property for the requested page.
<i>getRequestIp()</i>	Returns the IP address of the machine that requested a page.
<i>getSessionId()</i>	Provides a means of receiving an identification number that is used to identify a special user. Session Ids are defined by using cookies. Cookies are short text files that are stored on the client and can be used to identify users and keep track of their actions.
<i>getServletPath()</i> deprecated	Returns the relative path of the directory in which the servlets reside on the server. This method is used to write templates that are independent of the servlet path. The servlet path can be changed without affecting the template files when this method is used to refer to the path. In times of static export you should not use this method any longer. Instead you should use the link tag mentioned in the section about the static export.
<i>getPathUri()</i>	Returns the path of the requested file. To create proper links you should use the link tag.
<i>getFileUri()</i>	Returns the file name of the requested file. To create proper links you should use the link tag.
<i>getUri()</i>	Returns the link to the requested file parsed through the <code>getLinkSubstitution</code> method. If needed you can submit parameters in the tagcontent (like <code>"cmsframe=main&amp;id=7"</code> )
<i>getUriWithParameter()</i>	Returns the link to the requested file parsed through the <code>getLinkSubstitution</code> method. If needed you can submit parameters in the tagcontent. These parameters will be added to the parameters in the requestcontext.
<i>getStyleSheet()</i>	Is useful for inserting the appropriate style sheet definition for different browsers in a template. Two data blocks with the predefined names <code>&lt;stylesheet-ie&gt;</code> and <code>&lt;stylesheet-ns&gt;</code> have to be used in the template to specify the name of the style sheet files for the browser. The <i>getStyleSheet</i> method returns the path to the appropriate style sheet based on the browser that is used by the client. By using this method and different style sheets for Netscape Navigator and Windows Explorer, the appropriate style sheet is used if the method manages to detect the user's browser.
<i>getQueryString()</i> deprecated	Returns a string that contains the parameters that have been passed with the URL when the page was requested. You should use the method <code>getUriWithParameters()</code> .
<i>getFrameQueryString()</i> deprecated	Provides access to the parameters that have been passed with the URL in a frame. In general, the parameters are only accessible in the file that was requested and contains the frames. Each frame usually contains a different HTML page. The parameters that are passed to the parent file are not directly accessible in the frames. You should use the method <code>getUriWithParameters()</code> .

Method	Description
<i>parameters()</i>	Is used for debugging. It returns all of the parameters for the current template file that are passed with the URL. It also returns additional parameters for internal use that contain the name of the element, the complete file name of the body element, and the controlling Java class for this element.
<i>getFrameTarget()</i>	Is used to specify the target for a link in a template that works both in a version with and without frames. The parameter <i>cmsframe</i> is passed with the URL and used to select the version that does not use frames by setting it to <i>plain</i> . If <i>cmsframe</i> is passed as the parameter and is equal to <i>plain</i> , the target will be set to the empty string, meaning that the method returns <code>target=""</code> . If <i>cmsframe</i> is not passed as the parameter with the URL, the method returns the string <code>target="tagcontent."</code> The <i>tagcontent</i> is the parameter that is passed to the method by defining it in the method tag. If the <i>tagcontent</i> parameter is not passed, the method returns <code>target="_top"</code> for the frames version. This is a syntax example for the method: <code>&lt;method name="getFrameTarget"&gt;main&lt;/method&gt;</code> In this case the method returns the string: <code>target="main"</code> if the frame version is used, and <code>target=""</code> for the version that does not use frames. <b>Note:</b> this method is only useful if you are creating both a frames and a non-frames version of a website.

Table 17.1.: XML template methods

Table 17.1 describes the standard methods that are defined in the *com.opencms.template.CmsXmlTemplate* class.

As we have already seen, there are a number of basic methods that are used to define or get a template's data blocks. They are usually used in user-defined methods or in the *getContent()* method of a class.

Table 17.2 provides a short overview of the most important of these methods.

Method	Description
<b>CmsXmlTemplateFile Methods</b>	<b>CmsXmlTemplateFile Methods</b>
String <i>getDataValue</i> (String)	Returns the text and CDATA content of a data block.
String <i>getProcessedDataValue</i> (String)	Returns the text and CDATA content of a processed data block.
void <i>setData</i> (String, String)	Creates a data block with the passed string content in the data block hash table.
byte[] <i>getContent</i> (...)	Is invoked automatically to create the content of a template file.
byte[] <i>startProcessing</i> (...)	Starts creating the content of a template file.

Table 17.2.: Important methods for template programming

### 17.3.9. Using the element cache for XML templates

The creation of dynamic content with XML templates is a relatively performance and time consuming task (especially because the mediocre runtime performance of the XML template mechanism). Therefore there is an element caching mechanism provided to speed up the delivery of pages created with the XML templates.

*Please note:* The element cache is only required if you use XML templates, in case you use JSP based templates you do not need to bother about it. For JSP templates there is a new cache available called the FlexCache. The FlexCache is described in detail in the interactive part of the documentation.

If the element caching is activated the content of elements will only be created dynamically when a specific element is used the first time. At the second request the content of the element will be fetched from the element cache. The caching only affects resources in the online project, resources in offline projects are not cached.

#### Activating the element cache

The element cache is activated by setting the appropriate parameters in the configuration-file `opencms.properties`:

```
# Element cache parameters
#####
elementcache.enabled = true
elementcache.uri = 10000
elementcache.elements = 50000
elementcache.variants = 100
```

These entries activate the element cache and specify how many uris and elements will be cached. If the maximum number of objects in the cache is reached, the oldest objects in the cache will be removed. The content of an element can depend on different parameters like the current user or url parameters passed with the request. The element cache is able to cache these variants of an element. The parameter `variants` specifies a maximum number of variants of an element that will be cached. Normally an element has only few variants, but in cases when many variants of an element can be generated, the `variants` parameter prevents the creation of too many objects in the cache.

#### Clearing the element cache

The element cache can be cleared by selecting the icon `clear elementcache` from the administration view (see figure 17.14).

This administration-icon is only visible for administrators and only if the element cache has been enabled in the `opencms.properties`. When this point is selected a window will

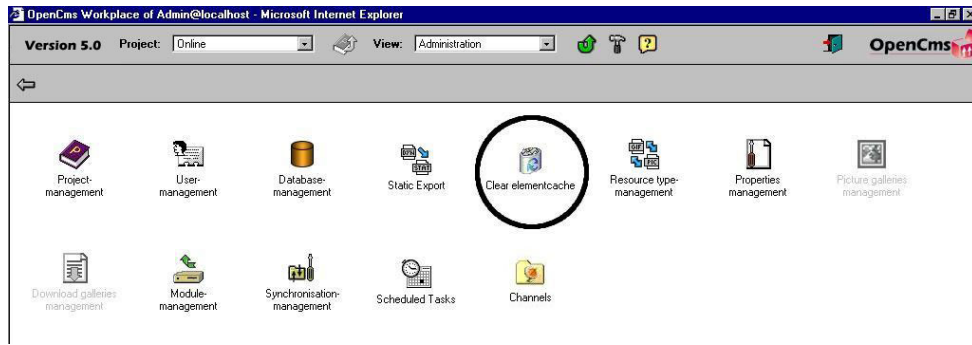


Figure 17.14.: Clear elementcache icon

pop up and show the number of uris and elements that are actually in the cache. You can then select to clear the cache or cancel the operation.

### Using the element caching in your own template classes

The class *CmsCacheDirectives* provides the opportunity for the module developer to specify how pages will be cached with the element cache. Every template class has to implement the method *getCacheDirectives()*. Here is a sourcecode example how this could be done:

```
public CmsCacheDirectives getCacheDirectives(CmsObjet cms,
      String templateFile, String elementName,
      Hashtable parameters, String templateSelector) {
    // create a CmsCacheDirectives object:
    CmsCacheDirectives cd =
        new CmsCacheDirectives(true, false, false, false, true);
    // adding the group to the cache key:
    cd.setCacheGroups(true);
    // return the CacheDirectives:
    return cd;
}
```

(The methods *isCacheable()* and *getKey()* that were used to control the caching for a preliminary version of the element cache are no longer necessary and will not be supported in future releases.) Inside this method an object of type *CmsCacheDirectives* is created and returned. This object contains information how the element should be cached. So far the class *CmsCacheDirectives* has been used to store informations about server and client-side caching and streaming. In addition this class now stores informations about the internal caching. These informations specify the different variants of elements that will be cached.

The class *CmsCacheDirectives* provides three different constructors:

- `CmsCacheDirectives(boolean internal, boolean proxyPriv, boolean proxyPub, boolean export, boolean stream)`

This constructor sets the values for:

- internal: the element can be cached in the element cache
- proxyPriv: the proxy private flag in the http-response header can be set
- proxyPub: the proxy public flag in the http-response header can be set
- export: the element can be exported (static export)
- stream: the page containing this element can be delivered in streaming mode (this means parts of the content can be delivered without waiting until the complete content of the page is generated). Streaming is usually possible if no redirect is triggered inside the template class.

- `CmsCacheDirectives(boolean)`

This constructor sets all five boolean parameters of the first constructor to the same boolean value.

- `CmsCacheDirectives(boolean internal, boolean stream)`

This constructor only sets the values for internal caching and streaming. The other values can be set separately to a boolean value by using the methods *setProxyPublicCacheable()*, *setProxyPrivateCacheable()* and *setExport()*. The values that are not set are determined at runtime by using the following table:

	proxy public	proxy private	export
internal cacheable	x	x	x
no parameters in the key			x
user/group not in the key	x	x	x
readable for guest-user	x		x
no timeout			x
no internal flag	x	x	x

The proxy public flag for example is set if the internal cacheable flag is set, user or group are not in the key, the resource is readable for guest-users and no internal flag is set for this resource. The determination of these values at runtime slows down the caching mechanism, therefore it is recommended to set these values manually if possible.

## Methods in the class *CmsCacheDirectives* to control the element caching

The way elements are cached can be controlled by using several methods of the class *CmsCacheDirectives*. Essential are the methods that affect the way the cache key is generated. If an element is internal cacheable and the caching properties in the

*CmsCacheDirectives* object have not been adjusted the XML template mechanism generates a variant of this element when it is requested for the first time and every following request will result in an delivery of this variant from the cache. If someone wants several variants of an element to be cached he has to specify which variants should be cached. For example if an element is different for different groups of users (a login box that has a form for guest users to login and a generic message for a user that is already logged in) the method *setCacheGroups(true)* provides a way to add the current group of the user to the cache key. This results in a creation of variants of this element for every different group. The complete set of methods to control the caching are:

- *setCacheUri(boolean uriCache)*  
If set to true, this method adds the uri to the cache key. Variants of the element for every different uri will be created.
- *setCacheUser(boolean userCache)*  
If set to true, this method adds the name of the user to the cache key. Variants of the element for every different user will be created.
- *setCacheGroups(boolean groupCache)*  
If set to true, this method adds the name of the group to the cache key. Variants of the element for every different group will be created.
- *setCacheGroups(Vector groupNames)*  
Some elements can be static for specific groups but dynamically for others. For example a login box that shows a login form for guest users and a personalized message for users that are already logged in. This element can be cached for the group Guests but not for the group of users that are logged in. For this reason the method *setCacheGroups* is overloaded and provides the opportunity to specify a Vector of groupnames that contains the groups for which variants of the element will be cached. For any other groups the element will be created dynamically.
- *setCacheParameters(Vector parameterNames)*  
The Vector passed to this method contains the name of parameters that should be added to the cache key. For every combination of values of these parameters a variant of the element will be created. Another variant will be created if a parameter is not given in the request.
- *setNoCacheParameters(Vector parameterNames)*  
This method provides a way to specify the name of parameters that prevent an element from being cacheable. An registration form for example can be fetched from the cache when the form is requested but should be dynamically processed when the form is send back. When the name of a parameter that is send with the form is passed to this method any occurrence of this parameter in the request will cause the element to be created dynamically.

- *setTimeout(CmsTimeout timeout)*

This method provides a way to specify a period of time for which an element is valid. The element will be cached but will be created new when this period of time has passed. This is particularly useful for applications like a news-modul where the news content is static but should be updated after a certain period of time (e.g. every 60 minutes). To set a duration of validity a *CmsTimeout* object has to be created. So far only one constructor which takes a number of minutes exists. The element will then be created new at 0:00 am and every x minutes from this point of time. For example if the timeout was adjusted to 40 minutes the element will be created new at 0:00 am and then at 0:40 am, 1:20 am, 2:00 am and so forth. The element will at least be created new every 24 hours, this means the counting starts new every day at 0:00 am. Useful values for the number of minutes are values between 5 and 1440 minutes (24 hours).

A technical note: the element will be checked for validity only when requested. An element will therefore be created new when it is requested and the duration of validity has passed. When an element is no longer valid all its variants will be deleted from the cache. This means the *setTimeout()* method can also be used for elements that have several variants.

- *renewAfterEveryPublish()*

If a project is published every element will be deleted from the cache if the template or class of the element has been changed in the project. Also every elements which have variants for different uris will be deleted from the cache. This is necessary because the hierarchy of files and folders in the project could have changed and cause a different output of the element even if the template of the element hasn't changed. Generally the programmer has not to care about deleting elements from the cache. In rare cases these rules for deleting elements from the cache may not be sufficient (for example if an element itself uses the content of other resources in the virtual file-system). For this reason the method *renewAfterEveryPublish()* exists to force the system to delete elements from cache when a project is published.

### Defining caching dependencies with the method `registerVariantDeps()`

The method *registerVariantDeps()* allows to define further dependencies for cached elements. This method can be used inside the *getContent()* method or user-methods to register resources in the VFS, contentdefinition objects and classes that affect the caching of the current element. It is not mandatory to use this method when using the element cache, but it allows a more subtle way to define dependencies and should be used whenever possible.

This is the signature of the method *registerVariantDeps()*:

```
protected protected void registerVariantDeps(
```

```
CmsObject cms, String templateName, String elementName,  
String templateSelector, Hashtable parameters, Vector vfsDeps,  
Vector cosDeps, Vector cosClassDeps)  
throws CmsException
```

The parameters have the following meaning:

- cms (type *com.opencms.file.CmsObject*) provides access to system resources.
- templateName (type *java.lang.String*) the absolute path to the template. This parameter is passed to the *getContent()* method and can be get in a user-method by calling *getAbsolutePathname()* on the *A\_CmsXmlContent* object.
- elementName (type *java.lang.String*) only needed if this parameter is used in the *getCacheDirectives()*, can be null otherwise.
- templateSelector (type *java.lang.String*) like the elementName this parameter can be null if it is not used inside the *getCacheDirectives()* method.
- parameters (type *java.util.Hashtable*) hash table with URL-parameters and other parameters added by the system. Inside a user-method the *userObject* contains this parameters hash table.
- vfsDeps (type *java.util.Vector*) this Vector contains all *com.opencms.CmsResource* objects (not the names of the resources !) which should force the element's variant to be created new if any of these resorces (files and folders) changes.
- cosDeps (type *java.util.Vector*) this Vector contains *com.opencms.defaults.A\_CmsContentDefinition* objects that are used to create the element and affect its output. When one of these resources changes the element's variant will be created new.
- cosClassDeps (type *java.util.Vector*) Here you can add class objects of content definition classes. If one object of the class changes the element's variant will be created new.

Here is a code sample how the method *registerVariantDeps()* is used in the standard navigation class:

```
public Object getNavCurrent(CmsObject cms, String tagcontent  
    A_CmsXmlContent doc, Object userObject)  
    throws CmsException {  
    // current folder
```



```
String currentFolder =
    cms.getRequestContext().currentFolder().getAbsolutePath();
// register this folder for changes
Vector vfsDeps = new Vector();
vfsDeps.add(cms.readFolder(currentFolder));
registerVariantDeps(cms, doc.getAbsoluteFilename(),
    null, null, (Hashtable)userObject,
    vfsDeps, null, null);
}
```



# 18. Building applications with XML templates

## 18.1. Building a navigation

In OpenCms, the navigation concept is based on a representation of files and folders of the VFS. To make a file or a folder appear in the navigation two properties must be added to the resource: The navigation position and the navigation text.

Whenever you create a new file or folder, a checkmark in the "New Page" / "New Folder" dialogue indicates whether this resource will be added to the navigation. You can then enter the navigation parameters position and text. Of course, those parameters can also be added, changed or deleted later.

If navigation entries should be grouped, then those elements have to be put in one folder. A folder can contain HTML files or further folders. An HTML file named `index.html` placed in a folder is being used as an overview for this folder. This means that if the folder name is, for example, `/news/sport/`, clicking on this navigation entry will cause the file `/news/sport/index.html` to be displayed. Because of this, the file does not need to be included in the navigation.

If `index.html` does not exist, the navigation link will point to the current folder. If you want the default file name to be changed from `index.html` to another name (e.g. `news.html`), a property `NavIndex` containing the new file name has to be added to the folder.

The OpenCms navigation properties can be easily read from a JSP using the OpenCms taglib or scriptlet API. How to do this is explained in the interactive part of the documentation.

In case you do not want to use JSP, you could also use the proprietary XML template mechanism to build navigation elements. *As of OpenCms 5.0, we recommend the use of JSP to build navigation elements.*

## 18.2. Navigations with XML Templates

The XML template mechanism brings along a set of methods in the class `CmsXmlNav` that help you build a simple navigation. However, if a navigation with extended functionality

is needed, a new class extending `CmsXmlNav` has to be written for with XML Templates (this is not required if you use JSP)

To display the navigation, an element `nav` must be defined in the frametemplate of the files. In the element definition of this element, the class `CmsXmlNav` and a template file `navigation` must be defined. The template of this element is used to define the required HTML-code to build the navigation. Example: First the frametemplate:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE><![CDATA[
    <HTML>
    <HEAD>
        <TITLE>]]><method name="getTitle"/><![CDATA[</TITLE>
    </HEAD>
    <BODY>
    <TABLE border width="100%" height="100%">
        <TR height="30%">
        <TH colspan=2 width="100%" align="center">
            The Head section
        </TH>
        </TR>
        <TR height="70%">
        <TD width="20%" align="center" valign="top">
            ]]><element name="nav"/><![CDATA[
        </TD>
        <TD width="80%" align="center">
            ]]><element name="contenttemplate"/><![CDATA[
        </TD>
    </TR>
    </TABLE>
    </BODY>
    </HTML>]]>
</TEMPLATE>

<ELEMENTDEF name="nav">
    <CLASS>com.opencms.defaults.CmsXmlNav</CLASS>
    <TEMPLATE>/system/modules/org.opencms.default/elements/navigation</TEMPLATE>
</ELEMENTDEF>

</XMLTEMPLATE>
```

Now the definition of the navigation element in `/system/modules/org.opencms.default/elements/`

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

<naventry>
  <![CDATA[
    <tr>
      <td>
        <a href=""><process>navlink</process><![CDATA[">
          <process>navtext</process><![CDATA[
        </a>
      </td>
    </tr>
  ]]>
</naventry>

<navcurrent>
  <![CDATA[
    <tr>
      <td>
        ]]><process>navtext</process><![CDATA[
      </td>
    </tr>
  ]]>
</navcurrent>

<TEMPLATE>
  <![CDATA[
    <TABLE border width="100%" height="100%">
      ]]><method name="getNavCurrent"/><![CDATA[
    </TABLE>
  ]]>
</TEMPLATE>
</XMLTEMPLATE>
```

Several data block definitions and one method are used in this example. The definition of the *naventry* data block is compulsory, otherwise no element of navigation would be shown. It is used to generate navigation entries. The *navcurrent* data block is used to show the current position in the navigation. It can be used to highlight the current navigation entry, e.g. using a different font color or style.

The method *getNavCurrent* is a method defined in class *CmsXmlNav* and returns the navigation of the current folder.

The following methods defined in CmsXmlNav class can be used to build different types of navigation. Here is the definition of methods, tools and data blocks defined in the CmsXmlNav class:

Data blocks that have to be defined in the templates managed by CmsXmlNav:

- navtext
- navlink
- naventry
- navcurrent
- navstart
- navend

Additional data blocks that can be of use in a navigation structure:

- navlevel
- navcount

Methods that are defined in CmsXmlNav:

- getNavCurrent
- getNavFold
- getNavTree
- getNavParent
- getNavRoot

Additional methods that could be helpful in templates:

- getFolderCurrent
- getFolderParent
- getFolderRoot
- getPropertyCurrent
- getPropertyParent
- getPropertyRoot
- getPropertyUri

**Definition of data blocks:****- navtext:**

This data block is used to display the navigation text. The navigation text is defined in the *NavText* property of the corresponding file or folder.

**- navlink:**

This data block shows the path of the corresponding navigation entry.

**- naventry:**

The definition of this data block is compulsory, because this data block defines the layout of a navigation entry. For each navigation entry this data block is used. Example:

```
<naventry>
<![CDATA[
<tr>
  <td>
    <a href=""><process>navlink</process><![CDATA[ class="nav"
target="_blank">
      ]]><process>navtext</process><![CDATA[
    </a>
  </td>
</tr>
]]>
</naventry>
```

**- navcurrent:**

With the definition of this data block, the layout of the current navigation position is set. If this data block is not defined, the *<naventry>* data block is used. Example:

```
<navcurrent>
<![CDATA[
<tr>
```

```
<td>
  <font color="red">]]><process>navtext</process><![CDATA[</font>
</td>
</tr>
]]>
</navcurrent>
```

#### - navstart and navend:

These data blocks are used in nested navigation system, so if the methods *getNavFold* or *getNavTree* are used then these data blocks can be used to format the navigation. These methods work recursively and after each call these data blocks are used to format the navigation entries. For example, a definition of the data block `<navstart>` as a `<ul>` tag and the `<navend>` data block as a `</ul>` tag creates a nested `<ul> ... </ul>` list. Example:

```
<navstart><![CDATA[<ul>]]></navstart>
<naventry><![CDATA[
  <li>
    <a href=""><process>navlink</process><![CDATA[">
      ]]]><process>navtext</process><![CDATA[
    </a>]]>
</naventry>
<navcurrent><![CDATA[<li>]]><process>navtext</process></navcurrent>
<navend><![CDATA[</ul>]]></navend>
```

If the `<navstart>` and `<navend>` are not defined then they are ignored.

#### - navlevel:

This data block contains the depth of the navigation entry. This data block could be used in a nested navigation system, e.g. the definition of a group of style sheets depending on the depth of navigation. Example:

```
<navstart><![CDATA[<ul>]]></navstart>
<naventry><![CDATA[
  <li class="nav_"]><process>navlevel</process><![CDATA[">
  <a class="nav_"]><process>navlevel</process><![CDATA[">
```



```

    href="]]><process>navlink</process><![CDATA[">
    ]]><process>navtext</process><![CDATA[
    </a>]]>
</naventry>
<navcurrent><![CDATA[
<li class="nav_]]><process>navlevel</process><![CDATA[">
]]><process>navtext</process>
</navcurrent>
<navend><![CDATA[</ul>]]></navend>

```

With `<navlevel>`, the depth of the navigation entry is determined. With a corresponding definition of the style sheet in a .css file (e.g. `nav_1.....nav_2.....nav_3....., ....`), each level of the navigation can be formatted in different way.

Should the level number be given as a parameter in a method, then the real depth will be calculated from the difference of depth between the current folder and the given depth in the method.

#### - **navcount:**

An additional data block that returns the number of navigation entries in the navigation.

### Definition of Methods:

#### - **getNavCurrent:**

This method gets the navigation of the current folder, using the *naventry* and *navcurrent* data blocks.

#### - **getNavFold:**

This method gets the navigation of a specified folder. If an entry is clicked, then the navigation of that entry will be added to the current navigation. Because of folding functionality this method is called `getNavFold`.

This method uses the `<naventry>`, `<navcurrent>`, `<navstart>` and `<navend>` data blocks.

A level parameter can be set as follows:

```
<method name="getNavFold/>
```

```
<method name="getNavFold">level</method>
```

The level parameter defines the starting folder of navigation from which the navigation will be constructed. For example, if the current folder is `/news/sport/football/liga/first/` and the method definition is `<method name="getNavFold">2</method>`, then the starting folder for the navigation will be `/sport/` and all elements starting from `/sport/` will be displayed.

If no level parameter is defined, the starting point is set to the root folder.

#### - **getNavTree:**

This method gets the navigation starting from a folder and shows the wholer navigation tree recursively. Therefore, this method is called `getNavTree`.

Level and depth parameter can be set as follows:

```
<method name="getNavTree"/>  
<method name="getNavTree">level</method>  
<method name="getNavTree">level,depth</method>
```

The level parameter defines the starting folder from which the navigation will be constructed. For example, if the current folder is `/news/sport/football/liga/first/` and the method definition is `<method name="getNavTree">2</method>` then the starting folder for the navigation will be `/sport/` and all elements starting from `/sport/` folder will be displayed.

If no level parameter is defined, the starting point is set to the root folder.

The depth parameter defines how many levels of folders will be displayed in the navigation tree. For example, if the current folder is `/news/sport/football/liga/first/` and the method definition is `<method name="getNavTree">1,3</method>` then the starting folder for the navigation will be `/news/` and all elements from `/news/` to `/news/sport/football/` will be displayed.

#### - **getNavParent:**

This method gets the navigation from the parent folder(s), depending on the given depth parameter. For example, if the current Folder is `/news/sport/football/liga/first/` and the

method definition is `<method name="getNavParent">2</method>` then the navigation of the folder `/news/sport/football/` will be displayed (two levels up). If no depth parameter is given, the navigation of the current folder will be displayed. If the number given as depth parameter exceeds the sum of possible folders, the root folder will be chosen.

#### - **getNavRoot:**

This method gets a navigation tree starting from the root folder, depending on the given parameter. If, for example, the current folder is `/news/sport/football/liga/first/` and the method definition is `<method name="getNavRoot">2</method>`, then the navigation of `/news/sport/` will be displayed (second level starting from root). If no depth parameter is given, the current folder's navigation is chosen. If the number given as depth parameter is too high, the current folder will be chosen.

#### - **getFolderCurrent:**

This method gets the current folder. It provides the same functionality as `getPathUri` in class `CmsXmlTemplate` used in many templates.

#### - **getFolderParent:**

This method gets the parent folder, depending on the depth parameter given. For example, if the current folder is `/news/sport/football/liga/first/` and the method definition is `<method name="getFolderParent">2</method>`, the folder returned by this method will be `/news/sport/football/`. If no parameter is given, the current folder will be returned. If the number given as depth parameter is too high, the root folder will be returned.

#### - **getFolderRoot:**

This method gets a specific folder underneath the root folder, depending on the depth parameter given. If, for example, the current folder is `/news/sport/football/liga/first/` and the method definition is `<method name="getFolderRoot">2</method>`, then the folder `/news/sport/` will be displayed (second level starting from root). If no depth parameter is given, the root folder will be returned. If the number given as depth parameter is too high, the current folder will be returned.

**- getPropertyCurrent:**

This method gets a property of the current folder. This method is used as follows:

```
<method name="getPropertyCurrent">property</method>
```

where *property* is the name of the property to be displayed. If there is no such property name, nothing is returned.

**- getPropertyParent:**

This method gets the property of a parent folder, depending on the parameters given.

This method is used as follows:

```
<method name="getPropertyParent">level,property</method>
```

where *property* is the name of the property to be read. If there is no such property name, nothing is returned.

*level* is the number of folders to go up. If, for example, the current folder is */news/sport/football/liga/first/* and the method definition is

```
<method name="getPropertyParent">2,NavPic</method>
```

, the value of the *NavPic* property of */news/sport/football/* would be returned by this method. If the number given as level parameter is too high, the same property of the root folder is read.

**- getPropertyRoot:**

This method gets the property of a specified folder underneath the root folder, depending on the level parameter given. This method is as follows:

```
<method name="getPropertyRoot">level,property</method>
```

where *property* is the name of the property to be read. If there is no such property name, nothing is returned.

*level* specifies the number of folders to go down from the root folder. If, for example, the current folder is */news/sport/football/liga/first/* and the method definition is 

```
<method name="getPropertyRoot">2,NavPic</method>
```

, the value of the property *NavPic* of */news/sport/* will be returned by this method. If the number given as level parameter is too high, the same property of the current folder is returned.

**- getPropertyUri:**

This method returns the specified property of the current uri. This method is used as follows:

```
<method name="getPropertyUri">property</method>
```

where *property* is the name of the property to be read. If there is no such property name, nothing is returned.

It is very common to use an image instead of Text in navigation. Therefore, here is an example that uses images instead of text:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<naventry>
<![CDATA[
<tr>
  <td>
    <a href=""><process>navlink</process><![CDATA["
      onmouseover="JavaScript:document.image_
      ]]><process>count</process><![CDATA[.value=
      '/pics/templates/]]><process>navtext</process>
      <![CDATA[_light.gif'"
      onmouseout="JavaScript:document.image_]]>
      <process>count</process><![CDATA[.value=
      '/pics/templates/]]><process>navtext</process><![CDATA[.gif'">
      <img name="image_]]><process>count</process><![CDATA["
      src="/pics/templates/
      ]]><process>navtext</process><![CDATA[.gif">
    </a>
  </td>
</tr>
]]>
</naventry>
<navcurrent>
<![CDATA[
<tr>
  <td>
    
    </td>
</tr>
]]>
</navcurrent>
<TEMPLATE>
<![CDATA[
```

```
<TABLE border width="100%" height="100%">
  ]]><method name="getNavCurrent"/><![CDATA[
</TABLE>
]]>
</TEMPLATE>
</XMLTEMPLATE>
```

To build a navigation using images instead of text, an image name must be written in the property *NavText*. To get mouse over and mouse out effects, another picture with the same name but with a suffix is needed so that javascript can replace it. In the example above, the image name is built with the count data block (so *image\_1*, *image\_2*,.... would be generated). Then, the picture for the mouse over effect is included with the count data block and the suffix *\_light*. So if the picture name is *nav.gif*, then the mouse over picture would be *nav\_light.gif*.

## 18.3. Frames

### 18.3.1. Frames with XML templates

The usage of frames is possible with proprietary XML templates. One big difference to the usage of frames in standard HTML is that in with XML templates, the frameset and all frames are defined together in one file. In this part, we will create a simple page with two frames to show the basic structure. The frameset and the inserted parts (templates) are defined in the frametemplate. The complete code for a frametemplate with 2 frames is shown here:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE><![CDATA[
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Seite mit Frames</TITLE></HEAD>
<FRAMESET ROWS="100,*">
  <FRAME NAME="frame_head" src="]]>
    <METHOD name="getUri"/>
    <METHOD name="getFrameQueryString">temp_head</METHOD>
    <![CDATA[">
  <FRAME NAME="frame_body" src="]]>
    <METHOD name="getUri"/>
    <METHOD name="getFrameQueryString">temp_body</METHOD>
```

```

        <![CDATA[">
</FRAMESET>
</HTML>]]>
</TEMPLATE>
<TEMPLATE name="temp_head"><![CDATA[
    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
    <HTML>
    <HEAD><TITLE>Seite mit Frames</TITLE></HEAD>
    <BODY>
        <P ALIGN="center">
            ]]><ELEMENT name="nav_head"/><![CDATA[
                </P>
            </BODY>
        </HTML>]]>
</TEMPLATE>
<TEMPLATE name="temp_body"><![CDATA[
    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
    <HTML>
    <HEAD><TITLE>Seite mit Frames</TITLE></HEAD>
    <BODY>
        <P ALIGN="center">
            ]]><ELEMENT name="contenttemplate"/><![CDATA[
                </P>
            </BODY>
        </HTML>]]>
</TEMPLATE>
<ELEMENTDEF name="nav_head">
    <CLASS>com.opencms.defaults.CmsXmlNav</CLASS>

<TEMPLATE>/system/modules/org.opencms.default/elements/nav_head</TEMPLATE>
</ELEMENTDEF>
</XMLTEMPLATE>

```

The frameset is here defined within the default template (the one without a name) with the normal `<FRAMESET>` tag. The content of the frames is defined in the other templates (temp\_head and temp\_body). The link to the frames is created by using the two XML template methods `getUri()` and `getFrameQueryString()`. The method `getUri()` returns the URI of the currently requested page and the method `getFrameQueryString()` returns the original query string of the request together with an additional parameter `cmsframe` which determines what template is to be used. The value of the `cmsframe` parameter is passed to the `getFrameQueryString()` method as the tagcontent of the `<METHOD>` tag. For example the line:

```
<METHOD name="getFrameQueryString">temp_head</METHOD>
```

results in this text:

```
?cmsframe=temp_head
```

It would be possible to simply add the parameter by hand:

```
<![CDATA[?cmsframe=temp_head
```

This would cause problems if further parameters are appended to the URL because they would not be included in this string. The method *getFrameQueryString()* solves this problem and handles further parameters. Thus it is recommended to use the method always.

Our frametemplate includes two elements, one for the navigation (nav\_head) and one content element. Of course, there has to be an elementdefinition for the navigation element at the end of the XML-file. The navigation template for this example looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<NAVENTRY><![CDATA[
  <A HREF="]]>
  <PROCESS>navlink</PROCESS>
  <METHOD name="getFrameQueryString">temp_body</METHOD>
  <![CDATA[" ]]]>
  <METHOD name="getFrameTarget">frame_body</METHOD>
  <![CDATA[>]]>
  <PROCESS>navtext</PROCESS><![CDATA[
  </A>]]>
</NAVENTRY>
<TEMPLATE>
  <![CDATA[<TABLE BORDER=1 CELLPADDING=2 CELLSPACING=0><TR><TD>]]>
  <METHOD name="getNavRoot">1</METHOD>
  <![CDATA[</TD></TR></TABLE>]]>
</TEMPLATE>
</XMLTEMPLATE>
```

A reference to a page that should be displayed in another frame in standard HTML looks like this:

```
<a href="aName.htm" target="aFrame">aNavText</a>
```



The parameter *aFrame* determines the frame in which the source should be displayed, and the parameter *aName.htm* determines the source that should be inserted. Here the parameter *aName.htm* has to be expanded by a parameter that defines what template has to be selected to insert. This is again done by the *templateselector*, and the parameter *cmsframe* is again inserted by the method *getFrameQueryString()* to allow further parameters.

```
<A HREF="" ]]>
<PROCESS>navlink</PROCESS>
<METHOD name="getFrameQueryString">temp_body</METHOD>
<![CDATA[" ]]>
<METHOD name="getFrameTarget">frame_body</METHOD>
<![CDATA[>]]>
<PROCESS>navtext</PROCESS><![CDATA[
</A>]]>
```

The parameter *target* is here inserted by the method *getFrameTarget()*. Of course, the parameter could be simply added by inserting *target="frame\_body"*, but you will see in the next paragraph why the usage of the method has advantages. To understand the difference, the above code block is now shown without the usage of methods:

```
<A HREF="" ]]>
<PROCESS>navlink</PROCESS>
<![CDATA[?cmsframe="temp_body" TARGET="frame_body">]]>
<PROCESS>navtext</PROCESS><![CDATA[
</A>]]>
```

Maybe this helps to understand which text is inserted by the methods. But again, it is always recommended to make usage of these methods, to write templates that can be easily reused and provide full functionality.

Pages which use our frame structure can now easily be created. You only have to choose the above mastertemplate. The output for three inserted pages could for example look like figure 18.1.

### 18.3.2. Frames- and noframes versions of a website with XML templates

Often a website has to be build in two versions: one with frames, and one without frames. This way users with older browsers are able to visit the site as well. These two versions can be achieved with one frametemplate. All you have to do is to extend the frametemplate by one further template. This template should be named *plain*. It defines the layout for the site without frames. One implementation of the *plain*-template is shown now:

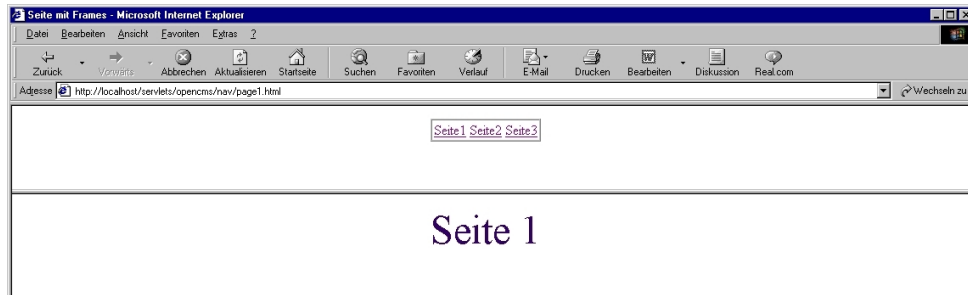


Figure 18.1.: A page with frames

```

<?xml version="1.0" encoding="ISO-8859-1"?>
...
<TEMPLATE name="plain"><![CDATA[
<HTML>
<HEAD>
  <TITLE>]]><method name="getTitle"/><![CDATA[</TITLE>
</HEAD>
<BODY>
  <TABLE border width="100%" height="100%">
  <TR height="30%">
    <TD align="center">]]>
    <element name="nav_head"/> <![CDATA[</TD>
  </TR>
  <TR height="70%">
    <TD align="center">]]>
    <element name="body"/> <![CDATA[</TD>
  </TR>
  </TABLE>
</BODY>
</HTML>]]>
</TEMPLATE>
<ELEMENTDEF name="nav_head">
...
</XMLTEMPLATE>

```

This template inserts exactly the same subtemplates as the one that uses frames. Even the template for the navigation is the same. They are only set inside a table instead of frames. Now it becomes useful that we implemented the navigation with the *getFrameTarget()* method. The navigation would not work with a table, if the parameter *target=...* had simply be added. The *getFrameTarget()* method distinguishes between the frames and the noframes version and drops the parameter *target* if necessary. If you implemented

the templates like this, a simple call to the URL of a page that uses this mastertemplate would start the frames version, which is defined in the default template. You can choose the noframes version by adding the parameter `?cmsframe=plain` to the URL. The output would then look like figure 18.2.

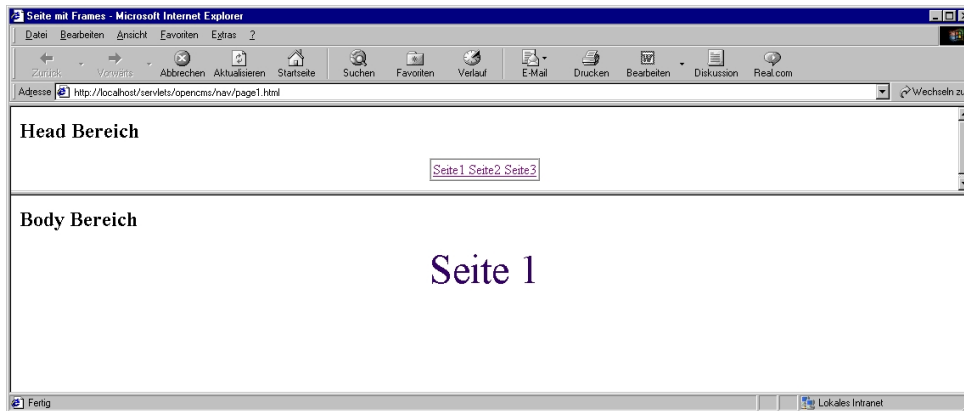


Figure 18.2.: A version without frames

## 18.4. The CmsObject Class (accessing system resources)

There is a programming interface that provides access to the system and all of its resources. The majority of the interface is part of the *CmsObject* that provides methods to access system resources. These resources can be for example files, user data or session-related data. The *CmsObject* enables these resources to be manipulated (i.e. deleted, renamed, moved) and properties of these resources to be requested. Since all of the operations performed by the *CmsObject* are executed by the user that is currently accessing the system, access to the system resources is limited by that user's access permissions. Using the API (Application Programming Interface) provided by the *CmsObject* enables you to create customized classes that perform various kinds of operations. The next sections will explain how common tasks can be accomplished by using the *CmsObject* and its API. Only a short collection of the available *CmsObject* methods is used in the following examples. The complete API can be found as JavaDoc in the appendix.

### 18.4.1. Accessing user data

The OpenCms user management can be accessed via the API to read or write the users maintained by the system. The following example will read all users of the system and will display them in a simple HTML table.

All of the following examples that use the *CmsObject* generate highly dynamic content, therefore it is necessary to override the *getCacheDirectives()* method in the template

classes of this examples and set the internal caching to false in the returned *CmsCacheDirectives* object. See section 17.3.9 (page 155) for details about the element caching.

First, all of the users have to be read from the OpenCms system. This is done in the *getContent* method of the template class, and is achieved by using the following *CmsObject* method:

```
// get all users
```

```
Vector users=cms.getUsers();
```

This returns a vector of *CmsUser* objects, each of which contains the data of a single OpenCms user. The individual *CmsUsers* are extracted from the vector and their login, first and last names as well as their e-mail address are read. See the appendix for a complete overview of the *CmsUser* object methods.

The HTML table is built in a similar way to the table in the animal list example (see page 139): a row of the table is defined as a data block in the template file, that contains several *<process>* tags that must be filled by the Java class. The rows are generated for each *CmsUser* entry in order to create a complete list of users.

The complete *getContent* Method of this example looks like this:

```
public byte[] getContent(CmsObject cms, String templateFile, String
    elementName, Hashtable parameters, String
    templateSelector) throws CmsException {
CmsXmlTemplateFile template = getOwnTemplateFile(cms, templateFile,
    elementName, parameters,
    templateSelector);
    // get all users
    Vector users=cms.getUsers();
    String list="";
    // browse through all users
    for (int i=0;i<users.size();i++) {
        //get a single user
        CmsUser user=(CmsUser)users.elementAt(i);
        template.setData("name",user.getName());
        template.setData("firstname",user.getFirstname());
        template.setData("lastname",user.getLastname());
        template.setData("email",user.getEmail());
        String row=template.getProcessedDataValue("row");
```

```
        list+=row;
    }
    template.setData("list", list);
    return startProcessing(cms, template, elementName, parameters,
        templateSelector);
}
```

This example can easily be extended to read additional user information that is stored in the OpenCms database by adding the access methods to the other data fields.

## Reading Resource Properties

Resources in the OpenCms VFS can have additional properties. By default, these properties are used to store the resource title, but they could also be used to store additional information that is used by the template class. For example this could be the name of an image that must be different for all pages that use a special template, or information on where the Java class can find additional data. Another typical use for the resource properties is to store information that is necessary to build a dynamic navigation.

The following example shows how to read all of a resource's properties and display them in an HTML table.

First, the name of the current resource - the URI - must be read. This is done by using the *CmsRequestContext*. There, all of the data that belongs to the actual request, i.e. User, Project, HttpRequest, HttpResponse and Session are stored:

```
// get the uri of the requested file
String uri=cms.getRequestContext().getUri();
```

The `getUri` method returns the complete path of the current resource in the VFS. See the appendix for a complete overview of the *CmsRequestContext*'s methods.

In the next step all of the *resource*'s properties are read. This is done using a *CmsObject* method that returns a hash table that contains the *properties*' names and values:

```
// get all properties of this resource
Hashtable prop=cms.readAllProperties(uri);
```

Now that all of the data is fetched from the system, the output must be generated. This is done in almost the exact same way that it was in the previous example:

```
public byte[] getContent(CmsObject cms, String templateFile, String
    elementName, Hashtable parameters, String
    templateSelector) throws CmsException {
    CmsXmlTemplateFile template = getOwnTemplateFile(cms, templateFile,
        elementName, parameters,
        templateSelector);
    // get the uri of the requested file
    String uri=cms.getRequestContext().getUri();
    // get all properties of this resource
    Hashtable prop=cms.readAllProperties(uri);
    Enumeration enum=prop.keys();
    String list="";
    while (enum.hasMoreElements()) {
        String key=(String) enum.nextElement();
        String value=(String)prop.get(key);
        template.setData("property",key);
        template.setData("value",value);
        String row=template.getProcessedDataValue("row");
        list+=row;
    }
    template.setData("filename",uri);
    template.setData("list",list);
    return startProcessing(cms, template, elementName,
        parameters,templateSelector);
}
```

As an alternative, a single resource property can be read with the following line of code:

```
String value=cms.readProperty(uri,key);
```

This will either return the value of the property with the name "propertyname," or null if this property does not exist.

### 18.4.2. Resource access

A typical task for a template programmer is to read the resources that are stored in the VFS. The *CmsObject* contains several methods that enable a programmer to read either a single resource - this could be a file or a folder - or a complete vector of subresources - the files or subfolders of a specified folder.

The next example will create a list of all of the folders that are stored in the VFS, that contains their complete path and title (if existing). It consists of a simple *getContent()* method that calls a recursive *getFolders()* method that produces the HTML output of

folder list of a given root folder. The manner in which the HTML output is created is similar to that shown in the previous examples.

```
public class CmsExample21 extends CmsXmlTemplate
public byte[] getContent(CmsObject cms, String templateFile, String
elementName, Hashtable parameters, String templateSelector)
throws CmsException
CmsXmlTemplateFile template = getOwnTemplateFile(cms, templateFile,
elementName, parameters, templateSelector);
    String list=getFolders(cms,template,"/");
    template.setData("filename","/");
    template.setData("list",list);
return startProcessing(cms, template, elementName, parameters,
templateSelector);
}
```

The core functions of this template class are located in the *getFolders()* method:

```
private String getFolders(CmsObject cms, CmsXmlTemplateFile
template,String root)
    throws CmsException{
    String list="";
// get all subfolders of the root folder Vector folders=cms.
getSubFolders(root);
    // build the list of folders
    for (int i=0;i<folders.size();i++) {
        CmsFolder folder=(CmsFolder)folders.elementAt(i);
        String foldername=folder.getAbsolutePath();
String foldertitle=cms.readProperty(foldername,"Title");
template.setData("resource",foldername);
template.setData("name",foldertitle);
        String row=template.getProcessedDataValue("row");
        list+=row;
        list+=getFolders(cms,template,foldername);
    }
    return list;
}
```

In this method, all of the subfolders of a given folder named *root* are read and stored in a vector:

```
// get all subfolders of the root folder Vector folders=cms.
getSubFolders(root);
```

The individual folders of this vector - alphabetically sorted CmsFolder objects - are then extracted. The required information, folder name and folder title, is put into the "row" data block that is defined in the template. The method is recursively called for each folder that is found, so that its subfolders are added to the list as well.

The example can easily be extended to read the files of a folder as well. This is done in a way that is very similar of that used to read a folder's subfolders:

```
// get all files of the root folder
Vector files=cms.GetFilesInFolder(root);
```

This creates a vector of CmsFile objects that can be accessed in almost the same way as the CmsFolder objects. See the appendix for a complete list of CmsFolder and CmsFile object's methods.

### 18.4.3. Session management

Because the HTTP protocol is a stateless protocol there is usually no way to recognize if consecutive requests are generated by the same client. Each request is completely independent and has no influence or memory of past or future requests. There is no built-in feature to keep track of the user's actions in the HTTP protocol. The system provides a session management that enables user actions to be tracked. This session management is built on the session tracking feature of the original Servlet API . The user is recognized by placing a cookie on the client side. A cookie is a little text file with a maximum amount of 4096 bytes that is sent by the server and stored on the client. This cookie contains a unique id string to identify a user. Session management can also be used to store data. The session management layer stores the class that is associated with a particular client in much the same way a cookie is stored. Retrieving the class back again in a subsequent request is a simple matter of calling a method from the session management class. Session management is essential when using the OpenCms workplace, because OpenCms' entire user management relies on the session management feature.

A session will be created or can be fetched by calling the *getSession()* method: *CmsSession session =(CmsSession)cms.getRequestContext().getSession(true)*; If the Boolean value is true and the session does not exist, a new session is created. If the value false is passed with no session or with an expired session, null is returned. First, you have to get the CmsRequestContext object that contains the access to the related data. The specified parameter indicates that the session will be created if it does not already exist. If you specify false as the parameter value for the *getSession()* method, it will return null if a session did not previously exist. The returned object is of the type *CmsSession* and can be used in a manner similar of that of the original HTTP session objects of the Servlet API. However, only some of the original methods provided by the Servlet API's HTTP Session interface are provided by the *CmsSession* class. These methods are:



```
public Object getValue(String name);
public void putValue(String name, Object value);
public void removeValue(String name);
```

As the name of the methods implies, they are used to fetch, store, or delete session-related data. A session can be used much like a class hash table when storing data in it. You can store every Java object with a string as a key value to access the object. It is **important** to remove used and changed values before redirecting them if they are no longer needed. You can easily archive them using the above mentioned *removeValue()* method. Below is an example of how this is done:

```
if (session.getValue("myValue") != null)
    session.removeValue("myValue");
```

#### 18.4.4. A first example using session management

The first example consists of two pages that show how to store data in the session. In the first page, you can select different colors that are represented by links with attached query strings. If you click on a link, the attached query string is processed by the Java program, stored in the session and displayed in the second page of the body template by using the template selector mechanism. You can see the selected color from the first page: it is used for the text and the background color. Clicking on the link takes you back to the first page of the template. Here, the color that you previously selected is displayed again. The template consists of two pages that are alternately displayed. The active template is selected by the template selector in the controlling Java class:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMLTEMPLATE>
<TEMPLATE>
<![CDATA [
<body bgcolor=""]><PROCESS>bgcolor</PROCESS><![CDATA [">
<table width=70% align=center valign=abscenter border>
<th><h1>Session Example</h1><br>
<h2>Select a color</h2>
<tr height=200 align=center><td><h2>Last color selected:<br>
]]><PROCESS>lastcolor</PROCESS><![CDATA [</h2>
</td></tr>
</table>
<table align=center valign=abscenter border=0>
<tr><td align=center><h2>
    <a href="]]><METHOD
```



Figure 18.3.: Default template output Blue.

```

name="getServletPath"/><![CDATA[SessionBeispiel2.html?red">Red</a>
. . .
    </h2></td>
</td></tr>
</table>
]]>
</TEMPLATE>
<TEMPLATE name="page2">
<![CDATA[
. . .
    <h2>You just selected color:<br><h2></th>
    <tr height=200 align=center ><td><h2>
]]><PROCESS>bgcolor</PROCESS>
<![CDATA[</h2></td></tr>
. . .

```

This figure shows the default template output (the last color selected was Blue) ... (figure 18.3).

... and the template section named "page2" is activated by using the `templateSelector` after clicking on a link (as you can see the color Green was selected here) (figure 18.4).

Here you can see the controlling Java method that stores the data in the session and switches between the pages by setting the template selector. In the session statement, a new session is created, or an existing one is fetched. The color that was selected is attached to the HTTP request as follows: `http://www.opencms.org/sessionexample?colorvalue`. To



Figure 18.4.: Default template output Green.

get the query string that is attached to the HTTP link, the method *getQueryString()* is called. If a color was saved in the previous session, it is fetched again, and displayed on the first page as the last color that was selected. On the first page, the user is able to select one of three colors, whilst on the second page only the "back" link can be selected. Depending on the link with the attached query string that is selected by the user, the template selector is set to either the first or the second page:

```
public byte[] getContent(CmsObject cms, String templateFile, String
elementName, Hashtable parameters, String templateSelector) throws
CmsException {
    //init
    String colorValue;
    String lastColorValue="";
    CmsXmlTemplateFile templateDocument =
getOwnTemplateFile(cms, templateFile, elementName, parameters,
templateSelector);
    //create new or fetch existing session
    CmsSession session = (CmsSession)
cms.getRequestContext().getSession(true);
    //get the query string from the link
    colorValue = ""+((HttpServletRequest)
cms.getRequestContext().getRequest().
getOriginalRequest()).getQueryString();
    //get the stored value from the session
```

```
        lastColorValue = (String) session.getValue("colors2");
        //go back to start page
        if (colorValue == "back") {
            colorValue="null";
            templateSelector="default";
        }
        //color chosen?
        if (colorValue != "null"){
            session.putValue("colors2",colorValue);
            templateSelector="page2";//show second page
        }
        //initial state of the page if (colorValue.equals("null") ||
colorValue.equals("")) ||
colorValue.equals("back")){
            colorValue="white";
            templateSelector="default";
        }
        //set process data
        templateDocument.setData("bgcolor",colorValue);
        templateDocument.setData("lastcolor",lastColorValue);
return startProcessing(cms, templateDocument,
elementName, parameters, templateSelector);
}
}
```

### 18.4.5. Session content

This example dumps the keys and corresponding values that are stored in a session. The below example displays the preceding example's session keys and values. As you can see, the value "red" attached to the clicked link "Red" is stored in key "colors2" of the session. The other keys and values were generated by the system automatically for internal use (figure 18.5).

The keys of the fetched session are stored in a string array, and the corresponding values are read out in a loop by the Java method. After fetching the session, the session key strings are created by using the *getValueNames()* method. By looping through the entries of this string array the corresponding values are read out by the *getValues()* method. Finally the data is set in the template document by using the *setData()* method:

```
public byte[] getContent(CmsObject cms, String templateFile, String
elementName, Hashtable parameters, String templateSelector) throws
CmsException {
```

<b>Session Example</b>	
Data stored in the session:	
Session keys	Session values
adminposition view colors2 USER_PREFERENCES	/system/workplace/administration/database/ /system/workplace/action/explorer.html red {filelist=4607}

Figure 18.5.: Session Content Example.

```

//init
String tempKey="";
String tempValue="";
CmsXmlTemplateFile templateDocument =
getOwnTemplateFile(cms, templateFile, elementName, parameters,
templateSelector);
    //create new or fetch existing session
CmsSession session = (CmsSession)
cms.getRequestContext().getSession(true);
    //copy sessionkeys into stringarray
String [] sessionKey = session.getValueNames();
    //loop through sessionkeys
for(int i=0;i < sessionKey.length;i++) {
    tempKey += sessionKey[i]+"<br>";
tempValue += session.getValue(sessionKey[i])+"<br>";
}

    // set data in xml-document
templateDocument.setData("sessionkey", ""+tempKey);
templateDocument.setData("sessionvalue", ""+tempValue);
    // Finally start the processing
return startProcessing(cms, templateDocument,
elementName, parameters, templateSelector);
}

```

### 18.4.6. Session history

As mentioned above, you can store any object you like in the session, e.g. strings, arrays or vectors. The following example makes use of this feature by storing data of the past query strings in the session. The example shows the past five color links you clicked and



Figure 18.6.: Session history Example

deletes the oldest one, if you clicked more than five times (figure 18.6).

This is done by the *getContent()* method described below. The values are stored using a string array in the session. If the array is not empty, it is fetched from the session. The new color value that is selected by the user is appended to the first empty entry of the array. If there is no empty entry left in the string array, the first entry is deleted by shifting the other entries up by one position. The new entry is moved to the (now empty) last position. The array is stored again in the session, and the parameters for the template are processed.

```
public byte[] getContent(CmsObject cms, String templateFile,
String elementName, Hashtable parameters, String templateSelector)
throws CmsException {
    //init
    String allColorValues = "";
    String colorValue="";
    String[] colorArray= "", "", "", "", "", "", "" ;
    //set data
    CmsXmlTemplateFile templateDocument =
    getOwnTemplateFile(cms, templateFile, elementName, parameters,
    templateSelector);
    //create new or fetch existing session
    CmsSession session = (CmsSession)
    cms.getRequestContext().getSession(true);
    //get query string from link
```

```
        colorValue =
        ""+((HttpServletRequest)cms.getRequestContext()
        .getRequest().getOriginalRequest()).getQueryString();
        // get array data from the session
        if (session.getValue("colorArray") !=null)colorArray = (String [])
        session.getValue("colorArray");
        // fill array with data
        for (int i=0;i<6;i++){
            if (colorArray[i] == ""){
                colorArray[i]=colorValue;
                break;
            }
        }
        // if array filled shift array data
        if (colorArray[5] != "") {
            for(int i=0;i<6;i++){
                colorArray[i]=colorArray[i+1];
            }
            colorArray[5]="";
        }
        for (int i=0;i<6;i++) allColorValues += "<br>" +colorArray[i];
        //store data in the session
        session.putValue("colorArray",colorArray);
        session.putValue("colors",colorValue);
        //set parameters for processing
        if (colorValue.equals("null") ||colorValue.equals(""))
        colorValue="white";
        templateDocument.setData("bgcolor",colorValue)
;templateDocument.setData("button",allColorValues);
        return startProcessing(cms, templateDocument, elementName,
        parameters, templateSelector);
    }
}
```

Session management is a useful feature when building web sites because it enables you to implement advanced features such as HTML forms that extend over multiple pages, online shopping functionality or personalized web content. Session management extends the abilities of cookies that are only capable of storing text data of a limited size (4096 Bytes) on the client. With session management, it is possible to store binary and class information. The session management is, of course, not only usable with query strings but also with forms.

## 18.5. Forms

HTML forms are used to add functionality to a website. With HTML forms, you can receive feedback from the surfers or enable them to order products, product-related materials, or anything else you'd like to make available. The input that is received from the user can be evaluated in a number of ways. Traditionally, this is done by a cgi program that processes the data on the server. Major drawbacks of this approach are the inefficiency and the security holes. When using cgi programs to process the data, every request to the cgi program creates a new process, which produces a significant load on the server. The cgi approach does not come with a security model that prevents incorrect cgi programs from crashing the server. These disadvantages have been overcome by Java servlets, which provide a way to write efficient, platform independent, and secure server-applications. Servlets are special Java classes that can be executed by a server that supports the Java servlet API. Servlets take advantage of the built in security model and the platform independence of the Java language. An instance of a servlet is loaded either when the server is started or when the servlet is first accessed. Concurrent requests to one servlet are handled by multiple threads of the servlet. After a servlet is first loaded and initialized, it is generally not unloaded until the server shuts down. This ensures the servlet class is loaded and initialized once when the servlet is accessed for the first time. OpenCms itself is a servlet that enables the evaluation of HTML forms and writing customized Java classes. Although not servlets themselves, these classes have the same advantages as servlets: they are efficient, portable, and secure! This chapter explains how input fields, radio buttons or select buttons are used in forms. It also discusses programming techniques that can (and should) be used with forms, such as validating and error checking input, as well as redisplaying input in input fields. Starting with the simple redisplay of text that was typed in an input field, you will learn how to use radio buttons and evaluate input data for errors (empty fields or incorrect inputs).

### 18.5.1. A first example using forms

In a first and simple example, you will see how a form can be used for plain text input and to redisplay the input on a confirmation page after submitting the form.

Because the form shouldn't be evaluated when it is requested for the first time, a little trick can help to detect if the form is being displayed for the first time or if it has already been filled out. A hidden input field in the form acts as a flag that indicates if the form has been sent or not. The value of this hidden field is set in the Java class. When the page is first requested, the value of the field is not defined. The Java class detects this and omits the processing of the form's input. This is an excerpt from the template :

```
<FORM>...  
  <INPUT type="hidden" name="action"
```



**Please type in something**

Textarea #1:

Textarea #2:

Figure 18.7.: A simple form

```
value="]]><process>setaction</process><![CDATA[">
...</FORM>
```

The Java class reads in the parameter *action* and if it has no value (the form has not been sent) sets its value to the string "default" by setting the data block *setaction*:

```
...
String action = (String) parameters.get("action");
//no button pressed!
if (action == null || action.equals("")) {
    templateSelector="default";
    template.setData("setaction","default");
...

```

Figure 18.7 shows a screenshot of the form. It consists of two different input fields where text can be entered.

After typing in the text, you can click on the OK button to send the form, or the Reset button to delete the input and restore the initial state of the form. The input will be processed by a Java class and a confirmation page will be shown which shows the text that was typed in the fields. Figure 18.8 shows the message that will be presented to the user.

As you can see, the confirmation page shows the user's input again so that the user can see what was sent to the server. The layout and content of the confirmation page is defined in the same file as the form inside a template with the name *reply*. The Java class controls what template will be used by setting the *templateselector*:

**You typed in:**

```
Textarea #1:Something in here...
Textarea #2:field 2
```

Figure 18.8.: the confirmation page

```
...
templateSelector="reply";
...
```

Here is the complete sourcecode of the *getContent()* method of the Java class:

```
public byte[] getContent(CmsObject cms, String templateFile,
    String elementName, Hashtable parameters,
    String templateSelector) throws CmsException {
    //init
    CmsXmlTemplateFile templateDocument =getOwnTemplateFile(cms,
        templateFile, elementName,
        parameters,templateSelector);
    //button pressed?
    String action = (String) parameters.get("action");
    if (action == null || action.equals("")) {
        //no button pressed!
        templateSelector="default";
        templateDocument.setData("setaction","default");
    } else {
        //a button was pressed
        //get parameters from the default template
        String comment = ((String) parameters.get("comment"));
        String comment2 = ((String) parameters.get("comment2"));
        //set the template selector
        templateSelector="reply";
        //set the parameters in the reply template
        templateDocument.setData("setaction","reply");
        templateDocument.setData("comment",comment);
        templateDocument.setData("comment2",comment2);
    }
}
```

```

    }
    return startProcessing(cms, templateDocument,
        elementName, parameters, templateSelector);
}

```

## 18.5.2. Presetting values

In order to use preset values, the controlling Java class has to set the values of the fields with a preset value or with the input the user has already given. This can be accomplished by setting the value of a field via the process tag:

```

<TEXTAREA cols=30 rows=8 name=comment>
]]><process>comment</process><![CDATA[
</TEXTAREA>

```

The value of the input field will be the value of the data block comment. This data block is specified in the template as an empty data block: If you did not initialize the data block, you will get an error message that the data block is unknown. You should always define the value as an empty element at beginning of the body template:

```

<XMLTEMPLATE>...
<comment></comment> ...
<TEMPLATE>...

```

A value is preset by entering text. When the template is first started, the data block is set with this value:

```

<XMLTEMPLATE>...
<comment>Please fill in something here!</comment>...
<TEMPLATE>...

```

If the value is not defined in the Java class with a different value, the data block will be empty or set to the start value, if it was defined in the template. To show the user's input in the field, the input first has to be fetched from the parameters that were passed to the *getContent()* method, which will set the data block comment to this value. Below is a piece of code that shows how to do this:

```

String comment = (String) parameters.get("comment");
template.setData("comment", comment);

```

The value of the field is first fetched from the parameters hash table, which stores the parameters that are passed when the form is submitted. The data block is then set to this value. When started for the first time, the output of the example page with the preset comment data block "Please fill in something here!" would look like in figure [18.9](#).

### Please type in something

Textarea #1:

```
Please fill in something
here!
```

Textarea #2:

Figure 18.9.: Presetting values

### 18.5.3. Evaluating forms

It is desirable to evaluate input fields for empty or invalid data and to give feedback to the user if input fields were filled out incorrectly. Form evaluation is a typical task that is performed by a programmer. The error checking and validation processes have to be provided by the controlling Java class. In the following example, the user is asked to fill out the input fields with his name, e-mail address and telephone number. The fields marked with an \* are required, the telephone number is optional (figure 18.10).

After posting the form data to the server by clicking on the OK button, the input fields are checked by the Java class, to see if any of the required (\*) fields were left empty. The page is redisplayed with the data that the user typed in and error messages. The user is

please insert your name and email address  
all fields that have to be filled out are marked with \*

Last name:  \*

First name:  \*

Email:  \*

Tel.:

Figure 18.10.: Evaluating forms Pic 1.

please insert your name and email address  
all fields that have to be filled out are marked with \*

Last name: \*

First name:

Email: \*

Tel.:

### The form wasn't filled out correctly!

Your input is not complete.

You left out: first name.

Figure 18.11.: Evaluating forms Pic 2.

informed that some of the fields were not correctly filled out. In the above example, the user forgot to fill out the first name input field. The following message is displayed and access to the confirmation page denied (figure 18.11).

The Java class uses preset data blocks with text phrases in the template to display the error message in the error process at the end of the form tag. If the input is correct, the error data block remains empty (as initialized in the template). If there are errors (one or more of the first three fields are empty) the page is redisplayed with the error message using the text in the *init* section:

```
...
<!--init-->
<FIELD_OMITTED><![CDATA[Your input is not complete.<br></br>
You left out:  ]]>
</FIELD_OMITTED>
<MISSING_LAST_NAME><![CDATA[last name]]></MISSING_LAST_NAME>
<MISSING_FIRST_NAME>
<![CDATA[first name]]>
</MISSING_FIRST_NAME>
<MISSING_EMAIL><![CDATA[e-mail]]></MISSING_EMAIL>
<GENERAL_ERROR>
<![CDATA[<font color=#FF0000><h3>The form wasn't filled out correctly!
</h3></font>]]>
</GENERAL_ERROR>
<KOMMA><![CDATA[,  ]]></KOMMA>
<POINT>.</POINT>
```

```
<BREAK><![CDATA[<br></br>]]></BREAK>
<NO_VALUE><![CDATA[none]]></NO_VALUE>
<setaction></setaction>
<first_name></first_name>
<last_name></last_name>
<tel></tel>
<email></email>
<error></error>
<!--end init-->
...
<FORM method="post">
...
<INPUT type=text maxLength=40 name=last_name
value=""]><process>last_name</process>
<![CDATA[" size=40]>*
...
<INPUT type="hidden" name="action" value="default">
</FORM>
]]><process>error</process><![CDATA[...
```

After the form data is sent, the Java code fetches the data from the parameters hash table. It then checks the data for missing input by looking for empty strings. If empty fields are found, an error message is generated, and the missing data field names are concatenated with data blocks that contain text from the template to create a more readable output of the error message. If one field is left empty, the integer fields are incremented and the Boolean value of the error is set to true. The missing fields are concatenated using the data block text contained in the template. Last but not least, the error text is returned. Below you can see the code of the Java methods that provide the evaluation of the input fields:

```
private StringBuffer checkInput(Hashtable parameters, CmsXmlTemplateFile
template) throws CmsException {
    boolean error = false;
    boolean emailError = false;
    StringBuffer errorText = new StringBuffer();
    String[] missingFields = new String[3];
    int fields = 0;
    String lastName = ((String) parameters.get("last_name"));
    String firstName = ((String) parameters.get("first_name"));
    String email = ((String) parameters.get("email"));
    String tel = ((String) parameters.get("tel"));
    // check if the fields last_name, first_name
```

```

    // and email are filled out
    if (lastName.equals(""))
        missingFields[fields++] = "missing_last_name";
    if (firstName.equals(""))
        missingFields[fields++] = "missing_first_name";
    if (email.equals("")) missingFields[fields++] =
        "missing_email";
    // create error message when fields are omitted
    if (fields > 0) {
        error = true;
        errorText.append(template.getDataValue("field_omitted"));
        for (int i = 0; i < fields; i++) {
            errorText.append(template.getDataValue(missingFields[i]));
            if (i < fields - 1) {
                errorText.append(template.getDataValue("komma"));
            }
        }
        errorText.append(template.getDataValue("point"));
        errorText.append(template.getDataValue("break"));
    }
    if (error) {
        errorText.insert(0,template.getDataValue("general_error"));
        if (emailError) {
            errorText.append(template.getDataValue("email_error"));
        }
    }
    return errorText;
}

```

The method *getContent()* calls the above seen *checkInput()* method, which returns the error message. The rest of the coding is more or less the same as seen in the previous examples:

```

...
    //no button pressed!
    if (action == null || action.equals("")) {
        templateSelector="default";
        template.setData("setaction","default");
    //the ok button was pressed
    } else {
        String errorText="";
        //check input for errors

```

```
        errorText = checkInput(parameters,template).toString();
        String lastName = ((String)parameters.get("last_name"));
        String firstName = ((String)parameters.get("first_name"));
        String email = ((String) parameters.get("email"));
        String tel = ((String) parameters.get("tel"));
        // set the fields with the given input in any case
        // (even if empty String)
        template.setData("last_name", lastName);
        template.setData("first_name", firstName);
        template.setData("email", email);
        template.setData("tel", tel);
        // if no error show reply template
        if (errorText.length() == 0) {
            templateSelector = "reply";
        } else {
            // show default page again with error message
            templateSelector = "default";
            // set error text
            template.setData("error", ""+errorText);
        }
    }
    return startProcessing(cms, template, elementName,parameters,
        templateSelector);
}
```

#### 18.5.4. Validating Forms

In order to perform a detailed check of the input data, the text must be checked for invalid content. The above example is extended by a simple e-mail checker. The Java method considers the e-mail address to be valid if it contains a dot (.) and an at (@) sign. If this is not the case, the user sees an error message like in figure [18.12](#).

To validate the e-mail address, the Java method *checkInput()* is simply extended in the mail checking section by calling the *validEmail()* method:

```
...
if (email.equals("")) {
    missingFields[fields++] = "missing_email";
} else {
    if (!validEmail(email)) {
        error = true;
        emailError = true;
    }
}
```



please insert your name and email address  
all fields that have to be filled out are marked with \*

Last name:  \*

First name:  \*

Email:  \*

Tel.:

### The form wasn't filled out correctly!

The given e-mail address is not correct.

Figure 18.12.: Validating forms Pic 1.

```
} }
```

...

The method *validEmail()* returns true if the e-mail address is valid or false otherwise:

```
private boolean validEmail(String email) {
    if (email != null
        && email.indexOf('.') != -1
        && email.indexOf('@') != -1) {
        return true;
    } else {
        return false;    }
}
```

### 18.5.5. Using radio buttons and checkboxes

HTML provides checkboxes and radio buttons for limited selections. Radio buttons enable users to specify a selection by clicking on a field rather than by typing in text. Checkboxes represent a group of labeled buttons in HTML. The user is able to check none, one, or more of these buttons that belong to the same group, which is defined by the name of the value. The checkbox is defined in HTML as an input field tag of the type checkbox:

```
<input type=checkbox name="my_name" value="some_value">
```

Radio buttons are very similar to checkboxes, except for the fact that the user has to check exactly one button of the same group. The group is defined by the name of the

**Please give use your feedback**

How do you rate our website?

Excellent  
 Good  
 Average  
 Bad

What do you think has to be improved?

Design  
 Navigation  
 Contents

OK Reset

**Please give use your feedback**

How do you rate our website?

Excellent  
 Good  
 Average  
 Bad

What do you think has to be improved?

Design  
 Navigation  
 Contents

OK Reset

Figure 18.13.: Usage of Radio buttons

radio buttons. Clicking on another radio button switches the old selection to the new one. The defined value is sent by the form when the data is confirmed by clicking on the OK button. Radio buttons are defined in HTML forms by:

```
<input type=radio name="my_name" value="some_value">
```

Figure 18.13 is an example of a form that uses both radio buttons and checkboxes. The user's selection consists of one of four radio buttons within one group, and of three checkboxes within another group.

The radio button labeled *Good* and the checkboxes labeled *Design* and *Contents* were clicked by the user before the confirmation button was activated (figure 18.13). The Java method *checkRadio()* is used to test whether a radio button was checked and then return the value *checked*. The Java method is called by a method tag with two parameters: name and value of the radio button or checkbox in the body template. The parameters are given as a comma separated list in the string in the *<METHOD>* tag. The first value is the name of the radio button or checkbox and the second the value it has when it is checked. The method checks whether the radio button or checkbox was activated by getting the value of the parameter and comparing it to the value it should have when it is selected. If the button was activated the method returns *checked* or an empty String otherwise. The string *checked* inside the *<INPUT>* tag in HTML indicates that this radio button or checkbox has to be rendered as selected. Here is an example how a checkbox is inserted in the template:

```
...
<INPUT type=checkbox name=design value="design"]]>
<method name="checkRadio">design,design
</method><![CDATA [>Design
...

```

## Thank you for your feedback

You have sent the following data:

You rated our website good.

You suggested improving the design of our website.

Figure 18.14.: Radio buttons Pic 3.

The method *checkRadio()* is defined in this way:

```
public Object checkRadio(CmsObject cms, String tagcontent,
    A_CmsXmlContent doc, Object userObject)
    throws CmsException {
    Hashtable parameters = (Hashtable) userObject;
    String returnValue = "";
    int kommaIndex = tagcontent.indexOf(",");
    if (kommaIndex > 0) {
        String radioGroup = tagcontent.substring(0, kommaIndex);
        String radioName = tagcontent.substring(kommaIndex + 1);
        String selected = (String) parameters.get(radioGroup);
        if (selected != null && selected.equals(radioName)) {
            returnValue = "checked";
        }
    }
    return returnValue;
}
```

The *getContent()* method processes the checked boxes by getting them from the parameter's hash table and displaying the result on the confirmation page (the template section "reply") of the template. Predefined data blocks in the template that contain text are used to process a readable error message. Figure 18.14 shows the message that the *getContent()* method produces if the input is complete (a radio button and one or more checkboxes were checked).

## 18.5.6. Presetting the values of select boxes and radio buttons in Java

As we have seen in the previous examples, presetting the values of radio buttons is a bit tricky. The XML template mechanism tries to take care of this problem by providing a special template class *com.opencms.defaults.CmsXmlFormTemplate* that facilitates the control of select boxes and radio buttons from within the controlling Java class. This template class allows radio buttons and select boxes to be selected before the body template is started. If you want to create radio buttons or select boxes with Java, your Java class has to extend the *CmsXmlFormTemplate* class rather than the normally used template class *com.opencms.template.CmsXmlTemplate*.

The standard HTML input tag select box that is used in the template has to be replaced by the following method tag, which is structured in much the same way the original HTML tag is. In its plainest form the tag looks like this:

```
]]><select name="select_1" method="my_selectorMethod"/><![CDATA[
```

The standard radio button input field is replaced by the following method tag:

```
]]><radiobutton name="radio_1" method="my_selectorMethod" order="row"/><![CDATA[
```

The main difference between the two basic tags shown above is that you can display the radio buttons in a row or in a column layout.

The layout of the radio buttons and select boxes is defined in a special file. Without this file, the new radio and select tags will not work. You need to extend your workplace by the *HTMLFormDefs* file in the */system/workplace/templates* directory because this file is used by the methods *handleRadiobuttonTag()* and *handleSelectTag()* of the *CmsXmlFormTemplate* class.

If you want to use additional statements in these tags (e.g. to change the size) have a look at the *HTMLFormDef* file that can be found in your */system/workplace/templates* directory.

Below is an example with the values "elite" for the radio button and "other" for the select box set by the Java method (figure 18.15).

You can activate the radio button that should be preselected by setting the data in the session by a method in your Java class (normally the *getContent()* method). In this example, the Java code

```
session.putValue("checkradio", "my_radioname_1");
```

activates the radio button *my\_radioname\_1*. The layout is set with "order=" to rows or columns.

The template calls the method *my\_radioMethod()* in the radio element. The checked radio button that was stored in the session is displayed with the other names and values that were added to this method.

**Radiobutton value was set by Java!**

What is your rating ?

- elite
- mostly harmless
- harmless
- poor

Please select something from the checkbox:

Figure 18.15.: Radio buttons in Java.

```
public Integer my_radioMethod (CmsObject cms, Vector values,
    Vector names, Hashtable parameters)
    throws CmsException {
    int index;
    String checkradioValue = (String)session.getValue("radio_1");
    // add values for the checkbox
    values.addElement("my_radiovalue_1");
    ...
    // add corresponding names for the checkboxvalues
    names.addElement("my_radioname_1");
    ...
    // used for the assignment of the button
    if checkradioValue.equals("my_radiovalue_1") index = 0;
    ...
    return new Integer(index);
}
```

To set the initial value (e.g. in the *getContent()* method or even in the *my\_radio\_method()* itself), you have to create a new session (or fetch an existing one) and store the value that is to be displayed when the body template is first started in the session (here *my\_radiovalue\_1()*):

```
...
CmsSession session = (CmsSession)
cms.getRequestContext().getSession(true);
...
session.putValue("radio_1", "my_radiovalue_1");
```

You can activate the select box entry that should be initially displayed by setting the data to activate the select box item to *my\_name* in the session by a method in your Java class (normally the *getContent()* method). Your Java code should look like this:

```
session.putValue("checkselect","my_selectname_1");
```

The template calls the method *my\_selectorMethod()* in the select element tag. The selected select box item that was stored in the session is displayed. The other select box names become visible after the select box is opened.

```
public Integer my_selectorMethod (CmsObject cms, Vector values, Vector
names, Hashtable parameters)
    throws CmsException {
    int index;
    String checkselectValue = (String)
    session.getValue("selectbox_1");
    // add values for the selectbox
    values.addElement("my_selectvalue_1");
    ...
    // add corresponding names for the selectboxvalues
    names.addElement("my_selectname_1");
    ...
    // used for the assignment of the button
    if checkselectValue.equals("my_selectvalue_1") index = 0;
    ...
    return new Integer(index);
}
```

To set the initial value (e.g. in the *getContent()* method or even in the *my\_selector\_method()* itself), you have to create a new session (or fetch an existing one) and store the value that is to be displayed when the template is first started in the session (here *my\_selectvalue\_1()*):

```
...
CmsSession session = (CmsSession)
cms.getRequestContext().getSession(true);
...
session.putValue("selectbox_1","my_selectvalue_1");
```

### 18.5.7. Using session management with forms over multiple pages

The next example shows how you can use session management to create an HTML feedback form that extends over multiple pages. A session provides a way to identify a user over multiple subsequent requests and store user-related data.

Without session management, it would be impossible to process data that was entered on the first form page in subsequent form pages (because it would not be possible to recognize that the two requests came from the same user). With session management, the data is stored in the session related data area, and is therefore saved for later use.

The example consists of two subsequent HTML forms that request feedback from the user. On the first page, the user rates the website and on the following page he/she is asked to provide personal information such as name and e-mail address. When all of the information has been provided, a confirmation page will show the user's input. Because of the session management, the data that were sent on the first page are saved in the session and are available when the confirmation page is created. In this example, all of the values in the session are stored in a newly created hash table. This is the easiest way to ensure that other values in the session are not affected by your activities. Imagine multiple programmers developing different HTML forms using session management. They can easily get confused if they store parameters with the same name in the session. It is safest to create an extra hash table and to store in it the session with a (hopefully) unique name.

Below is a piece of code that shows how this is done :

```
if (action == null || action.equals("")) {
    if (session.getValue("myForm") != null) {
        session.removeValue("myForm");
    }
    session.putValue("myForm", new Hashtable());
    template.setData("setaction", "page1");
} else {
    if (action.equals("page1")) {
        statements...
    }
}
```

Here you can see how the program recognizes on which page the user currently is. The hidden HTML form input field *action* is used to indicate the current page. The value of the parameter *action* will be an empty string when the site is requested for the first time. In this case, the system checks if the hash table that stores the parameters is already contained in the session. If this is the case, it will be removed and a new one will be created to store the new values. The value of the *action* parameter is then set to *page1*, and when the page is requested again the program will check the input fields of *page1*.

When the fields are filled out correctly the values are stored in the hash table:

```
Hashtable formData = (Hashtable) session.getValue("myForm");
formData.put("name");
formData.put("email");
```

These values can be fetched again later using the *get()* method of the hash table. The rest of the program is similar to the above examples because the validation of the HTML input fields is performed in the same way as in the previous examples. Basically, session management allows you to store data that is related to a special session over a period of time. In general, the session automatically expires if it has been inactive for more than 30 minutes.

### 18.5.8. Sending e-mails

After the forms have been processed you have all the information from the user that you need. To determine the usability of the collected information (e.g. for the owner of the website) it has to be forwarded to someone for analysis. One of the easiest and most common ways of exchanging information on the web is sending an e-mail. All you have to do to send the collected information by e-mail, is to import a special class and add some code to the method that will execute the mail delivery routine. The XML template class used to send e-mails is the class *com.opencms.defaults.CmsMail*. This class has to be included in an import statement in your template class if you want to use it to send an e-mail.

To allow easy changes of your data input, you should store every text that should appear in the email in data blocks in your template. This enables you to change the text without the need to recompile your Java code. Below is an example of such an email text data block in a template:

```
...
</TEMPLATE>
<EMAILTEXT>
<![CDATA [
Hello ]]><PROCESS>name</PROCESS><![CDATA [,
This is an automatic email reply.
Your data:
Name:  ]]><PROCESS>name</PROCESS><![CDATA [
Surname:  ]]><PROCESS>surname</PROCESS><![CDATA [
email:  ]]><PROCESS>email</PROCESS><![CDATA [
Tel.:  ]]><PROCESS>number</PROCESS><![CDATA [
Date:  ]]><PROCESS>date</PROCESS><![CDATA [
Thank you!
]]>
</EMAILTEXT>
</XMLTEMPLATE>
```

In the *getContent()* method the data blocks are fetched from the template. If they are fetched for the first time, the data blocks are initialized as empty strings. The data



is validated after the confirmation button is pressed (clicked on). If the validation is successful, the data is sent to the confirmation page of the template and the whole e-mail text is fetched as one single string by the *getContent()* method. The e-mail text can now be sent as the e-mail content:

```
...
//read the content (blank at the first start)
String vorname=(String)parameters.get("surname");
String name=(String)parameters.get("name");
...
// check if the form is displayed for the first time.
//If so, preset all input fields with blanks
if ((action==null) || (action.length()<1)){
template.setData("surname","");
template.setData("name","");...
} else {...
// the form is not called the first time,
// analyze the given data
...
//data was validated, set data in body template
template.setData("name", name);
String mailText = template.getProcessedDataValue("EMAILTEXT");
...
//send mail and start the processing
```

It is important to validate the collected values **before** you send the e-mail. To send the e-mail in Java, a code fragment like the following has to be added to your *getContent()* method:

```
...
//send mail and start the processing
//the recipient(s) of the mail
String receiver [] = {"receiver_1@somewhere.com", "receiver_2..."};
CmsMail mail =
new CmsMail(cms, sender, receiver, subject, content, "text/plain");
mail.start(); //start thread to send the mail
...
```

The variables passed to the constructor of the *CmsMail* class are:

- cms (type `com.opencms.file.CmsObject`) object to access system resources.

- sender (type `java.lang.String`) the e-mail address of the mail sender.
- receiver (type `java.lang.String[]`) one or more e-mail addresses of e-mail recipients.
- subject (type `java.lang.String`) the subject of the e-mail.
- content (type `java.lang.String`) the body (content) of the e-mail.
- type (type `java.lang.String`) the mime-type of the e-mail content. The mime-type is usually set to "text/plain" for pure text mails.

Note that there are other constructors of the class *CmsMail* that allow to send e-mail to cc and bcc recipients. Just have a look at the class if you need this functionality.

You can attach additional files to the mail by using the method:

```
addAttachement(String content,String type)
```

**Note:** The sending of the e-mail runs as a separate thread. This means the program is not blocked, even if the mail can not be sent right away!

The mail server used to send the mails can be configured in the registry of OpenCms. The registry resides in the file *registry.xml* in the folder *WEB-INF/config* of your webapplication.

Here is the part of the *registry.xml* file where the mail server is configured:

```
<smtpserver>my.smtp.server</smtpserver>  
<smtpserver2>alternative.smtp.server</smtpserver2>  
<defaultmailsender>nobody@nowhere.com</defaultmailsender>
```

The tag `<smtpserver>` contains the server that will be used first. If this server cannot be accessed the system tries to use the server given in the tag `<smtpserver2>`. The `<defaultmailsender>` is used for the e-mail address of the sender if no sender was given when creating the e-mail.

### 18.5.9. Personalization

XML templates enable the creation of personalized web content and the management of users and their profiles. With OpenCms' built in user management, you can store user-related data and use advanced features such as the management of different groups of users. Web page content is easily adapted to each user's profile and preferences. In order to provide personalized web content, the user's data and preferences must be known by the system. A user that wants to take advantage of personalized web content first has to register. He/she can either specify their preferences manually or let them be automatically



Figure 18.16.: Personalization Pic 1.

generated by enabling the system to analyze their surfing habits. The next example shows how a user is identified by means of a simple login box and how the user data is extracted from the system. The example program presents a login screen to the user. After the user has successfully logged in, his/her personal data such as name and address are displayed on the screen. To test the example, you have to enter the name and password of an existing OpenCms user. You can either create a new user for the purpose of a test, or use the one that you use to access the system. When you test the example, be aware that you will be working on the system under the user account that you logged in under. You have to log in again in order to work under your old account. The login screen is a simple HTML form with two text input fields (figure 18.16). The login is performed by the method *login()* of the class *CmsObject*. The method has the following structure:

```
public String login(String username, String password)
throws CmsException;
```

The method returns the login name when the login was successful, otherwise it throws a *com.opencms.core.CmsException*. The exception can be captured and an appropriate error message showed to the user.

```
...
try {
    cms.loginUser(name, pass);
    ...
    templateSelector = "logged_in";
} catch (Exception e) {
    e.printStackTrace(System.err);
    template.setData("setaction", "send");
    templateSelector = "error";
}
...

```

The user data is accessed via the methods of the *com.opencms.file.CmsUser* class. Several get methods can be used to fetch user data such as the first and last name and the e-mail

**Hello some User!**

```
Personal data:
Last name:      User
First name:     some
e-mail:         e@mail.de
Street:
Zip Code:
City:
Member of groups: Users
                  Guests
```

Figure 18.17.: Personalization Pic 2.

address. A hash table is used to store additional information, which is defined and passed using the methods:

```
public setAdditionalInfo(String key, Object obj);
public Object getAdditionalInfo(String key);
```

Figure 18.17 shows the output for the above logged in example user.

In addition to the name and e-mail address, additional information fields can be used to capture use-related data that is essential to create personalized content.

An alternative to using the additional information hash table would be to customize OpenCms' user management. Since OpenCms is an open source project you are free to modify the user management by inheriting the existing classes or extending the existing ones with additional features.

# Index

- addAttachement, 210
- Back Office, 59
- Backoffice modules, 71
- Bean class, 54
- bodys, 130
  
- caching, 155
- CDATA, 129
- check() , 79
- checkbox, 201
- CmsCacheDirectives, 138, 156, 180
- cmsframe, 176
- CmsObject, 101
- CmsPlausibilizationException, 79
- CmsXmlTemplateFile, 141
- Concurrency, 53
- content, 129
- Content Definition, 59
- contenttemplates, 130
  
- data blocks, 131
- database, 104
- defaultbodies, 130
- Distributed Objects, 54
  
- EJB, 53
- element, 164, 165
- element cache, 155
- ELEMENTDEF, 164, 165
- elements, 130
- Enumeration keys, 77
- error message, 197
  
- export property, 91
- exportname, 92
  
- file upload, 79
- frames, 174
- frametemplates, 130
  
- getCacheDirectives(), 138, 156, 179
- getContent, 133
- getContentDefinitionClass(), 75
- getCreateUrl(CmsObject cms), 75
- getDeleteUrl(CmsObject cms), 75
- getEditUrl(CmsObject cms), 75
- getFieldMethods(), 66
- getFieldNames(), 65
- getFilterMethods(), 65
- getFolderCurrent, 166, 171
- getFolderParent, 166, 171
- getFolderRoot, 166, 171
- getFrameQueryString, 175
- getFrameTarget, 176
- getLinkSubstitution, 90
- getNavCurrent, 165, 166, 169
- getNavFold, 166, 169
- getNavParent, 166, 170
- getNavRoot, 166, 171, 176
- getNavTree, 166
- getNavtree, 170
- getOwnTemplateFile, 141
- getPropertyCurrent, 166, 172
- getPropertyParent, 166, 172
- getPropertyRoot, 166, 172

- getPropertyUri, [166](#), [172](#)
- getRequestContext(), [184](#)
- getSession, [184](#)
- getSetupUrl() , [78](#)
- getSubFolders, [183](#)
- getTitle, [122](#), [164](#)
- getUniqueId(), [66](#)
  
- history, [37](#), [116](#)
- Home interface, [54](#)
- https, [91](#)
  
- internal, [130](#)
- isCacheable(), [156](#)
- isLockable(), [66](#)
  
- Javascript, [150](#)
  
- Launcher Manager, [101](#)
- link tag, [90](#)
  
- Naming, [54](#)
- navcount, [166](#), [169](#)
- navcurrent, [166](#), [167](#)
- navend, [166](#), [168](#)
- naventry, [166](#)
- navlevel, [166](#), [168](#)
- navlink, [166](#), [167](#)
- navstart, [166](#), [168](#)
- navtext, [166](#), [167](#)
  
- Persistence, [54](#)
- Primary key, [54](#)
- Project permanent, [114](#)
- Project temporary, [114](#)
- Projectmechanism, [113](#)
- publish project, [116](#)
- publish resource, [37](#)
  
- radio, [202](#)
- Remote interface, [54](#)
- resource broker, [101](#)
- resource type, [105](#)
- resources backup, [113](#)
- resources data structure, [113](#)
- resources offline, [113](#)
- resources online, [113](#)
  
- Security, [54](#)
- Server File System, [97](#)
- servlet engine, [102](#)
- session management, [184](#)
- session.putValue, [206](#)
- setup page, [78](#)
- Static Export, [89](#)
- Stylesheets, [150](#)
- synchronization, [97](#)
- synchronizationmanagement, [97](#)
- synchronize, [97](#)
- syncpath, [97](#)
- syncproject, [97](#)
- syncresource, [97](#)
  
- TEMPLATE, [129](#)
- Template Mechanism, [101](#)
- templates, [129](#)
- Transactions, [54](#)
  
- undeleate, [36](#)
- undo, [36](#)
  
- validEmail, [201](#)
- VFS, [106](#)
- Virtual File System, [97](#), [106](#)
  
- XML, [128](#)
- XML-tags, [128](#)
- XMLTEMPLATE, [129](#)