Infopark CMS Fiona

System Administration / Development

Infopark CMS Fiona

System Administration / Development

While every precaution has been taken in the preparation of all our technical documents, we make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein. All trademarks and copyrights referred to in this document are the property of their respective owners. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without our prior consent.

Contents

1 Introductory Comments 10

2 The Architecture of Infopark CMS Fiona 11

2.1 Tasks of the CMS Components 12

2.1.1 Portal Manager 12

2.1.2 Content Management Server 12

2.1.3 Template Engine 13

2.1.4 Infopark Search Cartridge 14

2.2 Communication between CMS Components 14

2.3 Scaling and Reliability 15

2.3.1 Live Webserver 15

2.3.2 Template Engine 15

2.3.3 Portal Manager and Infopark Search Cartridge 15

2.3.4 Content Management Server 15

3 Configuring Infopark's CMS 16

3.1 Preparing the Demo System for Productive Use 16

3.2 The File Structure of the CMS 17

3.3 Removing the Demo Content 18

3.4 Serving Static Data Using Trifork Application Server 19

3.5 Changing Passwords 20

3.5.1 Content Management Server 20

3.5.2 Template Engine 21

3.5.3 Content Management Server, Portal Manager and GUI 21

3.5.4 Trifork Application Server 21

3.6 Configuring the HTML-Editor 22

3.6.1 Adapting XHTML Validation 22

3.6.2 Integrating Dictionaries for Spelling-Checking 22

3.7 Updating the HTML Editor (EOP) 23

3.8 Updating Trifork T4 24

3.8.1 Updating Trifork T4 under Unix 24

3.8.2 Updating Trifork T4 under Windows 25

3.9 Enabling HTTPS in Trifork T4 26

3.9.1 Generating Key Pairs 26

3.9.2 Certifying a Key Pair in the Keystore 27

3.10 Displaying Dynamically Generated Content in the Preview 27
3.10.1 Java Server Pages 28
3.10.2 PHP Scripts 28
3.10.3 Active Server Pages 29
3.11 Respecting CMS Read Permissions in the Preview 29
3.12 Configuring Internet Access for the Link Checker 30
3.12.1 Link Checker Configuration 31
3.12.2 Firewall Configuration 31
3.12.3 Proxy Configuration of the CMS Systems 31
3.13 Configuring Access to the Online Help 31
3.13.1 Making the PDF documentation available 32
3.14 Configuring Reminder E-Mail Notification 33
3.14.1 Specifying the Mail Server 33
3.14.2 Setting an Execution Schedule for Sending E-Mail 33
3.14.3 Specifying the E-Mail Addresses of Users 34
3.14.4 Changing the E-Mail Message 34
3.15 Configuring Web Applications 34
3.15.1 Defining MIME Types 34
3.15.2 Configuring Error Pages 35
3.15.3 Defining the Base URL 36
3.16 Integrating a Database 36
3.16.1 Procedure 36
3.16.2 General Notes about Database Usage 38
3.16.3 Oracle 38
3.16.4 Sybase 40
3.16.5 MySQL 40
3.16.6 MS SQL Server 42
3.16.7 DB2 43
3.17 Integrating the Template Engine 43
3.18 Integrating the Search Server 45
3.18.1 Advanced Search in the Editorial System 45
3.18.2 Incremental Export 46
3.18.3 Static Export 46
3.19 Creating CMS Instances 47

4 The Starting Procedure of the CMS Applications 50
4.1 Tasks at Startup 50
4.2 Registering Tcl Procedures 51

5 The System Configuration of the CMS Applications 52
5.1 Format of the Configuration Files 52
5.2 Entries in the System Configuration 53
5.2.1 content 53
5.2.2 export 55
5.2.3 gui 59
5.2.4 indexing 62
5.2.5 licenseKey 64
5.2.6 searching 64
5.2.7 server 65
5.2.8 tuning 67
5.2.9 userManagement 68
5.2.10 guiPreferences 70
5.2.11 rolePreferences 72
5.2.12 Priority 72
5.2.13 Allowed Values 72

6 Tcl Interface Functions 74
6.1 File Conversion Functions of the Content Management Server 74
6.2 System Procedures (Functions) 75
6.2.1 Post-Action Function (Including E-Mail Notification) 75
6.2.2 Standard Post Action Function 76
6.2.3 E-Mail Notification 76
6.2.4 Link Functions 76
6.2.5 Thumbnail Function 79
6.2.6 Function for Checking Passwords 79
6.2.7 SystemExecute Procedures 80
6.2.8 Formatter Procedures 81
6.3 The User Manager Interface 81
6.3.1 Requirements for Connecting an External User Manager 83
6.3.2 Particular Requirements for an LDAP Server 84
6.3.3 Integrating an LDAP Server 84

6.3.4 The User Manager API 86
6.4 The Concept 86
6.5 Procedures of the User Manager API 86
6.5.1 Parameters of the LDAP/ADS User Manager Configuration 88

7 Configuring Functions and the Appearance of the Content Navigator 93
7.1 Defining Menu Entries 94
7.2 Defining the Menu and the Toolbar 94
7.3 Configuring the Details View 96
7.3.1 Tabbed Details Overview 96
7.3.2 Adding Fields to the Details Overview Easily 97
7.3.3 The Configuration Files for the Details Views 98
7.4 Velocity Templates 101
7.4.1 Keywords in the context of the current CMS file 101
7.4.2 Keywords in the ObjectTool ($object.key) 102
7.4.3 Keywords in the ContentTool 103
7.4.4 Keywords in the AttributeGroupTool 104
7.4.5 Keywords in the SortOrderTool 104
7.4.6 Keywords in the AttributeTool 105
7.4.7 Keywords in the LinkListTool 105
7.4.8 Keywords in the LinkTool 106
7.5 Configuring Themes 106
7.6 Authentication 107

8 Configuring Custom Commands 108
8.1 Definition of a Menu Command 108
8.2 Tcl Procedures 111
8.3 Wizards 112
8.3.1 Wizard Tags 115
8.4 Example of Usage 123
8.4.1 Reloading the File Hierarchy 124
8.4.2 Selecting a File 124
8.4.3 Informing the GUI about Content Changes (from Fiona 6.6) 124
8.4.4 Sample Wizard 127
8.5 Servlets 128
8.6 Opening other Pages with a Menu Command 129

9 Export Options ... 130
9.1 Incremental Export ... 130
9.2 Static Export ... 130

10 The PDF Generator ... 131
10.1 How the PDF Generator Works ... 132
10.2 Installing the PDF Generator ... 132
10.3 Configuring the PDF-Generator ... 133

11 Saving and Restoring Data ... 134
11.0.1 General Data Backup ... 134
11.0.2 Saving the Content, System and Runtime Configuration ... 134
11.1 Complete Dump/Restore ... 135
11.1.1 Saving Data ... 135
11.1.2 Restoring Data ... 135
11.2 Partial Dump/Restore ... 137
11.2.1 Saving Data ... 138
11.2.2 Restoring Data ... 139

12 Migration ... 142
12.1 Migration Requirements ... 142
12.1.1 System Requirements ... 142
12.1.2 Database ... 142
12.1.3 Installation of the current version of CMS Fiona ... 143
12.1.4 Making the old installation accessible ... 143
12.1.5 Creating instances in the new Fiona installation ... 143
12.1.6 Transferring update records ... 143
12.1.7 Hard disk space ... 144
12.2 In-Place Migration ... 144
12.2.1 Migrating a Distributed Installation ... 145
12.2.2 Migrating and Changing the Database Product ... 145
12.3 Testing the Migrated Instance or Putting It into Operation ... 145
12.4 Migrating Older Fiona Versions ... 146

13 The ContentService Interface ... 148
13.1 Basic Principles ... 148
13.2 Querying the List of Available CMS Files ... 148

13.3 Downloading CMS Files 149
13.4 Uploading Files 149
13.5 Deactivating CMS Files 150
13.6 Using the Content Service Interface 150
13.6.1 Setting up the Content Service as a Job 150

14 The Streaming Interface 151
14.1 Basic Principles 151
14.2 Using Streaming 152

15 The XML Interface 154
15.1 Using HTTP to Access CMS Components 154
15.2 Communication via XML Documents 154
15.3 CRUL Payloads 155
15.3.1 Header Element 156
15.3.2 Request Element 157
15.3.3 Response Element 158
15.4 From Tcl to XML 159
15.5 Requests as Templates and the Presentation of the Result Data 165
15.6 Error handling 169
15.7 Examples 170

16 The CRUL DTD 172

1

1 Introductory Comments

This document is designed for system administrators who would like to install and configure Infopark CMS Fiona. It is also designed for developers who would like to write e.g. installation-specific callback or filter procedures.

Both groups should know the operating system environment in which the CMS is to be installed or administrated. Experience with the database and webserver configuration is required.

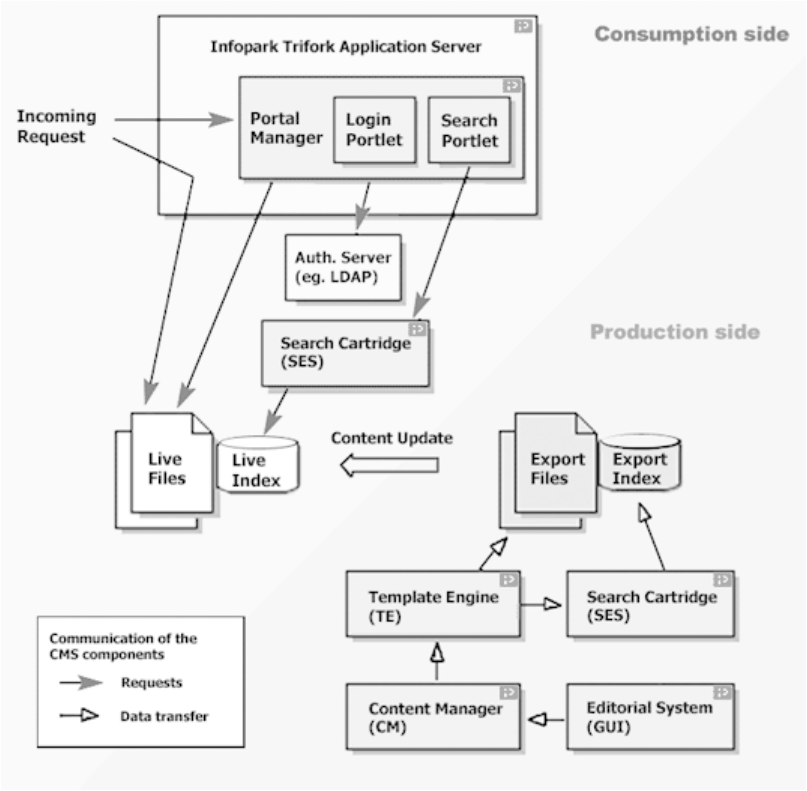
2

2 The Architecture of Infopark CMS Fiona

Infopark CMS Fiona consists of several components with which content can be edited (editorial system) and made available (live system) within the context of an online-publication (for example on a website).

This section describes the tasks of the components with respect to the system as a whole and how they communicate with each other in order to fulfill their respective tasks.

Jumping ahead with the function descriptions in the next section, the following diagram shows the architecture of the editorial system and the live system as well as the data flow during a content update and the request forwarding on the live system:



The Infopark Search Cartridge is an optional component which needs to be licensed separately.

2.1 Tasks of the CMS Components

2.1.1 Portal Manager

Like the Template Engine, the Portal Manager is an optional component for the live system. It has been implemented as an J2EE compliant web application that runs in every J2EE application server. It serves documents from any directory tree, it contains a portlet container, and provides personalization functions (access control, news channels, user-specific portlet settings).

Access control allows you to deliver documents or parts of them only to the members of certain user groups. Whether a user belongs to a group can be determined via an LDAP interface, for example.

The Portal Manager's Portlet engine is compatible with JSR 168 portlets. Portlet calls can be placed into documents directly in the editorial system.

Using the Template Engine, news feeds can be generated which the Portal Manager delivers as RSS feeds, taking into account the access permissions of individual users. These feeds can be read using the news portlet or any RSS news reader. Different feeds (channels) and the access permissions can be maintained in the editorial system.

2.1.2 Content Management Server

The Content Management Server is the main component of the Infopark CMS Fiona. It serves as an editorial system which is also used to maintain content. The Content Management Server is often also called "Content Manager". The server can be operated conveniently with a browser via the separate HTML user interface (HTML GUI). The HTML GUI is a web application and communicates with the Content Manager via the XML-based protocol CRUL. For previewing dynamically generated pages, a suitable websever can be integrated.

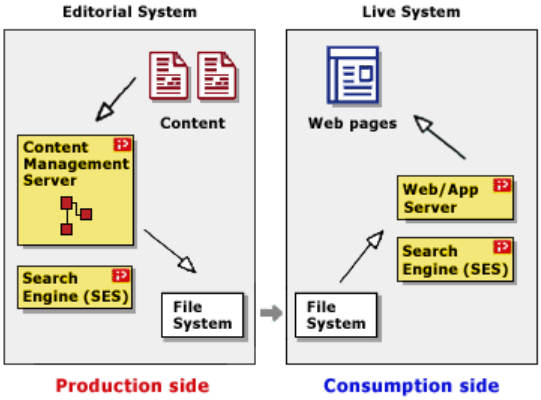
The content is maintained in a folder hierarchy. Analogous to directories with their subdirectories and files, a folder is a CMS file that can have files in it such as documents or images or further folders. The hierarchy formed by the folders - the file tree - is called the folder hierarchy.

By the so called export the files in the folder hierarchy are converted into web pages, i. e. to files in the file system. During this process the hierarchy is preserved so that the result of a complete export is your complete website.

When files are exported, layout files (also called "layouts") you have created are used. They are special files, which are not exported but contain instructions, which control the export of the files. To a certain extent, a layout file forms the frame within which folders and documents are exported. In this way, layouts can give the created web documents a consistent appearance. They allow you to reuse content and to define the layout of your complete site at a few central places.

Static Export

The Content Manager can export your file hierarchy, or sections of it, to a directory. The documents which are generated during this process can then be transferred to the live system using, for example, a script. They are not subject to any change until they are updated after the next export. The following figure illustrates this.



Using this export method, the folder hierarchy or a subtree of it is exported as a whole. For larger websites or greater performance demands, the Template Engine provides an optimized method which is based on caching and precomputing (incremental export).

Next to the Content Management Server, the optional Infopark Search Server can be used on the editorial system for indexing and searching content using a great variety of search criteria. The Infopark Search Server can also be used on the live system, even independently of the Template Engine, if required.

The Content Management Server uses a database to store most of user and administrative data.

2.1.3 Template Engine

The Template Engine is a CMS component that can be used on the live system. Its task is to incrementally export content in the background. Especially for larger websites the Template Engine has numerous advantages:

- Using layout files, content is exported on the live server. This happens in the background, and the exported web documents are stored in a directory hierarchy which is offline. Therefore, the documents on the live website are neither replaced nor deleted by this procedure.
- Content is exported incrementally. This means that only files which have been modified are exported, and deleted ones are removed. This saves the live server's resources.
- The website which has been created and updated in the background is regularly activated. Activation does not require data to be copied. Instead the webserver is directed to the directory tree which has been offline until then. Therefore, this happens very fast so that the webserver can immediately deliver the static documents.
- After activation, the directory tree which is now offline is synchronized with the live hierarchy to make the status quo available for further updating. This makes it unnecessary to re-export the complete folder hierarchy after every activation procedure.

If the Infopark Search Server is used, the Template Engine also activates the indexes which have been updated simultaneously.

The Template Engine receives modified file versions from the Content Management Server. The data is transferred via the XML interface (i. e. via HTTP connections) of the applications. The instruction to activate the updated website is also issued by the Content Manager.

The Template Engine requires a database to store file information, for example. The exported documents can be served with any webserver whose document root directory is configurable, for example with the Trifork Application Server delivered with the CMS.

2.1.4 Infopark Search Cartridge

The Infopark Search Cartridge is an optional component of Infopark CMS Fiona. It consists of the Search Server and the Autonomy (previously Verity) Search Engine, and makes high-quality search functions available on the side of the editorial system as well as the live system. The Search Cartridge can be used independently of the Template Engine.

On the editorial system the Search Server indexes files together with their contents and administrative data (such as the file format). Via the Content Navigator (the HTML user interface of the CMS) of the Content Manager you can comfortably search for files that match one or more search criteria.

On the live system the Search Server indexes the exported files, i. e. the pages the webserver is to deliver. On the basis of supplied Java functions, the search forms used on the live server can be created and maintained in the editorial system.

All indexing and search requests are transferred from the Content Manager, the Template Engine, or Java Server Pages to the Search Server which communicates them to the Search Engine and returns its results.

The Infopark Search Cartridge does not require a database to store its indexes. It uses so-called collections consisting of several files in a proprietary format.

2.2 Communication between CMS Components

The CMS server applications have an XML interface they use to communicate with each other and which allows clients to exchange data with the applications. An application or a client sends an XML document to another application by sending an HTTP POST request to the target application, specifying the URL `/xml`. The target application processes the data and instructions contained in the XML document and returns the result in the HTTP response document.

Since the CMS applications maintain different types of data, the XML documents they are able to process and to return are structured differently. The structure of the XML documents is defined by a DTD (Document Type Definition). Each DTD therefore describes the format of the documents which client and server must observe if they want to communicate syntactically correct and „understand" each other.

CRUL

CRUL, the Content Retrieval and Update Language is the most extensive and an important protocol for application developers. Using CRUL, clients can almost completely control the Content Management Server, i. e. retrieve and modify data, for example. The HTML user interface (NPS Navigator) also uses CRUL.

MISE

MISE, the Method of Interacting with Search Engines is the protocol used for requests to the Search Server. This server is part of the Infopark Search Cartridge.

MISE allows you to index files and web documents, to remove indexed documents from the index, and to make search queries.

2.3 Scaling and Reliability

Websites with an increasing number of visitors require a scalable system, i. e. a system whose response time improves significantly if its hardware is extended. Often measures to increase reliabilty also need to be taken.

2.3.1 Live Webserver

The performance of the system running the live webserver can be enhanced by adding additional processors to it. Furthermore, it is possible to operate several computers with webservers among which the requests are distributed evenly by means of a load balancer. This also enhances reliability. The Template Engine includes a mechanism for providing the local directory hierarchies of the individual webservers with documents.

2.3.2 Template Engine

Frequent or extensive content updates sometimes require all the documents on a website to be exported again. As a consequence, the export phase may become rather time-consuming which has the effect that new content is activated with a delay. Next to restructuring the content in order to minimize the number of dependencies between the documents, the Template Engine can be run on a dedicated computer equipped with several processors. The master-slave architecture of the Template Engine makes it possible to define the number of slaves that are to export the content in parallel.

2.3.3 Portal Manager and Infopark Search Cartridge

A large number of personalization requests to a portal can cause high load which leads to performance collapses. If the Portal Manager is running on the same system as the webserver, non-personalized documents will be delivered late as well. If, however, the Portal Manager is operated on a separate machine, non-personalized documents are still available, even if the Portal Manager's reaction is delayed because the hardware is working at full capacity.

This is also true for the Search Server which is part of the Infopark Search Cartridge. If the Server is run on a separate computer, a large number of simultaneous search requests will not automatically cause dynamic documents to be delivered with a delay as well.

2.3.4 Content Management Server

If many users are working simultaneously with the Content Management Server via the Content Navigator, insufficient hardware can lead to performance collapses. Also in this case, additional processors or dedicated machines for the webservers, the HTML GUI, or the Content Manager will significantly improve response time.

3

3 Configuring Infopark's CMS

3.1 Preparing the Demo System for Productive Use

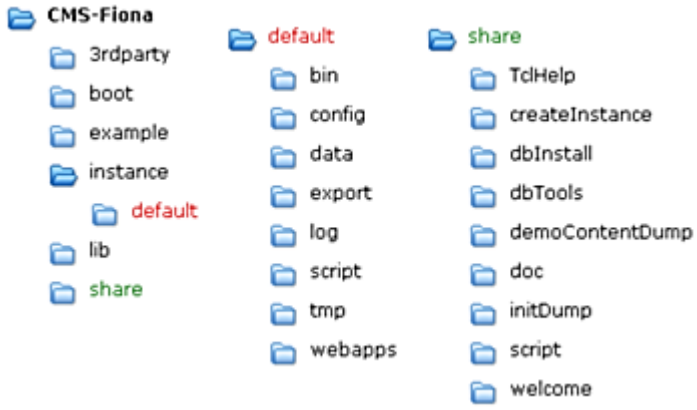
Normally, the requirements regarding network and data security as well as performance are different from running a demo version. For this reason, the following points should be taken into account when switching to productive use of Infopark CMS Fiona.

- Change the password of the root user and other passwords. See Changing Passwords.
- Replace the supplied SQLite database with a professional database. With respect to performance and data security, SQLite does not meet the requirements of productively used systems and therefore has not been approved for productive use. See Integrating a Different Database.
- Set-up a backup procedure. Create a dump in regular intervals using a cron job, for example. See Dumping and Restoring Data.
- For preserving the performance of a Sybase database, the scripts for updating the indexes need to be executed in regular intervals. See Notes on Database / Operating System Combinations.
- If you imported the demo content, remove it by restoring your own dump or the supplied initialization dump. See Dumping and Restoring Data or Removing the Demo Content, respectively.
- If you wish to export content incrementally, integrate the Template Engine. See Integrating the Template Engine. How static data can be served using the Trifork application server is described in section Serving Static Data Using the Trifork Server.
- If dynamic content is to be served, install a web server. For the preview, see Displaying Dynamically Generated Content in the Preview. The information in this section also applies to the live server.
- Immediately after delivery, the GUI can be operated using SSL. For this, use the port configured in the Trifork server (default: 8443). The port can be changed via the administration console of the Trifork server: `http://my.npsserver:8090/console`. Use your CMS host name instead of *my.npsserver*.

  If an official certificate is to be integrated instead of the supplied dummy certificate, place an Apache webserver with active mod_ssl in front of the Trifork server. Alternatively, you can generate a new private key in the JKS Keystore using the key tool and then send the corresponding certification request to the certification authority. Keys or certificates originally intended for use with the Apache webserver cannot be used directly in the Trifork server.
- If your content does not contain JSPs, you can increase the performance of the Trifork server by removing the `-devel` option from the `TRIFORK_ARGS` in the file `config/rc.npsd.conf`. (The server compiles JSPs dynamically only if the `-devel` option has been specified.)

3.2 The File Structure of the CMS

After the installation, Infopark CMS Fiona can be found in a single directory. This directory has the following structure:



The subdirectories have the following contents:

- **3rdparty** contains third party software. In particular, *ImageMagick* for creating thumbnails, *htmlExport* for converting files into the HTML format, the *libFoundation*, *Tcl*, as well as *vdk* (components of the Infopark Search Cartridge) are included.
- **boot** contains a script with which all existing instances can be started. This script can be included in the runlevels of your Linux distribution in order to have the instances automatically started and stopped together with the operating system.
- **example** includes a simple example wizard (as a servlet), a test program for demonstrating the the XMLAPI as well as a simple portlet (for the Portal Manager). These examples are provided as source code so that they can be used as a guide in your own projects.
- instance contains an individual directory for each instance. Each instance directory contains further directories that have the following contents:
  - **bin**: Scripts for starting and stopping the components of the instance.
  - **config**: the configuration files that are read when the instance starts.
  - **data**: Data of the instance. Each component saves its data in an individual directory. With the Content Manager (`cm`) and the Template Engine (`te`), for example, this directory includes the file `nps.db`, i. e. the storage file of the SQLite database, if no other database has been integrated. The Content Manager and the Template Engine here also store additional data if the `storeBlobsInDatabase` parameter has been set to NO. The Search Engine Server stores data relevant for queries and collections, among other things, in this directory.
  - **export**: is the directory in which the files produced by the incremental export of the Template Engine are stored.
  - **log**: the log files of the instance
  - **script**: instance-specific scripts
  - **tmp**: temporary files used by the preview or the upload function, for example.
  - **webapps**: the web applications of the instance, i. e. the GUI, for example.
- **lib** contains libraries, binaries and Java archives required by the CMS.
- **share** contains files required by all instances. In particular, these are:
  - **TclHelp**: the help pages for the Tcl interface.

- **createInstance**: scripts with which a new instance can be created.
- **dbInstall**: scripts with which a new database for a CMS component can be created.
- **dbTools**: scripts for optimizing the database performance of Sybase and DB2 DBMS.
- **demoContentDump**: a dump of the demo content supplied with Fiona. You can use this dump to restore a demo content that has been modified. For details, please refer to section Restoring Data.
- **doc**: the CMS documentation in HTML and PDF format.
- **initDump**: a dump with which an instance is automatically initialized. When this dump is read in, the following elements are created:
  - the fields `conversionResults`, `lastConversion`, and `linklist`
  - the file formats `document`, `generic`, `image`, `publication`, `officDocument`, and *template*
  - the system jobs `systemTransferUpdates` and `systemPublish`
  - the files `Rootpub` and `mastertemplate`
  - a task for the user `root`
  - in the user management, the user group `admins` and the user `root`
- **newinstance.tgz**: an archive that serves as the basis of new instances.
- **script**: scripts such as general systemExecute procedures, formatters or custom menu commands, used by all instances.
- **welcome**: HTML pages that are displayed after the installation.

3.3 Removing the Demo Content

If you imported the demo content and now wish to remove it, please proceed as follows:

Change to the directory containing the instances:

```
cd instance
```

Deactivate the web applications:

```
default/bin/rc.npsd undeploy
```

Stop the CMS applications:

```
default/bin/rc.npsd stop
```

In the `default` instance open the file `bin/rc.npsd.conf` and write down the path of the Trifork Application Server as well as the Java directory.

Delete the `default` instance:

```
rm -rf default
```

Create the default instance again:

```
../share/createInstance/createInstance
```

Enter `default` as the name of the instance. When asked to specify the paths mentioned above, please enter them. If no paths were given, just press Enter.

Start the CMS and activate the web applications again:

```
default/bin/rc.npsd start
default/bin/rc.npsd deploy
```

3.4 Serving Static Data Using Trifork Application Server

Trifork Application Server supplied with Fiona can not only be used for serving the Java Server Pages of the HTML user interface but also for static web pages. This means that the files exported by the Template Engine can be displayed without using a webserver. Proceed as follows to configure the Trifork server correspondingly:

First, create a new web application directory named `Export`:

```
cd instance/myInstance/webapps
mkdir Export
cd Export
mkdir WEB-INF
mkdir META-INF
```

In the `WEB-INF` directory create the file `web.xml` and place the following content into it:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <display-name>NPSExport</display-name>
  <mime-mapping>
    <extension>html</extension>
    <mime-type>text/html;charset=UTF-8</mime-type>
  </mime-mapping>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Into the `META-INF` directory, place the file `trifork-app-conf.xml` containing the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE trifork-app-conf PUBLIC '-//Trifork Technologies//DTD
    Trifork Application Descriptor1.0//EN'
    'http://trifork.com/j2ee/dtds/trifork-app-conf_1_0.dtd'>
<trifork-app-conf>
  <role-mapping />
  <web-app>
    <display-name>NPSExport</display-name>
    <context-root>/NPS/Export</context-root>
  </web-app>
</trifork-app-conf>
```

Furthermore, place the file `Export.map` containing the following into the current directory `NPS~/instance/default/webapps/Export`:

```
<?xml version="1.0"?>
<jar name="NPSExport.war">
  <dir path="WEB-INF"
    from="/opt/Infopark/NPS/instance/default/webapps/Export/WEB-INF"/>
  <dir path="META-INF"
    from="/opt/Infopark/NPS/instance/default/webapps/Export/META-INF"/>
  <dir path=""
    from="/opt/Infopark/NPS/instance/default/export/online/docs" />
</jar>
```

If you did not install Fiona into the `/opt/Infopark/NPS` directory, please adapt the paths in the file above.

Then execute the following command in a shell to make the files in the Export directory available as a web application (the following two lines are one command line, not two):

```
TriforkInstallDir/domains/default/bin/trifork archive deploy
  -uadministrator -padminPassword -inplace default Export.map
```

The default password of the Trifork administrator is `trifork`. After the command has been executed successfully, you can access your exported content using the following URL:

```
http://mein.npsserver:8080/NPS/Export
```

Instead of `my.npsserver` provide your CMS host name. If the page `index.html` can not be found, please make sure that the Template Engine has been integrated properly and that the user runnig the Trifork server has read access to the export directory of the Template Engine.

3.5 Changing Passwords

On systems used productively it is required to change the passwords of the applications listed below. Afterwards these applications need to be restarted. In the examples below, `default` is used as the instance name. Replace this name by the name of the instance actually concerned.

3.5.1 Content Management Server

Change to the `NPS/instance/default/bin` directory and execute the Content Management Server in an operating system shell using the command `./CM -single` (UNIX) or `CM.bat -single` (Windows).

Now change the password of the `root` user by executing the command

```
user withLogin root set password newPassword
```

at the server's prompt, replacing *`newPassword`* with the password to be used from now on. Then log-off from the Content Management Server using `exit`.

Please make sure that only authorized persons log-in to the Content Management Server directly, i. e. using the command line argument `-single`.

3.5.2 Template Engine

For the Content Management Server to be able to send updated content to the Template Engine, a user name and a password need to be set in the Template Engine.

This log-in data can be modified by editing the `export.xml` file contained in the `NPS/instance/default/config` directory using a text editor. Change the log-in data contained in the section

```
<incrementalUpdate>
  <isActive>false</isActive>
  <login>root</login>
  <password>changeme</password>
</incrementalUpdate>
```

and save the modified file.

3.5.3 Content Management Server, Portal Manager and GUI

For the CMS applications to be able to communicate with each other, a `secret` phrase common to all three applications is required. Please set the value of the following configuration keys to this `secret`:

GUI

- Version 6.0.x: `instanceSecret` in the `NPS/instance/default/webapps/GUI/WEB-INF/config/basicConfig.properties` file.
- From version 6.5.0: `instanceSecret` in the `NPS/instance/default/webapps/GUI/WEB-INF/basicConfig.properties` file.
- `secret` in the `tokenManager` bean included in the `NPS/instance/default/webapps/GUI/WEB-INF/pm.xml` file.

Portal Manager

- `secret` in the `tokenManager` bean included in the `NPS/instance/default/webapps/PM/WEB-INF/pm.xml` file.

Content Management Server

- `instanceSecret` in the `NPS/instance/default/config/server.xml` file.

Afterwards, the web applications must be deployed. Use the script command `rc.npsd deploy` (up to version 6.0.x Unix only) or `deployWebapps.bat` (Windows, only up to version 6.0.x) for this. These commands can be found in the `bin` directory of the `default` instance.

3.5.4 Trifork Application Server

Trifork Application Server initially is protected by a default user name and password. This log-in data can be used to log-in to the administration console which can be accessed via the URL `http://localhost:8090/console`. The user name is `administrator` and the password is `trifork`.

The password can be changed in the *Security -> Realms -> default* section of the administration console. Alternatively, it can be changed via an operating system shell using the command:

```
trifork realm updateuser administrator "myPassword"
```

`myPassword` represents the new password. It is also required to specify the new password in the file `rc.npsd.conf` (up to version 6.0.x only Unix) or `setInstanceEnv.bat` (Windows, only up to version 6.0.x), which is located in the directory `NPS/instance/default/bin`, by modifying the value of the respective variables in this file.

3.6 Configuring the HTML-Editor

3.6.1 Adapting XHTML Validation

The HTML editor Edit-On-Pro (EOP) included in Infopark CMS Fiona only processes valid XHTML. When loading and saving and when switching from the HTML source view to the WYSIWYG view the editor checks whether the document conforms to XHTML. For this, it validates the document against an extended XHTML 1.0 Transitional DTD that can be found in the file `instance/`*`instanceName`*`/webapps/GUI/NPS/eop4/eopro/xhtml1-transitional-eop.dtd` located in the installation directory. Up to version 6.0.x the path is `instance/`*`instanceName`*`/webapps/GUI/eop4/`.

Custom extensions to the DTD, such as individual elements and attributes can be added to the file `instance/default/webapps/GUI/NPS/eop4/eopro/eopcustomelements.dtd`. If the document does not conform to the extended DTD, the EOP tries to automatically repair it using *Tidy*. In this process called clean-up all elements and attributes not conforming to the DTD are removed. The *clean-up* process can be configured via the `cleanupprocess` entry in the file `instance/default/webapps/GUI/NPS/eop4/config.xml`. An in-depth description of the EOP can be found in the EOP integration manual in the file `instance/default/webapps/GUI/NPS/eop4/doc/pdf/eop4integrationmanual.pdf`.

In the configuration file of the EOP custom tags such as `<npsobj>` can be defined. *Tidy* treats these tags like valid tags. You can define such tags by means of the following entry:

```
<customtags>
  <customtag name="npsobj" alwaysshow="true"></customtag>
</customtags>
```

Furthermore, in the `config.xml` file an individual configuration file can be specified for *Tidy* with which extended options can be configured. If a clean-up processor other than *Tidy* is to be used, you can specify here other classes for repairing documents.

You can switch off *Tidy* in the configuration file (`cleanupprocess`). However, if this is done, files not conforming to XHTML can no longer be loaded.

3.6.2 Integrating Dictionaries for Spelling-Checking

In the spelling-checking dialog of the HTML Editor, the list of the configured dictionaries, for example for different languages, is offered to the user. You can adapt this list to your own needs by editing the file `instance/default/webapps/GUI/NPS/eop4/config.xml`. For each dictionary a `spellingchecker` entry in the `spellingcheckers` section exists. This entry references a dictionary file in the directory `instance/default/webapps/GUI/NPS/eop4/eopro/`.

3.7 Updating the HTML Editor (EOP)

Infopark does not recommend to update RealObjects' edit-on Pro (EOP) HTML editor, which is part of an Infopark CMS Fiona installation, if there is no reason to do so. Furthermore, responsibility for the compatibility of EOP with Fiona is accepted only for those EOP versions delivered with our CMS.

If you still wish to update your EOP, please proceed as described below. Note that EOP 5.x is only compatible with CMS Fiona 6.x and not with earlier Fiona or NPS versions.

The following example describes how to update the EOP editor in CMS Fiona 6.6.0. However, the update procedure is similar for Fiona 6.x. Do not perform an update if you do not have a valid EOP license.

1. Download the latest EOP ZIP archive from RealObjects' website.
2. Extract the `eopro` directory of the ZIP archive to a temporary directory.
3. Make a backup of the original `eopro` directory located in `instance/`*`instanceName`*`/webapps/GUI/NPS/eop/eopro` of your Fiona installation directory.
4. Copy all files from the new temporary `eopro` directory into the existing `eopro` folder part of the GUI web application, overwriting existing files.
5. Make sure that the `eopro` directory contains a valid license file (`licensekey.xml`). If you encounter problems with the license key, please contact Infopark customer support.
6. The following files are required for Fiona and must remain in the `eopro` folder, otherwise the editor will not work properly. These files are not contained in the EOP archive you downloaded.

   - `eopro/eopcustomelements.dtd`
   - `eopro/licensekey.xml`
   - `eopro/NpsCleanup.class` (up to 6.7.2)
   - `eopro/infopark-eop_cleanup_callback-6.7.3.jar` (from 6.7.3)
   - `eopro/npstemplate.html`
   - `eopro/xhtml1-transitional-eop.dtd`
   - `eopro/xhtml-lat1.ent`
   - `eopro/xhtml-special.ent`
   - `eopro/xhtml-symbol.ent`
7. If you are running Fiona in a version prior to 6.7.3, you should also apply the following patch. Since Java 1.6., Update 20, stricter security regulations are effective. They prevent `*.class` files from being subsequently loaded. This also affects the EOP plugin supplied by Infopark (`NpsCleanup.class`). To meet the requirements, Infopark offers this plugin as a signed JAR archive for download.

   After having downloaded the archive, copy the jar file contained in it into the `eopro` directory. Then delete the `NpsCleanup.class` file from the same directory. For the EOP to subsequently load the Java archive, the file `editonpro.js` needs to be edited. This file is also located in the `eopro` directory. In this file, please locate the following lines:

```
…
// Loads the applet
this.loadEditor = function() {
// Determines whether to enable caching or not
…
```

Add the new jar archive to the file as shown below:

```
…
// Loads the applet
this.loadEditor = function() {
this.addArchive("infopark-eop_cleanup_callback-6.7.3.jar","1.0.0.0");
// Determines whether to enable caching or not
…
```

8. Restart the GUI. If you click the blue *info* button, the new version number should be displayed. If this is not the case, please clear the Java and/or the browser cache and try again. You can now start using the editor.

3.8 Updating Trifork T4

3.8.1 Updating Trifork T4 under Unix

We recommend to use the Trifork Server included in the Fiona version installed. If this version causes problems, please contact our customer support to obtain a download link for a different version.

After downloading, proceed as follows to update Trifork T4 Application Server under Linux.

1. Stop the existing Trifork T4 server

   Stop Trifork T4 using `rc.npsd stop trifork`.
2. Install the new version of Trifork T4

   Unpack the Trifork archive, for example `trifork-4.1.23-jdk.tar.gz`, into a new directory (give it a name similar to the archive name). Then change into this directory and execute `./install`.

   Now copy the file `server/license/license.txt` from the original Trifork directory to `server/license` in the new installation directory of Trifork T4.
3. Activate the server

   Adapt the configuration of all CMS Fiona instances to the new Trifork installation. For this, correct the installation directory of Trifork T4 in the file `instance/`*`instanceName`*`/bin/rc.npsd.conf`.

   Please ensure that the `JAVA_HOME` environment variable points to the Java version to be used by Trifork T4 (for example to `/usr/java/jdk1.6.0_05/`) and that the directory `$JAVA_HOME/bin` is found first via the `PATH` variable.

   Prior to version 6.7.0: For recent Trifork versions we recommend to use Java 1.5. If your applications have been running under Java 1.4 and you do not wish to change this, please also unpack the archive `server.lib.endorsed.zip` into the installation directory of the new Trifork T4 software to provide for the libraries that are missing from Java 1.4.

   From version 6.7.0: Java 1.6 is required.
4. Redeploying the web applications

   The Trifork T4 software should be up-to-date now. As a final step the server must be restarted and the web applications need to be deployed again. For this please execute the following two commands:

```
instance/instanceName/bin/rc.npsd start trifork
instance/instanceName/bin/rc.npsd deploy
```

5. Uninstalling the old Trifork T4 version

   Uninstall the old Trifork T4 version by deleting the directory into which it was installed.

3.8.2 Updating Trifork T4 under Windows

Please proceed as follows to update Trifork T4 Application Server under Windows.

1. Stop and uninstall the existing Trifork T4

   Please stop the existing Trifork service by terminating *Trifork Enterprise Application Server* via *Control Panel > Administration > Services*.

   For not loosing the license, please copy from the installation directory of Trifork T4 the file `server\license\license.txt` to your desktop.

   Uninstall the Trifork service.
   Up to version 6.0.4 of Infopark CMS Fiona:

   ```
   instance\default\bin\uninstallTriforkService.bat
   ```

   Version 6.5.0 and later:

   ```
   instance\default\bin\rc.npsd.bat uninstallService trifork
   ```

   Now remove the Trifork T4 software by means of Windows Control Panel (Software).
2. Installing the new version of Trifork T4

   Install the new version of Trifork T4 by executing the Trifork installation program, for example *trifork-4.1.23-jdk.exe*. Note that the path of the installation directory must not contain space characters.

   Then copy the `license.txt` file from the desktop to `server\license` in the installation directory of Trifork T4.
3. Activate the server

   Adapt the configuration of all CMS Fiona instances to the new Trifork installation. For this, correct the installation directory of Trifork T4 in the file `instance\instanceName\bin\setInstanceEnv.bat` (up to Infopark CMS Fiona 6.0.4) or `instance\instanceName\config\rc.npsd.conf`, respectively (from version 6.5.0).

   If you use Infopark CMS Fiona 6.5.0 the file `domains\default\bin\service\default\service.ini` in the Trifork installation directory needs to be adapted. Insert the following into the `[Java-Options]` section:

   ```
   ; Infopark options
   -Xmx256m
   -Xms256m
   -XX:MaxPermSize=128m
   ```

Install a new trifork service.
Up to Infopark CMS Fiona 6.0.4:

```
instance\default\bin\installTriforkService.bat
```

From version 6.5.0:

```
instance\default\bin\rc.npsd.bat installService trifork
```

Then start this service.

4. Deploying the web applications again

   Finally, the web applications of CMS Fiona need to be deployed into the new Trifork server:
   Up to Infopark CMS Fiona 6.0.4:

   ```
   instance\instanceName\bin\deployWebApps.bat
   ```

   From version 6.5.0:

   ```
   instance\instanceName\bin\rc.npsd.bat deploy
   ```

   The new version of the Trifork server is now ready for operation.

3.9 Enabling HTTPS in Trifork T4

Trifork Application Server supports HTTPS if it has been activated in the Management Console. You can access the console via the URL `http://meinTriforkServer:8090/console`. The default user name is `administrator` and the password is `trifork`. After logging in, enable the HTTPS option in the HTTP, DEFAULT_ENDPOINT section and specify the desired port.

After this change, the Trifork server needs to be restarted. Your server can then be reached via HTTPS.

As a default, the Trifork server uses a supplied key pair and a corresponding SSL certificate. However, you can also generate your own key pairs and certificates. For this, you require Sun Microsystems' `keytool` program which is supplied with the Trifork server. This program can be found below the installation directory, in *javaDir*`/bin` where *javaDir* is the directory of the JDK used, for example in `trifork-4.1.26/jdk-1.5.0/bin`.

3.9.1 Generating Key Pairs

Key pairs are stored in a so-called keystore. Since several servers can be operated using one Trifork server, the keystore to be used can be selected by means of the Management Console. See the HTTP section mentioned above.

When creating a key pair, an alias needs to be specified for it. The alias is an identifier that can be used to refer to the key pair later on. Furthermore, you can specify a so-called keystore. If the keystore is not specified, the key pair is stored in the default keystore. If a nonexistent keystore is supplied, it will be created automatically. Details about this and the explanations given in the following can be found in Sun's JDK documentation:

http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Security6.html
http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html

Use the following syntax to create a key pair with the alias *alias-name* and to store it in the specified keystore. For accessing the key pair as well as the keystore, passwords can be specified.

```
keytool -genkey -alias alias-name -keyalg RSA -keypass changeit -storepass changeit -
keystore keystore.jks
```

By default, the supplied keystore, `keystore.jks`, is protected with the password `changeit`. Enter the password in the Management Console in the section mentioned above (HTTP, DEFAULT_ENDPOINT) so that the Trifork server can access the keystore.

3.9.2 Certifying a Key Pair in the Keystore

A key pair for the secure communication between a server and the clients is considered trustworthy if it has been certified. It has become the task of certificate authorities such as VeriSign, Thawte etc. to do this. If a browser comes across a certificate that has been certified by such an authority, it is automatically considered trustworthy provided that the browser has been configured correspondingly.

In many environments it is sufficient to use a self-signed certificate. In this case, the website owner acts as the issuer of the certificate. Therefore, when his website is accessed for the first time, the browser will ask the user whether he trusts the certificate. If he does, the browser stores the certificate in the pool for trustworthy certificates. Thus, the next time the user visits this website, the browser does not need not ask again.

For this to work, a certificate for the key pair concerned needs to be exported from the keystore. This certificate is then imported into the so-called truststore. Here is an example for the two steps applied to the `alias-name` key pair and the exported certificate named `name.cer`. As the keystore and the truststore, `keystore.jks` and `cacerts.jks`, respectively, are used.

```
keytool -export -alias alias-name -storepass changeit -file name.cer -keystore
 keystore.jks
keytool -import -v -trustcacerts -alias alias-name -file name.cer -keystore cacerts.jks \
  -keypass changeit -storepass changeit
```

In total, three files, `keystore.jks`, `cacerts.jks`, and *name*`.cer`, were created in the `bin` directory of the Java JDK. To make these files known to the Trifork server, please move them to the *installDir*`/domains/default/servers/default/config` directory. Before doing this, make a backup of the existing files `keystore.jks` and `cacerts.jks`. Using the Management Console, section Security, SSL, you can check whether the Trifork server recognizes the new certificate.

For converting existing certificates (`*.pem`; `*.der`; `*.p12`) to the Java keystore format, free tools available on the internet can be used. Individual certificate containers can be converted to other formats by means of OpenSSL (http://www.openssl.org/support/faq.html).

3.10 Displaying Dynamically Generated Content in the Preview

For static content the GUI itself generates preview pages. It lets the Content Manager compute the pages and then delivers them directly, not by means of an additional web server. Dynamic content, on the other hand, can only be generated by means of the appropriate applications for a webserver. The GUI places pages with dynamic content into the file system. Then it sends a request to the preview

server that generates the content. The content types to be handled by the webserver, their location in the file system, and the request URL are defined in the system configuration (see gui).

The preview server needs to be accessible only from the GUI because the GUI functions as a proxy for dynamic content. However, if security requirements are low, you can use the same webserver for previewing dynamic content and for the live server. In both cases the webserver needs to be configured correspondingly, i. e., for example, a location must be defined that points to the directory which has been set in the Content Manager.

In the following sections it is explained how to proceed for generating typical kinds of dynamic content. If required, these types can be combined. However, it might be necessary to use several server components.

Please note that 3rd party software (i.e. the user account under which it is running) must be given read access to the CMS files and directories concerned.

3.10.1 Java Server Pages

Java Server Pages (http://java.sun.com/products/jsp/) are text files containing a mixture of HTML text and Java code. In order to be able to open JSPs in the preview or on the live system, a servlet container like the Trifork or the Tomcat server is required. They compile the pages and execute them at request of the browser. Every so-called (Java) application server includes a servlet container.

To enable JSP support it is also necessary to integrate a JDK (Java Development Kit) into your system; the Trifork installation already includes a JDK. It is also possible to use the JDK offered by Sun on their website (http://java.sun.com/javase/downloads/). Please observe the respective installation requirements.

To make JSPs available in the live system, your servlet container must be configured in such a way that it responds to requests for JSPs (URL pattern: `*.jsp`) and delivers the files from the file tree into which Fiona exports them. When the Template Engine is used, this is the CMS installation directory, extended by `export/online/docs`.

For the preview the supplied Trifork-Server can be used. For this, an additional web application needs to be configured. This web application can be addressed either via an individual port or via a prefix on the same port as the GUI.

It is possible to use the JDK 1.4 provided by Sun on their website. Please take account of the respective installation requirements.

3.10.2 PHP Scripts

HTML pages can contain Hypertext Processor scripts (PHP , `http://www.php.net/`). In the pages, the scripts are included in `<?php ... ?>`. It is recommended to use `php` as content type, if only a small part of the website is dynamic. If all or most of the pages have dynamic content, `html` can be used as content type as an alternative.

The webserver commonly used to dynamically generate content using PHP is the Apache webserver plus a special module (`mod_php`, which is part of the PHP distribution). The module will be loaded if it is added to the `httpd.conf` configuration file, and the dynamic content is passed to the module by assigning the `application/x-httpd-php` MIME type to it.

For the preview, a separate Apache webserver must be configured in accordance with the live server. The preview server is required to deliver a directory accessible by the GUI (declare it in the `httpd.conf` file as `<Directory>` and direct it to an URL by means of an alias).

3.10.3 Active Server Pages

On Windows Systems it is possible to use Internet Information Services (IIS) instead of the Apache webserver for the preview and the live server. If Active Server Pages (http://msdn.microsoft.com/asp.net/) are to be used, IIS must be used both as live server and as server for the dynamic preview. To make use of IIS, please proceed as follows after you have installed NPS:

Open the *Internet Services Manager* in the *Administrative Tools* menu of the Control Panel and add one or two webservers, one for the live server and one for the dynamic preview, if required. Then change the properties of these webservers:

For the live server specify `export\online\docs`, located below the CMS installation directory, as *local path* on the *Home Directory* tab. On the *Documents* tab add to the list of standard documents the name `index` to which the file name extension is added (for example `index.html`).

For the preview server specify on the *Home Directory* tab as *local path* a directory path that can be accessed by the GUI (if required as a network resource). This directory path (as it is seen by the GUI) also needs to be set as the value of the `dynamicPreviewDirectory` system configuration entry. On the *Web site* tab enter as TCP port the number that follows the host name in the value of `dynamicPreviewUrl`.

Finally, restart the *World Wide Web Publishing Service* via the Services list of the Control Panel.

3.11 Respecting CMS Read Permissions in the Preview

In its initial state, Infopark CMS Fiona determines the access permissions for preview pages from the live read permissions.

To control whether editors have access to the preview page of a CMS file, it might be more appropriate to use the editoral read permission of the file instead of the live permissions.

This can be achieved by configuring the Portal Manager's authorizer accordingly. To do this, activate the preview authorizer of the CM instead of the default authorizer in the `pm.xml` file of the GUI web application.

```
…
<bean id="authorizationManager"
    class="com.infopark.pm.user.DefaultAuthorizationManager">
  <property name="authorizers">
    <list>
      <!--<bean class="com.infopark.pm.user.DefaultAuthorizer"/>-->
      <bean class="com.infopark.cm.htmlgui.browse.preview.Authorizer" />
    </list>
  </property>
</bean>
…
```

By default, if no live permissions have been set, no authorization is performed for preview pages.

To make the permission filter call the authorization manager even if no live permissions have been set, the `authorizeAnyway` property needs to be set in the `pm-filter.xml` file of the GUI web application:

```
…
<bean id="pmPermissionFilter" class="com.infopark.pm.PermissionFilter">
  <property name="authorizeAnyway" value="true" />
</bean>
…
```

You can also use the default authorizer in addition to the preview authorizer of the CM. However, when doing this, access is always granted if no live permissions have been assigned.

Background Information

As a default, preview pages for CMS files are subject to the same read permissions as the corresponding live pages delivered by the Portal Manager. This is due to the fact that preview pages are delivered by the Portal Manager as well, thus causing authorization to be based on live permissions only.

To check whether a visitor is permitted to access a specific page, the Portal Manager's permission filter examines the live read permissions of the corresponding CMS file. If no user groups have been assigned to these permissions, the permission filter grants access to this file, unless the `authorizeAnyway` property has been set to `true`. If user groups have been assigned, or if this property is `true`, the permission filter passes control to the authorization manager.

To finally find an answer to the question whether the visitor may access the page, the authorization manager calls the authorizers specified in the configuration. Two authorizers exist, the Portal Manager's default authorizer and the authorizer of the Content Manager.

Authorizer	Effect
`DefaultAuthorizer` (PM)	Grants access to a page if the logged-in visitor is a member of one of the user groups that have been assigned to the live read permissions of the corresponding CMS file. Access is also granted if no live read permission groups have been specified.
`Authorizer` (CM)	Grants access to a page if the logged-in visitor has read access to the corresponding CMS file. This is the case if the visitor is a superuser, or if he was granted administration or read permission for the CMS file.

As described above, you can use any of the two authorizers, either individually or combined, to control access to the preview and live pages.

3.12 Configuring Internet Access for the Link Checker

From version 6.5.0 Infopark CMS Fiona includes a link checker with which the availability of external `http` and `https` links can be tested. For this to work, the Content Management Server requires access to the internet.

The link checker does not verify SSL certificates.

3.12.1 Link Checker Configuration

URLs matching particular patterns can be excluded from link checking. Excluded URLs are alway treated as if they were available, meaning that they will never show up in the Content Navigator's list of unreachable link destinations.

3.12.2 Firewall Configuration

If the Content Management Server (CM) is operated behind a firewall (i. e. in a DMZ), please ensure that it can establish internet connections. For this, the following ports need to be enabled in the firewall:

- `http`
- `https`
- `dns` for resolving external DNS names

3.12.3 Proxy Configuration of the CMS Systems

For checking external link destinations the Content Management Server can make use of a proxy. For this, set the `http_proxy` or `all_proxy` for the CMS user to your proxy server. See the libcurl documentation.

Example for Linux with bash:

```
export http_proxy=http://ourproxy.company.dmz:3128/
```

If the proxy requires authentication, enter the following instead:

```
export http_proxy=http://username:password@proxyintern.firma.dmz:3128/
```

3.13 Configuring Access to the Online Help

From version 6.7.0

The editorial system offers the documentation as PDF files to the users if Infopark's knowledge base cannot be accessed. It is no longer required to configure the origin of the online documentation.

From version 6.7.1, the PDF files are contained in a separate package that can be downloaded in the download section. Unpack the files into the directory `instance/`*`instanceName`*`/webapps/GUI/NPS/doc` to offer them to the users if they cannot access the Infopark Knowledge Base.

From version 6.0.3 Patch Pack 7 to version 6.6.1

Static help pages are provided and delivered (offline help). However, access to the online help from within the Content Navigator is possible and can be enabled by means of a configuration parameter (see below). If the client machines have access to the Infopark Portal, we recommend using the online help since it is continually updated and can be searched easily.

The online help can be activated in these versions by editing the `web.xml` file located in the GUI web application (`instance/`*instanceName*/webapps/GUI/WEB-INF/web.xml). *instanceName* stands for the name of the instance concerned.

The configuration parameters can be found among the initialization parameters of the `HelpFilter` filter. The following parameter can be configured:

Parameter	Description	Example

`useOnlineHelp`	`true`: activates the online access to the knowledge base server.
	
	`false` (default): displays the static help pages delivered within the package.
		`<init-param>`
		`  <param-name>useOnlineHelp</param-name>`
		`  <param-value>true</param-value>`
		`</init-param>`

From version 6.0.3 to 6.0.3, Patch Pack 6

The help function of the Content Navigator fetches the help pages directly from the knowledge base server (online help).

3.13.1 Making the PDF documentation available

The PDF manuals for Infopark's products can be made available to the users of the Content Navigators by letting the *Manuals* (or *Documentation*) menu item in the *Help* menu point to an overview page on which the manuals are linked. To do this, proceed as follows:

1. Copy the manuals

   Place the manuals into a directory below the GUI web application, for example `doc`. Depending on the version of Infopark CMS Fiona, the PDF files can be found in different locations in the directory hierarchy of the CMS installation or must be downloaded separately. If you have more than one instance, all of them can link to this copy of the documentation.
2. Create `documentation.html`

   Please create a file named `documentation.html` and place it in the `webapps/GUI/` directory of the instance concerned. In this file, link to the PDF documents you would like to make available to the users. The file might have the following contents, for example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Infopark CMS Manuals</title>
  </head>
  <body>
    <div id="title"><h1>Infopark CMS Fiona Documentation</h1></div>
    <br /><br />
    <table cellpadding=5 width=600 border=0>
    <tr>
      <th align="left" bgcolor="#DDDDDD" colspan=2>Der Content Navigator</th>
    </tr>
    <tr>
     <td bgcolor="#EEEEEE"><FONT SIZE="-1"
```

```
     FACE="Helvetica,Arial,sans-serif">Describes the usage of the HTML user interface
 for editors</font></td>
     <td bgcolor="#EEEEEE"><a href="doc/Navigator-en.pdf">PDF</a></td>
    </tr>
    <tr><td colspan=2> </td></tr>

    <!-- More PDF files -->

    </table>
  </body>
</html>
```

3. Adapt `itemRegistry.xml`

   Place the previous *Manuals* entry (`bean id="online_help_index"`) inside a comment and replace it with the following lines (please adapt the URL according to your installation):

```
<bean id="online_help_index"
 class="com.infopark.cm.htmlgui.browse.menuaction.Redirect">
<property name="titles"><map>
<entry key="de"><value>Handb374cher</value></entry>
<entry key="en"><value>Manuals</value></entry>
<entry key="fr"><value>Manuels</value></entry>
<entry key="it"><value>Libretti di istruzioni</value></entry>
<entry key="es"><value>Libros de instrucciones</value></entry>
</map></property>
<property name="uri"><value>http://<server>:8080/NPS/documentation.html</value></
property>
<property name="isExternal"><value>true</value></property>
</bean>
```

4. Redeploy the GUI

   Redeploy the GUI for the changes to become effektive:

```
bin> ./rc.npsd deploy GUI
```

   Please note that deploying causes the GUI connections of the users to break. Therefore, the users will not be able to save the changes they have made to the content. For this reason, nobody should work with the GUI during the deployment procedure.

3.14 Configuring Reminder E-Mail Notification

Automatic e-mail notification for due reminders is set up in the following steps:

3.14.1 Specifying the Mail Server

To enable the system to send e-mail notifications it is required to specify the IP address of the mail server in the `server.email.host` system configuration entry.

3.14.2 Setting an Execution Schedule for Sending E-Mail

The notification e-mail messages are sent by means of a dedicated system job, `systemSendReminderNotifications`. The job's task is to check each file in the CMS for due reminders. For each due reminder the job sends an e-mail message to the recipients specied in the reminder.

You can specify the intervals at which the e-mail messages are sent in the execution schedule of the job. Depending on your requirements you might wish to have them sent on a daily or weekly basis, for example.

3.14.3 Specifying the E-Mail Addresses of Users

The reminder notification messages are sent to those users in the reminder's distribution list whose e-mail addresses have been specified in the user management.

3.14.4 Changing the E-Mail Message

You can replace the standard notification message text with your individual text. This is accomplished by redefining the *sendReminderNotification* procedure in a tcl file of your choice. Place this file in the script directory `cm/serverCmds` of the instance concerned. You might use the standard procedure as a template. It has the same name and is defined in the `reminderNotification.tcl` file which is located in the standard script directory.

3.15 Configuring Web Applications

By default, every instance of Infopark CMS Fiona includes three web applications:

- **GUI**: The HTML user interface
- **PM**: The Portal Manager
- **PM-PL**: The portlet web application for the Playland demo content

The web applications are located in the `webapps` directory below the instance directory, for example in `instance/default/webapps`.

This section is about configuring the web applications. Information already included in other parts of the documentation is not included here.

3.15.1 Defining MIME Types

In Infopark CMS Fiona, Version 6.6.1 or later, the MIME types of a web application are defined in a different location and in a different format than in previous versions:

Up to Version 6.6.0

File path: *instanceName*`/webapps/`*WEBAPP*`/WEB-INF/mime-types.properties`

Format:

```
image/png: png
image/wmf: wmf
image/x-icon: ico
text/css: css
text/html: html htm php shtml asp jsp
```

From Version 6.6.1

File path: *instanceName*`/webapps/`*WEBAPP*`/META-INF/mime.types`

Format:

```
image/wmf               wmf
image/x-icon            ico
text/css                css
text/html               html htm php shtml asp jsp
```

The file must not contain empty lines or comment lines. Whitespace between the first column and the file name extension must consist of tabs only.

If the changes described above have not been made, stylesheets are not evaluated, especially in Firefox. If you request the stylesheet directly by entering its URL in the browser's address field, its content is not displayed as text in the browser (it would, if the MIME type definition were correct). Instead, the file is offered for download, and the MIME type displayed in the download window is `application/octet-stream`.

If you use Tomcat instead of Trifork Application Server please define the MIME types in a file named `.mime.types` in the home directory of the Tomcat user.

After completing these changes, the web applications need to be deployed again.

3.15.2 Configuring Error Pages

The error pages of a web application can be defined in the corresponding file *`WEBAPP`*`/WEB-INF/web.xml` using the `error-page` entries. Here is an example:

```
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/global/errors/500.html</location>
</error-page>

<error-page>
  <error-code>401</error-code>
  <location>/global/errors/401.html</location>
</error-page>

<error-page>
  <error-code>403</error-code>
  <location>/global/errors/403.html</location>
</error-page>

<error-page>
  <error-code>404</error-code>
  <location>/global/errors/404.html</location>
</error-page>
```

The system searches for the error pages directly below the web application directory. For example, the page for a 401 error defined according to the example above is *`WEBAPP`*`/global/errors/401.html`. Additionally, the Portal Manager will, if the error page does not exist in the specified location, also search for the page below the `documentSource` which is defined in the `/webapps/PM/WEB-INF/pm.xml` file. In the case mentioned, the path of the error page would be *`cmsDir`*`/instance/`*`instanceName`*`/export/online/docs/global/errors/401.html`.

If virtual hosts have been defined in the PM (`VirtualHostConfig`), the error pages defined in the `web.xml` file must be located in `PM/`*`YourVirtualHost`*`/global/errors` instead of `PM/global/errors/`.

If your error page includes other elements, for example stylesheets or images, you need to set the base directory because the links to these additional elements are relative. The base directory can be set by means of the following Javascript code. Include this code in the layout that generates the `HEAD` section of your error pages.

```
<script type="text/javascript">
  <!--
    document.write("<base href='" + location.protocol + "//" +
    location.host + "${document.url}'/>");
  -->
</script>
```

This code ensures that the parts included into an error page are taken from the correct directory, regardless of where the error page is actually located.

3.15.3 Defining the Base URL

The path under which a web application can be reached can be specified using the `context-root` parameter in the *`WEBAPP`*`/META-INF/trifork-app-conf.xml` file. Here is an example:

```
<trifork-app-conf>
    <role-mapping />
    <web-app>
        <display-name>New Webapp</display-name>
        <context-root>/new/path</context-root>
    </web-app>
</trifork-app-conf>
```

After the web application has been deployed again, it can be reached using the URL `http://server:8080/new/path`.

3.16 Integrating a Database

Infopark CMS Fiona includes an SQLite database which the system uses after the installation.

The Content Manager and the Template Engine automatically create the required tables in the SQLite database if they do not yet exist. There is no need to manually set-up or configure the database unless changes have been made to the standard configuration.

Please note that SQLite has not been approved for productive use.

3.16.1 Procedure

If you wish to integrate a database different from the default one or from the one currently used, please proceed as described below.

These instructions also apply if you disabled database storage using `storeBlobsInDatabase="false"` and now wish to activate it again.

- Check whether the installation requirements are met, especially those concerning the database you use.
- Please install your database if you have not already done so. The appropriate database client (mysql, isql, sqlplus) needs to be installed on the machine running the CMS component (Content Manager, Template Engine) that requires the database.

- Stop the CMS services. Under Linux use

```
instance/instanceName/bin/rc.npsd stop
```

  for this. Replace *instanceName* with the name of the instance concerned (the default name is `default`). Under Windows, please use the corresponding commands in the *Services* section of the Windows control panel.
- If you would like to create a new database for the Content Manager, first use

```
instance/instanceName/bin/CM -dump dumpDir
```

  to backup the content and the configuration. See also Dumping and Restoring Data.
- Execute

```
share/dbInstall/install-db
```

  and answer the questions of the script. For this, it is required to know the database, i. e. its name, the user name, the password, and the name of the database server. The script generates a configuration file for the database of the CM (`cmdb.xml`) or the TE (`tedb.xml`), depending on what you asked it to do. (In versions prior to 6.6.1, the script created one of the files `mysql.xml`, `sybase.xml`, `oracle.xml` or `mssql.xml` which needed to be renamed.)
  The database user under whose login the CMS accesses the database needs to be given the permission to drop constraints. This is required for importing the initial data as well as for restoring data later on.
- Copy the database configuration file to `instance/`*`instanceName`*`/config` (*instanceName* stands for the instance concerned).
- Make sure that the name of this configuration file is specified in the `instance/`*`instanceName`*`/config/server.xml` file under

```
<server>
  <cm>
    ...
    <database fileName="cmdb.xml"/>
  </cm>
  <te>
    ...
    <database fileName="tedb.xml"/>
  </te>
  ...
</server>
```

  Even if the same database product is used, different configuration files are required for the Content Manager and the Template Engine.
- If a database for the Content Manager has been created and the data has been backuped prior to this, restore them now:

```
instance/instanceName/bin/CM -restore dumpDir
```

See also Dumping and Restoring Data.

- Restart the CMS services using

```
instance/instanceName/bin/rc.npsd start
```

on Unix systems or the services menu in the Windows control panel, respectively. The Content Management Server and the Template Engine will create the database tables required when started for the first time.

3.16.2 General Notes about Database Usage

Access permissions

The installation script accesses the database with administrator permissions to create the table spaces.

No compressed database tables

CMS Fiona does not support compressed database tables.

Providing memory

While the CMS is running, the database allocates as much memory to the CMS tables as the database or the default configuration specifies. This means that the amount of memory assigned each time more memory is required is not defined by the CMS but by the database.

Monitoring database operation

It is not the task of the CMS but of the database administrator to optimize the database with respect to memory assignment and performance as well as data and access security to the requirements of daily operation. Especially with larger productive systems, it is necessary to continuously monitor the database.

Content storage location

If a database other than the default SQLite database is to be used and blobs (content and part of the configuration) are to be stored in the database, this needs to be specified before the CMS application concerned is executed for the first time (see server). Changing this configuration option afterwards will lead to loss of data.

Special notes on the individual databases can be found in the following sections.

3.16.3 Oracle

Oracle 9 databases must be operated using the client for Oracle 10. Please observe the following:

- The `LD_LIBRARY_PATH` of the database user needs to point to the `oracle10/lib` directory. If this environment variable contains the library paths of more than one Oracle client versions, `oracle10/lib` must be found first.
- Please check whether the database configuration file pointed to from within `instance/instance_name/config/server.xml` (mostly

`<database fileName="oracle.xml" />`) has the proper format. When using the Oracle 10 client, it should look as follows (and contain the correct login data):

```
<?xml version="1.0"?>
<configuration>
    <adaptor>oracle</adaptor>
    <version>10</version>
    <server>ORACLE</server>
    <user>user</user>
    <password>pass</password>
    <storeBlobsInDatabase>...</storeBlobsInDatabase>
</configuration>
```

The Oracle database client must have been installed on the system on which the CMS is to be or has been installed. This includes the installation of `sqlplus` on the CMS system and of the Oracle Listener on the database system. The CMS system user must be able to run `sqlplus`. This needs to be checked prior to installing the CMS or switching to an Oracle database by executing the following command:

```
sqlplus system/password@ServerId
```

Enter as *`password`* the password of the database user `system` and as *`ServerId`* the Oracle SID.

Please make sure that the required services such as the Listener or the local database are started automatically after a server restart.

Furthermore, the database server needs to be configured to use the UTF-8 character set. Prior to installing the CMS, the value of the database parameter `NLS_LENGTH_SEMANTICS` must have been set to `CHAR` (instead of `BYTE`) so that sufficient space is allocated for strings.

Also, for the CMS to work properly and to cooperate with the database, the environment variable `ORACLE_HOME` must be set correctly. The environment variable `NLS_LANG` needs to be set to *`Language_Country`*`.UTF8`. In this string, *`Language`* and *`Country`* are placeholders for valid combinations of identifiers specifying the language and the country, for example `GERMAN_GERMANY.UTF8`. For the valid combinations, please refer to the database documentation. From Fiona 6.5.0, you can also use the `AL32UTF8` character set, if `UTF8` must not be used in your Oracle environment.

The Tcl scripts with which the databases are installed are located in `share/dbInstall`. Please note that no optimization parameters are used when the databases are installed. If required, they can be added manually to the respective Tcl script. For example, the operation for creating an Oracle database can be extended in `share/dbInstall/oracle.tcl` to control memory allocation:

```
DEFAULT STORAGE (INITIAL 10M NEXT 5M MINEXTENTS 1 MAXEXTENTS UNLIMITED PCTINCREASE 0)
```

For further details please refer to your database documentation.

Good database performance requires that the indexes of all tables are updated regularly.

```
ANALYZE TABLE table_name COMPUTE STATISTICS FOR ALL
INDEXED COLUMNS FOR ALL INDEXES;
```

3.16.4 Sybase

Sybase is supported up to Fiona 6.7.2. From then on, Sybase support is discontinued.

If an already existing device is specified during the installation of Infopark CMS Fiona and a different size is to be assigned to this device, then the new size is ignored. Therefore, if you would like to change the size of the device, it must be deleted and the Sybase Server restarted prior to installing the CMS. Please refer to your database documentation for details.

The Content Management Server and the Template Engine should not be operated using the same Sybase database server since this might cause problems concerning the transmission of update records.

When configuring the Sybase database, we recommend to also install the character set CP 850 and to define it as the default. As "Sort Order" we recommend to use "Binary ordering, for use with code page 850".

To preserve the database performance, it is absolutely necessary to execute the `update statistics` database command for all tables at regular intervals. For this purpose, the standard installation includes in the `share/dbTools/` directory the `sybaseUpdateStatisticsCM.sql` and `sybaseUpdateStatisticsTE.sql` scripts for the Content Manager and the Template Engine, respectively. The scripts can be executed using the following command:

```
isql -Uuser -Ppassword -Sserver -iscript
```

The parameters have the following meaning:

- *`user`*: Database user of the CM, TE, or PM, respectively
- *`password`*: The respective database user's password
- *`server`*: Name of the database server
- *`script`*: The file to run

Further information on the `update statistics` SQL command can be found in the Sybase database documentation.

3.16.5 MySQL

MySQL is supported from CMS Fiona 6.6.

For the supported database version, please refer to the system requirements. We recommend using the commercial version of MySQL including support, in case you require competent assistence. Such a version can be purchased from Infopark.

If you plan to store large blobs (100 MB or more) in the 32-bit version of the database, we recommend testing the system with large blobs prior to using it productively. Due to the MySQL configuration it might be necessary to use the 64-bit version of the database server.

When creating the database tables, CMS Fiona uses the *InnoDB* storage engine by default. Using this engine is obligatory.

To be able to use CMS Fiona with a MySQL database, MySQL must have been installed on the machine running the CMS. To check whether the CMS can connect to the database, run the following commands as the CMS user and provide the database administrator's login data:

```
mysqladmin -u MySQL-Administrator -p -h MySQL-Server ping
mysql -u MySQL-Administrator -p -h MySQL-Server -e ""
```

Please note that the standard settings of the MySQL server (`my.cnf`) need to be changed for running CMS Fiona:

- The MySQL server needs to be operated with the `READ-COMMITTED` global transaction isolation level. Please specify this value for the `transaction-isolation` option in the `[mysqld]` section.

```
transaction_isolation = read-committed
```

- Furthermore, the database server needs to be configured to use the `UTF-8` character set.

```
character-set-server = utf8
default-character-set = utf8
```

- The global SQL mode of the MySQL server must be set to `TRADITIONAL`. Please specify this value for the `sql_mode` option in the `[mysqld]` section.

```
sql_mode = traditional
```

Please note that the standard settings of the MySQL server also need to be changed for being able to handle large amounts of data:

- In the `innodb_data_file_path` option delete the database file size restriction (`max:128M`) or set it to the required value.
- To ensure best performance, the option `innodb_buffer_pool_size` should be increased. Depending on the database size and the amount of available RAM, it should be set to at least `512M`.
- For single large transactions (large binaries), adapt the `innodb_log_file_size` (to approximately 25% of `innodb_buffer_pool_size`) and `innodb_log_buffer_size` options.
- Likewise, we recommend setting the `max_allowed_packet` option to `1G`.
- To prevent the database server from closing its connections to the CMS, increase the values of `net_write_timeout` to `1800` and `wait_timeout` to at least `3600`. As an alternative to changing `wait_timeout`, you can set the `slaveIdleTimeout` parameter of the CMS to a value less than `wait_timeout`. This can be done in the `tuning.xml` file.
- We recommend activating `innodb_file_per_table` and to deactivate `innodb_data_file_path` in the MySQL configuration file `my.cnf`. This change has the effect that an individual file is used for each database table, making data access faster. See below for a description of how to proceed.

If your MySQL server is not running on the standard port 3306, the port needs to be specified in the database configuration of the instance concerned (`mysql.xml` in the `config` directory). The port is queried by the `install-db` script used for database creation.

if a MySQL database is created using `install-db`, it will be configured to permit access only from the machine on which the script was executed. To permit access from a different machine, the access permissions must be changed on the MySQL server accordingly.

Activating `innodb_file_per_table` for Databases

After activating `innodb_file_per_table` and deactivating `innodb_data_file_path` in the MySQL configuration file, `my.cnf`, the databases concerned need to be deleted and created again for the change to become effective. To do this, please proceed as follows:

1. Completely stop the CMS.
2. Save the contents of the CM database concerned using `CM -dump`.
3. Change the MySQL parameters in the file `my.cnf` and restart the server.
4. Delete the databases of the CM and the TE using the `mysql` command line tool.
5. Create the database of the CM and the TE according to your database configuration `cmdb.xml` (and `tedb.xml`, respectively).
6. Restore the dumped data using `CM -restore`.
7. Start your CMS again.

Good database performance requires that the indexes of all tables are updated regularly.

```
analyze table tablename1, tablename2, ...
flush tables;
```

3.16.6 MS SQL Server

MS SQL Server up to CMS Fiona 6.0.x

If you use the Content Manager under Windows with an MS SQL 7 database, please note the following: For non-ASCII characters (like German Umlauts) to be stored correctly in the database, you need to install the database with the *Custom Installation* option and select *CP 850 (Multilingual)* as *Character Set* in the *Character Set / Sort Order* dialog. Furthermore, the option *Unicode Collating* must be set to *General Unicode*.

For MS SQL 2000, please carry out a custom installation and select one of the Codepage 850 options (*case insensitive* or *binary order*) in the *Collation Settings* dialog.

Good database performance requires that the database statistics are updated regularly.

MS SQL Server from CMS Fiona 6.5.0

From version 6.5.0, the Fiona database adapter uses the ODBC interface. It is therefore required that an individual ODBC data source exists for each Fiona server application (CM and TE) when configuring the database in Fiona.

To be able to store larger blobs, the "SQL Server Native Client" needs to be used instead of the standard "SQL Server" database driver. The "SQL Server Native Client" can be downloaded from the following Microsoft website:

http://www.microsoft.com/downloads/details.aspx?FamilyId=C6C3E9EF-BA29-4A43-8D69-A2BED18FE73C&displaylang=en#SNAC

Install the driver, then proceed as follows to add the data sources on the CMS machine (i.e. on the MS SQL client machine):

Open the *Administration* settings from the Control Panel, select *Data Sources (ODBC)* -> *System DSN* -> *Add*. As the driver select "SQL Server Native Client", then choose the MS SQL Server on which the CMS database is located as the SQL Server to connect to. Additionally, the standard database of this ODBC connection must be set to the corresponding CMS database.

Good database performance requires that the database statistics are updated regularly.

3.16.7 DB2

DB2 is supported under Solaris or Linux up to Fiona 6.6.0. From then on, DB2 support is discontinued.

A DB2 client or server must already be present on the system on which Infopark CMS Fiona is to be installed. Only clients are supported that communicate via TCP/IP with the database.

If the CMS is not to be installed on the DB2 server machine, the installation must be performed as the DB2 client user. Prior to this, the DB2 server must have been made known as a node to the client (see the DB2 documentation for details). If the CMS is to be installed on the DB2 server itself, then it must be installed as a DB2 instance owner. The database installation script (`install-db`) queries the corresponding data. If the CMS is not being installed on the DB2 server, specify the node name when asked for the *Node/Instance*. If the CMS is installed on the machine that also serves as the DB2 server, do not specify its TCP/IP node name. Instead, use the instance name or the name of a so-called local node, if such a node has been set up.

Please note that the following restrictions apply: The length of the user-defined database name must not exceed eight characters. Furthermore, the maximum blob size is limited to 1 GB.

For good database performance, the `runstats` command must be executed in regular intervals for all tables. For this purpose, the standard installation includes the sample shell script `db2runstats.sh` which is located in the `share/dbTools/` directory. The script needs to be adapted to the installation (database name, user, password, CM or TE).

3.17 Integrating the Template Engine

The Template Engine is responsible for incrementally exporting the content stored in the Content Management Server. To integrate the Template Engine into Fiona, please proceed as follows. Note, that under UNIX all commands need to be executed as the user who installed the CMS.

- For all database types except SQLite:
  Create the database for the Template Engine first. Then edit the corresponding configuration file (for example `config/oracle.xml`) and enter its name in the `server.xml` file in the section `te`, entry `database`. The default is `<database fileName="sqlite.xml"/>`
- Edit the file `config/export.xml` and set the value of `export.incrementalUpdate.isActive` to `true`.
- If you wish to provide a search function on the live server, edit the file `config/indexing.xml` and set `indexing.incrementalExport.isActive` to `true`. Modify `collectionSelection` according to your preferences.
- If necessary, create the required collections, by running the SES in single mode and executing the `createCollection` command.
- To start and stop the Template Engine together with the other CMS applications in the future, add the `TE` to the list of applications in the start/stop script `instance/default/config/rc.npsd.conf`:

```
set conf(apps) [list CM TE SES trifork]
```

Please make sure that the CM is listed before the trifork because at startup the GUI requests data from the CM. The order of the other application shortcuts is irrelevant.

- Run the Content Management Server in single mode to reset the incremental export and to publish the content:

Attention: The command `incrExport reset` must only be used in single mode. It must only be executed if it is guaranteed that no write operations are performed on the data since this would corrupt the exported data. Therefore it is required to make sure the the CM is not running as a server and no other CMs are running or started in single mode while this command is being executed.

```
# Under UNIX
cd ~/FionaDir/instance/default/bin
./rc.npsd stop CM
./CM -single
incrExport reset
exit

# Under Windows
cd C:\Program Files\Infopark\FionaDir\instance\default\bin
rc.npsd.bat stop CM
CM -single
incrExport reset
exit
```

See also the notes on jobs below.

- Now start the CMS applications and execute the `systemPublish` job:

```
# Under UNIX
cd ~/FionaDir/instance/default/bin
./rc.npsd start
./client
connect localhost 3001 root demo
job withName systemPublish exec
exit

# Under Windows
cd C:\Program Files\Infopark\FionaDir\instance\default\bin
rc.npsd.bat
client.bat
connect localhost 3001 root demo
job withName systemPublish exec
exit
```

- Using the the GUI or the Tcl client, configure the execution schedule of the Content Management Server's `systemPublish` job (see below) or execute this job manually each time you want the content to be transferred to the Template Engine.
- Install a webserver to serve the exported documents from the directory `instance/default/export/online/docs`. Alternatively, configure the Trifork server so that it delivers the static documents.

Notes on Jobs

The predefined `systemPublish` job of the Content Management Server transfers updated data and versions from the Content Manager to the Template Engine. Then it instructs the Template Engine to export the data transferred and make the documents available to the server. This job can be executed manually using

```
job withName systemPublish exec
```

Additionally or alternatively the execution schedule of the job can be set so that the job is regularly executed, for example at midnight.:

```
job withName systemPublish set schedule {{minutes 0 hours 0}}
```

To transfer the updated data to the Template Engine without publishing them immediately, the `systemTransferUpdates` job can be used. This should be done after a large amount of data has been uploaded or updated.

For large websites where content is often modified it is recommended to use the `systemTransferUpdates` job several times a day to spread the load. The `systemPublish` job for initiating the publication process should be used less often.

Using the following example commands, you can define execution schedules for transferring the update information to the Template Engine every 15 minutes on working days and to publish the data daily at midnight:

```
job withName systemTransferUpdates set schedule \
  {{minutes {0 15 30 45} weekdays {1 2 3 4 5}}}
job withName systemPublish set schedule {{minutes 0 hours 0}}
```

The execution schedules can be edited comfortably in the GUI.

3.18 Integrating the Search Server

If you wish to subsequently integrate the Search Server into your system, there are three possible uses you should take account of.

3.18.1 Advanced Search in the Editorial System

For the advanced search to be available in the editorial system, please proceed as follows:

1. In the file *`CMS_directory`*`/instance/`*`instace_name`*`/config/indexing.xml` set the `indexing.advancedSearch.isActive` parameter to `true`.

2. To make these changes take effect, please restart the editorial system (i.e. the Content Manager and das GUI):

```
CMS_directory/instance/instace_name/bin/rc.npsd restart CM GUI
```

3. If the Search Server is not running, please start it now:

```
CMS_directory/instance/instace_name/bin/rc.npsd start SES
```

4. Make sure that the collection is filled with data. For this open a Tcl shell, connect to the Content Manager and enter the command `indexAllObjects`:

```
CMS_directory/instance/instace_name/bin/client localhost 3101 root
password:
CM> indexAllObjects
```

   The duration of the indexing process depends on the amount of content to be indexed.

3.18.2 Incremental Export

To use the Search Server for indexing the live content, please proceed as follows:

1. In the file *CMS_directory*`/instance/`*instance_name*`/config/indexing.xml` set the parameter `indexing.incrementalExport.isActive` to `true`.
2. For this configuration change to become effective, it is required to restart the Template Engine:

```
CMS_directory/instance/instance_name/bin/rc.npsd restart TE
```

3. If the Search Server is not running, please start it now:

```
CMS_directory/instance/instance_name/bin/rc.npsd start SES
```

4. Start the Tcl client, open a connection to the Template Engine, and execute the commands `obj touchAll` and `app publish` to export the content and to have it indexed in this process:

```
CMS_directory/instance/instance_name/bin/client teTclHost teTclPort root
TE> obj touchAll
TE> app publish
```

   The duration of this process depends on the amount of data.

To make the search available on the live server, a search form needs to be integrated into the content and the search results need to be computed and displayed dynamically. This can be done by means of the search portlet and the Portal Manager, for example.

3.18.3 Static Export

To index the content for which a static export has been made, please proceed as follows:

1. In the file *CMS_directory*/instance/*instance_name*/config/indexing.xml set the parameter indexing.staticExport.isActive to true.
2. Since the CMS does not create a collection for the static export on its own, please create one. For this, start the SES in single mode and create a new, non-switchable collection:

```
CMS_directory/instance/instance_name/bin/SES -single
% createCollection static-live-docs
```

   For the content to be indexed into this new collection during the static export, please define the rules according to which the collection is selected when content is indexed. This definition is part of the configuration file *CMS_directory*/instance/*instance_name*/config/indexing.xml. Enter something like the following as the value of the indexing.staticExport.collectionSelection entry:

```
<collectionSelection>
  <select collection="static-live-docs">
    <isEqual name="objType" value="image" negate="true" />
    <isEqual name="mimeType" value="application/zip" negate="true" />
    <matches name="blobLength" value="0" negate="true" />
  </select>
</collectionSelection>
```

3. For this configuration change to become effective, the Content Manager needs to be restarted:

```
CMS_directory/instance/instance_name/bin/rc.npsd restart CM
```

4. Since a new collection was created, the Search Server also needs to be started again:

```
CMS_directory/instance/instance_name/bin/rc.npsd restart SES
```

5. Initiate a static export to fill the collection, i.e. to index the content. For this, open a Tcl shell, connect to the Content Manager, and exexute the exportSubtree command (or your export job, in case you have defined one):

```
CMS_directory/instance/instance_name/bin/client localhost 3101 root
password:
CM> obj withPath path exportSubtree filePrefix fsPath
```

3.19 Creating CMS Instances

Instances of Infopark CMS Fiona make it possible to maintain different sets of content plus configuration data while still using common application resources. Common resources are, for example, the executable CMS applications, third party software, and the localization files. When updating common resources, they need to be replaced only once instead of several times as would be the case with several complete installations. This significantly reduces the administration effort.

The content and its structure, the meta data, the configuration and the scripts, on the other hand, are a part of each instance.

Each instance of a CMS application is represented by its own instance directory below the `instance` directory. After the installation exactly one instance exists. It can be found in the `default` directory.

Proceed as follows to create a new CMS instance. Basically, the new instance will be set up like the `default` instance after the installation of the CMS.

- For every instance an individual license file with at least 10 CU (*Concurrent Users*) is required. Please contact Infopark to have your licence divided (in accordance with your contract) into the number of licence files required or to acquire additional licences.
- Execute the script `/share/createInstance/createInstance` (Unix) or `\share \createInstance\createInstance.bat` (Windows). (The path is relative to the Fiona installation directory.)
- The script asks you to enter the path of the licence file:
  `Enter the path to the new instance's license file:`
  Input here the path of the license file belonging to the new instance.
- Afterwards the name of the new instance is queried:
  `Enter the instance name:`
  from version 6.5: Use only lower-case letters from a to z (no umlauts or other special characters), digits and underscores. The name may be up to 9 characters long and needs to start with a letter. The reason for this is that the instance name becomes part of database table names generated by the CMS. Several database-related restrictions apply to the length of names and the characters they may be composed of.
- The script then asks for the first port of the port range:
  `Enter the port range start (e.g. 3101):`
  Enter here the smallest port number to be used by the CMS. The script will create further (greater) port numbers, starting from the specified number.
- Then a directory named like the new instance is created below `Fiona/instance`. In this directory `.../share/newinstance.zip` is unpacked. The licence is copied into the `config` directory created by the unpacking procedure.

- Then the configuration files of the new instance are adapted. This includes, among other things, the ports and the names of the web applications. The output of the script informs you of the parameters and the respective files in which they have been set.
- Under Windows the services are then installed.
- The result is an individual CMS instance consisting of the CM, SES, and the GUI and PM-PL web applications. PM-PL provides the portlets for preview pages opened from the GUI. For deploying the web applications use the `rc.npsd deploy` (Unix) or `rc.npsd.bat deploy` (Windows) command to be found in the `bin` directory of the new instance.
- If required, prepare the instance for productive use by integrating a different database, activating the incremental export and configuring a different user manager.

The new instance is now ready to be used and the CMS applications can be started using `instance/instanceName/bin/rc.npsd start`. At startup, the applications automatically create their database tables in the configured databases. Likewise, the Search Engine Server automatically creates its preset collections. The HTML user interface (GUI) can be accessed via the URL path *instance_name*`/NPS`, where *instance_name* is the name of the new instance. You can log into the new instance as the `root` user using the empty password.

4

4 The Starting Procedure of the CMS Applications

4.1 Tasks at Startup

At startup, all CMS server applications first read their system configuration. Then they check the availability of the database configured and create new database tables if required. With SQLite, the database itself is also created if it does not exist.

During the initialization procedure, Tcl scripts are read from the following directories.

```
share/script/common/clientCmds
share/script/app/clientCmds
share/script/common/serverCmds
share/script/app/serverCmds

instance/default/script/common/clientCmds
instance/default/script/app/clientCmds
instance/default/script/common/serverCmds
instance/default/script/app/serverCmds
```

In the paths listed above, *app* stands for the application-specific script directory *cm* or *ses*. Please note that the Template Engine reads the scripts of the Content Management Server at startup. Scripts in the `common/clientCmds` and `common/serverCmds` directories are read by all applications at startup.

Scripts contained in the directories listed in the first group are common to all instances and are read by the corresponding CMS applications, independently of the instance.

You can place your instance-specific scripts in the directories listed in the second group. These scripts are read after the scripts of the first group so that procedures stemming from the first group can be redefined by instance-specific procedures if required.

When a dump is generated, only the instance-specific scripts are saved. Analogously, when a dump is restored, the instance-specific script directories are reset to the contents of the dump (in fact, the instance-specific script directories are deleted prior to restoring the dump). See also Dumping and Restoring Data.

The script files contained in the directories listed above are sorted alphabetically (ASCII) and then read in ascending order.

The server commands from the `serverCmds` directories are available only in commandline mode while the client commands (from the `clientCmds` directory) can be used in single mode (commandline parameter `-single`) and in the Tcl client (under a different name, if applicable, see below) as well as in Tcl procedures.

Finally the Content Manager executes the script `usermanAPI.tcl` at startup. The functions defined in this file represent the interface the applications use for communicating with the integrated or an external user manager.

4.2 Registering Tcl Procedures

The CMS applications with Tcl interfaces provide the administrator with a Tcl interpreter in command-line mode (command-line argument `-single`). The instructions and procedures executed in this interpreter have complete access to the system.

Normally, users who use a Tcl shell to connect to the Content Manager or the Template Engine should not have complete access to the system. For this reason, all procedures which are to be executable in the Tcl shell need to be registered explicitly. This is done by using the `safeInterp` procedure:

```
safeInterp alias serverProc clientProc
```

Using this ccommand, the *`serverProc`* server procedure is registered under the name *`clientProc`* and made available in Tcl clients.

The `getRegisteredCommands` procedure supplies the name of all procedures which a Tcl client should forward for execution to the Tcl server. In standard installations, this procedure is called in the start script of the Tcl client.

5

5 The System Configuration of the CMS Applications

The system configuration is a hierarchical data structure which the CMS applications read from the file *nps.xml* at startup. This file is located in the *config* directory below each instance directory.

Please note that an application must be restarted after the system configuration was changed. When changes to the configuration of the Content Management Server have been made, the HTML GUI must be restarted as well because it caches configuration data of the Content Manager. Use the `rc.npsd restart` command or the corresponding command from the *Services* menu in the Windows control panel.

5.1 Format of the Configuration Files

Each entry in the system configuration has a name under which a value is stored. Under the name `keepOldVersions`, for example, the value `true` can be stored:

```
<configuration>
  ...
  <content>
    <keepOldVersions>true</keepOldVersions>
  ...
  </content>
</configuration>
```

Each system configuration entry is represented by an XML element. The name of the opening and closing tag of such an element corresponds to the name of the entry, and the content of an XML element corresponds to the value of the entry.

The value of an entry in the system configuration can not only be a string, but also a list of strings or a so-called dictionary. A dictionary stores a list of name-value pairs together under one name. The system configuration is itself a dictionary represented in the configuration files by the root element `configuration` .

A main entry in the system configuration is therefore an entry in the `configuration` dictionary (such as `keepOldVersions` in the above example). A dictionary contains any number of entries as its value, however no additional text:

```
<configuration>
  ...
  <mimeTypes>
    <html>text/html</html>
    <css>text/css</css>
    <gif>image/gif</gif>
    <zip>application/zip</zip>
  </mimeTypes>
```

```
  ...
</configuration>
```

Each value can also be a list. A list contains unnamed values in a defined order. Each element of a list is represented in the configuration files by any XML element, preferably `listitem` (*list item*) or the corresponding CRUL element (see the XML Interface documentation). The following example shows how the list of global permissions to be available in the Content Manager is specified. Note that in lists the tag attribute `type` must be given and assigned the value `list`.

```
<configuration>
  ...
  <globalPermissions type="list">
    <globalPermission>permissionGlobalRoot</globalPermission>
    <globalPermission>permissionGlobalUserEdit</globalPermission>
    <globalPermission>
      permissionGlobalUserAttributeEdit
    </globalPermission>
    <globalPermission>permissionGlobalRTCEdit</globalPermission>
    <globalPermission>permissionGlobalExport</globalPermission>
    <globalPermission>permissionGlobalMirrorHandling</globalPermission>
  </globalPermissions>  ...
</configuration>
```

Each list element can itself be a list or a dictionary. Furthermore, the values of dictionary entries can be lists or dictionaries again.

Dividing Configuration Files

The size of the configuration files can be reduced considerably by storing sections of the configuration in separate files. This replacement mechanism ist often used in the standard installation. In this way, the content of the `server` element, for example, with which client-server connections are configured, can be imported from the *server.xml* file.

```
<configuration>
  <server fileName="server.xml"/>
  ...
</configuration>
```

Using the tag attribute `fileName`, the name of the file whose versions are to replace the respective XML element is specified. The specified file name or path is relative to the configuration directory.

5.2 Entries in the System Configuration

The system configuration is a hierarchical data structure that is divided into several files. When the Content Manager, the Template Engine, or the Search Engine Server starts, these files are read and joined into one large hierarchy of system configuration entries. All files can be found in the `config` directory of the CMS instance.

5.2.1 content

The elements of this system configuration entry, which can be found in the instance-specific `config/content.xml` file, refer to the content.

- `anonymousWorkflowLog`: Determines whether users initiating a workflow action or a file command remain anonymous (`true`) or are logged (`false`).

- `caseSensitiveObjectNames`: The value of this key (`true` or `false`) determines whether the Content Manager distinguishes between upper and lower case letters with names of CMS files. Under Windows the value should be `false` whereas it should be `true` under Solaris and Linux. These values also are the default values. After the Content Manager has been started for the first time the value of this entry must not be changed any more. The value `false` only ensures that a folder never contains two files whose names differ only in case. The value does not influence, for example, how layout files are searched for during the export. Example: If a layout file named `microNav` exists, the exporter will not find it (independently of the value of `caseSensitiveObjectNames`) if the base layout contains the following instruction: `<npsobj insertvalue="template" name="micronav" />`
- `generateThumbnails`: Determines whether thumbnails are generated for files that contain images (`true`) or not (`false`). Thumbnails are generated by calling the `generateThumbnail` callback in the `share/script/cm/serverCmds/generateThumbnail.tcl` file. By default, this callback calls a third party application (see the callback code).
- (from Version 6.7.1): Determines whether images con be imported into resource files (type `generic`) (`true` or `false`).
- `inputCharsets`: The list of valid character sets for the `inputCharset` user configuration entry. When a file is imported, one of these character sets can be specified as the character set of the file. Example:

```
<inputCharsets type="list">
  <listitem>iso8859-1</listitem>
  <listitem>iso8859-2</listitem>
  <listitem>utf-8</listitem>
</inputCharsets>
```

- The complete list of the supported character sets can be determined with the Tcl command `encoding names`.
- `invalidAnchorTargets`: The list of invalid link targets. If a link in a version contains one of the targets listed here, the version counts as incomplete and cannot be released. Example:

```
<invalidAnchorTargets type="list">
  <listitem>myGlobalFrame</listitem>
  <listitem>aFrameName</listitem>
</invalidAnchorTargets>
```

- `keepOldVersions`: Activates or deactivates the versioning function and in doing so determines whether previously released versions should be kept or not (`true` or `false`).
- : Here you may define exceptions for external link checking available from version 6.5.0 of the CMS. Each entry contains a pattern of valid URLs. HTTP codes less than 400 will always be considered valid. Additional valid HTTP codes may be specified. Without code pattern, any matching URL is valid. URLs and codes may be abbreviated by means of an asterisk. Furthermore, the response timeout in seconds (default: 5) can be specified (from Version 6.7.2)

```
<linkChecker>
  <whiteList>
    <entry urlPattern="*" httpCodePatterns="401,403"/>
    <entry urlPattern="http://www.wikipedia.org/wiki/*"/>
```

```
  </whiteList>
  <timeout>5</timeout>
</linkChecker>
```

- `mimeTypes`: With this entry file name extensionss are assigned to MIME types. The dictionary also determines the allowed file name extensions. The Content Manager uses this assignment in the preview, for example. Example:

```
<mimeTypes>
  <html>text/html</html>
  <css>text/css</css>
  <gif>image/gif</gif>
  <zip>application/zip</zip>
</mimeTypes>
```

  Please note that the main content (field `body`) of a version belonging to a file of the *document* or *folder* type is only processed by the Content Manager if the file name extension of the version has been assigned the `text/html` MIME type. Otherwise the contents of the main content is treated as pure text.
- `sortKey`: The name of the field according to which files in folders are sorted when no sort key has been defined for these folders.
- `thumbnailSize`: The size of the thumbnails (preview images) the `generateThumbnail` callback creates (see Thumbnail Function). Since thumbnails are square, the value defines both the height and the width of the image.
- `validAnchorTargets`: The list of valid link targets. Editors can only use these targets when they edit links. Example:

```
<validAnchorTargets type="list">
  <listitem>myFrame1</listitem>
  <listitem>myFrame2</listitem>
</validAnchorTargets>
```

5.2.2 export

This configuration element, which can be found in the instance-specific `config/export.xml` file, defines the export of files by the Content Management Server. The subelements have the following meaning:

- `abortOnError`: Using this key (value `false` or `true`) you determine whether an export is aborted when an error occurs (if, for example, nonexistent fields are queried using NPSOBJ).
- `absoluteFsPathPrefix`: This option is only effective if links are exported with absolute paths (subentry `exportAbsolutePaths`). `absoluteFsPathPrefix` defines the character string that is used as a prefix for all file paths in links. Therefore, this option allows you to place the exported files into directories located somewhere below the webserver's document root directory.
- `absoluteUrlPrefix`: same as `absoluteFsPathPrefix`, but applied to the path in URLs.

- `charset`: The character set and its encoding of the exported documents. The value can be one of the element names in the `charsetMap` subentry.
- `charsetMap`: The element name corresponds to the name of the character set in Tcl, whereas the element's content is the name used in HTML and XML. One of the element names listed here can be specified as the value of the `charset` subentry. If `charsetMap` has not been defined, then <utf-8>UTF-8</utf-8> is used.
- `convertNpspm` (NPS 6.0.3 and later): The value of this entry controls how the CMS handles `npspm` elements during the export. The element has the following subelements.
  - `isActive`: Determines by means of the value `true` or `false` whether `npspm` tags (with which Portal Manager functions can be included in exported documents), are converted to the Velocity template language or not.
  - `hideForbiddenLinks`: Controls by means of the values `true` or `false` whether links to documents under access control can even be clicked (no = `true`) if the website visitor is not allowed to access the target document. If so, the corresponding error page configured in the `web.xml` file in the instance-specific directory `webapps/PM/WEB-INF` as the value of an `error-page` element is displayed (see the examples in this file). By this means, the website visitor can be directed to a login page, for example.
  - `mapping`: With this entry, file name extensions are associated with the language into which `npspm` tags are translated in exported pages. Example:

```
<mapping>
  <htm>velocity</htm>
  <html>velocity</html>
  <jsp>jsp</jsp>
</mapping>
```

- `defaultDocType`: The value of this entry can be either `HTML` or `XML` sein, depending on the desired format of the tags generated by the CMS.
- `exportAbsolutePaths`: defines whether links to files are to be exported with absolute (`true`) or relative (`false`) paths. This option must be switched on if files that are included with the NPSOBJ-insertvalue-dynamiclink instruction contain links to files. In this case, relative links in the files included would fail to work properly since the location of the originally exported files is not predictable.
- `exportBaseDir`: Determines the directory, into which the Template Engine places the exportierted files creates subdirectories if required. The default value is `export`.
- `guiUrl`: The value of this entry determines the URL up to the web application of the GUI. The supplied `systemExecute` command `editLink` uses this entry. Furthermore, the HTML editor (edit on pro) uses this URL to determine the URL of preview images displayed in WYSIWYG mode. Therefore a URL must be specified as `guiUrl` that can be resolved externally. Example:

```
<guiUrl>http://localhost:8080/NPS</guiUrl>
```

- `incrementalUpdate`: defines whether the Content Management Server sends data for the incremental update to the Template Engine. Furthermore it defines how the Template Engine can be reached.
  - `isActive`: activates (`true`) or deactivates (`false`) the incremental update

  - `login`: login under which the Content Management Server can log-in to the Template Engine.
  - `password`: The password for logging-in.
- `referenceAttributes`: Under this entry a list of version fields can be specified that are to trigger a reference type dependency instead of a content change dependency. This has the effect that files depending on such a field are not invalidated when the value of the field changes. Add a field to this list if it is used exclusively for creating links to the files concerned, for example in automatically generated navigations. Do not specify fields of the `linklist` type here. Example:

```
<export>
  ...
  <referenceAttributes type="list">
    <referenceAttribute>showInNav</referenceAttribute>
    <referenceAttribute>hideInNav</referenceAttribute>
  </referenceAttributes>
</export>
```

- `tclDynamicLinkFormatterCommands`: This dictionary defines the alias names of the formatting procedures that can be called when NPSOBJ insertvalue-dynamiclink instructions are evaluated.

  The corresponding `formatter` routines are defined in the file `dynamicLinkFormatters.tcl` located in the `script/cm/serverCmds` directory below the instance directory (i. e., for example below `instance/default`). This directory is also used by the Template Engine.

  Please note that the web pages on the live server must be exported again if Tcl procedures affecting the export results are modified. For this purpose, the `obj touchAll` command can be used. Example:

```
<tclDynamicLinkFormatterCommands>
  <phpInclude>formatter:phpInclude</phpInclude>
  <phpIncludeOnce>formatter:phpIncludeOnce</phpIncludeOnce>
  <phpRequire>formatter:phpRequire</phpRequire>
  <phpRequireOnce>formatter:phpRequireOnce</phpRequireOnce>
  <include>formatter:include</include>
</tclDynamicLinkFormatterCommands>
```

- `tclFormatterCommands`: The dictionary contains procedure alias names, which can be used as `formatter` attribute values of NPSOBJ-insertvalue-var instructions. Here, the names of the procedures which are to be called when the NPSOBJ instructions are evaluated are assigned to the alias names. Example:

```
<tclFormatterCommands>
  <phpVardef>phpVardef</phpVardef>
</tclFormatterCommands>
```

- `tclSystemExecuteCommands`: This dictionary defines the NPSOBJ systemExecute commands that can be used in layouts. The commands must have been made available as server commands (as with `tclDynamicLinkFormatterCommands`). Example:

```
<export>
  ...
  <tclSystemExecuteCommands>
    <procedureAliasName>trueProcedureName</procedureAliasName>
  </tclSystemExecuteCommands>
</export>
```

- `toclistSortLanguage`: Specifies the local language setting to be used for sorting file lists alpabetically (`npsobj list="toclist"`). The language setting consists of two lowercase letters for the language followed by two uppercase letters for the country (for example `de_DE` for German). If this system configuration entry is not present, the operating system setting is used, i. e. the value is taken from the LC_COLLATE environment variable.
- `validDateTimeOutputFormats`: The valid date formats are stored in this dictionary as subelements. These formats can be specified in NPSOBJ elements as values of the *format* attribute in order to format date values. Example:

```
<validDateTimeOutputFormats>
  <dayMonthYear>%d/%m/%Y</dayMonthYear>
</validDateTimeOutputFormats>
```

Date and time formats as they can be defined in this system configuration entry are strings. You can use the following for current system-time parts in these strings:

Place-Holder	Meaning
`%d`	day as two-digit number
`%m`	month as two-digit number
`%y`	year as two-digit number
`%Y`	year as 4-digit number (with century)
`%j`	day of the year as 3-digit number (001-366)
`%H`	hour as two-digit number (24h format)
`%I`	hour as two-digit number (12h format)
`%M`	minutes as two-digit number
`%S`	seconds as two-digit number
`%p`	am/pm
`%Z`	the timezone name
`%z`	timezone difference to GMT (±HHMM)
`%w`	weekday as two-digit number (0=Sunday)
`%%`	a percentage sign

- `wantLinkCallback`: Activates or deactivates link callback functions (`YES` or `NO`) with which links can be modified during export or preview.

- `writeLiveServerReadPerms`: Determines whether the live server read permissions (group names assigned to the files) are transferred to the Template Engine during a content update (`true`) or not (`false`).

5.2.3 gui

This system configuration entry, which can be found in the instance-specific `config/gui.xml` file, configures the properties of the Content Navigator as well as the behaviour of the GUI web applications.

- `customCommands`: Determines the custom extensions of the Content Manager pages (see Configuring Custom Commands).
- `dynamicPreviewDirectory`: If the dynamic preview has been activated, specify here the directory in which the exported preview files are to be made available to the webserver. Example: `/absolute/dirpath`
- The preview web server needs to be configured to read the files from the specified directory using the character set defined in the `export.exportCharset` system configuration entry. If the Apache web server is used, the configuration explained in the `dynamicPreviewUrl` section below is sufficient.
- `dynamicPreviewExtensions`: List of file name extensions (without the period) to be dealt with by the webserver. If the list is empty, the dynamic preview is deactivated, otherwise the connection to the preview server needs to be configured by means of the `dynamicPreviewDirectory` and `dynamicPreviewUrl` entries. Entries in this list are formed using `extension` elements. Example:

```
<dynamicPreviewExtensions>
  <extension>php</extension>
  ...
</dynamicPreviewExtensions>
```

- `dynamicPreviewUrl`: URL under which the preview webserver accesses the `dynamicPreviewDirectory`. To this URL the path of the file whose version is to be further processed dynamically is appended.
  Example: `http://localhost:3100/preview`. If a file has the path `/path/page.php` the URL `http://localhost:3100/preview/path/page.php` is requested. In this case, the preview server needs to be configured in such a way that `/preview/` points to the `dynamicPreviewDirectory`.

  The preview web server needs to deliver files in the character set defined in the `export.exportCharset` system configuration entry. The character encoding needs to be specified in the Content-Length header. If the Apache web server is used, it is normally sufficient to specify `AddDefaultCharset` *`charset`* in the file `httpd.conf`, i. e. `AddDefaultCharset UTF-8`, for example.
- `fontFamily`: Fonts from which the user can choose when configuring page display. Each font is defined by specifying its name in an `item` subelement. Please note that the browser possibly does not support all fonts and replaces missing fonts with existing ones, if necessary.
- `fontSize`: Font sizes from which the user can choose when configuring page display. Each font size is defined by specifying its point size in an `item` subelement (example: `12pt`).
- `jreSpecification`: Under Windows, the applet for executing local applications requires the Java runtime environment 1.4.2_17 or later (see also the system requirements). If JRE 1.5 or later is installed in addition to JRE 1.4.2_x, this system configuration entry can be used to force

the Internet Explorer to nevertheless use JRE 1.4.2_x for this applet. To do this, modify the the predefined entry in the following way:

```
<jreSpecification name="1.4.2_17">
  <clsId>CAFEEFAC-0014-0002-0007-ABCDEFFEDCBA</clsId>
  <codebase>http://java.sun.com/update/1.4.2/
    jinstall-1_4_2_17-windows-i586.cab#Version=1,4,2,17</codebase>
</jreSpecification>
```

The example above refers to the JRE version 1.4.2_17. If you use a different version, please adapt the version number accordingly.

- `previewMasterTemplates`: Names of the layouts that can be selected for the preview. Each layout is defined in a `template` subelement. Example:

```
<previewMasterTemplates>
  <template>
    <name>mastertemplate</name>
    <title lang="de">Mastertemplate</title>
    <title lang="en">master template</title>
  </template>
</previewMasterTemplates>
```

- `roles`:
  This dictionary contains the role definitions. (In the Content Navigator, roles are called user interfaces.) Each role in this dictionary is defined by means of a `role` element. For the possible values, please refer to the section `rolePreferences`. Example:

```
<roles>
  <role>
    <name>previewer</name>
    <title lang="de">Nur Vorschau</title>
    <title lang="en">Preview Only</title>
    <availableFor>
      <group>editors</group>
    </availableFor>
    <values>
      <browserName fixed="true">pb</browserName>
    </values>
  </role>
</roles>
```

Please note: If you remove a role currently in use by users, these users will not be able to log into the GUI until they have been given an existing role. To do this, use the following command: `userConfigForUser` *`login`* `setTexts guiPreferences.currentRole` *`newRoleName`*

- `tinymceConfig`: When TinyMCE is initialized as an editor for the main content, its configuration is passed to it as a JavaScript object. By means of this configuration entry, `tinymceConfig`, a configuration can be specified as a JSON string for every user role. Furthermore, a (`default`) configuration can be specified for users with roles not given here. Example:

```
<tinymceConfig>
  <default>{
    theme_advanced_buttons2: ""
  }</default>
  <restricted>{
    theme_advanced_buttons1: "bold,italic,underline",
    theme_advanced_buttons2: ""
  }</restricted>
</tinymceConfig>
```

This system configuration entry is optional. If it is missing or neither includes an entry for the user's role nor a `default` entry, the presets of TinyMCE are used. If, on the other hand, an entry for the user's role, or, as a fallback, a `default` entry exists, the settings of this entry extend or replace the presets of the TinyMCE configuration.

- `webDav`: When a file is copied to or created in a web folder, the format of the corresponding new CMS file is determined by means of this system configuration entry and the name extension of the new file. The default format for name extensions not listed here is `generic`. Furthermore, the format of folders (directories in web folders) can be specified here using the defaultPublicationClass subentry. Example:

```
<webDav>
  <objectClasses type="dictionary">
    <html>document</html>
    <wri>generic</wri>
    <css>generic</css>
    <js>generic</js>
    <txt>generic</txt>
    <pdf>generic</pdf>
    <doc>generic</doc>
    <gif>image</gif>
    <jpg>image</jpg>
    <jpeg>image</jpeg>
    <png>image</png>
  </objectClasses>
  <defaultPublicationClass>publication</defaultPublicationClass>
</webDav>
```

TinyMCE configuration presets

TinyMCE and its configuration should never be modified since the changes would be reverted the next time CMS Fiona is updated. Essentially, the presets define a toolbar consisting of two rows that include the elements (buttons or drop-down boxes) named below.

```
{
    mode: "specific_textareas",
    editor_selector: "mceEditor",
    dialog_type: "modal",
    theme: "advanced",
    plugins: "npsfilebrowser,paste,searchreplace,table",
    theme_advanced_toolbar_location: "top",
```

```
    theme_advanced_buttons1:

 "bold,italic,underline,strikethrough,|,sup,sub,|,justifyleft,justifycenter,justifyright,justifyfull,|,bullist,
    theme_advanced_buttons2:

 "undo,redo,|,search,replace,|,removeformat,visualaid,|,npsLinkBrowser,npsImageBrowser,|,tablecontrols,|,code",
    theme_advanced_buttons3: "",
    theme_advanced_toolbar_align: "left",
    theme_advanced_resizing: true,
    theme_advanced_statusbar_location: "bottom",
    skin: "o2k7",
    skin_variant: "silver",
    convert_urls: false,
    valid_elements: "*[*]"
}
```

5.2.4 indexing

This configuration element, which can be found in the instance-specific `config/indexing.xml` file, determines the details of content indexing by the Content Management Server and the Template Engine.

- `advancedSearch`: Configures the indexing when the advanced search is used in the Content Management Server. The element has the same subentries as `incrementalExport`.
- `contentPreprocessors`: This element defines preprocessors, that are called before versions are indexed. If no preprocessors are to be called, `<contentPreprocessors />` must be specified. Example for an internal and an external preprocessor definition:

```
<contentPreprocessors type=list>
  <preprocessor>
    <processor type="internal"/>
    <mimeTypes type="list">
      <mimeType>application/vnd.ms-excel</mimeType>
      <mimeType>application/vnd.ms-powerpoint</mimeType>
      <mimeType>application/msword</mimeType>
    </mimeTypes>
  </preprocessor>
  <preprocessor>
    <processor type="external">bin/tclsh</processor>
    <processorArguments type="list">
      <argument>pdfToTextWrapper.tcl</argument>
    </processorArguments>
    <mimeTypes type="list">
      <mimeType>application/pdf</mimeType>
    </mimeTypes>
  </preprocessor>
  <preprocessor>
    <!-- Another preprocessor for other MIME types -->
  </preprocessor>
</contentPreprocessors>
```

Each preprocessor is responsible for at least one MIME type. As with all lists, The `contentPreprocessors` Element has an obligatory attribute, `type="list"`. This element consists

of subelements each of which defines a preprocessor. Each `preprocessor` subelement has the following subelements:

- `mimeTypes` defines the MIME types of the versions to be processed by this preprocessor.

  Attributes: `type` with the value `list` (obligatory).

  Content: For each MIME type a `mimeType` element, whose value is the respective MIME type (for example `text/html`).
- `processor` defines the preprocessor for versions with one of the specified MIME types.

  Attributes: `type` with one of the following values: `internal`, `external`, `ignore`, `ignoreBlob`. Default: `external`.

  Content, if `type` has the value `internal`: The blob is filtered by the Verity filter application before it is indexed.

  Content, if `type` has the value `ignore`: the version is not indexed; the content of the element is ignored.

  Content, if `type` has the value `ignoreBlob`: empty. All fields except the main content are indexed. The main content is not converted (normally, all field values are converted to plain text before a version is indexed).

  Content, if `type` has the value `external`: The data to be indexed is passed to the program specified. Further arguments can be passed to it by means of the `processorArguments` element. For further explanations on the external preprocessor facility please refer to the Search Server documentation.
- `processorArguments` is optional. This element defines the arguments to be passed to the program defined as `processor`.

  Attributes: `type` with the value `list` (obligatory).

  Content: Each commandline argument is specified as the content of an `argument` subelement.

  Note: Up to version 6.7.0, the commandline arguments need to be provided directly as the value of the `processorArguments` element (e.g. `<processorArguments>pdfToTextWrapper.tcl</processorArguments>`).

- `incrementalExport`: Configures the indexing for the incremental export. The element has the following subentries:
  - `isActive`: Switches indexing on (`true`) or off (`false`).
  - `collectionSelection`: Defines rules that determine the collection to be used for indexing a document. Example:

```
<collectionSelection>
  <select collection="cm-contents">
    <isEqual name="state" value="edited"/>
  </select>
  <select collection="cm-contents">
    <isEqual name="state" value="released"/>
  </select>
</collectionSelection>
```

In each `select` element `collection` determines a collection into which a document is indexed if all of the rules contained in the element apply. The rules contained in a `select` element are AND-related. An OR relation can be formed by using more than one `select` element in which the same collection name is specified. If the `collection` attribute is omitted, the document is not indexed if the rules apply. The rules are processed one by one. The first set of rules that applies determines the collection into which the document is indexed. Each rule is represented by one element and can be reversed by adding the tag attribute `negate="true"`. The following rules exist:

- `isEqual`: This rule applies if the value of the file or version field specified by means of the `name` tag attribute exactly corresponds to the string `value`. Example:
  `<isEqual name="mimeType" value="application/x-shockwave-flash" />`
- `isTrue`: This rule applies if the file or version field specified by means of the `name` tag attribute has the value `true`, `yes`, or `1`.
- `isFalse`: This rule applies if the file or version field specified by means of the `name` tag attribute has the value `false`, `no`, or `0` hat.
- `hasPrefix`: This rule applies if the value of the file or version field specified by means of the `name` tag attribute begins with the string `value`. Example:
  `<hasPrefix name="mimeType" value="application/" />`
- `hasSuffix`: This rule applies if the value of the file or version field specified by means of the `name` tag attribute begins with the string `value`. Example:
  `<hasSuffix name="mimeType" value="/zip" />`
- `matches`: This rule applies if the value of the file or version field specified by means of the `name` tag attribute contains a string that matches the regular expression specified as `value`. Example:
  `<matches name="collspec" value=".*live.*" />`

- `staticExport`: Configures the indexing for the static export by the Content Management Server. The element has the same subentries as `incrementalExport`.
- `vseLocale`: Determines the locale (language specific settings) the Verity Search Cartridge is to use. `uni`, `germanx`, and `englishx` are available by default (additional locales can be acquired). `uni` is a universal locale that uses the UTF-8 character encoding. However, no language-specific search query functions such as stemming or typographical tolerance can be used. The value specified is applied to all collections. If this value is changed, all collections need to be created again.

5.2.5 licenseKey

This element of the system configuration contains the license file. Example:

```
<licenseKey fileName="license.xml"/>
```

The structure of the license file is described in a separate document.

5.2.6 searching

This configuration element, which can be found in the instance-specific `config/searching.xml` file, determines the details of the search for content on the live system and of the advanced search on the editorial system. Example:

```
<searching>
  <preprocessor>tclProcName</preprocessor>
```

```
  <postprocessor>tclProcName<postprocessor>
  <searchResultFormatterCommands>
    <procedureAliasName>trueProcedureName</procedureAliasName>
    <uppercase>uppercase</uppercase>
  </searchResultFormatterCommands>
</searching>
```

The elements below the `searching` element have the following meaning:

- `postprocessor`: This element is optional. Its content can be the name of a Tcl procedure which is called after a search request has been processed. To the Tcl procedure the XML-Fragment `searchResults`, i.e. the search result as a string, is passed as the only argument. The procedure can modify this fragment and must return it as a syntactically correct XML string.
- `preprocessor`: This element is optional. Its content can be the name of a Tcl procedure which is called before the search process itself is triggered. The XML fragment `search`, originating from the search request, is passed to the procedure as the only argument. The procedure can modify this fragment if desired. The return value of the procedure is required to be a string that forms a syntactically correct `search` element of a search query.
- `searchResultFormatterCommands`: This element is optional. Its subelements define the alias names and the real names of Tcl procedures used for formatting the field values of documents returned by a search. Each subelement assigns an alias to a name. The element name corresponds to the alias, the element's content to the procedure name. To such a procedure the value to be formatted is passed as the only argument. The procedure's return value is the formatted string.

The Tcl procedures referenced in the system configuration elements described above should be implemented in a Tcl script file which is sourced during start-up from one of the Search Server's script directories (`share/script/ses/serverCmds` and `instance/`*`instanceName`*`/script/ses/serverCmds`).

5.2.7 server

The connection data for clients is stored in this dictionary which can be found in the instance-specific `config/server.xml` file. The dictionary has the following entries:

- `cm`: Connection Data of the Content Management Server.
  - `database`: Dictionary whose content determines the database configuration Example:

    ```
    <database filename="sqlite.xml" />
    ```

    Subelements can be:

    - `adaptor`: Name of the database adapter.
    - `database`: Name of the database (not SQLite).
    - `password`: Password of the database user (not SQLite).
    - `server`: Name of the database server (not SQLite, DB2).
    - `storeBlobsInDatabase`: Using this configuration value, you can determine whether blobs (versions, user settings, etc.) are to be saved in the database (`true`, the default) or stored in the file system as files (`false`). Never change this value after the first start of the application unless appropriate precautions have been taken. Otherwise, your data will be corrupted.
    - `user`: Login used for logging into the database (not SQLite).

  - `httpConnectHost`: Other CMS applications read the value of this entry to determine the name of the server under which the application can be reached. The value is obligatory.
  - `httpHost`: The server name or the IP address to which the server binds its HTTP port. If nothing is specified here, the server can be reached under all available names of the machine. If `localhost` is specified here, the server can be accessed from local clients.
  - `httpPort`: HTTP port on which the application can be reached.
  - `httpProxyHost` (optional): Name or IP address of a proxy server via which the application can be reached.
  - `httpProxyPort` (optional): Proxy server port.
  - `stateHost`: Name of the computer via which the condition of the CMS application can be queried internally.
  - `statePort`: State server port.
  - `tclHost`: The name of the machine on which the Tcl server of the CMS applications can be reached.
  - `tclPort`: Tcl server port.
- `email`: Defines the connection parameters for e-mail notification (from version 6.5 including reminder notification). Example:

```
<email>
  <!-- leave 'host' empty to deactive email services-->
  <host>localhost</host>
</email>
```

- `log`: The subelements of this dictionary define which type of information is logged in which file. Example:

```
<configuration>
  <!-- Valid "info" log levels are 0, 1, 2, 3.
  Messages will go to the specified log file.-->
  <logger name="info" level="2" logFile="info.log" />
  <logger name="error" logFile="error.log" />
  <logger name="alert" logFile="error.log" />
  <logger name="warning" logFile="error.log" />
  <logger name="debug" logFile="debug.log" />
</configuration>
```

- The `name` attribute determines the log type, `logFile` the log file. By means of `logFile` an individual log file can be specified for each log type. Releative paths refer to the CMS directory *log*. The log type `info` allows you to determine the amount of information to log. For this, use the `level` attribute (0 = nothing, 3 = much).
- For the GUI, logging is configured in the separate file `instance/webapps/NPS5/WEB-INF/log4j-config.xml`.
- `ses`: Connection data of the Search Engine Server. Like `cm`, however without the `database` element.
- `rpcUrlPrefix`: Prefix of the URL the Content Management Server uses to talk to the Java Kernel.

- `te`: Connection data of the Template Engine. The Template Engine is provided with data from the Content Management Server via HTTP. This element contains the same elements as `cm`.

5.2.8 tuning

By means of the elements of this system configuration entry, which can be found in the instance-specific `config/tuning.xml` file, the operation of Fiona can be optimized.

- `clearCachesFrequency`: Number of commands or exported files after which the Tcl Server clears its cache.
- `export`: Parameters for optimizing the export:
  - `acceptableFailures`: The maximum number of file export failures not causing the export process to terminate.
  - `maxNumberOfExportedObjects`: The maximum number of files a Template Engine slave exports before the master replaces it with a new slave.
  - `maxParallelExports`: The maximum number of slaves of a Template Engine master exporting data simultaneously. This value should only be increased, if the export does not fully occupy the system, i. e. if unused CPU and I/O resources still exist. At least one slave should always be available for external requests, meaning that `maxParallelExports` should always be less than `maxSlaves`.
  - `maxRecursionLevel`: The maximum recursion depth in nested NPSOBJ instructions.
  - `timeout`: The maximum number of seconds the Template Engine master waits for a slave to export the file assigned to it.
  - `usesAllThreshold`: The maximum number of files on which a file may depend. If this number is exceeded, the dependencies are replaced with a single general dependency.
- `indexing`: The following parameters ensure that during high system load not too many documents need to be indexed at once and that during low system load not too much time elapses between two indexing processes.
  - `filterTimeout`: Time (in milliseconds) the Verity filter application (located in *bin/IF* below the Instance directory) may take for filtering a blob (default: 300, used by the SES). The filters use the value 0 to switch off the timeout.
  - `interval`: Period of time after which the accumulated documents are indexed at the latest.
  - `maxBulkSize`: Maximum number of documents that may accumulate without starting an indexing process (i. e. indexing is forced to take place when this number is exceeded).
  - `optimizationInterval`: The interval at which the Search Engine Server optimizes its collections. 0 or a negative value disables optimization.
- `jobMaxLogLength`: The value of this entry determines the maximum number of log entries that are to be kept.
- `master`: Defines parameters that refer to the main instance of the respective application.
  - `cm`: Defines the optimization parameters of the Content Management Server.
    - `busySlavesWarningThreshold`:
      Specifies the number of busy slaves reached or exceeded that causes a warning to be written to the log file. If no value has been specified, or the value is `0` or less, or greater than `maxSlaves`, `maxSlaves` is used. Alteratively, a percentage can be specified which then refers to the value of `maxSlaves`.

Example: Setting `maxSlaves` to `5`, and `busySlavesWarningThreshold` to `70%` results in a warning threshold of 4 slaves.

- `maxSlaves`: Maximum number of slaves permitted to run simultaneously.
- `minIdleSlaves`: Minimum number of idle slaves. The master server ensures that at least this number of slaves are ready to accept requests.
- `slaveExecArguments`: Command line arguments passed to a slave.

- `ses`: Defines the optimization parameters of the Search Engine Server. This elements has the same subelements as `cm` .
- `slaveExecMaxFailures`: Maximum number of slave execution failures after which the master server terminates.
- `slaveIdleTimeout`: Time in seconds after a slave not processing a request is considered idle.
- `slaveShutdownTimeout`: Time in seconds a slave may take to terminate.
- `slaveStartupTimeout`: Time in seconds after which an executed slave must be ready.
- `te`: Defines the optimization parameters of the Template Engine. This element has the same subelements as `cm`.
- `maxSearchResultSize`:
  Limits the number of results the standard and advanced search (`obj where` and `obj search`, respectively) returns to the number specified.

- `minStreamingDataLength`: The size in KB a main content of a version or the contents of a resource needs to have for the application to send it to the Search Engine Server via the streaming interface. If this value is zero, streaming is not used and the data is inserted into the payload sent to the Search Engine Server instead. The Content Management Server also uses streaming when communicating with the HTML-GUI.
- `slave`: Defines parameters relating to the slave instances of the respective applications.

  - `masterTimeout`: Number of milliseconds a slave communicating with the master server waits for it to fully read the data. If the data could not be read in this period of time, the slave terminates assuming that the communication between itself and the master is disturbed. 0 (zero) means, that no timeout is used (the slave waits forever).
  - `maxNumberOfRequests`: The maximum number of requests a slave accepts.
  - `requestTimeout`: The time in seconds after which a slave running idle closes its connection to the client to be free for new requests.
- `streamingTicketValidityTimeInterval`: The value of this entry determines how long streaming tickets are valid. The value is given as an integer number which is interpreted as the period of time in seconds. A streaming which is ticket older than specified here is deleted.
- `userManProxy`: Settings of the user manager proxy.

  - `cacheAuthentication`: `true` (which is the default) enables the authentication cache. This value should be set to `false` with external user managers.

5.2.9 userManagement

In this section of the system configuration, which can be found in the instance-specific `config/userManagement.xml` file, the user management both of the editorial system and the live system can be configured. Furthermore the user preferences can be found here. The dictionary can have the following subentries:

- `editorial`: Determines the properties of the user management of the editorial system.

- `live`: Determines the properties of the user management of the live system.
- `preferences`: In this dictionary the presets of user-specific configuration values for the GUI are stored. The Content Management Server uses these values if a user has not configured his individual settings. Except for the values contained in the `guiPreferences` entry, the settings refer to the GUI from NPS 5.2 and later if not stated otherwise. The dictionary kann have the following entries:
  - `dateTimeInputFormatName`: Date input format. Example: `dayMonthYear`
  - `dateTimeOutputFormatName`: Format of combined date and time output. Example: `europeanShortDateTime`.
  - `defaultTemplate`: The name of the master template to be used for the export and the preview. Default: *mastertemplate*.
  - `displayTitleFormat`: The format in which the value of the `displayTitle` parameter is returned. The parameter is used by the HTML user interface to find the title of the most entities to be displayed in the Content Manager (fields, file formats, workflows, etc.). `%s` can be used up to two times as a place-holder in the format string. The first instance of `%s` is replaced with the title of the respective entity (e.g. the title of a workflow), the second instance with the primary key (e.g. the name of a workflow).
  - `guiPreferences`: User-specific presets for the Content Navigator. The subelements are described in the section *guiPreferences*.
  - `inputCharset`: The default character set of the documents the user imports via Tcl (selectable as *Standard* character set when importing via the GUI).
  - `languages`: the languages preferred by the user. Each language is represented by a `language` element. The first element determines the language of the HTML user interface. Example:

```
<languages type="list">
  <language>de</language>
  <language>en</language>
</languages>
```

  - `listDisplayRows`: The number of lines in search result lists and in the file hierarchy display.
  - `maxHierarchyDepth`: The maximum number of levels to be displayed in the file hierarchy.
  - `maxHierarchyLines`: The maximum number of lines to be displayed in the file hierarchy.
  - `preferEditedContents`: Determines whether draft versions (`YES`) or released versions (`NO`) of files displayed in the preview are to be preferred. If `YES` is selected, the released version is only used if a file neither has a draft nor a committed version. If `NO` is selected a file is treated as if it were not present, if no it does not have a released version. This setting also affects the *exportBlob* of a version.
  - `preferEditedTemplates`: The same as `preferEditedContents` but for layout files. The setting also affects the `visibleExportTemplates` of files.
  - `textAreaHeight`: The height of multi-line text input fields.
  - `textAreaWidth`: Width of multi-line text input fields.
  - `textFieldWidth`: Width of single-line text input fields.
  - `timeZone`: The user-specific time zone setting. The list of time zone names can be retrieved using the `systemConfig validTimeZoneNames` Tcl command. Example: `CET`
- `globalPermissions`: The value of this entry is a list of names of global permissions. Example:

```
<globalPermissions type="list">
  <globalPermission>permissionGlobalRoot</globalPermission>
  <globalPermission>permissionGlobalUserEdit</globalPermission>
  <globalPermission>permissionGlobalUserAttributeEdit</globalPermission>
  <globalPermission>permissionGlobalRTCEdit</globalPermission>
  <globalPermission>permissionGlobalExport</globalPermission>
  <globalPermission>permissionGlobalMirrorHandling</globalPermission>
</globalPermissions>
```

You can define here additional permissions that can be used in the Content Management Server to restrict the use of file formats to particular user groups.

5.2.10 guiPreferences

There are two types of user preferences: role and non-role preferences. The value of a role preference depends on the user's current role. The (fallback) values are defined for every role in the section `gui`. Their values can be overridden by the user's configuration if the specific preference is not marked as `fixed` in the role. See the section *Role Preferences* for details. Providing values for role preferences in the `guiPreferences` is optional.

Non-role preferences, on the other hand, are by definition role-independent. All non-role preferences must be present in the section `guiPreferences`. They must provide the default value.

`guiPreferences`: User-specific presets for the Content Navigator. In addition to the optional elements this element must have the following subelements:

- `browserStates`: Determines the properties of the menu area (menu and column view).
  - `cb` determines the properties of the column view.
  - `tb` refers to the hierarchical view.
  Both have `browserWidth` as a subelement, which defines the position of the horizontal split bar. `cb` additionally contains `numCols` (the number of columns). `tb` additionally contains `root` (first folder of the partial tree displayed) as well as `expandedNodes` (the nodes, subelement `node`) displayed in the columns. Example:

```
<browserStates>
  <cb>
    <browserWidth>60</browserWidth>
    <numCols>5</numCols>
  </cb>
  <tb>
    <browserWidth>40</browserWidth>
    <root />
    <expandedNodes />
  </tb>
</browserStates>
```

- `currentRole`: Name of the user interface to be used for new users. This name can be used to make part of the user interface accessible to particular users (and hide them from others). The

name must be one of the interface names defined in `roles`. By means of the `fixed="true"` attribute a user's access to the GUI can be restricted to one interface definition, meaning that she or he cannot switch to a different interface. Example:

```
userConfigForUser mueller setElements guiPreferences.currentRole \
"<currentRole fixed='true'>editor</currentRole>"
```

- `externalEditorEncoding`: The character set to which the versions of documents, folders and layouts are converted before they are transferred to a client for editing with a local application. The value must be an encoding name accepted by Java.
- `externalEditors`: The external editors a user has configured for himself. For each file name extension (content type) `externalEditors` has a subelement, whose name is the file name extension and whose contents is the command line with which the editor is executed. Example:

```
<externalEditors>
  <html>notepad.exe</html>
  <txt>notepad.exe</txt>
</externalEditors>
```

- `objClassHistory`: File formats the user recently used in the Content Navigator. Any number of file formats can be specified. However, the GUI offers only the first 5 file formats from this list to the user. Each element in this list is represented by an `objClass` element. Example:

```
<objClassHistory>
  <objClass>publication</objClass>
  <objClass>document</objClass>
</objClassHistory>
```

- `startArea`: The section of the HTML user interface that is displayed after the user has logged-in to the system. The following values can be specified:
  - `browse_page` (Content Navigator)
  - `wizard_page` (Wizard selection page)

  up to version 6.0.x:
  - `object_hierarchy` (hierarchy)
  - `object_search` (Search)
  - `object_tasks` (Tasks)
  - `configuration_objClasses` (Configuration - File formats)
  - `configuration_attributes` (Configuration - fields)
  - `configuration_workflows` (Configuration - Workflows)
  - `configuration_jobs` (Configuration - Jobs)
  - `configuration_channels` (Configuration - Channels)
  - `users_users` (User Management - Users)
  - `users_groups` (User Management - Groups)
  - `users_user_attributes` (User Management - User fields)

  - `preferences` (Personal preferences)
- `startingPublication`:The folder that is selected, after the user has logged-in.
- `theme`: The default theme name.
- `userMenu`: Bookmarks of new users.
- `userStyle`: Font and other display options.
- `wantWorkflowConfirmation`: Determines whether the confirmation page is to be displayed (`YES`) or not (`NO`) after workflow actions have been performed in the GUI. This setting does not apply to the *give* action.

5.2.11 rolePreferences

5.2.12 Priority

If an element has set the attribute `fixed="true"`, its value overrides the one chosen by the user (or the default value, if nothing was chosen yet). As long as such a value is in effect, the user is not allowed to change the according option.

5.2.13 Allowed Values

All of the following elements are allowed inside the `role` subelement named `values` in the section `gui`. In addition to this, all elements except `preferEditedContents`, `preferEditedTemplates`, and `defaultTemplate` can optionally be used inside `guiPreferences`.

- `browserName`: The name of the current browser (to be seen on the left-hand side of the Content Navigator). Permitted values are:
  - `cb` (column browser)
  - `db` (details browser)
  - `tb` (tree browser)
  - `lb` (list browser)
  - `pb` (preview browser)
  - `ib` (icon/thumbnail browser)

  Example:

```
<browserName>ib</browserName>
```

- `browserShowsDocuments`: Defines whether files of the document type are shown in the browser (folders are always shown). Permitted values are `true` and `false`.
- `browserShowsGenerics`: Defines whether resources are shown in the browser. Permitted values are `true` and `false`.
- `browserShowsImages`: Defines whether image files are shown in the browser. Permitted values are `true` and `false`.
- `browserShowsInactive`: Defines whether inactive files are shown in the browser. Permitted values are `true` and `false`. Example:

```
```

```
<browserShowsInactive fixed="true">false</browserShowsInactive>
```

- `browserShowsTemplates`: Defines whether templates are shown in the browser. Permitted values are `true` and `false`.
- `browserShowsTitles`: Defines whether the browser displays the titles instead of the names of files. Permitted values are `true` and `false`.
- `defaultTemplate`: See the explanation in the section userManagement.
- `preferEditedContents`: See the explanation in the section userManagement.
- `preferEditedTemplates`: See the explanation in the section userManagement.

6

6 Tcl Interface Functions

6.1 File Conversion Functions of the Content Management Server

The Content Management Server is supplied with HTML filter software with which files in many formats can be converted, including the images they contain, if desired.

For the Content Navigator, a conversion wizard is available with which the users can easily convert a resource file. This process generates additional files in NPS containing the content converted to HTML and, if applicable, also the images. For details on creating wizards see also section Wizards.

For converting files the wizards call procedures from the supplied Tcl library `share/script/cm/serverCmds/converterLib.tcl`. This library also contains interface routines that call the Verity filter application.

For converting an office document without subsequently generating files in the CMS, the procedure `nps::convertWithVerity` can be used in a Tcl Shell. Example:

```
nps::convertWithVerity aWord.doc msAdvert /tmp/Converted
```

This converts the document `aWord.doc` into the HTML file `/tmp/Konvertiert/msAdvert.html` and places the associated images (if images images existed in the document) in the same directory.

To convert an office document and store it in the CMS as a folder (including images in this folder, if applicable) the following procedure can be used:

```
msOfficeConverter::_convert resourceId
```

As `resourceId` the procedure expects the ID of a CMS file of the resource type. Of this file the main content of the draft version is converted. If the file does not have a draft version, it is created from the relesed version.

In the draft version of the file to be converted, the fields `conversionResults` and `lastConversion` need to exist. These fields have the following tasks:

Field	Task
`conversionResults`	This field is of the *Linklist* type. During conversion, links to additional files (usually links to images) are stored in this field. If a new main content is

	subsequently imported into the resource and the resource is converted once again, the files to which the links in the field point are deleted.
`lastConversion`	The field is of the date type. The conversion procedure stores the time of conversion in it.

The procedure returns the ID of the folder created.

6.2 System Procedures (Functions)

System procedures are called when the CMS performs particular actions. They have no write access to the content in the CMS or to the runtime configuration (file formats, fields, etc.). However, if system procedures have not been registered in the safe interpreter, they do have write access to files in the file system.

6.2.1 Post-Action Function (Including E-Mail Notification)

The post-action function makes it possible to react to file-specific actions such as the release. The action might, for example, trigger the execution of a third-party application or send a request..

Every time a file-specific action (creation, workflow action, deletion) has been performed in the Content Manager, it calls the procedure `notificationCmd`.

To this procedure, five arguments are passed:

- `logType`: The action type. It is one of:
  - `action_comment` (File was commented)
  - `action_create_subobj` (File was created in the folder)
  - `action_delete_subobj` (File was deleted from folder)
  - `action_admin_obj` (A file field was modified)
  - `action_edit_obj` (Editing process started)
  - `action_give_obj` (Draft version was given to someone)
  - `action_take_obj` (Draft version was taken)
  - `action_forward_obj` (Draft version was forwarded)
  - `action_commit_obj` (Draft version was committed)
  - `action_reject_obj` (Draft version was rejected)
  - `action_revert_obj` (Draft version was reverted)
  - `action_sign_obj` (Committed version was signed)
  - `action_release_obj` (Draft or committed version was released)
  - `action_unrelease_obj` (Released version was unreleased)
  - `action_deactivate` (Released version was deactivated, available from version 6.5.0)
- `objectId`: ID of the file with which the action was performed.
- `userLogin`: The login of the user who performed the action.
- `receiverLogin`: The login of the user or the name of the group who/which receives the file.
- `logText` : The comment of a workflow action

6.2.2 Standard Post Action Function

Up to version 6.7.0: The supplied post-action function calls the e-mail notification function (see below).

From version 6.7.1: The supplied post-action function calls all notifications that have been placed in the `script/cm/serverCmds/notifications` directories. The e-mail notification function can be found in the central `share` directory. Further custom post-action functions can be placed into the instance-specific directory *instanceName*`/script/cm/serverCmds/notifications`. The name of the script file must be *name*`Notification.tcl`, and the procedure *name*`Notification` must be defined in it. For further information about this mechanism please refer to the file `share/script/cm/serverCmds/notificationCmd.tcl`.

6.2.3 E-Mail Notification

The supplied E-mail notification sample sends messages for particular actions to the users responsible next in the workflow.

The e-mail notification will only work if the e-mail server to use has been specified and set-up properly.

Specifying the E-Mail Server

The E-Mail-Server can be specified in the `instance/default/config/server.xml` file. Enter the host name of a reachable SMPT mail server as the value of the `email.host` key. Example:

```
<email>
  <host>mail.my.domain</host>
</email>
```

Deactivating E-Mail notification

Up to version 6.7.0: The standard e-mail notification can be deactivated by providing an empty definition of `emailNotificationCmd`. For this, create the file `script/cm/serverCmds/notificationCmd.tcl` in the instance directory and place the following line into it:

```
proc emailNotificationCmd {args} {}
```

From version 6.7.1: Create the file `script/cm/serverCmds/notifications/emailNotification.tcl` in the instance directory and place the following line into it:

```
::notifications::remove emailNotification
```

6.2.4 Link Functions

The Content Manager and the Template Engine execute link functions when versions are exported or displayed in the integrated, the separate or the live preview. The functions give you the possibility to modify link tags and therefore allow *URL-Rewriting*. For each tag that may contain a URI (*Uniform Resource Identifier*), two functions are called, one for the opening and one for the closing tag.

Calling Tcl procedures during export can negatively influence the speed of the export.
The link functions are therefore disabled by default. They can be enabled by setting the

export.wantLinkCallback system configuration entry to YES. The functions are disabled again by setting the value to NO.

The names of the functions are linkCallback for the opening and linkCallbackCloseTag for the closing tag.

The linkCallback Procedure

The Content Manager and the Template Engine call the linkCallback procedure according to the following definition:

```
proc linkCallback {tagName srcId absSrcPath \
  expId absExpPath relExpPath attrList} {}
```

The arguments have the following meaning:

- tagName: the name of the link tag.
- srcId: the ID of the source file, i.e. of the file containing the link tag.
- absSrcPath: the absolute path of the source file.
- expId: the ID of the file to be exported.
- absExpPath: the absolute path of the file to be exported.
- relExpPath: the path of the file to be exported, relative to the source file.
- attrList: the list of attributes of the tag. Each attribute is itself a list with the following elements:
  - attrName: the name of the tag attribute.
  - attrValue: the value of the tag attribute.
  - linkId: the ID of the link.
  - destId: the ID of the target file.
  - absDestPath: the absolute path of the target file (only for URL attributes, otherwise the empty string).
  - relDestPath: the path of the target file, relative to the source file (only for URL attributes, otherwise the empty string).

Please note that an argument contains an empty string if the value does not exist. This is valid for, e.g. destId in external links. The procedure must return a list with the following elements:

- tagName: the possibly changed name of the link tag.
- attrList: the list of possibly changed attributes of the tag. Each attribute is itself a list with the following elements:
  - attrName: the name of the tag attribute.
  - attrValue: the value of the tag attribute.
  - reserved1: the value of this element is ignored.
  - reserved2: the value of this element is ignored.
  - reserved3: the value of this element is ignored.
  - reserved4: the value of this element is ignored.

Because the Content Manager and the Template Engine expect the elements reserved1 up to reserved4 in the subelements of attrlist, you can create the results list by assembling it from the first and last arguments of linkCallback.

Example (up to version 6.7.0)

The following example shows how to change the `http` prefix into an `https` prefix for external links.

```
proc linkCallback {tagName srcId absSrcPath expId absExpPath
  relExpPath attrList} {
  if { "$absExpPath" == "" || "$tagName" != "a" } {
    return [list $tagName $attrList]
  }
  set newList ""
  foreach attribute $attrList {
    set key [lindex $attribute 0]
    set value [lindex $attribute 1]
    if { "$key" == "href" || "$key" == "title" } {
      regsub "^http:" $value "https:" value
    }
    lappend newList [list $key $value {} {} {} {}]
  }
  return [list $tagName $newList]
}
proc linkCallbackCloseTag {tagName expId absExpPath} {
  return $tagName
}
```

From Infopark CMS Fiona 6.7.1: The link callback procedure supplied is able to call a custom procedure for every link tag to process. Such custom procedures can be registered easily (see the example below). This functionality is documented in the callback script file `share/script/cm/serverCmds/linkCallback.tcl`.

Example (from version 6.7.1)

```
::linkCallback::registerHandler "a" "addBlankTarget"

namespace eval ::linkCallback {
  proc addBlankTarget \
      {tagName srcId absSrcPath expId absExpPath relExpPath attrList} {
    if {[existsAttr href] && [string match "http*" [attrValue href]]} {
      addMissingAttrValue target _blank
      return [dataAsCallbackResult]
    }
    return [unmodified]
  }
}
```

The `linkCallbackCloseTag` Procedure

The `linkCallbackCloseTag` procedure is called up by the Content Manager and the Template Engine according to the following definition:

```
proc linkCallbackCloseTag {tagName expId absExpPath} {}
```

The arguments have the following meaning:

- `tagName`: the name of the link tag.
- `expId`: the ID of the file to be exported.
- `absExpPath`: the absolute path of the file to be exported.

Both CMS applications expect the possibly changed name of the link tag as a return value.

6.2.5 Thumbnail Function

Files of the types *image* (`image`) or *resource* (`generic`) are used to store images or any kind of binary data in the CMS. The binary data is stored in the version of the respective file as a blob (binary large object).

The thumbnail function offers you the possibility to create small preview images (thumbnails) from this content data. The HTML user interface of the Content Manager accesses these thumbnails via the `thumbnail` version field in order to support the user during file selection.

The Content Manager always calls the `generateThumbnail` Tcl function after data has been assigned to the draft version of an image file or a resource. Its function is to calculate a thumbnail from the binary data passed to it. The Content Manager includes a `generateThumbnail` function which fulfills this task with the aid of the `generateThumbnailForGif`, `generateThumbnailForJpeg`, and `generateThumbnailForPng` functions built into the Content Manager. The Content Manager stores the thumbnail in the `thumbnail` version field.

The `generateThumbnail` function is contained in the `generateThumbnail.tcl` script. This script is to be found in the `serverCmds` directory under the `config` directory of the Content Manager. The callback function can create thumbnails for versions of the `jpg`, `jpeg`, `gif`, and `png` types (capitalization is irrelevant). It can be extended if necessary to also generate thumbnails for data of other types.

The following arguments are passed to the `generateThumbnails` function in the order given below.

- `contentId`: the ID of the respective version.
- `objId`: the ID of the file to which the version with the `contentId` ID belongs.
- `blobFile`: the name of the file containing the source document from which the thumbnail is to be created.
- `contentType`: the file name extension of the version.
- `thumbnailSize`: the size of the thumbnail in pixels (same value for width and height). It can be configured using the `thumbnailSize` system configuration entry.

The function returns the base-64-encoded thumbnail image. The Content Manager stores this image as the value of the `thumbnail` field in the respective version.

6.2.6 Function for Checking Passwords

When a user password is set using the command

```
user withLogin login set password
```

or a new user is created specifying the password using

```
user create login newuser defaultGroup testers password herPassword
```

the Content Management Server executes the `passwordPolicyCheck` Tcl procedure. If a new user is created in the GUI, he can only login after a password has been given to him by means of the corresponding dialog or Tcl procedure.

In this procedure, administrators can implement the rules passwords must obey and reject inacceptable passwords using `error {error message}`. The error message is then displayed in the Tcl shell and in the GUI. The procedure is not executed for passwords that have already been assigned.

A sample implementation can be found in the file `share/script/cm/serverCmds/passwordPolicyCheck.tcl` which can be found in the CMS directory. It rejects passwords whose length is less than 4 characters:

```
proc passwordPolicyCheck {password} {
  if {[string length $password] < 4}  {
    error "Das Passwort ist zu kurz. / The password is too short."
  }
}
```

6.2.7 SystemExecute Procedures

The system-execute mechanism of the Content Manager and the Template Engine is used during export (and in the Content Manager also when files are previewed) to export file or version data which cannot be determined using NPSOBJ-insertvalue instructions. SystemExecute procedures are called as follows:

```
<npsobj insertvalue="systemExecute" name="procAlias">content</npsobj>
```

This instruction executes the procedure with the *procAlias* alias and inserts the return value into the export file instead of the instruction. SystemExecute alias names are registered in the `export` system configuration entry as the value of the `tclSystemExecuteCommands` key. The corresponding entry in the configuration file of the Content Manager or the Template Engine has the following format:

```
<tclSystemExecuteCommands>
  <procAlias>sysExecProc</procAlias>
</tclSystemExecuteCommands>
```

With the above configuration entry, the `name="procAlias"` part in the NPSOBJ instruction causes the `sysExecProc` procedure to be called. SystemExecute procedures need to be stored in the instance-specific `script/cm/serverCmds` directory and are sourced during the start-up process of the Content Manager and the Template Engine. Please note that the Tcl code of system procedures only has read-access to the CMS data.

The ID of the file or link in whose context the instruction is evaluated is passed as the first argument to a systemExecute procedure. As the second argument it receives the evaluated content of the calling NPSOBJ element. The following example procedure `sysExecProc` returns (as a result) a formatted text which contains both these values.

```
proc sysExecProc {objId content} {
  set result "<b>ObjId: $objId</b>\n"
  append result "<pre>$content</pre>\n"
  return $result;
}
```

File dependencies on the live server

SystemExecute procedures have read access not only to the file being exported but also to all other files. If you use the Template Engine and a referenced file was updated, then the file containing the

systemExecute call has possibly also become out-of-date and needs to be exported again. This file depends on the referenced file.

If, for example, during the export of file A a systemExecute procedure is called which reads one or more of file B's version fields and has them incorporated into A's export result, then modifying B's version fields should invalidate not only B's but also A's web page. This would cause A to be exported again when a web site visitor requests A's page. By defining a dependency A's web page can be kept up-to-date even if no changes are made to the file A itself.

Dependencies can be made known to the Template Engine by using the global `additionalDependencies` variable in the systemExecute procedure. In `additionalDependencies` you can store a list of value pairs. Each pair consists of a dependency type and a file ID. The following example expresses that the file which is currently beeing exported depends on the contents of the file with the path `/path/to/file`:

```
global additionalDependencies
set additionalDependencies [list content [obj withPath /path/to/file]]
```

The dependency type determines which kind of changes to the specified file cause the file dependent on it to be deleted from the cache and exported again. In the following overview the available dependency types and their modification conditions are listed:

Dependency type	Modification conditions
`children`	The number of subfiles in a folder, one of its subfiles itself or the sorting in the folder has changed.
`content`	The draft version of a file has changed.
`object`	A file field has changed.
`reference`	A field which is relevant for links (such as `title` or `destinationUrl`) has changed.
`usesAll`	Any file or its draft version has changed. With this type the file ID is irrelevant and will be ignored. Nevertheless it must be specified.

6.2.8 Formatter Procedures

Formatter procedures serve to reformat field values during the export.

Formatters can be called by the `insertvalue var` and `insertvalue dynamiclink` NPSOBJ instructions by specifying the `formatter` tag attribute. For further details and examples please refer to the documentation of these two NPSOBJ instructions.

6.3 The User Manager Interface

Very frequently, the Content Management Server (CM) needs to determine whether a particular user may access a CMS file in a particular way, or whether a user belongs to a particular user group. To resolve such issues, the Content Management Server calls appropriate external (i.e. not built-in) Tcl procedures. These procedures constitute the User Manager API. By changing the implementation of

the procedures part of the API, other user managers than the built-in one can be queried to have these questions answered.

Two user managers can be connected to the Content Management Server, one for the editorial system, and one for the live system. Since implementations for the built-in user manager (the default one) as well as for LDAP and ADS have been provided, it should be sufficient to configure the implementation to be used.

This can be done by means of the `userManagement` system configuration entry and its `editorial` and `live` subentries. Both refer to a configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<userManagement>
  <editorial fileName="um_ldap_nis.xml"/>
  <live fileName="um_none.xml"/>
  ...
</userManagement>
```

A configuration file such as `um_ldap_nis.xml` above specifies the interface module to be used:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <handler>ldap_nis</handler>
  <config>
    <-- Server settings -->
    <host>ldap.server</host>
    <!-- LDAP account with extensive read permissions for file queries
         (empty value: anonymous LDAP requests) -->
    <bindDn>cn=manager,dc=company,dc=de</bindDn>
    <bindPassword>password</bindPassword>
    <!-- The following server specifications are optional -->
    <port>389</port>
    <!--  LDAP protocol version (alternative: 2)  -->
    <protocolVersion>3</protocolVersion>
    <!--  connection type (default: not encrypted; further options: ssl, tls -->
    <secureConnection>tls</secureConnection>

    <-- Groups -->
    <groupSearchBase>ou=groups,dc=company,dc=de</groupSearchBase>
    <groupFilter>(objectclass=posixGroup)</groupFilter>
    <groupResolver>
      <name>simple</name>
      <properties>
        <dnFormat>cn=%s,ou=groups,dc=company,dc=de</dnFormat>
      </properties>
    </groupResolver>
    <!-- LDAP attributes providing the value of a usermanAPI key -->
    <groupAttributeMapping>
      <name>cn</name>
      <realName>description</name>
    </groupAttributeMapping>

    <-- The following specifications are not relevant for live user integration
        (we're dealing with the CM here which does not have live users) -->
    <-- User -->
    <userSearchBase>ou=people,dc=company,dc=de</userSearchBase>
    <userFilter>(objectclass=posixAccount)</userFilter>
    <userResolver>
      ...
    </userResolver>
    <userAttributeMapping>
      ...
    </userAttributeMapping>

    <-- Group membership -->
    <!-- From version 6.5.0: group membership attribute can be configured -->
```

```
    <groupToUserRelationAttribute>memberUid</groupToUserRelationAttribute>
    <!-- The existing relationAttribute either contains the user name (true) or his DN
 (false; default) -->
    <memberValueIsLogin>true</memberValueIsLogin>

    <!-- Granting global permissions. The resolver referenced by name is loaded
         from cm/serverCmds/userman/lib/ldap/resolver/perm/<name>.tcl geladen -->
    <globalPermissionResolver>
      <name>simple</name>
      <properties/>
    </globalPermissionResolver>

    <!-- Known superusers (a list of logins) -->
    <superUsers type="list">
      <login>root</login>
    </superUsers>
  </config>
</configuration>
```

The handler subelement references a connector Tcl script located in the instance-specific directory

```
/NPS/instance/instName/script/cm/serverCmds/userman/handler
```

or in the common directory

```
/NPS/share/script/cm/serverCmds/userman/handler
```

The task of the referenced script is to read the configuration parameters and to implement the required Tcl interface procedures. You can use the script files supplied as templates if you wish to create individual connectors. The corresponding interface is described in section The User Manager API.

The following connection variants are supplied with Fiona

- Data source: internal user management
- Dummy for no live user management (*none*)
- Data source: ADS
- Data source: LDAP in
  - two normal variants and
  - a simple approach

An external user manager needs to meet particular general requirements if it is to be attached to Infopark CMS Fiona. Please also note the requirements an LDAP server must meet.

6.3.1 Requirements for Connecting an External User Manager

For an external user manager to be able to return sensible data via the user manager interface, the following conditions must be fulfilled:

1. The user manager in use must be able to handle both individual users as well as user groups. It must be possible to allocate users to groups and to remove them from groups again. Users and groups must be able to possess attributes whose values must be able to be queried.
2. The user manager must be able to handle global permissions. This means that it must be able to allocate the name of a global permission it has been given to a user or user group and to be able to make this allocation public upon request. It does not need to handle global permissions itself since these belong to the responsibilities of the CMS applications.

3. The user manager must offer access functions which make it possible to implement the Tcl procedures described in the following section. Ideally, the user manager has a Tcl interface or the possibility to create such an interface.
4. The user manager almost exclusively decides which criteria it uses to judge whether a user is a superuser or not. The CMS application simply demands that all users who have the `permissionGlobalRoot` permission are identified as superusers. If the user manager cannot do this, the API procedure userWithLoginIsSuperUser can perform this task as a substitute.
5. The user manager should be able to offer defined and, most important, detailed error messages when operations partly or completely fail. The error messages must be able to be forwarded to the CMS application via the Tcl interface.

The CMS application is not intended to have write access to the user manager.

6.3.2 Particular Requirements for an LDAP Server

In order to connect a CMS application to a user manager based on LDAP, the LDAP server must only fulfill a small number of conditions:

1. First, two schemes must be defined which allow users and user groups to be identified as Content Manager users or Content Manager groups respectively. For example, there could be a scheme called `fionaUser` and one called `fionaGroup` to which particular LDAP entries have been allocated which are to represent the users and the groups (e.g. `inetOrgPerson` and *groupOfUniqueNames*).
2. An entry which represents a Content Manager user must possess an attribute (e.g. `uid`) which is suitable for defining a login. The login must be unique for all entries, i. e. the user must be identified uniquely by this field. Additionally, only values must be accepted into this field which satisfy the general criteria for logins.
3. An entry for a Content Manager user must possess an additional attribute which corresponds to the user password in the Content Manager. The value of this attribute must satisfy the general criteria for passwords. Passwords should be stored in unencrypted form or encrypted using the `crypt` method. Alternatively, base64-encoding can be used.
4. An entry which represents a Content Manager user group must possess an attribute which is suitable for defining a group name. The login must be unique for all entries, i. e. the group must be identified uniquely by this field.
5. It must be possible to allocate the user groups of the CMS application and users to each other. This means that either the user entries of the CMS application must possess attributes in which the groups to which they belong are stored or the groups must contain a corresponding attribute for the users who belong to them.
6. Additionally, both group entries and user entries should possess attributes in which the granted global permissions are stored. The permissions need simply be saved as strings since the user manager does not have to be able to administer the relevant permissions in the Content Manager. It must be possible to determine whether a user is a superuser or not. To do this, e.g. the users can be allocated the `superuser` attribute.

6.3.3 Integrating an LDAP Server

In order to connect an LDAP server to Infopark CMS Fiona, please first configure the required connectors by specifying their configuration file in the `userManagement.xml` configuration file. Make sure that the connector configuration file is complete and its contents is correct. Please also read the Notes on Configuring LDAP.

Then check and adapt the connector script used. It is located in the `share/script/cm/serverCmds/userman/handler/` directory.

The LDAP integration can now be tested, for example by calling the Tcl commands `listGroups` and `listSecondaryGroups` and checking their results.

If the group names of the connected servers are not returned, enable logging using the following command:

```
CM -single
% ::usermanager::type::handler::client::activateDebugging [level]
```

Here is an example:

```
% ::usermanager::live::ldap::client::activateDebugging
% ::usermanager::editorial::ldap::client::activateDebugging
```

Then execute the commands once more and check the logs again.

If no connection is established, please check the server names and name resolution as well as the ports and other parameters relevant to the network connection. If the users and groups returned or their access permissions are not what you expected, check whether the LDAP parameters set in the configuration file specified in the `userManagement.xml` file correspond to the configuration of the server (see also the next section).

Notes on Configuring LDAP

An LDAP user login (DN = *distinguished name*) can be described like in the following example:

```
uid="larry",ou=people,o=company.com
```

In the CMS system configuration this is specified using `uid` for `userIdColName` and `ou=people,o=company.com` for `userSearchBase`. Groups have the following format:

```
cn="admins",cn=groups,o=company.com
```

Thus, `groupIdColName` is the first occurrence of `cn`, and `groupSearchBase` is `groups`. The members of a group are defined using `uniqueMember` attributes, each of them specifying a single user-DN. This means that `uniqueMember` can occur more than once. From version 6.5.0, the attribute used for specifying the group membership of users can be configured by means of the `groupToUserRelationAttribute` entry in the LDAP configuration file (for example `um_ldap.xml`).

Some user and group parameters in the Content Manager correspond to attributes or entries in LDAP which are queried instead (for example, `realName` in the Content Manager is `cn` in LDAP). The value of the `users` parameter is queried using `uniqueMember`*. This is done in the Content Manager's interface to the user manager. In this interface and in the LDAP server configuration permissions and their checking can easily be implemented (by using additional fields). In the supplied version of the interface file no groups exists. The only user configured is the `root` user who has the state of a superuser and is thus granted all permissions.

For the editorial system the `internal` connector is used, `none` for the live server. `internal` causes the internal user manager of the Content Manager to be used while `none` is ignorant to user data and thus returns nothing. `none` must not be specified for the editorial system.

6.3.4 The User Manager API

6.4 The Concept

The User Manager API includes two sets of procedures. Via these procedure sets, the Content Management Server accesses the respective user manager for the editorial system and for the live system to do the following, for example:

- determine the groups and the users of the editorial system as well as their permissions,
- authenticate users of the editorial system by means of a password,
- determine the groups of the live system.

The names of the procedures for accessing the user manager for the live system contain the character string `secondary`

The procedure sets for the internal user manager and some commonly known user managers such as LDAP and ADS are supplied with Fiona and can be found in the `instance`*`instName`*`/script/cm/serverCmds/userman/handler` directory as Tcl files. The corresponding configuration files (in which the Tcl files are referenced) can be found in the `instance`*`instName`*`/config` directory.

In order to make use of an existing procedure set,

- adapt the parameters in the corresponding configuration file to the system requirements,
- integrate the configuration file into the system configuration and
- restart the Content Management Server.

Please also refer to the detailed description of the structure of the configuration files.

In order to create and use a new procedure set, you can copy an existing Tcl interface file including its configuration file, redefine the procedures, and adapt and integrate the configuration file.

6.5 Procedures of the User Manager API

checkLoginAndPassword login password

The function tests whether a user with the login name *`login`* has the password *`password`*. The Content Manager passes the unencrypted password to the function. If the user possesses this password, the function returns 1; if not, it returns 0. In case of an error, the return value is the error message.

groupsWhere whereParams
secondaryGroupsWhere whereParams

This procedure returns the list of the names of user groups which fulfill the search criteria (*`whereParams`*). Search criteria are given as name-value-pairs. The only search criterion available is `groupText`.

groupWithNameExists name
secondaryGroupWithNameExists name

The procedure tests whether a user group exists with the name *`name`* and returns 1 if this is the case and 0 when the group does not exist.

groupWithNameGet name key

secondaryGroupWithNameGet name key

The procedure reads the value of the *key* attribute allocated to the group *name* and returns it. An external user manager must at least ensure that valid values are returned if `name` or `realName` is specified as `key`. In the editorial system additionally `displayTitle` needs to return the group title to be displayed. In case of an error, the error message is returned. The Tcl routine should always forward error messages of the external user manager.

groupWithNameHasGlobalPerm name permission

The procedure tests whether the user group with the name *name* has the global permission `permission` and returns 1 if this is the case. If the group does not have the permission, the procedure returns 0; in case of error, the return value is the error message.

listGroups
listSecondaryGroups

This procedure has no parameters. It returns the list of group names. In case of error, it returns the error message.

listUsers

This procedure has no parameters. It returns the list of user names. In case of an error it returns the error message.

typeForGroupGetKey key
typeForSecondaryGroupGetKey key

This procedure returns the type of a `group` parameter (`list` or `string`).

typeForUserGetKey key

This procedure returns the type of a `user` parameter (`list` or `string`).

usersWhere whereParams

This procedure returns the list of user logins for which the search criteria (*whereParams*) are fulfilled. Search criteria are given as name-value-pairs. In the integrated user manager, the only search criterion available is `userText`.

userWithLoginExists login

The procedure tests whether a user with the login *login* exists and returns 1 if this is the case and 0 when the user does not exist.

userWithLoginGet login key

The procedure reads the value of the *key* attribute allocated to the user *login* and returns it. An external user manager must ensure that valid values are then supplied when `login`, `realName`, `email`, `groups`, `displayTitle`, or `defaultGroup` is specified as `key`. In case of an error, the error message is returned. The Tcl routine should always forward error messages of the external user manager.

userWithLoginHasGlobalPerm login permission

The procedure tests whether the user with the login *login* has the global permission *permission* and returns 1 if this is the case. If the user does not have the permission, the procedure returns 0; in case of error, the return value is the error message.

The integrated user manager determines whether a user has a permission by testing whether the permission was given directly to the user or whether the user is a member of a group with this permission.

The user manager's only task is to allocate the names of global permissions to users and user groups. The administration of permissions is left to the applications which access it.

userWithLoginIsOwnerOf login ownedLogin

The procedure checks whether the user with the *login* login is the administrator (owner) of the user with the *ownedLogin* login. If this is the case, the procedure returns 1, otherwise it returns 0. In case of an error, the error message is returned.

This query tests whether the *login* user is permitted to modify the preferences of the *ownedLogin* user. A simple implementation of this function merely returns the result of `userWithLoginIsSuperUser` *login*.

userWithLoginIsSuperUser login

The Content Manager passes the login *login* to this procedure. The procedure tests whether the user with the login is a superuser and returns 1 if this is the case, and 0 when the user is not a superuser. In case of error, the procedure returns the error message.

6.5.1 Parameters of the LDAP/ADS User Manager Configuration

The LDAP/ADS User Manager connected to the Content Management Server can be configured by means of the following parameters:

host / port

By means of these two parameters, the host name (or the IP address) and the port, respectively, on which the LDAP/ADS server is running can be specified. Example:

```
<host>ldap.server</host>
<port>389</port>
```

protocolVersion

This parameter defines the version of the LDAP server (2 or 3). Example:

```
<protocolVersion>3</protocolVersion>
```

secureConnection

The encryption type. One of the following values can be specified:

- `none`: no encryption
- `ssl`: SSL encryption
- `tls`: TLS encryption

Example:

```
<secureConnection>none</secureConnection>
```

bindDn / bindPassword

The credentials (DN and password) the CM uses to identify itself to the LDAP server. Example:

```
<bindDn>cn=administrator,cn=Users,dc=company,dc=de</bindDn>
<bindPassword>password</bindPassword>
```

userSearchBase

The top-level DN of the subtree in the LDAP directory hierarchy that contains the users. Example:

```
<userSearchBase>ou=people,dc=company,dc=de</userSearchBase>
```

groupSearchBase

Like `userSearchBase` but referring to groups. Example:

```
<groupSearchBase>ou=groups,dc=company,dc=de</groupSearchBase>
```

groupToUserRelationAttribute

The name of the LDAP attribute used to associate users with the groups of which they are members. Example:

```
<groupToUserRelationAttribute>uniqueMember</groupToUserRelationAttribute>
```

userFilter

Using this parameter the users returned by the LDAP user manager can be filtered (restricted). Example:

```
<userFilter>(objectclass=inetOrgPerson)</userFilter>
```

groupFilter

Analogous to `userFilter`, this parameter allows to filter the user groups. Example:

```
<groupFilter>(objectclass=groupOfUniqueNames)</groupFilter>
```

userResolver

Allows to specify the method used to generate the LDAP query for determining the user data. Two methods are included in the CMS, `simple` and `lookup`.

- **`simple`**:

  If this method is used, the following assumptions are made:

  - The DN of all users has the same format throughout the LDAP directory. The DN is composed from the login and the format specified in the `dnFormat` parameter.

- All logins in the LDAP directory can be identified uniquely by means of this `dnFormat`. The LDAP filter specified by means of the `userFilter` parameter is therefore ignored.
- All users can be found in the specified LDAP directory entry (`scope=base`)

Example:

```
<userResolver>
  <name>simple</name>
  <properties>
    <dnFormat>uid=%s,dc=company,dc=de</dnFormat>
  </properties>
</userResolver>
```

In this format string, `%s` is replaced with the login to yield the DN which is then placed into the LDAP query. Assuming that the login is `maria`, the result would be:

```
uid=maria,dc=company,dc=de (objectclass=*) ... -scope base
```

- **`lookup`**:

  This is the default value of the userResolver. In this mode it is assumed that the location of the DN in the LDAP directory tree may be different from user to user.

  - In addition to the filters specified `userFilter` another filter is added which reads the login from the attribute specified as `idAttribute`. Thus, it is required to define the LDAP login attribute by means of `idAttribute`.
  - The LDAP query will be generated so that all of the LDAP tree is searched (`scope=subtree`).

  Example:

```
<userResolver>
  <name>lookup</name>
  <properties>
    <idAttribute>uid</idAttribute>
  </properties>
</userResolver>
```

  With `(objectclass=inetOrgPerson)` as the `userFilter` and `dc=company,dc=de` as the `userSearchBase`, the following LDAP query is generated for the `hermann` user login:

```
dc=company,dc=de (&(objectclass=inetOrgPerson)(uid=hermann)) ... -scope subtree
```

groupResolver

This parameter refers to groups and functions analogously to `userResolver`. Example:

```
<groupResolver>
  <name>lookup</name>
  <properties>
```

```
    <idAttribute>cn</idAttribute>
  </properties>
</groupResolver>
```

userAttributeMapping

Using this parameter, user fields in the CMS can be mapped to the corresponding LDAP attriibutes. A mapping for `login` should always exist. Example:

```
<userAttributeMapping>
  <login>uid</login>
  <realName>description</realName>
</userAttributeMapping>
```

groupAttributeMapping

The `groupAttributeMapping` parameter refers to groups and works analogously to `userAttributeMapping`. Example:

```
<groupAttributeMapping>
  <name>cn</name>
  <realName>description</realName>
</groupAttributeMapping>
```

globalPermissionResolver

This parameter specifies the method with which the global permissions are determined. See Mapping Global Permissions to Groups.

```
<globalPermissionResolver>
  <name>group</name>
  <properties>
      <groupNameFormat>admins_%s</groupNameFormat>
  </properties>
</globalPermissionResolver>
```

superUsers

This parameter serves to specify a list of users permitted to act as `superUser` in the CMS. Example:

```
<superUsers type="list">
  <login>root</login>
</superUsers>
```

memberValueIsLogin

This parameter specifies whether users-to-group assignments via the `groupToUserRelationAttribute` are made by means of the respective user's login or not. The default value is `false`. Examples:

- `member`: `uid=tester,dc=company,dc=de`
  `memberValueIsLogin`: `false`.
  This has the effect that the value of the `member` attribute is assumed to contain the DN and not just the login.
- `member`: `tester`
  `memberValueIsLogin`: `true` because the login is be used to determine group membership.

Example:

```
<memberValueIsLogin>false</memberValueIsLogin>
```

7

7 Configuring Functions and the Appearance of the Content Navigator

The menu items and the toolbar of the Content Navigator can be configured in order to adapt them to individual requirements. For example, menu items can be removed to avoid maloperation by unexperienced users. Reversely, additional menu commands can be placed in the menu to extend the functions of the GUI, for example by wizards.

Two possibilities for configuring the menu and the toolbar exist, the standard configuration and an additional configuration. The standard configuration is part of the GUI web application, while the additional configuration is part of the content, meaning that it is also saved and restored when content is dumped and restored. The additional configuration is a means of providing content-specific menu commands, i. e. menu commands that work only in conjunction with this content. For example, Infopark's demo content includes a wizard that relies on the existence of particular file formats. Placing the command for running this wizard into the standard configuration would not be sensible since the required formats are not present in the standard configuration. The items contained in the additional configuration extend the standard configuration or replace items of the same name.

The menus and the toolbar are configured by means of XML files. The standard configuration is located in the `webapps/GUI/WEB-INF` directory, the additional configuration in the instance-specific `config` directory, i. e. in `instance/default/config`, for example.

Both configurations (the standard and the additional configuration) consist of several parts described below:

Configuration	Comment
Standard configuration	

Menu
`itemRegistry.xml`
`menuBar.xml`
	Defines items for the menu and the toolbar
	Forms menus by combining menu items

Toolbar `toolBar.xml`	Menu items in the toolbar

Details views
`inspectorRegistry.xml`
`inspectors/*.vm`
	Defines the contents of the details view
	Velocity templates for displaying the contents

Search `searchRegistry.xml`	Defines the components of the search dialog
Additional configuration	

Menu / toolbar `contentMenu.xml`	Contains definitions as XML elements, analogous to `itemRegistry.xml`, `menubar.xml`, `toolBar.xml`
Detail views `contentInspectors.xml`	Contains definitions as XML elements, analogous to `inspectorRegistry.xml` and `inspectors/*.vm`

In the additional configuration, two parts that are separate in the standard configuration are combined: `contentMenu.xml` contains `<beans>` (as in the `itemRegistry.xml` file), `<menuBar>` (as in the `menuBar.xml` file), and `<toolBar>` (as in the `toolBar.xml` file). `contentInspectors.xml` contains `<beans>` defining the details overview as well as the detailed property views on their tabs; `<templates>` contains the Velocity templates as CDATA sections, i. e. not only links to them as in `inspectorRegistry.xml`.

7.1 Defining Menu Entries

All menu and toolbar items of the standard configuration are registered in the `itemRegistry.xml` file. It is recommended to not edit the `itemRegistry.xml` file since it is replaced during updates. Edit the file `contentMenu.xml` instead.

If the file is changed anyway, additional menu commands need to be of a general nature, meaning that they must only refer to file formats, fields etc. present in the standard configuration. As with other menu commands, the items can be referenced in the menu (`menuBar.xml`) or in the toolbar (`toolBar.xml`) and thus be integrated into the Content Navigator.

Every item in the `beans` element of `itemRegistry.xml` and `contentMenu.xml` is a `bean`. All `bean` elements have the same structure:

```
<beans>
  <bean id="..." class="...">
    <property name="...">...</property>
    ...
  </bean>
  ...
</beans>
```

The identifier specified as the `id` must be unique, as `class` a Java class known to the system is specified.

For details on defining additional menu commands see section Configuring Custom Commands.

If such commands depend on the existence of a particular folder hierarchy, of file formats, fields and the like, they need to be defined in the file `contentMenu.xml`. This file is part of the content and can be found in the instance-specific `config` directory.

7.2 Defining the Menu and the Toolbar

In the `menuBar.xml` and `toolBar.xml` files, the menu and the toolbar of the Content Navigator are composed from individual menu items (beans). The items available are listed in the `itemRegistry.xml` file. Other content-specific menu items and buttons on the toolbar can be defined in the `contentMenu.xml` file (see Configuring Custom Commands). Take a look at this example from the `menuBar.xml` file:

```
<menu name="file">
  <title lang="en">File</title>
  <title lang="de">Datei</title>
  <group name="object" contexts="object, select.object">
    <group name="new">
      <title lang="en">New</title>
      <title lang="de">Neu</title>
      <title lang="fr">Nouveau</title>
      <item>create_object_document</item>
      <item>create_object_publication</item>
      <item>create_object_image</item>
      <item>create_object_generic</item>
      <item roles="admin">create_object_template</item>
    </group>
    <item separated="true">import_file</item>
  </group>
  <item contexts="bookmark">create_bookmark_folder</item>
  <group separated="true" name="search" contexts="*.object">
    <item contexts="object, select.object">search_objects</item>
    <!-- insert saved searches here-->
  </group>
  <item contexts="object" separated="true">logout</item>
</menu>
```

The definitions of the items on the toolbar is specified analogously. However, it does not include a hierarchy (`submenu`) because the toolbar has no subitems:

```
<toolBar>
  <group name="object" contexts="object">
    <item>create_object_document</item>
    <item>create_object_publication</item>
    <item roles="admin">import_file</item>
    <item separated="true">finish_tasks</item>
    <item separated="true">search_panel</item>
  </group>
  <group name="edit" separated="true">
    <item>cut_into_clipboard</item>
    <item>copy_into_clipboard</item>
    <item contexts="object">paste_objects</item>
    <item contexts="bookmark">paste_bookmarks</item>
    <item contexts="object">delete_object</item>
    <item contexts="bookmark">delete_bookmarks</item>
  </group>
  <group name="view" separated="true" roles="admin">
    <item contexts="object">browser_preview</item>
    <item>browser_tree</item>
    <item>browser_columns</item>
    <item>browser_list</item>
    <item contexts="object">browser_icons</item>
    <separator/>
    <item contexts="object">preview</item>
    <item>view_mode</item>
    <item>refresh</item>
  </group>
  <separator/>
  <item>online_help</item>
  <item>logout</item>
</toolBar>
```

Each item in a menu or in a toolbar refers to a `bean` in the `beans` element of `itemRegistry.xml`. For example

```
<item>about_nps</item>
```

refers to the following item in `itemRegistry.xml` and includes it in the toolbar:

```
<bean id="about_nps"
    class="com.infopark.cm.htmlgui.browse.menuaction.OpenSubpage">
  <property name="titles"><map>
    <entry key="de"><value>Über Infopark CMS Fiona</value></entry>
    <entry key="en"><value>About Infopark CMS Fiona</value></entry></map>
  </property>
  <property name="subpage"><value>AboutNPS</value></property>
  <property name="selectionType"><value>none</value></property>
  <property name="dialogSize"><value>large</value></property>
</bean>
```

The `group` element gives you the possibility to group elements together without making this grouping visible in the menu or toolbar. If the group contains a dividing line (`separator`), and none of the items it contains (see below) can be selected (due to restrictions), the group will be completely hidden, i. e. including the separator. This helps to avoid two adjacent separators (without menu items between them).

Furthermore, the `group` element can be used to restrict the visibility of the items it contains to particular Content Navigator pages. This can be achieved by means of the `contexts` attribute that can have the following values:

- `object`: the items in this group are visible if a file has been selected in the main window of the Content Navigator.
- `bookmark`: the elements in this group are visible if bookmarks are being edited in the Content Navigator.
- `select.object`: the elements in the group are visible if a file is selected using the file selection dialog in the Content Navigator.
- `search.object`: the elements in the group are visible if a file is being searched for on the search page.

Using the `roles` attribute, groups of menu commands as well as individual menu commands can be restricted to particular user interfaces. See also the `gui.roles` system configuration entry in the `gui.xml` file.

The properties of beans of the class `com.infopark.cm.htmlgui.browse.menuaction.OpenSubpage` have the following meaning.

- `titles` determines the multi-language titles of dialog pages.
- `subpage` determines the names of a dialog page.
- `selectionType` determines the selection of files to which a menu command can be applied: `none` = globally (no file), `single` = a single file, `extended` = a list of files.
- `dialogSize` determines the size of the dialog window. Permitted values are `small`, `medium`, and `large`.

7.3 Configuring the Details View

The details section as well as the tabs that show up after clicking on *Details* can be configured, just like their contents.

7.3.1 Tabbed Details Overview

Starting with Infopark CMS Fiona 6.7.1, the details overview can have tabs to save space when displaying field sets.

Please include the following bean configuration in the `config/contentInspectors.xml` file to activate this feature system-wide:

```
<beans>
    <!-- tabbed overview inspectors. -->
    <bean id="publication-overview" parent="abstract-overview">
        <property name="template" value="tabbed-overview.vm" />
    </bean>
    <bean id="document-overview" parent="abstract-overview">
        <property name="template" value="tabbed-overview.vm" />
    </bean>
    <bean id="image-overview" parent="abstract-overview">
        <property name="template" value="tabbed-overview.vm" />
    </bean>
    <bean id="generic-overview" parent="abstract-overview">
        <property name="template" value="tabbed-overview.vm" />
    </bean>
    <bean id="template-overview" parent="abstract-overview">
        <property name="template" value="tabbed-overview.vm" />
    </bean>
</beans>
```

In addition to the *File info* tab, each field set whose name begins with `overview_` is displayed on a separate tab. The field sets can be configured individually for each file format.

The following script illustrates how the field sets `overview_en` and `overview_de` can be created for the `publication` file format using Tcl:

```
attributeGroup create identifier publication.overview_en title English
attributeGroup withIdentifier publication.overview_en addAttributes \
        titleEn summaryEn bodyEn

attributeGroup create identifier publication.overview_de title Deutsch
attributeGroup withIdentifier publication.overview_de addAttributes \
        titleDe summaryDe bodyDe
```

7.3.2 Adding Fields to the Details Overview Easily

The details overview on the right hand side of the Content Navigator is generated by Velocity templates. For each of the five file types that exist in Infopark CMS Fiona such a template is available.

In the details overview you can have more fields than the default ones displayed. To achieve this, call a macro, `attributeGroupInOverview` (supplied with the CMS), from within one or more of the five templates. This macro adds all the fields of the specified field group (here called "overview") to the details overview concerned:

```
#attributeGroupInOverview($content.getAttributeGroup('overview'))
```

The macro terminates silently if the specified field group does not exist, meaning that it is safe to use it. It can be found, like other macros, in the file `webapps/GUI/WEB-INF/inspectors/default_macros.vm`.

After this macro has been added to all *objType*`-overview.vm` files, anyone permitted to modify file formats in the CMS can add an `overview` field group to any file format. Then add the fields to be displayed in the extended details overview to the `overview` field group.

To prevent your changes from being overwritten when migrating the CMS to another version, place the contents of the template file directly into the `contentInspectors.xml` file:

```
<templates>
   <template name="publication-overview.vm"><![CDATA[
      ... previous content of publication-overview.vm ...
      #attributeGroupInOverview($content.getAttributeGroup('overview'))
   ]]></template>
</templates>
```

However, if you always wish to use the latest versions of the `*.vm` files, you should not only modify the templates but also overwrite the corresponding *objType*`-overview` beans. Specifying individual templates here gives you the possibilty to include the supplied template (using `parse`) as well as call the macro that adds the field group to the details overview.

If this procedure is to be applied to the overviews of several file types, and the titles of these overviews are identical, it is more efficient to join the multi-language titles of these views into an individual bean. This bean can then be referenced as `parent` in the view beans themselves. The following example demonstrates this for the `publication` details overview.

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

<contentInspectors>
  <beans>
    <bean id="overview-titles"
            class="com.infopark.cm.htmlgui.browse.inspector.View" abstract="true">
      <property name="titles"><map>
        <entry key="de" value="Eigenschaften"/>
        <entry key="en" value="Overview"/>
        <entry key="fr" value="Propriétés"/>
        <entry key="it" value="Proprietà"/>
        <entry key="es" value="Propiedades"/>
      </map></property>
    </bean>
    <bean id="publication-overview" parent="overview-titles">
      <property name="template" value="advanced_publication-overview.vm"/>
    </bean>
  </beans>
  <templates>
    <template name="advanced_publication-overview.vm"><![CDATA[
      #parse("publication-overview.vm")
      #attributeGroupInOverview($content.getAttributeGroup("overview"))
    ]]></template>
  </templates>
</contentInspectors>
```

7.3.3 The Configuration Files for the Details Views

The settings for the details views can be found in the following file:

Up to version 6.0.x: `webapps/GUI/WEB-INF/config/inspectorRegistry.xml`
From version 6.5.0: `webapps/GUI/WEB-INF/inspectorRegistry.xml`

The files are based on the Spring framework.

The contents of the individual details views (internally called inspectors) are displayed by means of Velocity templates that can be found in the *inspectors* directory below the directory mentioned above. If desired, the supplied Velocity templates can be modified or extended. For this the keywords listed in section Velocity Templates can be used.

In the file `inspectorRegistry.xml` a details view is called an `inspector`. What an inspector displayed is controlled by restrictions. An inspector can have several views.

```
<bean id="publication-inspector"
  class="com.infopark.cm.htmlgui.browse.inspector.Inspector">
<!-- restrictions:-->
  <property name="objType">
    <value>publication</value>
  </property>
<!-- views:-->
  <property name="overview">
    <ref bean="publication-overview"/>
  </property>
  <property name="details">
    <list>
      <ref bean="publication-details" />
      <ref bean="links" />
      <ref bean="publication-childData" />
      <ref bean="sourceCode" />
      <ref bean="publication-administration" />
      <ref bean="help" />
    </list>
  </property>
</bean>
```

In this example the inspector named `publication-inspector` is used if the type of the currently selected file is `publication`.

The following restrictions exist:

```
<property name="objType"><value>publication</value></property>
<property name="objClasses"><set>
  <value>referrer</value>
  <value>press</value></set>
</property>
<property name="roles"><set>
  <value>admin</value>
  <value>restricted</value></set>
</property>
```

The criteria available for deciding whether an inspector is used for displaying the properties of a file are file types, file formats and roles. However, it is not required to define restrictions. This is the case with the default inspector:

```
<bean id="default"
  class="com.infopark.cm.htmlgui.browse.inspector.Inspector">
<!-- no restrictions-->
<!-- views:-->
<property name="overview">
  <ref bean="default-overview"/></property>
</bean>
```

Restrictions for file formats and file types are mutually exclusive because the file format includes the definition of the file type.

The order of the inspectors in the configuration file is meaningless. Instead the inspectors are examined according to the following criteria. The first inspector that meets a criterion is selected and the search process is terminated.

- The inspectors in which file format restrictions are defined are examined first.
- The inspectors defining file type restrictions are examined next.
- The inspectors defining no restrictions are examined.
- The inspector with the ID default is searched for.
- No inspector is used and an error message is written into the log.

If ambiguous inspector configurations exist, a corresponding message is written to the log.

For each inspector at least one view, `overview`, must be configured:

```
<property name="overview">
  <ref bean="default-overview"/>
</property>
```

Optionally, additional details views can be configured. In an inspector configuration references to such views can be made:

```
<property name="details">
  <list>
    <ref bean="publication-details" />
    <ref bean="links" />
    <ref bean="publication-childData" />
    <ref bean="sourceCode" />
    <ref bean="publication-administration" />
    <ref bean="help" />
  </list>
</property>
```

The order of the references to views determines the order of the tabs displayed for them in the Content Navigator. Of course, the referenced views must be present, i. e. for each of them a view configuration must exist:

```
<bean id="publication-overview"
    class="com.infopark.cm.htmlgui.browse.inspector.View">
  <property name="titles">
    <map>
      <entry key="de"><value>Überblick</value></entry>
      <entry key="en"><value>Overview</value></entry>
    </map>
  </property>
  <property name="template">
    <value>publication-overview.vm</value>
  </property>
</bean>
<bean id="publication-details"
    class="com.infopark.cm.htmlgui.browse.inspector.View">
  <property name="titles">
    <map>
      <entry key="de"><value>Details</value></entry>
      <entry key="en"><value>Details</value></entry>
    </map>
  </property>
```

```
  <property name="template">
    <value>publication-details.vm</value>
  </property>
</bean>
[etc.]
```

A view configuration determines exactly one Velocity template. The referenced template files are searched for in the `webapps/GUI/WEB-INF/config/inspectors` directory.

7.4 Velocity Templates

Infopark CMS Fiona uses Velocity templates (file name extension `.vm`) to display the views of the Content Navigator. Velocity is an interpreter that can be used in servlet environments to output content by means of templates (see http://velocity.apache.org/). Administrators can write and adapt templates to have their own sets of CMS file fields displayed, and to change the appearance of these sets.

Along with the templates for displaying the details, the GUI always loads the template `macros.vm`. You can put your own macros into this file. However, when migrating to a new Fiona version, this file is not automatically copied to the new location. Therefore, it is safer to maintain macros in the `contentInspectors.xml` file located in the `config` directory of the instance concerned. An example:

```
<contentInspectors>
  ...
  <templates>
    <template name="macros.vm">
      <![CDATA[
        ## my macros
      ]]>
    </template>
  </templates>
  ...
</contentInspectors>
```

By means of the keywords below you can determine values from various contexts. Keywords that refer to the CMS entities file, version, link, field und field group can also be queried via Tcl and are explained in the Tcl reference.

7.4.1 Keywords in the context of the current CMS file

Keyword	Type	Meaning
`content`	`ContentTool`	Returns `editedContent` or `releasedContent`, depending on the user's preference and the availability of these versions
`editedContent`	`ContentTool`	The draft version of the file if it exists
`icons`	`String`	Scheme-based determination of icon URLs. Example: `<img src="$icons.get("publication.png")"/>`

lang	String	The language identifier of the user: "de", "en", ... Use an if-clause to localize your own strings: #if ($lang == "de") Hallo #elseif ($lang == "en") Hello #else Hi #end
localizer	String	Used internally to determine strings localized in the user's language
object	ObjectTool	The file currently to be displayed
releasedContent	ContentTool	The released version of the file, if it exists
role	String	The user interface assigned to or selected by the user. Query this key to hide or display fields depending on the user interface.

7.4.2 Keywords in the ObjectTool ($object.key)

Keyword	Type	Meaning
hasMirrors	AttributeTool	From version 6.5
id	AttributeTool	
isDeactivated	AttributeTool	From version 6.5
isMirror	AttributeTool	From version 6.5
mirrors	LinkListTool	From version 6.5
name	AttributeTool	
original	AttributeTool	for mirror files the path of the original file; from version 6.5
path	AttributeTool	
reminder	AttributeTool	Reminder; from version 6.5
type	AttributeTool	
visiblePath	AttributeTool	
objClass	AttributeTool	
isExportSuppressed	AttributeTool	
isExportable	AttributeTool	
isEdited	AttributeTool	
isReleased	AttributeTool	
editedContent	ContentTool	
releasedContent	ContentTool	
content	ContentTool	Version displayed according to the current display mode

`workflow`	`AttributeTool`	
`version`	`AttributeTool`	
`superlinks`	`LinkListTool`	
`state`	`AttributeTool`	State of the file depending on its existing versions (`released edited committed`)
`hasPermissionRead`	`AttributeTool`	
`hasPermissionWrite`	`AttributeTool`	
`hasPermissionRoot`	`AttributeTool`	
`hasPermissionCreateChildren`	`AttributeTool`	
`permissionRead`	`AttributeTool`	Outputs group names
`permissionCreateChildren`	`AttributeTool`	Outputs group names
`permissionLiveServerRead`	`AttributeTool`	Outputs group names
`permissionRoot`	`AttributeTool`	Outputs group names
`permissionWrite`	`AttributeTool`	Outputs group names

7.4.3 Keywords in the ContentTool

Keyword	Type	Meaning
`title`	`AttributeTool`	
`contentType`	`AttributeTool`	
`validFrom`	`AttributeTool`	
`validUntil`	`AttributeTool`	
`lastModified`	`AttributeTool`	
`lastChanged`	`AttributeTool`	Synonym for `lastModified`
`thumbnail`	`AttributeTool`	img-Link to the thumbnail
`preview`	`AttributeTool`	Inline preview for images, otherwise a-href link to the preview page
`getAttributeGroup(`*`name`*`)`	`AttributeGroupTool`	
`attributeGroups`	List of `AttributeGroupTool`	
`sourceCode`	`AttributeTool`	Source view

`blobLength`	`AttributeTool`	
`width`	`AttributeTool`	
`height`	`AttributeTool`	
`bodyTemplateName`	`AttributeTool`	
`mimeType`	`AttributeTool`	
`isComplete`	`AttributeTool`	
`reasonsForIncompleteState`	`AttributeTool`	
`channels`	`AttributeTool`	
`textLinks`	`LinkListTool`	Links in the main content
`body`	`AttributeTool`	The main content
`blob`	`AttributeTool`	Synonym for main content
`sortOrder`	`SortOrderTool`	
Names of custom version fields	`AttributeTool`	

7.4.4 Keywords in the AttributeGroupTool

Keyword	Type	Meaning
`editButton`	`String`	HTML code for edit button
`editMarkupOpen`	`String`	Opening HTML code for edit marker
`editMarkupClose`	`String`	Closing HTML code for edit marker
`title`	`String`	Localized title of the field group
`name`	`String`	Name of the field group
`identifier`	`String`	Identifier of the field group
`isDefaultGroup`	`boolean`	
`isEmpty`	`boolean`	
`attributes`	List of `AttributeTool`s	

7.4.5 Keywords in the SortOrderTool

Keyword	Type	Meaning
`editButton`	`String`	HTML code for edit button
`editMarkupOpen`	`String`	Opening HTML code for edit marker
`editMarkupClose`	`String`	Closing HTML code for edit marker

label	String	Localized label Sorting
getLabel(i)	String	Sort criterion with the index 1 <= i <= 3
getType(i)	AttributeTool	The type of the sort criterion with the index 1 <= i <= 3
getKey(i)	AttributeTool	The sort kriterion field with the index 1 <= i <= 3
getKeyLength(i)	AttributeTool	The length of the sort criterion field with the index 1 <= i <= 3
direction(i)	AttributeTool	Sort direction ascending or descending

7.4.6 Keywords in the AttributeTool

Keyword	Type	Meaning
editButton	String	HTML code for edit button
editMarkupOpen	String	Opening HTML code for edit marker
editMarkupClose	String	Closing HTML code for edit marker
displayValue	String	Formatted display value
label	String	Localized label
placeholderForEmpty	String	The value which is output instead of the empty value
type	String	date, enum, multienum, html, generic, image, linklist, signature, string, text, boolean, integer, href
isMandatory	boolean	1 for obligatory fields, otherwise 0

7.4.7 Keywords in the LinkListTool

Keyword	Type	Meaning
editMarkupOpen	String	Opening HTML code for edit marker
editMarkupClose	String	Closing HTML code for edit marker
editButton	String	HTML code for edit button
label	String	Localized label

`isEmptyAndEditingAllowed`	`boolean`	true, if the link list is empty and can be edited (this is not true for textlinks and superlinks)
`placeholderForEmpty`	`String`	The value displayed instead of the empty value
`type`	`String`	Always returns `linklist`
`isMandatory`	`boolean`	1 for obligatory fields, otherwise 0
`links`	List of `LinkTool`	The links in the link list

7.4.8 Keywords in the LinkTool

Keyword	Type	Meaning
`iconMarkup`	`String`	HTML code that includes an icon
`isComplete`	`boolean`	Specifies whether the links is complete
`displayUri`	`String`	The URI of the link, quoted as HTML text
`title`	`String`	The title of the link, can be zero
`followLinkMarkup`	`String`	HTML code for a link navigating to the link target when followed, can be zero
`sourceObject`	`ObjectTool`	Source of the link
`targetObject`	`ObjectTool`	Destination of the link

7.5 Configuring Themes

The appearance of the Content Navigator can be configured on a per-user basis. The settings required for this are accessible via the Content Navigator itself. Next to the font family and font size as well as individual color settings, several themes exist of which one can be chosen. A theme is a predefined set of colors, fonts, background images, and icons.

Each theme is located in an individual directory below the `webapps/GUI/NPS/themes` directory of the instance concerned and consists of image files plus a parameter file named `index.xml`. In the `default` theme, which has the title *Sky (Default)*, the parameters contained in this file are commented so that their meaning can be easily unterstood.

A new theme can be created by making a copy of a theme directory first. Then the copied images and the copied parameter file can be modified as desired. The new theme directory only needs to contain the images different from the default theme. This applies analogously to the parameters in the new `index.xml` file.

7.6 Authentication

From version 6.5.0, Infopark CMS Fiona supports three mechanisms for authenticating users of the editorial system.

1. Login form (the standard setting)
2. Basic authentication
3. NTLM for single sign on

For the standard setting no further configuration is required.

If you wish to use Basic Authentication or NTLM, please adapt the `instance/default/webapps/GUI/WEB-INF/acegi.xml` file. This file is a bean configuration file in the spring format.

Basic Authentication can be switched on by setting the `filterInvocationDefinitionSource` property of the `filterChainProxy` bean to the provided value, which is commented out by default.

For using NTLM, please adapt the `ntlmAuthentication` bean to your ADS configuration, in addition to specifying `filterChainProxy`. In the `instance/default/webapps/GUI/WEB-INF/web.xml` file, the `NtlmFilter` and its `filter-mapping` needs to be configured and commented-in.

If an ADS server is used for authentication, upper/lower case conflicts may arise with user names (ADS ignores case, the CMS does not). Therefore, as a preventive measure, you should set the `loginTransformer` property with ADS servers. By default, this setting, which is available from version 6.6, is not set.

To set the property in order to have user names automatically converted to lower case when logging in, please edit the GUI file `acegi.xml` located in the `webapps/GUI/WEB-INF` directory of your instance. Remove the comment characters from the `loginTransformer` property of the `abstractAuthenticationProvider` bean:

```
<bean id="abstractAuthenticationProvider" abstract="true">
  <property name="loginTransformer">
    <bean class="com.infopark.libs.util.LowercaseTransformer"/>
  </property>
</bean>
```

The GUI needs to be redeployed for the changes to take effect.

8 Configuring Custom Commands

The Content Navigator can be extended by additional menu commands (custom commands). Such a command consists of two parts: the program code that implements the functionality of the command plus a menu item in a configuration file. The menu item serves as the interface between the user (i. e. the Content Navigator) and the program code and is integrated into the menu (and if desired into the toolbar) by means of another configuration file. What the relevant files are and how they are related to each other is described in section Configuring Functions and the Appearance of the Content Navigator.

8.1 Definition of a Menu Command

The Content Navigator can be extended by additional commands to ease the work of editors and administrators.

Next to the Tcl or Java code that implements the functions of a custom command, the properties of such a command such as its name, multilingual titles, or required permissions need to be defined. Where and in which form this is done is described in this document. Finally, the command needs to be integrated into the menu or the toolbar or into the preview of the Content Navigator so that it can be used.

Three types of custom commands exist:

Custom Command Type	Short Description
Tcl procedure	Tcl Script whose output is displayed in the browser
Wizard	Tcl script that creates and controls GUI dialogs
Servlet call	JSP or servlet that runs in the session and may read-access its data.

Custom commands can be made available globally, i. e. independently of files, for exactly one file, and for any number of files. Every command is a `bean` entry in the `itemRegistry.xml` to be found in the `config` directory of the the GUI web application, or in the `contentMenu.xml` configuration file located in the instance-specific `config` directory. An example:

```
<bean id="editImageWizard"
    class="com.infopark.cm.htmlgui.browse.menuaction.CustomCommand">
  <property name="titles"><map>
    <entry key="en"><value>Image Editing</value></entry>
```

```
    <entry key="de"><value>Bildbearbeitung</value></entry> </map>
  </property>
  <property name="selectionType"><value>single</value></property>
  <property name="requiredObjectPermissions"><set>
    <value>write</value>
    <value>createChildren</value></set>
  </property>
  <property name="command">
    <value>editImage::render</value>
  </property>
  <property name="dialog"><value>true</value></property>
  <property name="objectTypes"><set>
    <value>image</value></set>
  </property>
  <property name="necessaryPermission">
    <value>wizardPermission</value>
  </property>
</bean>
```

The elements and attributes in such a definition have the following meaning:

- `bean`: Defines the custom command. The class to specify is the same for all commands.
  - `id`: a unique identifier.
  - `class`: must be `com.infopark.cm.htmlgui.browse.menuaction.CustomCommand`.
- `property name="titles"`: defines in the `map` subelement the titles displayed in the GUI in the available languages. These languages are specified in the `localizer.xy.xml` files (replace `xy` with the language code) the `share` directory.
- `property name="selectionType"`: Defines the scope of the command. In the `value` sublement one of the following values can be specified:
  - `none` for global commands,
  - `single` for commands that can be applied to exactly one file,
  - `extended` for commands that can be applied to any number of files.
- `property name="requiredObjectPermissions"` (from version 6.6): defines the file-specific permissions according to which file-specific menu commands (of the `single` or `extended selectionType`) are filtered. `single` menu commands are not offered if the permissions have not been granted, `extended` menu commands are offered, however the filtered and not the original file list is passed to the corresponding command script. The permissions `write`, `root`, and `createChildren` can be specified.
- `property name="necessaryPermission"`: defines in the `value` subelement the global permission which user must have been granted for being offered this command. You can specify one of the predefined global permissions well as any custom global permission such as `wizardPermission`, for example.
- `property name="command"`: Defines in the `value` subelement the name of the Tcl procedure to execute when the custom command is called. The procedure needs to be located in the instance-specific `script/cm/serverCmds` directory or in the corresponding directory below *`FionaDir`*`/share`.
- `property name="objectClasses"`: Defines in the `set` subelement a set of file formats to which the command is to be restricted. Inside `set`, each format is specified in a `value` element. If no file formats are specified then the command is not restricted to particular file formats.
- `property name="objectTypes"`: Analogous to the restriction to file formats, commands can be restricted to a selection of the five file types.

- property name="servletURI": Specifies that the command has been implemented as a servlet. The URI of the servlet, relative to the GUI web applikation, is specified in the value subelement. A property element with name="dialog" must not exist at the same time.
- property name="dialog": Determines that the command has been implemented as a wizard. A property element with name="servletURI" must not exist at the same time. The Tcl procedure that implements the wizard needs to be placed in the instance-specific script/cm/serverCmds directory or in the corresponding directory below *FionaDir*/share.
- property name="dialogSize": Determines the size of the dialog window. Possible values are:
  - tiny: see the dialog for changing the user's password as an example
  - small: see the dialog for specifying the locally installed applications as an example
  - large: see the window that is displayed when the *Help -> About Infopark CMS Fiona* menu command is selected from the menu
  - parent: the same size as the window from which the wizard was opened – normally the main window – (from version 6.6.0)
  - none: no window; all output of the wizard is ignored (from version 6.5.0)
- property name="outputsHtml": Specified whether the Tcl code returns HTML text (true or false). If true HTML-specific characters are not converted and the html, body, and head elements must not be present.
- property name="displayHeaderAndFooter": Specifies whether the header and footer areas are to be displayed or not (true or false, respectively). If this item is missing, true is assumed and the header and footer areas are displayed. If the header and footer areas are suppressed, not only the standard buttons but also the footer area buttons generated using the npsButton tags (inButtonArea) are not present.
- property name="descriptions" and property name="sortKey" (from version 6.7.1): These optional elements change the way in which a wizard is listed on the *Wizard Selection* start area. As with titles, descriptions define the subtitles displayed in the available languages according to the map subelement. By applying a sortKey, you can modify the order of the wizards in the selection list.

Hints for Translating NPS 5.2 / 5.5 Custom Commands to Infopark CMS Fiona

In previous versions of Infopark CMS Fiona (NPS), custom commands were defined in the customCommands.xml file. Each command was specified in a listitem element. When translating this file, a bean element needs to be created in the file itemRegistry.xml (or contentMenu.xml for content-specific commands). The contents of the command subelement becomes the id of the bean element. Each of these elements contains an attribute class="com.infopark.cm.htmlgui.browse.menuaction.CustomCommand".

The title elements of the listitem element need to be translated to a property element inside of bean, <property name="titles"><map>, containing an entry element for each of them. The language is set as the key attribute of the entry element. The contents of the title element given in the customCommands.xml file translates to a value element inside entry.

Furthermore, each bean element has a property subelement, <property name="selectionType"><value>, containing one of the following:

- none for listitem elements from globalCustomMenu in customCommands.xml
- single for listitem elements from objectCustomMenu
- extended for listitem elements from objectListCustomMenu

The other subelements of the `listitem` elements in `customCommands.xml` are translated as follows:

`command` becomes `<property name="command"><value>`. The alias names in `commands` become the values of the `property` elements. What, for example, reads

```
<commands>
  <commandAlias>tclProcedureName</commandAlias>
...
</commands>
```

translates to

```
<property name="command"><value>tclProcedureName</value></property>
```

The `servlet` element needs to be translated as `<property name="servletURI">` with the URI as the contents of `value`.

The `outputsHtml` element becomes `<property name="outputsHtml">` with `true` instead of `YES` and `false` instead of `NO` as `value`.

The `dialog` element translates to `<property name="dialog"><value>` with `true` or `false` as `value`.

For file format restrictions by means of `objectClasses` in `customCommands.xml`, `<property name="objectClasses"><set>` is used. Each `listitem` element translates to a `value` element with the same content.

8.2 Tcl Procedures

If you have defined an additional menu command as a Tcl script, the following steps are required to integrate this command into the Content Navigator:

- Store your procedure as a script file in the instance-specific `script/cm/serverCmds` directory.
- Make sure that the procedure is registered in the safe Tcl interpreter. You can place the corresponding Tcl command at the end of the script file. Example:

  ```
  safeInterp alias serverProcName clientProcName
  ```

  This step is only required, if the Tcl procedure is no wizard, i.e. if the value of the `dialog` property is `false`.
- Register the menu command in the `beans` section of the `config/contentMenu.xml` file by adding a corresponding bean. Specify as the `command` property the client alias registered in the safe interpreter.
- Add a reference to the created bean to the `menuBar` section of the `config/contentMenu.xml` file to integrate the menu command into the GUI. Additionally, or alternatively, you might make the command available in the toolbar by adding a reference to the bean to the `toolBar` section.
- Restart the CM and the Trifork server using `bin/rc.npsd restart CM trifork`.

Notes on creating the script file for the menu command

When a custom command is issued that causes a Tcl procedure to be executed, the Content Manager passes the following arguments to the procedure. The arguments depend on the scope of the command that was specified in the corresponding bean by means of the `selectionType` property. The `value` of this property can be one of:

Scope	Arguments
`none` (global)	None
`single` (a single file)	The ID of the file concerned as the only argument.
`extended` (any number of files)	A list containing the IDs of the files concerned, as the only argument.

The procedure needs to be defined using `args` so that the arguments are available as a list:

```
proc tclProcedureName {args} {
  foreach i $args {
  }
...
}
```

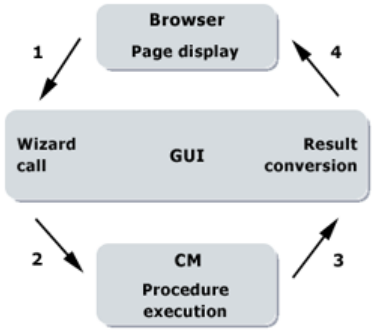
8.3 Wizards

Wizards consist of program code that initiates a sequence of dialogs in the HTML GUI of the Content Manager. Such a dialog sequence enables editors to solve a task in a series of predefined, optionally correctable steps. Wizards are implemented as Tcl scripts and thus can make use of all of the Content Manager's functionality. As examples and templates for developing your own wizards, you can use the wizards included in the Infopark Demo Content.

As with simple Tcl procedures, wizards are integrated into the Content Navigator by means of a definition in the file `contentMenu.xml`. This file can be found in the instance-specific `config` directory (see Configuring Custom Commands). In the definition, set the `dialog` property to `true` to distinguish it from a simple custom command.

The wizards delivered with the CMS are defined in the `itemRegistry.xml` file located in the instance-specific `webapps/GUI/WEB-INF` (`webapps/GUI/WEB-INF/config` in Fiona 6.0.x) directory.

When the custom command is called in the GUI, the latter sends a corresponding XML request to the Content Management Server. From this request the server extracts the name of the custom command and of the Tcl procedure and executes the procedure passing to it the arguments from the request. The procedure then evaluates the arguments passed to it and outputs the result as HTML text to the standard output device. This text is sent as the response to the request to the GUI which processes it and finally sends it to the browser.



A wizard execution cycle consists of four steps

The HTML wizard page generated by the Tcl script is displayed by the GUI as a form whose elements are filled in by the user (in order to move to the next processing round). Since the Tcl script needs to assign a name to the page to be displayed and since the user submits the form it contains using a submit button, the Tcl script receives the information it requires to advance to the next step or terminate the dialog sequence. The script can end this circle by responding with an empty result (containing nothing or just spaces). (Technically spoken, the GUI and the Tcl script form a state machine in which the current state is defined by the page name and the desired change of state by the submit action).

Page Generation

The Tcl procedure generates a dialog page by writing its HTML text into the standard output device. Since the GUI embeds this HTML text in a form contained in a page it must not use the elements `body`, `head` and `form`.

The page name must be set using the following element:

```
<npsPage name="PageName" title="Title">
  <!-- Contents of the page: HTML code plus wizard tags -->
</npsPage>
```

The value of the optional `title` field is used by the GUI as page title. When the form is submitted, the title, among other values (see below), is passed to the wizard.

Interaction with the User

The Tcl code can use the commonly known HTML form elements (i. e. output them to the standard channel) for collecting the user's input.

By means of several wizard tags, you can access functions of the Content Navigator, for example for displaying buttons, making use of selection dialogs and more.

Parameter Transfer

When the user calls a wizard, the corresponding procedure is called with two arguments. The procedure needs to be defined as follows:

```
proc wizardCommand {paramList args} {
   ...
}
```

The `paramlist` argument contains a list of name-value pairs that contain the variable names and the corresponding values stored in the session. The list is not newly created for each form. It is reused

instead, meaning that all dialogs share one name space. Thus, form fields must be unique across all dialogs to be able to call them in any order.

Initially, the list contains the following named values:

Name	Meaning
`wizard.page`	Name of the page generated using `npsPorm name="..."` from which the input originates. At the first execution of the wizard code, i. e. after the user has called the wizard, this parameter has the value `init`. By the value of this parameter the wizard code can determine the page on which the user has clicked a submit button.
`wizard.button`	Name of the submit button pressed last (generated using `npsButton name="..."`). When the wizard's Tcl code is called for the first time, this parameter has the value `init`. By means of the value of this button, the Tcl code can determine which action the user has chosen.
`wizard.tickets`	A list containing pairs of names and values. The names are identifiers, the values are ticket IDs. By means of these values, editors such as the HTML editor can be called from within wizards and a main content or the contents of an HTML field can be passed to them.
`wizard.language`	The user's language setting (de, en etc.)
`wizard.attributeName`	When editing a field, the name of the field is passed to the wizard in this variable..
`wizard.contentId`	When editing a field, the ID of the version concerned is passed to the wizard in this variable.

Every time the user clicks a submit button, this list of name-value pairs is updated and extended with the values of fields unknown until then. The GUI does not remove field names from the list that are not referenced in the current form.

As the second argument, `args`, the IDs of the files to be processed by the wizard are passed to it, depending on the wizard's menu item configuration. For menu items with the `selectionType single`, `args` is the ID of the file currently selected. For an `extended selectionType`, `args` is the list of IDs of the files selected on a GUI page that allows several files to be selected.

The following table summarizes the arguments passed to the wizard, depending on the `selectionType`:

Menu	Arguments
`none`	• List of name-value-pairs (`wizard.language`, `wizard.button`, `wizard.page` as well as input field names).
`single`	• List of name-value-pairs (like above). • ID of the file concerned.
`extended`	• List of name-value-pairs (like above). • A list containing the IDs of the files concerned.

Helper Procedures

Fiona includes a wizard library (`share/script/cm/serverCmds/wizardLib.tcl`) and a WizardController (`share/script/cm/serverCmds/wizards/lib/wizardController.tcl`) that make it much easier to create wizards.

8.3.1 Wizard Tags

npsButton

Using submit buttons, the user can jump to the previous or the next dialog in the dialog sequence or cancel the operation (i. e. the wizard). A submit button can be generated using the following element in two variants:

```
<npsButton name="Name" value="Title" inButtonArea="Placement" verify="Validation" />
<npsButton name="Name" image="Image File" inButtonArea="Placement" verify="Validation" />
```

The tag attributes have the following meaning:

- *Name* specifies the name of the button by which its function is defined.
- *Title* (if `image` has not been specified) defines the button's label text. `value` needs not be specified for the button names `back`, `next`, `execute`, `cancel`, `close` and `ok` since the GUI uses the corresponding localized values in these cases. This value is displayed by the browser as label.
- *Image File* (if `value` has not been specified) defines the name of an image file that is to function as a button with which the user can execute the function specified in `name`. The file is expected to be located in the current scheme directory (below `themes` in the GUI web application of the instance concerned).
- *Placement* specifies whether the button is displayed in the bottom area of the wizard page (`true`) or in place of the `npsButton` tag (`false`, the default setting).
- *Validation* (from Version 6.7.1) specifies whether the GUI is to validate the form fields when the button is clicked (`true`, the default setting) or not (`false`). The default value for the buttons named `back`, `cancel`, and `close` is `false`.

Next to the tag attributes specified, any numer of additional attributes can be specified in the `npsButton` tag. These attributes are taken over into the generated `input` tag.

npsEdit

Using the `npsStream` and `npsEdit` elements, content such as the main content or the value of an HTML field of a file version can be passed from within a wizard to an editor such as the HTML editor. Please proceed as follows to achieve this:

First, read the data to be edited as a stream and store the ticket ID that is returned in a variable. For the main content, you can use the following command for this:

```
set ticketId [obj withId objectID editedContent get blob.stream]
```

As the *objectID* you can use, for example, the ID you received via the `name` parameter of the `npsSelectObject` element. For version fields the field value needs to be queried first and then stored as a stream:

```
set value [content withId contentID get fieldName ]
set ticketId [stream uploadBase64 [base64:encode $value]]
```

In both cases the ticket ID is stored in a variable. The ID is required for passing the data to be edited to the Content Management Server by means of the `npsStream` instruction. Additionally, the `npsStream` instruction is given the MIME type of the data and the path of the CMS file concerned plus an ID, which can be chosen freely, for referring to the data later on in the `npsEdit` instruction:

```
set path [obj withId objectID get visiblePath]
npsStream ticket="$ticketId" mimeType="text/html" path="$path" id="htmlData"
```

Then the `npsEdit` instruction can be used, specifying the reference ID determined above (`htmlData`), to include an editor in the wizard page:

```
<npsEdit editor="html" stream="htmlData"/>
```

By means of the `editor` specification, all editors available in the CMS can be included: `html`: HTML editor, `external`: local application, `tinymce`: TinyMCE, `internal`: internal editor (text input area).

Please note that the text modified by means of the editor is only saved in a stream if one of the wizard buttons `ok` or `next` (observe case) is used to leave the wizard page from which the editor was opened.

Since selecting a file by means of a wizard, editing it, and storing it again can be done on different pages (meaning that the ticket ID may change in the course of editing), the GUI passes in the `wizard.tickets` variable a list of stream IDs and the ticket IDs associated with them to the wizard. This makes it possible to determine the ticket ID by means of the chosen stream ID (here `htmlData`):

```
proc wizardEditor {params args} {
  array set tickets [getParam wizard.tickets]
  set ticketId [getKey tickets htmlData]
...
```

Finally, the edited data can be written back to the file version. For a main content, use:

```
obj withId objectId editedContent set blob.stream $ticketId
```

For other fields, the content needs to be read first and then stored as a field value:

```
set value [base64:decode -asString [stream downloadBase64 $ticketId]]
obj withId objectId editedContent set fieldName $value
```

npsField

The `npsField` element is available from version 6.0.3. It provides functionality for creating input fields in forms. It supports convenient field-specific error handling and obligatory field values. The required HTML code is generated by the GUI. This ensures that the design is homogeneous.

Syntax

```
<npsField type="Type" name="Name" label="Label text"
    required="Obligatory" error="Error text" width="Field width" ...>
...
</npsField>
```

This creates a form field.

- *Type* is the type of the form field and can be `file`, `text`, `select`, `textarea` (from version 6.5), `date` or `selectObject` (both from version 6.7.0).
- *Name* is the name of the parameter created.
- *Label text* is the text with which the field is to be labelled.
- *Obligatory* may be `true` or `false` (the default). If the value is true `true`, an error is generated if the user does not specify the value concerned. The error message is displayed next to the field. With selection fields, the option `none` is not displayed if `*Obligatory*` is `true`.
- *Error text* can be set to an appropriate error message by the wizard programmer. It is displayed if the wizard has found an error in the value of the field. This message is also displayed next to the field, in a matching design.
- *Field width* specifies the width of the field in `ex`, which is the standard unit in the styles used. If required, this value is adjusted internally for a better display. The Firefox browser interprets this value incorrectly, which might lead to display errors.

Depending on the field type, additional parameters can be specified as attributes in the `npsField` tag.

Input Field Types and Their Parameters

Uploading files (type `file`)

```
<npsField type="file" name="name" mimetype="Mime type" ... />
```

This creates a file selection field.

- *Mime type* restricts the files available for selection. For details, please refer to the HTML specification.

Entering strings (type `text`)

```
<npsField type="text" name="name" value="some data"
    maxlength="maximum length" ... />
```

This creates a text input field.

- *maximum length* specifies the maximum number of characters that can be entered.

Selecting an option (type `select`)

```
<npsField type="select" name="name" selectiontype="single"
     rendermode="display mode" ... >
...
</npsField>
```

This creates one or more fields for selecting an option.

- *Display mode* specifies how the field is to be rendered:
  - `vertical`: Radio buttons will be displayed.
  - `list` (default): A drop-down menu is displayed.
- The `width` attribute is only taken into account if *Display mode* has the value `list`.
- Please specify the values available for selection using `npsFieldOption` elements.

Selecting several options (type `select`)

```
<npsField type="select" name="name" selectiontype="multi"
     rendermode="Display mode" size="Height" ... >
...
</npsField>
```

This displays one or more fields for selecting an option.

- *`Display mode`* specifies how the field is to be rendered:
  - `vertical`: Check boxes are rendered.
  - `list` (default): A multi-selection field is rendered.
- The `width` attribute is only taken into account if `*Display mode*` has the value `list`.
- *`Height`* is only taken into account if `*Display mode*` has the value list `list` and specifies the height of the field in lines.
- Please specify the values available for selection using `npsFieldOption`-Elementen.

Entering text (type `textarea`)

```
<npsField type="textarea" name="name" rows="number of rows"
     cols="number of columns" style="styles"... />
```

This creates a text area input field (available from version 6.5).

- *`number of rows`* specifies the height of the text area
- *`number of columns`* specifies the width of the text area.
- *`styles`* specifies (additional) CSS style.

Selecting a file (type `selectObject`)

```
<npsField type="selectObject" name="name" value="File ID"
    startId="Start file ID" startPath="Start file path"
    objectTypes="File types" selectMessage="Message".../>
```

This creates a file selection field which contains one input field to which a selection button is attached. If the button is clicked, the selection page is opened. After selecting a file on this page, the path of the file is placed into the input field.

- *`Start file ID`* and *`Start file path`* optionally specify the file initially selected in the selection dialog. Only one of these attributes may be specified.
- *`File ID`* optionally contains the ID of a file. The path of this file is placed into the input field as a default. This path overrides the *`Start file ID`* or *`Start file path`* presets.
- The optional *`File types`* restrict the files that appear in the selection dialog with respect to their type. The value of this attribute is a comma-separated list. Folders are always visible.
- *`Message`* can be optionally used to display a message to the user informing him about the purpose of the file selection.

Selecting a date (type `date`)

```
<npsField type="date" name="name" value="Date"
  format="Date format"... />
```

This creates a date selection field consisting of an input field and a selection button. If the button is clicked, the selection page is opened. Selecting a date on this page places the date into the input field.

- *Date* is the value of the file selection field as a 14-digit CMS date. The field value passed to the wizard is also a 14-digit CMS date.
- *Date format* is the optional format to be applied to the field value. Please refer to the validDateTimeOutputFormats system configuration element for a description of the formatting placeholders. If this value has not been specified, the date format of the respective user is used.

Usage in Wizards
Input elements, i.e. the corresponding npsField elements, can be comfortably generated in wizards by calling procedures. These procedures are available in the file share/script/cm/serverCmds/wizards/lib/layout.tcl.

Text field

```
::layout::textField name label value ...
```

This creates an npsField element of the type text.

- *name*, *label*, and *value* specify the value of the corresponding npsField element attributes. HTML specific characters will be converted.
- Additional npsField element attributes may be specified as key/value list, e.g. *maxLength*.

Example:

```
::layout::textField product "Product escription" [getParam product]\
    width 40 maxLength 40
```

Multi-Line Text Field

```
::layout::textArea name label value ...
```

This creates an npsField element of the type text.

- *name*, *label*, and *value* specify the value of the corresponding npsField element attributes. HTML specific characters will be converted.
- Additional npsField element attributes may be specified as key/value list, e.g. *maxLength*.

WYSIWYG HTML Editor (TinyMCE, from Version 6.7.1)

```
::layout::mceEditor name label value ...
```

This creates a multi-line text field which is replaced at runtime by the TinyMCE HTML editor. The parameters correspond to those of the multi-line text field.

In CMS Fiona 6.8.0 and later, the editor can be configured by means of the optional config argument. The value of this argument is expected to be the JSON representation of the TinyMCE configuration:

```
::layout::mceEditor … config {{theme_advanced_buttons1: "bold", …}}
```

TinyMCE requires the tinyMCE web application that can be deployed using the following command:

```
InstanceDirectory/bin/rc.npsd deploy tinyMCE
```

In CMS Fiona 6.8.0 and later, the `tinyMCE` web application is included in the list of the web applications to be deployed (`rc.npsd.conf`) by default. Thus, it is deployed if `rc.npsd deploy` is executed.

For properly displaying link paths contained in the field value to be edited, they need to be adapted prior to calling the editor:

```
set tinyMceTextArea [changeImgPathsToPreviewPaths $blob $objId]
```

Here, `tinyMceTextArea` is the name of the variable used for temporarily storing the adapted field value. `blob` is the field content to be processed and edited, and `objId` is the ID of the CMS file concerned. After editing, the paths need to be normalized again:

```
set blob [changePreviewPathsToImgPaths $tinyMceTextArea]
```

Multi selections

```
::layout::chooseMulti limit name valuesAndLabels selectedValues ?label? ...
```

Creates a list of checkboxes or one multi-selection element, depending on the size of *`valuesAndLabels`*. Checkboxes are used if the size is less than the given *`limit`*.

Single selections

```
::layout::chooseSingle limit name valuesAndLabels selectedValue ?label? ...
```

Creates a list of radio buttons or one popup menu, depending on the size of *`valuesAndLabels`*. Radio buttons are used, when the size is less than the given *`limit`*.

Radio buttons

```
::layout::radioButtons name valuesAndLabels selectedValue ?label? ...
```

This creates an `npsField` element with `npsFieldOption` elements. The selection is displayed as radio buttons.

- *`name`* specifies the parameter name of the HTML form.
- *`label`* (optional) specifies the description of the selection.
- *`valuesAndLabels`* specifies the list of options. An option is a list consisting of a value and an optional label.
- *`selectedValue`* specifies the currently selected value.
- Additional `npsField` element attributes may be specified as a list of name-value pairs, for example `required`.

Example:

```
::layout::radioButtons color {{ffff00 yellow} {ff0000 red} {0000ff blue}}\
    [getParam "color"] "Color" required true
```

Checkboxes

```
::layout::checkBoxes name valuesAndLabels selectedValues ?label? ...
```

This creates an `npsField` element consisting of `npsFieldOption` elements. The selection is displayed as independently activatable checkboxes.

- The parameters are analogous to `::layout::radioButtons`
- *`selectedValues`* contains the list of selected options.

```
::layout::checkBox name valueAndLabel selectedValue label ...
```

Creates a single checkbox (from version 6.7.1).

Popup Menu

```
::layout::dropdown name valuesAndLabels selectedValue ?label? ...
```

This creates an `npsField` element consisting of `npsFieldOption` elements. The selection is displayed as a popup menu.

- The parameters are analogous to `::layout::radioButtons`

Multi-selection list

```
::layout::select name valuesAndLabels selectedValues ?label? ...
```

This creates an `npsField` element consisting of `npsFieldOption` elements. The selection is displayed as a multi-selection field.

- The parameters are analogous to `::layout::checkButtons`

File upload

```
::layout::fileField name label ...
```

This creates an `npsField` element of the `file` type and with the name *`name`* and the description *`label`*.

File selection

```
::layout::selectObject name label id selectMessage ...
```

This creates an `npsField` element of the `selectObject` type and with the name *`name`*, the description *`label`*, the file ID *`id`*, and the message *`selectMessage`*.

Date selection

```
::layout::dateField name label value ...
```

Creates an `npsField` element of the `date` type and with the name *`name`*, the description *`label`*, and the value *`value`*.

Example of Usage

If, in a wizard, the following code is returned to the GUI, an HTML input field is created on the corresponding wizard page. It serves to upload a file to the server:

```
<npsField type="file" name="varName"></npsField>
```

For the file the user has uploaded, a streaming ticket is created in the Content Management Server. Through this ticket, the uploaded file can be accessed. The Content Management Server stores the ID of the ticket as well as the name of the file in two variables, namely:

```
$varName.fileName
$varName.ticketId
```

varName is the value of the tag attribute `name` in the `npsField` element. Inside the wizard the values can be accessed in the following way, for example:

```
proc wizard {params objId} {
   set uploadFileName [getParam meinUpload.fileName]
   set uploadTicket [getParam meinUpload.ticketId]
   ...
}
```

For the purpose of storing the uploaded file as a CMS file you can refer to the ticket in the corresponding Tcl command to have the uploaded file placed into the main content of the CMS file. However, before this is done, the name of the new CMS file as well as its file name extension is taken from the name of the uploaded file (if the name and the extension of the CMS file is not predefined or defined elsewhere):

```
# Determine name and name extension
set name [file rootname $uploadFileName]
set contentType [string trimleft [file extension $uploadFileName] "."]

# Create new file containing the contents of the uploaded file
obj withId $parent createAndLoad name $name objClass generic \
        contentType $contentType blob.stream $uploadTicket
```

In the example above, a new file is created from the `generic` file format. The file is placed into the folder with the `$parent` ID, and the main content is filled with the uploaded file that can be accessed via the ticket ID.

To store the uploaded file in the file system of the server instead of in the CMS, you can download it using the `stream downloadFile` command, again specifying the ticket ID of the file:</p>

```
set tmpFile "/tmp/uploads/$uploadFileName"
stream downloadFile $uploadTicket $tmpFile
```

Afterwards the file can be found in the directory `/tmp/uploads/`.

npsFieldOption

The `npsFieldOption` element is available from version 6.0.3. It serves to assign available values to (multi-)selection fields that were created using the `npsField` element:

```
<npsField type="select" ... >
    <npsFieldOption value="the value" label="label text" selected="preselected" />
    <npsFieldOption ... />
    ...
</npsField>
```

- *the value* is the option value.
- *label text* is the text that is displayed for the option. If the attribute label has not been specified, *the value* is used instead.
- *preselected* may be true or false (the default) and specifies whether this option is selected when the field is displayed.

npsFieldSet

This element creates a labeled border which looks like the borders the Content Navigator displays around messages, field values or form fields. The frame is displayed using the fieldset and legend tags. Syntax:

```
<npsFieldSet label="label">...</npsFieldSet>
```

npsFieldTable

This element creates a bordered box (<fieldSet>) containing a table (<table>). The table can be filled by using npsField tags. Table cell tags (<td>) are automatically added before and after the <npsField> tags. If other than <npsField> tags are used, the programmer needs to add the table cell tags by himself. Additionally, the appropriate table row tags (<tr>) need to be defined as well. Syntax:

```
<npsFieldTable label="label">...</npsFieldTable>
```

8.4 Example of Usage

```
<npsFieldTable label="Default values">
  <tr>
    <npsField type="text" name="lastName" label="First name" />
    <npsField type="text" name="firstName" label="Last name" />
  </tr>
  <tr>
    <npsField type="select" name="year" label="Year"
              selectiontype="single" rendermode="list" required="true">
      <npsFieldOption value="2013" selected="true" />
      <npsFieldOption value="2012" />
      <npsFieldOption value="2011" />
    </npsField>
    <npsField type="select" name="month" label="Month"
             selectiontype="single" rendermode="list" required="true">
      <npsFieldOption value="01" label="January" selected="true" />
      <npsFieldOption value="02" label="February"/>
      <npsFieldOption value="03" label="March"/>
    </npsField>
  </tr>
</npsFieldTable>
```

npsGuiControl

By means of subelements, this element allows you to perform particular actions in the GUI web application.

The available actions are `npsRefresh` for reloading the file hierarchy, `npsSelect` for selecting a file, and `npsCommit` for informing the GUI about changes to the draft version of a file.

8.4.1 Reloading the File Hierarchy

```
<npsGuiControl>
  <npsRefresh identifier="id" depth="depth"/>
</npsGuiControl>
```

`npsRefresh` is an empty element that has the following attributes:

- `identifier`: The value of this attribute, *`id`*, determines the ID of the CMS file to update.
- `depth` determines the scope of the update process and can be one of the following values:
  - `node`: Only the specified file is updated.
  - `children`: The file and, in case of folders, all its immediate subfiles are updated.
  - `subtree`: The subtree of which the file is the root is updated.

8.4.2 Selecting a File

```
<npsGuiControl>
  <npsSelect identifier="id"/>
</npsGuiControl>
```

By means of `npsSelect` a file in the file hierarchy can be selected. `identifier` has the same meaning as with `npsRefresh`.

8.4.3 Informing the GUI about Content Changes (from Fiona 6.6)

```
<npsGuiControl>
  <npsCommit value="true"/>
</npsGuiControl>
```

If you have specified a wizard for editing a field, the GUI ensures that the requirements for editing are met, for example by creating a draft version of the file concerned.

To inform the GUI about changes to the version, use `npsCommit`. If this is omitted, the version might be reverted, meaning that the changes are lost.

npsPage

The `npsPage` element creates a wizard page. The content of the element will be used as the HTML content of the page.

Syntax

```
<npsPage name="name" title="title" subtitle="subtitle"
  affectednodes="affected files" helpurl="help URL">
  HTML code
</npsPage>
```

- *`name`* is the name of the wizard page.

- *title* is the optional title of the wizard page.
- *subtitle* is an optional piece of explanatory text on the page.
- The optional *affected files* are the IDs of the files affected by the wizard. They are displayed on the page.
- *help URL* is an optional online help URL for this wizard page (available from version 6.7.0).

npsPreviewImage

This element serves to display images from the CMS folder hierarchy in wizards. It calculates a preview URL, places it inside an `img` tag, and writes this tag to the output stream. Syntax:

```
<npsPreviewImage src="/source/path" alt="Alt text" title="Title" other="HTML attribute" /
>
```

The `src` tag attribute is obligatory - it serves to specify the path of the file in the CMS file hierarchy. If the tag attributes `alt` and `title` are not specified, their values are taken from the CMS image file. All other attributes are taken over unmodified. Example:

```
<npsPreviewImage src="/images/anImage" style="float:left;
margin: 1em; margin-right: 3em;"/>
```

This causes the following HTML text to be generated (if *localhost* is the name of the preview server):

```
<img title="playland" style="float:left; margin: 1em; margin-right: 3em;"
  src="http://localhost:8080/NPS/preview/TRIFORK531261944775/-/mastertemplate/3/images/
anImage.gif;jsessionid=TRIFORK531261944775"
  alt="playland" />
```

npsSelectObject

From version 6.7.0, the `npsSelectObject` tag is obsolete. Please use the `npsField` tag with the `selectObject` type instead.

```
<npsSelectObject name="selectedObj" objectTypes="image,generic" />
```

Using this element the user of the wizard is given the opportunity to select a file on a subpage. For this, an input field and a selection button is displayed. When the button is clicked, the selection page is opened. If a file is selected there, its path is taken over into the input field. The `npsSelectObject` element has the following attributes:

- `name` contains the name of the parameter with which the ID of the file selected by the user is passed to the wizard.
- `value` optionally contains the ID of a file. The path of this file is places into the input field as a default. This path overrides the preset values of `startId` and `startPath` .
- `startId` and `startPath` allow you to specify the file initially selected in the file selection dialog. Only one of these two attributes may be given.
- Using `objectTypes`, the types of the files displayed in the selection dialog can be restricted by their type. The value of this attribute is a comma-separated list. Folders are always visible.
- Using `selectMessage`, you can have a text displayed to inform the user, for example, of the purpose of the file selection action.

- `isRequired` specifies whether a file needs to be selected in the selection dialog or whether the dialog may also be cancelled. A file must be selected only if the value of this attribute is `true`.

npsStream

Using this element in conjunction with `npsEdit`, field values can be transferred to and from the CMS so that they can be edited by means of a wizard. See `npsEdit`.

npsStreamImage

If an image is to be displayed in a wizard page, the Tcl code of the wizard needs to make the image available to the GUI. However, for security reasons the Content Management Server normally has no access to the file system of the GUI and vice versa.

An alternative possibility to have a wizard display an image is to transfer the image to the Content Manager by the means of streaming. The GUI or the browser can then request the image from the Content Manager.

The wizard first places the image file on the server using this command:

```
set ticketId [stream uploadFile path]
```

Then it can write the following tag to the standard output so that the GUI can generate an appropriate `img` tag.

```
<npsStreamImage ticketId='$ticketId' mimeType='$mimeType' />
```

The GUI translates this tag to the following `img` element. The URL in its `src` tag causes the Content Manager to deliver the image:

```
<img src="/NPS/streampreview?ticketId=34253&mimeType=image%2Fgif" />
```

Both tag attributes, `ticketId` and `mimeType` are obligatory.

npsThemeImage

This element generates an `img` tag whose URI points to an icon or another image from the current theme directory (*themes* subdirectory). The `src` attribute containes the file name of the image (without path). All other attributes are taken over into the `img` tag unmodified. An example:

```
<npsThemeImage src="template.png" />
<npsThemeImage src="template.png" alt="ALT" title="Title" bit="1" />
```

This generates the following HTML code:

```
<img src="/NPS/themes/default/template.png" />
<img src="/NPS/themes/default/template.png" alt="ALT" title="Title" bit="1" />
```

8.4.4 Sample Wizard

The WizardController makes it possible to write event-driven wizards. Furthermore it eases the creation of dialog elements. You can integrate the WizardController by means of the `useWizardController` procedure which is part of the `wizardLib.tcl` file. For details about how the WizardController works, please look into the `wizardController.tcl` file.

The following example is a simple wizard for editing the title of a file. For handling events, the WizardController calls procedures whose names it composes of the page name and the user's action (identified by the name of the button clicked). For clarity, the components of these procedure names are colored.

```
if {[app get appName] ne "CM"} {
   return
}

namespace eval ::enterTitle {

   useWizardController


   proc initRender {paramList args} {
        variable objId
        set objId $args
        return
   }


   proc handleInitPage {buttonName} {
        variable objId
        if {![obj withId $objId get isEdited]} {
            return [renderErrorPage "The file does not have a draft version." "Error"]
        }
        if {[whoami] ne [obj withId $objId editedContent get editor]} {
            return [renderErrorPage "You are not the editor of the file." "Error"]
        }
        return [renderEnterTitlePage]
   }


   proc handleCancelAction {pageName} {
        return ""
   }


   proc renderEnterTitlePage {} {
        set content [::layout::textField contentTitle Titel "New title"]
        return [renderPage -title "Set title" -buttons [list ok cancel]\
            "enterTitle" $content]
   }


   proc handleEnterTitleOkPageAction {} {
        variable objId
        obj withId $objId editedContent set title [getParam contentTitle]
        addNpsRefresh $objId
        return [renderSuccessPage]
   }


   proc renderSuccessPage {} {
        set content "<p>The title has been set.</p>"
        return [renderPage -title "Thank you very much!" -buttons [list ok] "success"
 $content]
   }
```

```
    proc handleSuccessOkPageAction {} {
         return ""
    }


    # Output any page of this wizard
    proc renderPage {args} {
        variable objId
        return [eval [list npsPage -affectedNodes [list $objId]] $args]
    }
}
```

8.5 Servlets

This type of custom command has the effect that the GUI sends a request using a configured URL. This URL is configured in the *itemRegistry.xml* file (see Configuring Custom Commands):

Since the GUI needs to be able to find the servlet, it must be placed in the GUI web application. Place the servlets class file in the subdirectory named like the servlet package name below

.../webapps/GUI/WEB-INF/classes/

Alternatively, the jar file can be placed in lib. Then register the servlet in

.../webapps/GUI/WEB-INF/web.xml

The paths above refer to the instance directory concerned.

The servlet is required to descend from the class *com.infopark.cm.htmlgui.browse.CustomServlet* which provides the following auxiliary functions:

`String getLogin(HttpServletRequest request)`
returns the user name of the current user.

`Locale getLocale(HttpServletRequest request)`
returns the locale (and thus also the language) of the current user.

`List getObjectIds(HttpServletRequest request)`
returns a list of the IDs of all files for which the command has been called (for global commands the empty list).

`String getReturnUri(HttpServletRequest request)`
returns an URI to be requested via a browser redirect for returning to the GUI.

`String getDialogId(HttpServletRequest request)`
returns the dialog ID required for subsequent requests.

Therefore, in the custom servlet the login of the current user and his language can be determined. If the custom command refers to a file or a list of files, the IDs of the files concerned are available as well (see the methods above).

Notes on usage

Since the servlet must not harm the GUI, nothing must be stored in the current session. Instead of the methods normally used for this purpose, please use the following auxiliary methods of the same name:

```
void setAttribute(HttpServletRequest request,
```

```
String key, Object value)

Object getAttribute(HttpServletRequest request, String key)

void removeAttribute(HttpServletRequest request, String key)
```

When the servlet initiates new requests, for example via a redirect or a form, the dialog ID must be passed as a request parameter (URL parameter, form field or the like) for the auxiliar methods to continue to work. This is not necessary when returning to the GUI.

The name of the parameter under which the dialog ID must be passed in the request is stored in the `DIALOG_ID` static string field.

8.6 Opening other Pages with a Menu Command

You can add menu commands to the Content Navigator that point to an external or internal web page. For this, add a bean according to the following example to the file `contentMenu.xml`. In this example, an external page is opened:

```
<bean id="linkToExternal" class="com.infopark.cm.htmlgui.browse.menuaction.Redirect">
  <property name="titles"><map>
    <entry key="de" value="link to myDomain" />
  </map></property>
  <property name="uri" value="http://www.mydomain.de" />
  <property name="isExternal" value="true" />
</bean>
```

The property `isExternal` specifies whether an external page or an internal (GUI) page is opened.

To assign the bean to a menu item, add its ID to the `menuBar` section of the `contentMenu.xml` file. For the changes to become effective, the Content Manager needs to be restarted, and the GUI needs to be redeployed.

9

9 Export Options

Infopark CMS Fiona exports files to generate web pages from their released versions. This is accomplished by means of layout files that add layout in the broadest sense to the content. Files with binary content are exported unmodified.

The CMS provides two possibilities to export content, the incremental and the static export.

9.1 Incremental Export

The incremental export is the standard export procedure for websites with a lot of traffic. By means of jobs configured in the Content Management Server, updated content is regularly transferred to the live server. There it is exported by the Template Engine.

One advantage of the incremental export is its better performance compared to the static export. This is because not the complete folder hierarchy but only files that have been modified need to be exported. Another advantage is that the live server is always up-to-date.

In the Content Management Server, a system job transfers updated content to the Template Engine and initiates the export in regular intervals.

9.2 Static Export

The static export has the function to export all files in the folder hierarchy, or a partial hierarchy. By this, a directory hierarchy is created which is formed analogously to the exported folder hierarchy.

This export method is suited especially for smaller websites that are uncritical with respect to performance and topicality. Furthermore, the static export is the procedure used for multi-channel or cross-media publishing. This is possible by using different template sets for the export.

The static export is initiated by means of the `exportSubtree` Tcl command.

10

10 The PDF Generator

The PDF generator is available and to be licensed as a separate product. It makes it possible to create PDF files from CMS folders and documents.

After the installation and configuration of the generator, the file menu of the Content Navigator contains a corresponding menu command. To create a PDF file, select a file or a folder with the *html* name extension in the folder hierarchy and choose the menu comand mentioned. The PDF file is created as a file of the *resource* type in the same folder as the source file. To its link list field `pdfSource` a link to the source folder is added. If required, add `pdfSource` to the file format used for PDF files.
If an existing PDF file is to be updated, the menu command is labelled correspondingly.

The output of the PDF generator is controlled by an XSL style sheet which by default supports the following features:

- Output in the format DIN A4
- Table of contents
- Bookmarks
- Running headers and footers
- Double-sided output
- The following HTML tags are processed:
  - Headlines (`h1` etc.); the headlines `h1` to `h4` are placed into the table of contents and the bookmark list.
  - Lists (`ul`, `ol` with different numbering types, `dl`)
  - Tables (including `border`, `rowspan`, `colspan`, `align`, correct column width using `colgroup`)
  - Embedded images; the dimensions can be specified using the width and height attributes)
  - Page breaks (br)
  - Others: `address`, `b`, `big`, `blockquote`, `cite`, `code`, `em`, `hr`, `i`, `kbd`, `nobr`, `p`, `pre`, `samp`, `small`, `strike`, `strong`, `sub`, `sup`, `title`, `tt`, `u`, `var` .

The following features are supported from version 1.1:

- PNG images

The following features are not supported by the style sheet:

- Links (`a`) to external URLs and within the document
- Centering using `center`

To adapt the output of the PDF generator to individual requirements, the corresponding style sheets can be written and integrated into the Content Navigator's menu.

10.1 How the PDF Generator Works

PDF files are generated by means of the preview function, using a layout file. The layout file might, for example, read out the main content of all files contained in a folder, concatenating them to a single HTML document which is then passed to the PDF generator for conversion. The versions of the files used for this depend, as is the case for the normal preview, on the view chosen in the *View* menu of the Content Navigator.

Prior to generating the PDF file, the corresponding HTML text is cleaned using *tidy*, and the following HTML elements are removed: `font`, `npspm`, `form`, `input`, `textarea`.

As mentioned above, an XSL style sheet file is used for converting the HTML document to a PDF file. The resulting PDF file is placed in the folder where the source file is located.

Both the XSL file and the layout file can be configured and are maintained in Infopark CMS Fiona itself.

The PDF generator is based on the Formatting Objects Processor 0.20.5 / 0.93 (Version 1.0 / 1.1).

10.2 Installing the PDF Generator

The PDF generator is delivered as a Java web archive named `infopark-pdf-generator.war`. It is contained in the `infopark-pdf-generator-`*`version`*`.zip` archive that can be obtained on request from our customer support. To install the PDF generator, stop the CMS and deploy the web archive in the following way:

- Place `infopark-pdf-generator.war` into a directory of your choice, for example into `instance/default/webapps` in the CMS directory.
- Add the license file `license.xml` to the `infopark-pdf-generator.war` file (adapt `/path/to` to match the directories into which you placed the files):

```
cd /tmp mkdir WEB-INF
cp /path/to/license.xml WEB-INF
zip -r /path/to/infopark-pdf-generator.war WEB-INF
rm -r WEB-INF
```

- Run the following Trifork command from the command line to create a new system container, `infopark-pdf-generator`:

```
TriforkDir/domains/default/bin/trifork system create infopark-pdf-generator
```

  `TriforkDir` refers to the directory of the Trifork Application Server.
- Run the following Trifork command from the command line to deploy the new web application:

```
TriforkDir/domains/default/bin/trifork archive deploy infopark-pdf-generator \
        /path/to/infopark-pdf-generator.war
```

Again, specify the correct Trifork path and the full path of the `infopark-pdf-generator.war` archive.

After the installation, the PDF generator should be configured.

10.3 Configuring the PDF-Generator

The configuration of the PDF generator is limited to the paths of the layout and XSL files to use, and its own URL (to which the GUI sends conversion requests). These parameters are specified directly in the instance-specific menu configuration, `/config/contentMenu.xml`, by which the PDF generator is integrated into the Content Navigator. See the beans `createPdfManual` and `updatePdfManual` in this file.

Please note that a bean definition in the `contentMenu.xml` file is not effective if the same bean has already been defined in the `itemRegistry.xml` file.

The relevant configuration parameters are:

- `pdfDownloadUrl`: the URL the GUI requests to have the PDF content generated and returned to itself.
- `xslPath`: CMS path to an XSL file (e.g. `/layout/pdfstylesheet.xsl`) containig the XSL style sheet by means of which the XHTML output of the layout file is formatted. If no style sheet is given, an internal style sheet is used.
- `pdfTemplateName`: the name of the CMS layout file used for generating the HTML input of the PDF generator.

After changes have been made to the configuration, the CM and the GUI must be restarted.

After a PDF file has been generated and stored in the CMS folder hierarchy as a resource, a link to the source folder is stored in this resource. This makes it possible to update the PDF without having to recall where its source folder is located. The link is stored in the `pdfSource` linklist field of the PDF resource. For this to work, this field must be created and added to the file format of the PDF resource. Sample field definition:

- Name: `pdfSource`
- Title: Source of the PDF document
- Description: In this field, the PDF Generator stores a link to the source folder to be able to find the source later on when the PDF file is updated.
- Type: Link list
- Searchable in Content Manager: No
- Searchable in live system: No
- Input field type: Link list field

11

11 Saving and Restoring Data

Infopark CMS Fiona is used to create, maintain, and publish valuable content. Not only the content but also the configuration of the system is the result of much work. Therefore, it is highly recommended to have a backup available, in case the hardware or software fails.

11.0.1 General Data Backup

Infopark recommends to create database dumps and to back-up the installation directory of Infopark CMS Fiona on a regular basis. The intervals at which the respective data is saved is application-specific. If the highest risk you can bear is to loose the content produced or modified in one day, the content needs to be saved once every day. This applies analogously to the data located in the installation or instance directory hierarchy, i.e. configuration files, scripts, wizards, etc. – in particular while the system is being extended.

Please note: If the content is stored in the file system instead of the database (see the `storeBlobsInDatabase` configuration value), the CMS must be stopped completely before making a backup.

Backups should never be stored on the medium on which the data is produced and used.

11.0.2 Saving the Content, System and Runtime Configuration

Infopark CMS Fiona's Content Management Server (CM) provides a mechanism (Dump/Restore) for partially or completely saving and restoring the content, the system configuration, and the runtime configuration (which includes field definitions, file formats, for example). This allows you to

- save your data to have a backup available in case of a system failure;
- migrate content to a newer version of Infopark CMS Fiona (up to version 6.5.0, with later versions, the database is migrated directly);
- duplicate content in the source instance of the CMS installation or transfer it to a different instance in order to use it as a basis for a new website, for example.

Infopark recommends (as an addition to general backup procedures) to regularly use the CM to create a complete dump and to treat it like a backup.

The following sections describe how to produce and restore a complete or partial dump using the CM.

11.1 Complete Dump/Restore

A complete dump (not the partial dump, however) includes files and the data associated with them (versions, links, file formats, fields etc.) as well as the system configuration.

11.1.1 Saving Data

Caution: Only start the dumping process if you have ensured that no write access to the data takes place while dumping. Otherwise the saved data will be incomplete or corrupted. Make sure that the CM is not running as a server and that no other CMs are running or started in single mode while data is being dumped.

It is not possible to dump CMS content and other data to an NFS mount, or to restore a CMS dump from an NFS mount (using `CM -dump` or `CM -restore`, respectively).

In order to save your data, please proceed as follows:

- Log on to the operating system as the CMS administrator. The CMS administrator is the user whose login was specified for this purpose during the installation of Infopark CMS Fiona.
- Stop the Content Management Server.
- Change to the instance-specific `bin` directory and execute the following command in a shell:

```
CM -dump dumpDir [resume]
```

Please specify as `dumpDir` the directory to which the CMS data is to be saved. A relative path refers to the current directory.

Using the `resume` option, an interrupted dumping procedure can be resumed.

11.1.2 Restoring Data

Caution: Only start the restore process if you have ensured that no write access to the data takes place while restoring. Otherwise the existing or restored data will be incomplete or corrupted. Make sure that the CM is not running as a server and that no other CMs are running or started in single mode while data is being restored.

It is not possible to dump CMS content and other data to an NFS mount, or to restore a CMS dump from an NFS mount (using `CM -dump` or `CM -restore`, respectively).

In order to restore the data saved previously, please proceed as follows:

- Log on to the operating system as CMS administrator. The CMS administrator is the user whose login was specified for this purpose during the installation.
- Change to the instance-specific `bin` directory.
- Stop the Content Management Server:

```
./rc.npsd stop CM
```

- Version 6.7.0 or later, and If you use the Rails Connector: Remove the Rails database tables using the following command:

```
./CM -unrailsify
```

- To restore the dump, execute the following command (the parameters are described below):

```
./CM -restore dumpDir [resume] [-sysConfig none | partial | full]
```

- Version 6.7.0 or later, and If you use the Rails Connector: Generate the Rails database tables using the following command:

```
./CM -railsify
```

- Start the Content Management Server:

```
./rc.npsd start CM
```

Please specify as `dumpDir` the directory in which the CMS data was saved. A relative path refers to the current directory. Using `resume`, an interrupted restoration process can be continued.

By means of the optional argument `-sysConfig`, you can determine the type of system configuration data to be imported. Use one of the following three keywords for this:

- `none`: The system configuration is not restored.
- `partial`: All configuration entries except the following ones, which are related to the server configuration, are restored from the dump:
  - `server, tuning, portal, export.incrementalUpdate, export.charsetMap, export.charset, export.exportAbsolutePaths, export.absoluteFsPathPrefix, export.absoluteUrlPrefix, export.abortOnError, export.validDateTimeOutputFormats, export.guiUrl, export.writeLiveServerReadPerms, indexing, gui.dynamicPreviewUrl, gui.dynamicPreviewDirectory, gui.fontFamily, gui.fontSize, gui.webDav, userManagement.editorial, userManagement.live`.
- `full`: The saved system configuration is restored completely.

The default `sysConfig` mode is `partial`.

The restoration process is interactive. If you are restoring data into a newer version of the CMS (only up to version 6.5.0, later versions use in-place migration instead of dump and restore) some field names might have become reserved keywords in the new version. The Content Manager will ask you in such cases to enter new names for the fields concerned. All references to the names concerned will be updated, except when they are located in Tcl scripts. Apart from the names beginning with a digit, the following names are invalid:

- anchors, archivedContentIds, blob, blobLength, body, bodyTemplateName, channels, children, codeForActionPreview, codeForPreview, codeForSourceView, codeForThumbnail, contentIds, contentType, createNewsItem, displayTitle, editor, encodedBlob, encodedExportBlob, exportBlob, exportCharset, exportFiles, exportMimeType, externalAttributes, externalAttrNames, externalAttrTypeDict, frameNames, freeLinks, getKeys, hasChildren, hasSuperLinks, height, id, isCommitted, isComplete, isEdited, isReleased, isRoot, lastChanged, linkListAttributes, mimeType, name, next, nextEditGroup, nextSignGroup, noPermissionLiveServerRead, objClass, objectId, objectsToRoot, objType, parent, path, permissionCreateChildren, permissionLiveServerRead, permissionRead, permissionRoot, permissionWrite, prefixPath, previous, reasonsForIncompleteState, releasedVersions, setKeys, signatureAttrNames, sortValue, subLinks, superLinks, superObjects, suppressExport, textLinks, thumbnail, title, toclist, validControlActionKeys, validCreateObjClasses, validFrom, validObjClasses, validPermissions, validSortKeys, validSortOrders, validSortTypes, validUntil, version, visibleExportTemplates, visibleName, visiblePath, width, workflow, workflowComment, workflowName, xmlBlob.

Attention: When fully restoring a dump to a Content Management Server, all the changes made intermediately to the system configuration will be overwritten.

Resuming an Interrupted Restoration Process

If, for example, the restoration process was cancelled because of insufficient hard disk space, it can be resumed with the following command:

```
CM -restore dumpDir resume
```

Migrating from NPS 5 to Infopark CMS Fiona (up to version 6.5.0)

In the new directory structure of Infopark CMS Fiona the supplied scripts are located in a directory that is different from the directory provided for custom scripts. The supplied scripts can be found in the CMS directory `share/script/...` while the custom scripts are located below the `script` directory of the instance concerned, for example in `instance/default/script/...`

The procedures contained in the custom (i. e. instance-specific) scripts override the procedures in the supplied scripts. If both a custom as well as a supplied script file contain a procedure named `example`, the CMS uses the procedure in the custom script file because it is sourced after the supplied script. This effect is by design. It allows customers to replace supplied script files with their own versions without having to delete or modify the supplied ones.

However, when migrating from NPS 5 to Infopark CMS Fiona all scripts originating from NPS 5 are placed in the instance-specific `script` directory of Fiona. Therefore, it is likely that newer scripts will not be used by Fiona because they are overridden by scripts from the NPS 5 installation. This is why all scripts originating from the previous version need to be deleted if they do not contain custom code. For scripts that have been modified you should check whether `share/script` contains a file of the same name. If so, the custom parts of the file should be transferred to a different file and the duplicate custom file should be deleted.

11.2 Partial Dump/Restore

When data is partially dumped and restored, the following entity types are taken into account:

Entity	Command keyword
Datei	`object` `objectTree` (Folders and the files they contain, recursively)
Field	`attribute`
File format	`objClass`
Workflow	`workflow`

The Content Manager additionally saves and restores the following entities referenced by other entities:

Entity	References that can be contained in the entity
File	Tasks, workflows, file formats, versions (including fields, links, and channels), log entries, file specific access permissions.
Field	Links
File format	Workflows, file formats, fields
Workflow	Fields

11.2.1 Saving Data

Attention: Up to CMS Fiona 6.7.3, with productively used systems, data must not be restored from partial dumps since this might result in data loss.

Caution: Only start the dumping process if you have ensured that no write access to the data takes place while dumping. Otherwise the saved data will be incomplete or corrupted. Make sure that the CM is not running as a server and that no other CMs are running or started in single mode while data is being dumped.

In order to partially save your data, please proceed as follows:

- Log on to the operating system as CMS administrator. The CMS administrator is the user whose login was specified for this purpose during the installation.
- Stop the Content Management Server.
- Change to the instance-specific `bin` directory and execute the following command in a shell.

```
CM -dump dumpDir (resume | {entity ident} | file deffile)
```

Please specify as `dumpDir` the directory to which the CMS data is to be saved. A relative path refers to the current directory.

Using the `resume` option, you can have the Content Manager continue an interrupted dumping procedure.

The type of the entity to be saved and its identifier (name, ID) can be specified optionally and more than once as *entity ident*. For *entity* use one of the keywords `object`, `objectTree`, `attribute`, `workflow`, or `objClass`.

If `object` or `objectTree` is specified as *entity*, specify a file ID or a file path as *ident*. For all other entities specifiy their respective name.

Using the option `file` *deffile*, the entities to be saved can be specified in a definition file instead of specifying them on the command line. In such a definition file each line consists of an *entity* type, followed by an *ident* which is the name (or path or ID for files) of the respective entity. Comments can be placed on separate lines beginning with a hash mark (#).

The following sample command line dumps data to the directory `myDir`. The file with the path `/en` and the file with the ID `12345` are dumped recursively (including the files they contain). The file format `newsdoc` is dumped as well:

```
CM -dump myDir objectTree /en objectTree 12345 objClass newsdoc
```

11.2.2 Restoring Data

Attention: Up to CMS Fiona 6.7.3, with productively used systems, data must not be restored from partial dumps since this might result in data loss.

Only start the restore process if you have ensured that no write access to the data takes place while restoring. Otherwise the existing or restored data will become incomplete or corrupted. Make sure that the CM is not running as a server and that no other CMs are running or started in single mode while data is being restored.

In order to restore your data, please proceed as follows:

- Log on to the operating system as NPS administrator. The NPS administrator is the user whose login was specified for this purpose during the installation.
- Stop the Content Management Server.
- Change to the *bin* directory of the instance concerned and execute the following command in a shell.

```
CM -restore dumpDir  targetPub \
  [{entity ident}| file deffile] \
  [-batch [{(entity | all) conflictStrategy}]]
```

Please specify as `dumpDir` the directory to which the data was dumped. A relative path refers to the current directory.

If the optional parts of the command line are completely omitted, the Content Manager assumes that all data contained in the dump is to be restored.

The *targetPub* parameter is required to specify a target CMS folder for the files to be restored.

Specify as *entity* one of the keywords `object`, `objectTree`, `attribute`, `workflow`, or `objClass`.

If `object` or `objectTree` is specified as *`entity`*, provide a file ID or a file path as *`ident`*. For all other entities provide their respective name.

Using the option `file` *`deffile`*, the entities to be restored can be provided by means of a definition file (*`deffile`*) instead of entering them on the command line. In such a definition file, every line consists of an *`entity`* type, followed by an *`ident`* which is the name (or path or ID for CMS files) of the respective entity. Comments can be placed on separate lines beginning with a hash mark (#).

By means of the `-batch` option, interactive conflict handling (see below) can be switched off. If `-batch` has been specified, one of the following conflict handling strategies can be specified for each entity type:

- `w` = Always overwrite
- `a` = Always restore under an automatically generated name
- `k` = Always skip the entity

You can use `all` for *`entity`* to set the strategy for all entity types. The default conflict handling strategy is `w` (always overwrite). However, in CMS Fiona 6.0.0 and later, a CMS file contained in a partial dump is never restored if the file already exists in the target system, independently of the chosen conflict handling strategy.

As with a full restoration process, an interrupted partial restoration can be continued using the `resume` option:

```
CM -restore dumpDir resume
```

Example: The following command restores the data from the dump located in the `myDir` directory to the destination folder with the ID `4812`. The complete folder hierarchies `/de` and `12345` as well as the file format `newsdoc` are to be restored:

```
CM -restore myDir 4812 objectTree /en objectTree 12345 objClass newsdoc
```

Conflict Handling

When a complete dump is restored, the original file IDs are preserved. This is not the case when data from a partial dump is restored. This would lead to conflicts if files with these IDs already exist. Therefore, files are created using the next available ID.

However, name conflicts can still occur, if files contained in partial dumps already exist under the same name in the destination folder. In this case an error message is displayed and the restoration process is aborted. If entities of the other types (fields and workflows, for example) already exist, the conflict is handled, meaning that you can decide how to proceed.

If the interactive conflict handling has not been switched off using `-batch`, all conflicts must be solved interactively. If an entity already exists, the Content Manager will offer the following options in the case of a conflict:

- `o` = Overwrite (replaces the existing entity with the saved one).
- `w` = Always overwrite (like `o`, but also for all other entities of the same type).
- `r` = Rename (restore the entity under a different name which is to be specified by the user).
- `a` = Always rename (like `r`, but using automatically generated new names for the other entities of the same type).

- `s` = Skip (the entity is not restored).
- `k` = Always skip (like `s`, but also for all other entities of the same type).

Mirror File Handling

From CMS Fiona 6.8.0, the partial dump/restore procedure also restores mirror files. As with other entities, restoring mirror files can lead to situations that require special treatment, e.g. if a dump contains mirror files but not their original files. In interactive mode (if `-batch` has not been specified) the user is asked how to proceed if such a situation occurs. In batch mode, on the other hand, the desired behavior of the Content Management Server can be defined in advance using up to two more keywords for *`entity`*:

- `unrestorableMirrors`: If mirror files cannot be restored at all (because mirror files of the target folder exist), the restoration process can be terminated by specifying `q` as the conflict handling strategy (this is the default behavior). If, instead, `k` has been specified, the process is continued without restoring mirror files.
- `replaceOriginal`: If the original of a mirror file is not contained in the dump but is present in the target system under the known path, strategy `a` restores the mirror file using the existing original file. Strategy `k` (the default) causes such mirror files to be skipped instead.

Thus, in batch mode, the restoration process is terminated if the dump contains mirror files, and if mirror files cannot be restored in principle. However, if mirror files are restorable, they are still not restored if their originals are missing from the dump. This is the default behavior.

In the following example, mirror files whose originals are not contained in the dump will nevertheless be restored if their originals are present in the target system. Otherwise, if the originals are neither contained in the dump nor in the target system, they will be omitted.

```
CM -restore sourceDir /target/folder -batch replaceOriginal a
```

Listing the Contents of a Partial Dump

The data that can be restored from a partial dump can comfortably be listed using the `-listDump` option:

```
CM -listDump dumpDir [entity]
```

`dumpDir` refers to the directory containing the dumped data. By specifying the entity type *`entity`*, the output can be restricted to the files of this type. Example:

```
CM -listDump dumpDir object
```

Additional Technical Information

The Content Manager saves the current state of the dumping procedure to the file `dumpstate.xml` in the respective dump directory.

In case the restoration process is interrupted, the Content Manager saves the current state of the process in the file `restorestate.xml` in the `tmp` directory below the instance directory as well as in the database. Only one unfinished restoration process can exist for each instance.

12

12 Migration

If you wish to update your existing installation of Infopark CMS Fiona to the latest version, a migration is required. This will transfer your content and your configuration into the new system. From Fiona 6.5.0 (in some cases from Fiona 6.0.4), the in-place migration method can be used. This method migrates your content, which is stored in the database, much faster than the old dump/restore method.

As the first step of an in-place migration, the configuration of the old installation is copied to the new installation to which it is then adapted. The database configuration is copied as well but will not be changed. Then, the contents of the database is migrated to make it suitable for the new Fiona-Version. *In-place* means that the database tables are modified directly.

We recommend to update Fiona on a test system first.

12.1 Migration Requirements

This guide refers to CMS Fiona 6.0.4 or later. With Sybase or MS SQL databases, this guide refers to CMS Fiona 6.5.0 or later. If your Fiona version is not covered by the version ranges mentioned, or if you wish to switch-over to a different database product in the migration process, please read the section on migrating older versions first.

Before you start migrating, please ensure that the following requirements are met.

12.1.1 System Requirements

The machine on which the new version is to be installed needs to meet the system requirements. Also, from version 6.7.0, a machine-specific license without ID limit is required for migrating in-place.

12.1.2 Database

Make a dump of the CM and TE databases. In the case of unexpected problems, you can revert to the original state using these dumps.

In-place migration is supported for the following database products and versions:

- Oracle 9 or later
- Sybase 15 or later (only up to CMS Fiona 6.7.2)
- DB2 7 or later (only up to CMS Fiona 6.6.0)
- MS-SQL Server 7 or later (from CMS Fiona 6.5.1)

- MySQL 5.0 (from CMS Fiona 6.6)

In-place migration is not available for SQLite databases.

12.1.3 Installation of the current version of CMS Fiona

Install the current version of CMS Fiona on the machine on which the migration is to be performed.

The current version is required to be installed only, there is no need to configure it. It will be configured by the in-place migration process.

12.1.4 Making the old installation accessible

The new installation requires access to the instance directories and the database of the old installation since the migration process copies the configuration from the old to the new installation and migrates the content. This requirement is met if the new Fiona version has been installed on the machine on which the old version is located.

However, if the new CMS Fiona version has been installed on a new server on which no file system access to the old instance directories exists, these directories need to be copied to the new server. Do not copy them into the new installation, copy them to a temporary directory instead.

If the database is also not available, or if you wish to migrate a copy of the database, Falls die Datenbank ebenfalls nicht erreichbar ist oder falls Sie die Migration mit einer Kopie der Datenbank durchführen wollen, you can use the database dumps of the CM and the TE you created at the beginning. After the database server has been set up (if required) and the dumps have been imported, adapt the database configuration in the old instance directories so that the database copies are used. Note that it is neither necessary nor possible to ever start the CMS server from one of the copied old instance directories.

12.1.5 Creating instances in the new Fiona installation

For every existing instance you wish to migrate, create a corresponding instance in the new installation using the original name. To every new instance the port range should be assigned that was assigned to the corresponding old instance. This helps to prevent the configuration of the web applications (which cannot be migrated automatically) and of the the CMS server configuration from becoming inconsistent.

The CMS server configuration is stored in the instance-specific `config/server.xml` file. If a different port range has been assigned to an instance, the server configuration and the configuration of the GUI web application need to be adjusted manually (with respect to the port range) after the migration process. The configuration of the GUI is stored in the instance-specific `webapps/GUI/WEB-INF/basicConfig.properties` file.

Please note that only during the migration of the CM (and not of the TE) the configuration of the TE is copied to the new instance and adapted to it. Thus, if the TE was active in the old installation, its configuration is covered by the migration only if it was run from the instance directory.

12.1.6 Transferring update records

If you are exporting the content incrementally, please ensure that prior to the migration all existing update records have been transferred to the TE. This can be done by running the `SystemPublish` job.

Unprocessed update records cause the migration process to be aborted and a corresponding message to be displayed.

12.1.7 Hard disk space

Please ensure that enough hard disk space is available. Take into account that

- the current and the new CMS Installation will be installed in parallel
- the `script`, `config`, and `data` directories are copied from the old to the new instance.

12.2 In-Place Migration

After the migration requirements have been met, proceed as follows to migrate CMS Fiona:

1. Stop the old CMS server (CM, SES, TE) as well as the Trifork Application Server.
2. Start the migration from the directory of the target instance by calling the CM with the `-migrate` option, passing it the source directory (e.g. `CMS-Fiona-6.0.4/instance/intranet/`) as an argument:

```
$ CMS-Fiona-current/instance/intranet/bin/CM -migrate /opt/Infopark/CMS-Fiona-6.0.4/
instance/intranet
[2010-02-27 09:36:22] (32764) [cm master info] migrate: START
Have you made a backup of the database, in case the migration fails? [Y/n] > y
Does the database user have the permission to alter the database schema? [Y/n] > y
```

3. Since the migration directly modifies the contents of the database, you need to confirm the existence of a backup. Furthermore, it is required that the CMS database user has the permission to modify the database schema. This includes, among other things, the permission to execute `ALTER TABLE`.

   After the questions have been confirmed with `y`, the migration is continued. From the old instance, the directories `config`, `script`, and `data` are copied. This might take some time, depending on the amount of data in the `data` directory. The license file is not copied since the new installation is assumed to have an individual license file.
4. Then several migration steps follow in which the database schema, the data itself, and the system configuration are modified:

```
[2010-02-27 09:36:26] (32764) [cm master info] migrate: Please wait. The migration may
 take a while...
[2010-02-27 09:36:26] (32764) [cm master info] migrate: Copying config, script and
 data directories from old instance
[2010-02-27 10:11:11] (32764) [cm master info] migrate: DB's destination schema
 version = 121
[2010-02-27 10:11:11] (32764) [cm master info] migrate: DB's current schema version =
 89
[2010-02-27 10:11:11] (32764) [cm master info] migrate: Performing migration step 99:
 CopyOldBinConfAndConvertItToTcl
[2010-02-27 10:11:11] (32764) [cm master info] migrate: Performing migration step 100:
 SupportMultipleInstances
[2010-02-27 10:13:46] (32764) [cm master info] migrate: Performing migration step 101:
 AddContentService
...
[2010-02-27 10:15:19] (32764) [cm master info] migrate: END
[2010-02-27 10:15:19] (32764) [cm master info] CM master process terminating
```

The output above indicates that the CM has successfully migrated the database.

5. If you are exporting your content incrementally using the Template Engine, you can now migrate the TE data:

```
$ CMS-Fiona-current/instance/intranet/bin/TE -migrate /opt/Infopark/CMS-Fiona-6.0.4/
instance/intranet
[2010-02-27 13:48:16] (20695) [te master info] migrate: START
Have you made a backup of the database, in case the migration fails? [Y/n] > y
Does the database user have the permission to alter the database schema? [Y/n] > y

[2010-02-27 13:48:21] (20695) [te master info] migrate: Please wait. The migration may
 take a while...
[2010-02-27 13:48:22] (20695) [te master info] migrate: DB's destination schema
 version = 121
[2010-02-27 13:48:22] (20695) [te master info] migrate: DB's current schema version =
 89
[2010-02-27 13:48:22] (20695) [te master info] migrate: Performing migration step 100:
 SupportMultipleInstances
...
[2010-02-27 13:49:04] (20695) [te master info] migrate: END
[2010-02-27 13:49:04] (20695) [te master info] TE master process terminating
```

6. Run the `CM -migrate` command the same way for all your CMS instances.

12.2.1 Migrating a Distributed Installation

For updating a Template Engine operated separately, a complete migration is required, including both the CM and the TE. The reason is that some components common to both TE and CM can only be migrated by performing a CM migration.

Please proceed as follows to migrate the Fiona installation on TE machine:

1. If the CM has not been in operation on the TE system until now, and therefore has no database, please create the database. Afterwards run the CM once to generate the database tables:

```
$ instance/intranet/bin/CM -single
% exit
```

2. Now, perform the migration as described here.

12.2.2 Migrating and Changing the Database Product

If the database product is changed, e.g. from Oracle to MySQL, the the dump/restore method must be used to transfer the content from the old to the new database. This can be done before or after the in-place migration described above. Please proceed as follows:

1. Dump the contents of the instance to be migrated.
2. Configure the target instance to use the target database system.
3. Stellen Sie den Content auf dem Zielsystem wieder her.

12.3 Testing the Migrated Instance or Putting It into Operation

Putting the new installation into operation or testing it requires some further adjustments.

1. Only when migrating to version 6.5.0: In the file `instance/InstanceName/config/rc.npsd.conf` of each instance, set the `triforkHome` entry to the installation path of the new Trifork. Note that the path needs to be enclosed with curly brackets.
2. Contents already exported on the production system using the Template Engine can still be used on the target system. For this, in the new instance, delete the contents of the directories `export/online` and `export/offline`. These directories themselves are symbolic links and need to be preserved.

   Copy or move the contents of the `export/online` and `export/offline` directories from the old instance into the corresponding directories of the new instance.
3. Web applications are not migrated automatically. If, in the old installation, you have made changes to the supplied web applications or added applications for your portlets, for example, please repeat these changes in the new instance. This is also required if the port configuration of the web applications was changed (the XML interface port of the GUI, for example); check `WEB-INF/basicConfig.properties` for each of them. It might also be necessary to adapt the context root path in `META-INF/trifork-app-conf.xml`.
4. The migration has copied the Tcl scripts contained in the `script` directory from the previous instance to the new one. It might be necessary to manually adapt the scripts to the new environment, meaning that absolute paths need to be corrected or Tcl commands need to be modified to reflect the syntax changes made by Infopark. For details on these changes, please refer to the release notes of the CMS version to which you upgraded plus the ones you left out.
5. You can now restart the applications. Please also deploy the web applications:

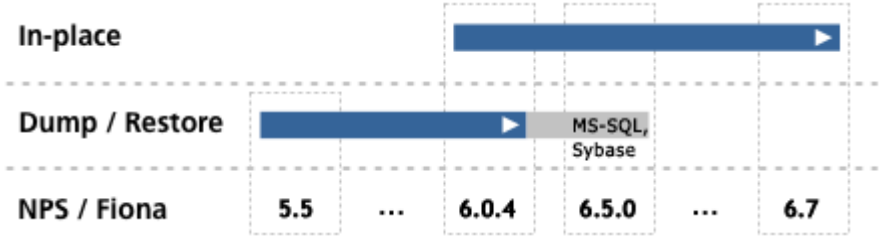
```
$ instance/intranet/bin/rc.npsd start CM SES TE trifork
[...]

$ instance/intranet/bin/rc.npsd deploy
[...]
```

6. To check whether your actions have been successful, open the GUI in the browser: `http://myserver:8080/InstanceName/NPS`. Please note that the format of the GUI URL has changed from `/NPSInstanceName` (up to version 6.0.4) to `/InstanceName/NPS` in version 6.5.
7. If you intend to use the new system as the production system, please adapt your backup tasks accordingly.

12.4 Migrating Older Fiona Versions

Migrating a version of Fiona prior to 6.5.0 may require additional steps to prepare the database for migration.

The method to use in this case depends on the version of the existing installation as well as on the future database system. The following diagram illustrates the migration methods available.



As can be seen from the diagram, it might be necessary to temporarily install an intermediate version of the CMS.

- If you are running a version earlier than 6.0.4, you need to temporarily install version 6.0.4 first and use the dump/restore method for upgrading to this version. Afterwards, you can move to the target version using the in-place migration.
- If the version of your current system is 6.0.4 or later, you can directly upgrade it to the latest version using the in-place migration method, except when using a Sybase or MS SQL database.
- If you operate your CMS with a Sybase or an MS SQL database server, the in-place migration is only available from version 6.5.0. In this case, please use version 6.5.0 as the intermediate version instead of 6.0.4.

If you also wish to switch over to a different database, the dump/restore method must be used to transfer your data from your current to the new database. This can be done before or after the in-place migration is performed.

Updating CMS Fiona from a version prior 6.0.4 (Sybase/MS-SQL: prior to 6.5.0):

1. Backup the content of the instance to migrate.
2. Oracle or MySQL: Install CMS Fiona 6.0.4 on the target system.
   Sybase or MS-SQL: Install CMS Fiona 6.5.0 on the target system.
3. Configure the CMS instance so that it uses the desired database.
4. Restore the content on the target system.
5. Install the latest version of CMS Fiona on the target system.
6. Perform an in-place migration on the target system.
7. Test the new version on the target system.

13

13 The ContentService Interface

13.1 Basic Principles

By means of the Content Service interface available from version 6.5.0, individual CMS files or hierarchy branches can be transferred from one Fiona instance to another one. The Content Service functionality is an extension to Infopark CMS Fiona which is to be licensed separately.

This interface is based on HTTP, i.e. by means of HTTP requests using a well-formed URL you can

- query the list of the available CMS files,
- download a CMS file from a Fiona instance,
- upload a CMS file to a Fiona instance, and
- deactivate a CMS file.

Example:

```
http://localhost:8080/default/NPS/cs/released/index
```

When downloading files, only their released versions are taken into account. When uploading files, these versions are created as draft versions. If a file on the target system already has a draft version, it is updated. Released versions present on the target system are not modified.

The base URL of the interface is `/instanceName/NPS/cs`. The Content Service interface can be reached via the HTTP port of the server application. The method used for authenticating is "Basic Authentication".

An HTTP client and an instance of the Content Management Servers can communicate with each other using the `GET`, `POST` (or `PUT`), and `DELETE` commands.

If the action was performed successfully, the server's response contains the status code 200 (`OK`), otherwise the corresponding error code.

In the following, the four available request types are explained.

13.2 Querying the List of Available CMS Files

Request Syntax

```
GET /default/NPS/cs/released/index
```

Server Response

A UTF-8-encoded text stream containing file paths. Each line of this stream corresponds to exactly one path of a CMS file and consists of four values separated by spaces:

- `lastChanged`: The value of the `lastChanged` field of the released version (time stamp in ISO format, 14 digits) or `00000000000000` if the file does not have a released version.
- `sourceLastChanged`: The value of the `sourceLastChanged` field of the file. The field `sourceLastChanged` is set when a file is created or updated via the Content Service (see below). This causes the `lastChanged` value of the source file to be assigned to the field.
- `objType`: The file type, i.e. one of the values `document`, `publication`, `image`, `generic`, or `template`.
- `path`: The path of the file.

The first line contains the column names.

Example:

```
lastChanged sourceLastChanged objType path
20100211123055 00000000000000 publication /
20100211123055 00000000000000 publication /company
20110103130455 20091111060623 image /company/logo
20110101220723 00000000000000 document /company/information
```

A script could process this response line by line in order to download the CMS files from their respective paths using further requests, and store the files.

13.3 Downloading CMS Files

Request Syntax

```
GET /default/NPS/cs/released/file/Path_to_CMS_file
```

Using URLs according to this scheme returns the CMS file with the path */Path_to_CMS_file* as stream. The stream has the MIME type `application/octet-stream`. If the file has no released version, the stream only includes the relevant file information and no content data.

Request Examples:

`GET /default/NPS/cs/released/file/` returns the base folder.

`GET /default/NPS/cs/released/file/index` also returns the base folder (as opposed to `GET /default/NPS/cs/released/index`).

`GET /default/NPS/cs/released/file/Internet/company/information` returns the file `information` available in the path `/Internet/company`.

13.4 Uploading Files

Request Syntax

```
PUT /default/NPS/cs/edited/file/Path_to_CMS_file
```

This creates the CMS file with the specified path using the stream as the data of the file to be created. If the file already exists, its file information and its draft version are updated.

Alternatively, the HTTP method `POST` can be used.

13.5 Deactivating CMS Files

Request Syntax

```
DELETE /default/NPS/cs/edited/file/Path_to_CMS_file
```

This deactivates the file under the specified path.

13.6 Using the Content Service Interface

The use of the Content Service function available from version 6.5.0 is exemplified by means of the `contentService` Tcl function included in Infopark CMS Fiona. You can find this function and additional hints on how it works in the `/share/script/cm/serverCmds/contentService.tcl` script file.

13.6.1 Setting up the Content Service as a Job

If you wish to transfer content by means of the Content Service function on a regular basis, it is recommended to define a job that calls the `contentService` function with the appropriate arguments. The following sample code transfers the `/news` folder from the `sourceInstance` instance to the `/intranet/content` folder of the `destinationInstance` instance:

```
contentService
  http://localhost:8080/sourceInstance myUser myPassword /news
  http://otherServer:8080/destinationInstance otherUser otherPassword /intranet/content
```

14

14 The Streaming Interface

14.1 Basic Principles

The Content Management Server, the Search Engine Server, and the Template Engine include a so-called streaming interface. This interface allows you to transfer large amounts of data to or from an application without using-up as much RAM as the data occupy on the hard disk. The data is split into several blocks that are transmitted one after the other and written to a file at their destination. This means that the amount of memory consumed during this process is limited to the size of a single block.

There are three types of operations that can be performed using the streaming interface: uploads, downloads, and transferring the output of custom commands. In each of these operations, a unique identifier is used to refer to the data that have been transferred or are to be transferred. For the three types, this means:

- **Upload**

  When files are uploaded to an application, a ticket is issued which must be specified instead of the data in the next operation. After a large blob has been transferred to the Content Management Server, for example, this blob can be assigned to a content by specifying the ticket in a request directed to the XML interface.
- **Download**

  If, for example, a blob is asked for by means of an XML request and the application is allowed to use streaming, a ticket is returned instead of the blob in the response if the amount of data exceeds the configured limit (see tuning). The data can then be fetched from the streaming interface, whereby the ticket is specified (see the example in the next section).
- **Custom Commands**

  Via the GUI (and thus via the XML interface) custom commands can be executed. Since the output of a command is normally viewed in the browser, the answer to the execution request is a streaming ticket. Specifying this ticket, the output of the custom command can continuously be retrieved, i. e. until the command terminates.

The streaming interface can be reached via the HTTP port of the server application. The base URL of the interface is */stream* . For downloads and custom commands the ticket ID follows as another URL component (see the examples in the following sections).

Uploaded or requested date are available for the time that has been configured in the `streamingTicketValidityTimeInterval` system configuration entry in the tuning section. Tickets and the data to which they belong are automatically deleted when their configured maximum age is exceeded.

14.2 Using Streaming

HTTP requests to the streaming interface of a CMS application must use one of the HTTP methods `POST` and `GET`. A request that is executed with another than these two methods is acknowledged with a response containing the HTTP error code 405 (method not allowed).

Uploads

Data is transferred with a `POST` request addressed to the `/stream` URL to a CMS application. For uploading data, the body of the HTTP POST request must contain the data to be transmitted. Keep-Alive is supported.

The application stores the received data in the file system and generates a ticket whose ID is returned to the client. Thus, the body of the HTTP response to the upload request contains the ticket ID only. This response contains the HTTP error code 200 (OK). If, however, an error occurs, the HTTP response does not contain a ticket ID and the HTTP error code 500 is returned (Internal Server Error).

Downloads

All downloads via the streaming interface are HTTP `GET` requests referring to the ticket ID. Therefore, the URL is formed as shown in the following example (specify your server host name instead of `my.cmsserver`):

```
http://my.cmsserver/stream/6528721
```

In the example above, 6528721 is the ticket ID that could have been requested via CRUL and returned in the response to this request:

```
<cm-request ...>
  <obj-where>
    <id>37634</id>
  </obj-where>
  <obj-get>
    <blob encoding="stream"/>
  </obj-get>
</cm-request>


<cm-response ...>
  <cm-code numeric="0" phrase="ok">
    <obj>
      <blob encoding="stream">78239162</blob>
    </obj>
  </cm-code>
</cm-response>
```

Now the blob can be fetched by means of a `GET` request to the streaming interface of the Content Management Server:

```
http://my.cm/stream/78239162
```

If no ticket ID is specified in the download URL or no ticket with the specified ID exists or a ticket which has become invalid is referenced, the HTTP response contains the HTTP error code 404 (Not found). However, if the ticket is valid, the HTTP response contains the HTTP error code 200 (OK) and the body contains the data stored under the ticket. Keep-Alive is also supported here.

Fetching the Output of Custom Commands

Custom commands can be executed by the Content Management Server only. As with downloads, you can fetch the output of such an additional command using streaming by sending a `GET` request to the server, specifying the ticket ID in the URL. This ID is issued when the custom command is executed. Errors resulting from invalid or missing ticket IDs are handled by the Content Management Server in the same manner as with downloads.

If the ticket is valid, the HTTP response contains the HTTP error code 200 (OK) and in the body the output of the custom command to which the ticket refers can be found. Keep-Alive is not supported for requests like these.

If the output of a custom command is integrated into a web page (as is the case with the GUI of the Content Management Server, the CMS Navigator), this page can be written in such a way that it is updated regularly. By this, the user of the page is kept up to date about the progress of the custom command.

To update a page every 15 seconds, use a meta tag according to the following example:

```
<meta http-equiv="Refresh" content="15; url=http://my.cm/stream/7891528">
```

15

15 The XML Interface

15.1 Using HTTP to Access CMS Components

The Content Manager, the Template Engine, the Search Server, the GUI, and the Portal Manager have a HTTP interface used for communicating with other components and applications. The CMS components transfer and receive data via the HTTP interface using HTTP.

An HTTP server is addressed via an URL which the server evaluates. A browser, e. g., connects to a webserver when a URL is entered in order to request the document corresponding to the URL.

Often, not only the path to the document is transferred to an HTTP server by means of the URL, but also a set of parameters. In requests to search engines in particular, the path mainly consists of parameters. A search engine does not return a static document as a response to a search request, but a document which it has created using the parameters in the URL.

The Content Manager uses this mechanism called *URL-encoding* to be able to make various data available to other components. In this way, e.g. the user interface with the following URL can transfer a request document to the NPS Content Manager using the POST command:

```
http://my.cm.server/xml
```

The Content Manager recognizes that the actual request is directed to its XML interface by the `/xml` part of the URL. It consequently expects the request as an XML document in the body of the request sent. An application or a CMS component can therefore address the XML interface of another CMS component via the `/xml` part of the URL.

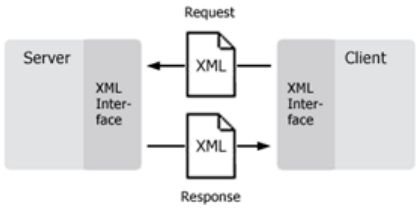
15.2 Communication via XML Documents

Several components of Infopark CMS Fiona have an XML interface. The components can communicate with each other via this interface and can, e.g. exchange information about files and versions.

A CMS component or another application can therefore send a request document to the Content Manager using HTTP in the manner described above. The request is encoded in the body of the request as an XML document. The Content Manager also returns its response as an XML document in the body of its HTTP response.

As a rule, a component can either encode request documents or response documents, however not both. The user interface can therefore send requests to the Content Manager and interpret its responses; however, it cannot create responses to requests. This is not necessary either because it does not maintain data which would be of interest to other components.

The XML documents exchanged by two CMS components follow strict request-response patterns, whereby what is meant by *Request* and *Response* is not request and response on the level of the HTTP protocol. In fact, the XML documents contain request and response XML elements. A component which creates requests in this sense and interprets their responses is a client. Components which can respond to requests are servers. This is illustrated by the figure *Request-Response Pattern*.

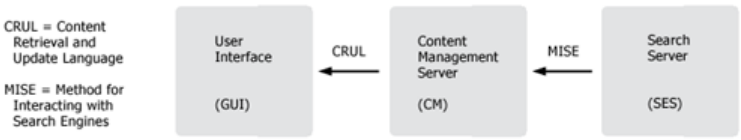


Request/response scheme (Client → Server → Client)

A CMS component can be client and server at the same time. This is the case with the Content Manager, which operates the user interface as a server and appears as a client to the Search Engine Server.

How request and response documents have to be constructed so that the respective components can accept and evaluate them is defined by two DTDs (DTD = *Document Type Definition*). A DTD is a protocol which specifies the hierarchy and the parts of an XML document.

The figure *Client/server relation between CMS components* shows which CMS components can communicate using which protocols (DTDs) and which client-server relationships the components have. The arrows always point from a server to a client.



Client/server releation between CMS components

As can be seen in the figure, the components use the CMS protocols CRUL and MISE.

MISE (*Method for Interacting with Search Engines*) defines the communication between the Content Manager and the Search Engine Server (part of the Infopark Search Cartridge). CRUL is used by the user interface to access the Content Manager. Using CRUL, application developers have full access to all the data administered in the Content Manager. Therefore, the XML Interface is described in the following sections with CRUL as an example. An example with a complete request-response pair can be found in section Examples.

15.3 CRUL Payloads

The XML documents exchanged by CMS components via the XML Interface are called *Payloads*. The `cm-payload` element is the root element of all request and response documents:

```
<!ELEMENT cm-payload (cm-header, (cm-response+ | cm-request+))>
<!ATTLIST cm-payload
  payload-id CDATA #REQUIRED
  timestamp CDATA #REQUIRED
  cm.version CDATA #REQUIRED
>
```

The attributes of `cm-payload` elements have the following meanings:

- `payload-id`
  Payload ID. This ID is generated by the creator of the payload and must be unique within a communication context. Such a context is formed by the Content Management Server (e.g. of a company) and all the clients that communicate with the server. As a rule, the `payload-id` is generated by an algorithm.
- `timestamp`
  Date and time (timestamp) of payload creation. The timestamp must be specified in canonical form as a 14-digit string (beginning on the left: year 4-digit, month 2-digit, day 2-digit, hour 2-digit, minutes 2-digit, seconds 2-digit) in GMT (Example: 20100716020223).
- `cm.version`
  Version of the XML Interface protocol. The construction of the payload is dependent on the version. At the time of writing this manual, the XML Interface protocol has the version number 2.0. You can download versions of CRUL from *www.infopark.com*.

In a request a client indicates which version of the XML Interface protocol it is using by specifying a value for `cm.version`.

The Content Management Server supports – in addition to the current version of the protocol – all versions which were previously valid. If the client is using one of these versions, the server creates a response payload in this version. Otherwise, it responds with an error message which indicates the protocol incompatibility. The server creates this message in its current version of the XML Interface protocol.

15.3.1 Header Element

The first subelement of all payloads is `cm-header`. The `cm-header` element contains information about the identity of the two parties that exchange the payload.

```
<!ELEMENT cm-header (cm-sender, cm-receiver?, cm-authentication?)>
<!ELEMENT cm-sender EMPTY>
<!ATTLIST cm-sender
  sender-id CDATA #REQUIRED
  name CDATA #REQUIRED>
<!ELEMENT cm-receiver EMPTY>
<!ATTLIST cm-receiver
  receiver-id CDATA #REQUIRED
  name CDATA #REQUIRED
>

<!ELEMENT cm-authentication EMPTY>
<!ATTLIST cm-authentication
  login CDATA #REQUIRED
  password CDATA #REQUIRED
>
```

cm-sender

The `cm-sender` element must always appear as the subelement of the `cm-header` element. It specifies the identity of the payload sender. The attributes of the `cm-sender` elements have the following meanings:

- `sender-id`

Payload sender ID. During the installation, each NPS Content Management Server and the clients are allocated an ID which is unique within the communication context. This ID is used in the creation of the payload as the `sender-id`.

- `name`
  Payload sender name. The name is a string which identifies the application which sends the payload. The Content Management Server uses *CM-Server* as the name.

cm-receiver

The `cm-receiver` element specifies the identity of the desired receiver of the payload. This element is optional as long as there is an individual network connection between the server and the client. In this case, the sender of the payload and the receiver are uniquely identified. If, however, between the server and the client there is a proxy server which serves several clients or servers, both the server and the client can use the `cm-receiver` element to inform the proxy server of the receiver. The attributes of the `cm-receiver` elements have the following meanings:

- `receiver-id`
  Payload receiver ID. This attribute has the same semantics as the `sender-id` attribute of the `cm-sender` element. It contains the ID of the NPS server or NPS client which is to receive the payload. The ID is unique within a communication context. For response payloads, this attribute is always a copy of the `sender-id` attribute of the `cm-sender` element in the corresponding request payload.
- `name`
  Name of the receiver. The name is a string which identifies the desired receiver application. For response payloads, this attribute is a copy of the `name` attribute of the `cm-sender` element in the corresponding request payload.

cm-authentication

The `cm-authentication` element is optional. It can only be used in request payloads. It contains information about the user for whom the request is to be processed. The attributes of the `cm-authentication` element have the following meanings:

- `login`
  The login of the user of the Content Management Server.
- `password`
  The password of the user (clear text).

15.3.2 Request Element

For request payloads, one or more `cm-request` elements follow the `cm-header` element in the `cm-payload` root element. These elements specify the operations which are to be executed by the CMS server.

```
<!ELEMENT cm-request (%cm.cm-request;)>
<!ATTLIST cm-request
  request-id CDATA #REQUIRED
  preclusive (true | false) "false">
```

The subelements of the `cm-request` element are defined in CRUL. The corresponding DTD is explained in the XML Reference. A `cm-request` element has the following attributes:

- `request-id`

Request ID. This ID is issued by the creator of the request payload. It must be unique within all payloads that are exchanged in a particular communication context. It is not sufficient that the ID is unique within the current payload.

- `preclusive`
  Truth value (`true` or `false`). The `preclusive` attribute allows the NPS client to mark the requests which are critical for the continued processing of payloads. If the processing of a request marked as `preclusive` fails, all further requests in the payload are not processed but are answered with an error message.

All requests in a payload are processed in sequence beginning with the first request. In this way, it is possible for a client to put together requests that are dependent on each other in one request payload. For example, in a single payload, a client can first create a file format and then create files based on this format. Using the `preclusive` attribute, the client can also ensure that the dependent request is only processed when the previous one has not caused an error.

15.3.3 Response Element

For response payloads, the `cm-payload` root element contains one or more `cm-response` elements after the `cm-header` element with which each result of the operations performed by the server is returned.

If a request payload has been completely recognized and processed by the server, each `cm-response` element in the response payload corresponds to a `cm-request` element in the request payload. In this case, the `cm-response` elements contain messages which refer to the contents of the requests (Request Level Message).

If, on the other hand, the Content Management Server receives an invalid request payload (e.g. without a `cm-header` element or with unrecognizable `cm-request` elements), it returns a response payload containing only one `cm-response` element with which the general error is reported. In this case, the `cm-response` element refers to the payload (Payload Level Message). A `cm-response` element is constructed as follows.

```
<!ELEMENT cm-response (cm-code*)>
<!ATTLIST cm-response
  response-id CDATA #REQUIRED
  payload-id CDATA #IMPLIED
  request-id CDATA #IMPLIED
  success ("true" | "false") #REQUIRED
>
```

The individual responses to the requests are returned in `cm-code` elements.

```
<!ELEMENT cm-code ANY>
<!ATTLIST cm-code
  numeric CDATA #REQUIRED
  phrase CDATA #REQUIRED
>
```

The attributes of the `cm-response` element have the following meanings:

- `response-id`
  Response ID. This ID is set by the creator of the response payload. It must be unique within all payloads that are exchanged in a particular communication context.
- `payload-id`

Request payload ID. It corresponds to the `payload-id` attribute of the request payload. This attribute is added by the server only when the `cm-response` element contains a payload level message in the first `cm-code` element. A client can consequently recognize by the occurrence of this attribute whether its request payload could be interpreted as such by the server (independent of the requests contained in it).

- `request-id`
  Request ID. It corresponds to the `receiver-id` attribute of the `cm-request` element in the request payload. This attribute is only present when the `cm-code` element contains a request level message.
- `success`
  The value of this attribute is `true` when the request could be successfully processed. Otherwise, it is `false`.

Please note that several operations can be performed within a request. In this case, the `success` attribute contains the logical *And* of the results of all operations performed. This means, `success` is only `true` when all operations were successful.

The results of the operations performed are returned with the aid of `cm-code` elements within the `cm-response` element. The `cm-code` elements contain a success or error message and other XML elements which represent the result of the operation or if necessary error information.

The attributes of the `cm-code` element have the following meanings:

- `numeric`
  Error number (or success message number). The messages corresponding to the numbers are described in section Error handling.
- `phrase`
  The description of the error.

The content of the `cm-code` element depends on the operation indicated in the request. In operations which could not be performed successfully, the content of the `cm-code` element depends on the error which occurred. The possible contents of the `cm-code` element are listed for each operation in the XML Reference.

15.4 From Tcl to XML

This section describes how applications can access data managed by the Content Management Server via its XML interface. In the examples below, payloads and requests are only given as fragments. How complete payloads and requests can be built is described in section CRUL Payloads.

The functions available via the XML Interface are mostly the same as those for the Tcl Interface, meaning that you can use either to create files or query the values of version fields, for example. This is possible because the Tcl Interface and the XML parser built into the Content Management Server call the same underlying application functions in most cases.

Classes and Instances

To be able to convert Tcl commands to XML fragments you must know that each group of commands refers to a class and that classes have instances. Therefore, the `attribute` group of commands refers to the class of fields, the `obj` group of commands refers to files, and so on. Instances of these classes are the individual fields or files. Using the commands of a group of commands, you can access a class entirely, e.g. all files, or individual instances. For example, the list of fields is obtained using the following command:

```
attribute list
```

Commands such as `list`, which you use to access an entire class, are called class methods. `create` is also a class method of most classes. An instance is added to a class using `create`.

```
attribute create name fruit type multienum
```

You can see by this example that class methods can have arguments. If you, e.g. add an instance to a class, you must be able to clearly reference the instance. For fields, this is done with the name. The name is a property (also "Parameter") of a field, as is, e.g. the type. In the above example, the method `create` is given the name of a parameter and its value as arguments, in pairs (`name` = `obst`, and `type` = `multienum`). Arguments are specified in this way for many Tcl commands.

Instances

For Tcl commands that refer to individual instances (such as a file or a user), the value of the parameter which clearly identifies the instance must be given. The following Tcl command obtains the password of the user whose login is `smith`:

```
user withLogin smith get password
```

In this command, `user` is the class, and the instance of this class to which the command refers, is clearly referenced by `smith`. A command with which an instance of a class is addressed always has a subcommand. The subcommand specifies the action to be taken with respect to the instance. In the above example, the subcommand `get` is used. The value of an instance variable, here `password`, can be obtained using `get`. In the following example, the parent folder of a file is set:

```
obj withId 4912 set parent aPublicationObjectId
```

Here, the file with the ID 4912 is the instance of the class `obj` to which the subcommand `set` is applied. Subcommands are called instance methods.

Methods in XML

Class methods and instance methods are encoded in CRUL requests as elements. The tag of the corresponding element is built from the class name and the method name, connected by a hyphen:

```
<attribute-create> ... </attribute-create>
```

Using this element, a field is created and the necessary name and values of the instance to be created are specified as subelements:

```
<cm-request ...>
  <attribute-create>
    <name>Address</name>
    <type>string</type>
  </attribute-create></cm-request>
```

As in the corresponding Tcl command (`attribute create name Address type string`), this request creates a response which contains the name of the created field:

```
<cm-response ...>
  <cm-code numeric="0" phrase="ok">
    <attribute>
      <name>Adresse</name>
    </attribute>
  </cm-code></cm-response>
```

In the Tcl Interface, the name of the field created is returned without additions using `attribute create`. In the response documents of the XML Interface, on the other hand, the return value is contained in the class name element (here `attribute`) as a parameter element (here `name`). Basically, the XML interface and the Tcl interface both supply the value of the primary key of the instance created for instancing, e.g. the ID for files, the login for users and the name for workflows.

Instance methods are applied to instances of a class. In the Tcl interface, a file instance can be referenced by `obj withId` or `obj withPath`, a field instance by `attribute withName`, a workflow by `workflow withName`, and so on. The following Tcl code sets the value of the `title` field to `report` for the file with the ID 4812. This is done by means of the `set` instance method.

```
obj withId 4812 set title report
```

Alternatively, you can use the class method `where` to access several instances at once. `where` produces a list of instances in which a given string is part of or equals the value of a parameter. The following code sets the value of the `journal` field to `no` for all files that have the `notice` format:

```
foreach id [obj where objClass Notice]
  {obj withId $id set journal No}
```

In the XML Interface, this Tcl command becomes the following request:

```
<cm-request ...>
  <obj-where>
    <objClass>notice</objClass>
  </obj-where>
  <obj-set>
    <journal>no</journal>
  </obj-set>
</cm-request>
```

Of course, the values of more than one field can be set simultaneously in the `obj-set` element. The `obj-where` element can also have several subelements in order to further restrict the instances to be modified. This makes it possible to refine the example above by restricting the operation to files that have a draft version:

```
<cm-request ...>
  <obj-where>
    <objClass>notice</objClass>
    <state>edited</state>
  </obj-where>
  <obj-set>
    <journal>no</journal>
  </obj-set>
</cm-request>
```

It is important that the class method `obj-where` is immediately followed by an instance method (here `obj-set`). This forms a loop in which the instance method is applied to each instance obtained by the `where` element. For all classes, except `obj`, the `where` element is the only one with which instances can

be obtained. It covers both the `with` logic (`withName`, `withId`, `withPath`) and the `list` class methods of the Tcl Interface. For `obj`, in addition to `where`, the class method `search` exists. `search` enables you to search for CMS files using the Infopark Search Cartridge.

If you search the names of instances in the Tcl Interface using the `where` method, you obtain those instances whose names contain the given string. On the other hand, the following XML fragment supplies only the field instance whose name exactly matches the given string.

```
<cm-request ...>
  <attribute-where>
    <name>attr</name>
  </attribute-where>
  <attribute-set>
    <editField parameter="type">textField</editField>
  </attribute-set>
</cm-request>
```

The reason for this is that the selectors `withName`, `withId` and `withPath` do not exist in the XML Interface. Their function is realized by the `name` parameter in `where` and `search` elements triggering an exact search. However, the parameter `nameLike` is available in order to be able to select instances whose names only contain a string (instead of corresponding to it). Therefore, the following XML fragment deletes all workflows whose names contain `special`:

```
<cm-request ...>
  <workflow-where>
    <nameLike>special</nameLike>
  </workflow-where>
  <workflow-delete/></cm-request>
```

It is also possible to obtain all the instances of a class using `where`. In the Tcl Interface, this task has the class method `list`, which does not exist in the XML Interface: Instead you use an empty `where` element such as in the following XML fragment with which you can release all files (as long as there is no reason that prevents this action):

```
<cm-request ...>
  <obj-where/>
  <obj-release/></cm-request>
```

This can be done using the following code in the Tcl Interface:

```
foreach id [obj list] {obj withId $id release}
```

Reading Instance Variables

Using the instance method `get`, parameters of the instances can be read that were obtained with the previous `where`. In the following example, the value of the `journal` field is queried for all fields of the `notice` file format:

```
<cm-request ...>
  <obj-where>
    <objClass>notice</objClass>
  </obj-where>
  <obj-get>
    <journal/>
  </obj-get>
```

```
</cm-request>
```

All direct and indirect subelements of a `get` element must be empty because the Content Management Server processes the element hierarchy in the `get` element as a template. For each instance obtained using the `where` element, the template is filled out with the corresponding value and inserted in the response request. This is explained in the section Requests as Templates and the Presentation of the Result Data.

Instance Method Arguments

Instances can have arguments, e.g. there is a method called `isOwnerOf` in the Tcl Interface. Using it, you can determine whether a user is the administrator of a user or a user group:

```
user withLogin smith isOwnerOf admins
```

The instance method `isOwnerOf` has the login of a user or a group name as an argument (here the group name `admins`). Arguments are encoded as attributes of the corresponding XML element. The Tcl command above can therefore be translated as follows:

```
<cm-request ...>
  <user-where>
    <login>smith</login>
  </user-where>
  <user-get>
    <isOwnerOf group="admins"/>
  </user-get></cm-request>
```

The above example illustrates an important difference between the Tcl Interface and the XML Interface: You can read-access instances only with the `get` method. As a consequence, all reading instance methods must be used within a `get` element (here `user-get`). Such instance methods look like instance variables (such as `name` for attributes or `realName` for users), they can however differ due to arguments. Thus, `isOwnerOf` has exactly one (variable) argument, namely `group` or `user`. Arguments of instance methods are (as shown in the above example) specified as tag elements (not as subelements).

This process of encoding instance methods as instance variables is identical for all classes. The following example shows how the instance methods `permissionGrantedTo` of the class `obj` are used:

```
<cm-request ...>
  <obj-where>
    <objClass>publication</objClass>
  </obj-where>
  <obj-get>
    <permissionGrantedTo permission="permissionWrite" user="elton"/>
  </obj-get></cm-request>
```

Referencing Subproperties

The class of fields is the only one in the Content Management Server whose instances have subproperties: For each field you can define the input field with which the user enters the value of the field. The definition of an input field consists of five parameters; among others its type, set in the Tcl Interface as follows:

```
attribute withName attr editField set type popup
```

In the XML Interface, the parameters are referenced with the tag attribute `parameter` in the `editfield` tag. The following example shows the equivalent of the above command in the XML Interface:

```
<cm-request ...>
  <attribute-where>
    <name>attr</name>
  </attribute-where>
  <attribute-set>
    <editField parameter="type">popup</editField>
  </attribute-set></cm-request>
```

Analogous to this, input field parameters are retrieved in the following way:

```
<cm-request ...>
  <attribute-where>
    <name>attr</name>
  </attribute-where>
  <attribute-get>
    <editField parameter="type"/>
  </attribute-get></cm-request>
```

Forming Search Requests

If you have installed the Infopark Search Cartridge and have integrated it into the Editorial System, you can search for files in both the Tcl Interface (with the `obj search` command) and in the XML Interface (with `obj-search`). How search requests are structured is described in the Tcl Reference as well as the Search Server documentation and the XML Reference.

The `obj-search` element can be followed by the same elements as `obj-where`, i.e. you can continue to process the file instances found by the search function using instance methods such as `obj-get` or `obj-set`.

Dimensions of Values

Some values of properties are available in the Content Management Server in several languages. Thus, it is possible for versions and other entities to have both a general `title` field and language-specific versions of this field (`title.english` etc.). This "ability" of an instance to have several "interpretations" or versions of a property is called a "dimension". The `title` field of a version or field, a file format and a workflow has the dimension "Language". You can access the versions of properties using a tag attribute in the corresponding XML tag.

"Language" is currently the only property dimension in the Content Manager. It is available in the `title`, `helpText` and the `description` parameters of most classes. The language dimension is encoded using the `lang` tag attribute:

```
<cm-request ...>
  <obj-where>
    <objClass>document</objClass>
  </obj-where>
  <obj-get>
    <title lang="en"/>
  </obj-get></cm-request>
```

Simple and Structured Values

The values of instance properties can be simple or structured values. A simple value is a string, a structured value is a list or a dictionary.

Whereas "string" is a common concept, "list" and "dictionary" require some explanation. A list is an ordered amount of any number of elements. Each element is represented in the XML Interface by a `listitem`:

```
<cm-request ...>
  <attribute-where>
    <name>channels</name>
  </attribute-where>
  <attribute-addEnumValues>
    <listitem>books</listitem>
    <listitem>computing</listitem>
    <listitem>music</listitem>
  </attribute-addEnumValues></cm-request>
```

In the example above, the three possible values `books`, `computing` and `music` are added to the `channels` attribute (we assume that it is of type `enum` or `multienum`). The order in such a list is a result of the order of the list elements. It is not generally relevant.

A dictionary is an unordered amount of any number of allocations of values to names. Each allocation is represented by a `dictitem` subelement which in turn consists of a name (`key`) and a value (`value`):

```
<myDictionary>
  <dictitem>
    <key>name</key>
    <value>theName</value>
  </dictitem>
  <dictitem>
  ...
  </dictitem>
</myDictionary>
```

Dictionaries structured with `dictitem` elements are rarely used in the XML interface (they are used for accessing the user preferences, for example). Most of the name-value pairs are encoded in CRUL using the name as element name and the value as the element's contents.

The value of a dictionary element and a list element can be a string or list or a dictionary.

15.5 Requests as Templates and the Presentation of the Result Data

For each request in a payload, a response is created in the result payload. For requests, which do not query instance properties, but set or create instances, the data are returned which would be returned by the corresponding Tcl command.

```
<cm-request ...>
  <obj-where>
    <id>2001</id>
  </obj-where>
  <obj-create>
    <name>simplepub</name>
    <objClass>publication</objClass>
  </obj-create>
</cm-request>
```

The above request which creates a file immediately below the base folder leads to the following response:

```
<cm-response ...>
  <cm-code numeric="0" phrase="ok">
    <obj>
      <id>32191</id>
    </obj>
  </cm-code>
</cm-response>
```

For requests with which instance properties are to be obtained using `where` and the following `get`, the structure of the request gives the structure of the response. A request is to a certain extent a template which the Content Management Server fills out in order to create the response. In the following example, the request shows that the response is to contain the name and the ID of the selected files.

```
<cm-request ...>
  <obj-where>
    <objClass>document</objClass>
  </obj-where>
  <obj-get>
    <name/>
    <id/>
  </obj-get>
</cm-request>
```

The response document to this request will contain a response in which the name and the ID are supplied for each file matching the criteria in the `obj-where` element:

```
<cm-response ...>
  <cm-code numeric="0" phrase="ok">
    <obj>
      <name>mary</name>
      <id>92674</id>
    </obj>
  </cm-code>
  <cm-code numeric="0" phrase="ok">
    <obj>
      <name>john</name>
      <id>72941</id>
    </obj>
  </cm-code>
</cm-response>
```

In both of these example responses above, you can see that for each instance selected with `where`, a `cm-code` element is created. The queried instance values are encoded as the content of the class element within such an element. In the above example, this is the `obj` element.

Some instance properties are references to instances of the same or a different class. Therefore, e.g. the `parent` of a file (except for the base folder) is a reference to a file instance. The members (`users`) of a user group (`group`) are user references, and the mandatory fields (`mandatoryAttributes`) of a file format are references to fields (`attribute`). As will be shown later, the properties of instances to which the references refer can also be queried. This is done by extending the request template in the `get` element.

The following is an example to show how the files contained in the base folder are queried and encoded in the response:

```
<cm-request ...>
  <obj-where>
    <path>/</path>
  </obj-where>
  <obj-get>
    <children/>
  </obj-get>
</cm-request>
```

The response has the following pattern:

```
<cm-response ...>
  <cm-code numeric="0" phrase="ok">
    <obj>
      <children>
        <listitem>92674</listitem>
        <listitem>72941</listitem>
      </children>
    </obj>
  </cm-code>
</cm-response>
```

In these cases, what is returned are the unique IDs of the instances; i.e. the IDs for files, versions and links and the names for instances of other classes. An ID is not encoded as the content of the corresponding property element, i. e. the ID of a file reference is not the content of an `id` element and the name of a field is not the content of a `name` element. Instead, the IDs are encoded as list elements using `listitem`. See the following example in which those groups are obtained of which a user is a member. The request

```
<cm-request ...>
  <user-where>
    <login>smith</login>
  </user-where>
  <user-get>
    <groups/>
  </user-get>
</cm-request>
```

is followed by a response according to the following pattern:

```
<cm-response ...>
  <cm-code numeric="0" phrase="ok">
    <user>
      <groups>
        <listitem>cmadmins</listitem>
        <listitem>sysadmins</listitem>
      </groups>
    </user>
  </cm-code>
</cm-response>
```

Extended Queries

If the data obtained in a request are instances of a class, you can extend the request to also obtain data from these instances. Therefore, in the above example, the queried groups are instances of the class `group` and parameter values of each of these instances can be queried in the same request.

```
<cm-request ...>
```

```
  <user-where>
    <login>smith</login>
  </user-where>
  <user-get>
    <groups>
      <group>
        <realName/>
        <users>
      </group>
    </groups>
  </user-get>
</cm-request>
```

The above request creates a response according to the following pattern:

```
<cm-response ...>
  <cm-code numeric="0" phrase="ok">
    <user>
      <groups>
        <group>
          <realName>CM Administrators</realName>
          <users>
            <listitem>smith</listitem>
            <listitem>roberts</listitem>
          </users>
        </group>
        <group>
          <realName>System Administrators</realName>
          <users>
            <listitem>smith</listitem>
          </users>
        </group>
      </groups>
    </user>
  </cm-code>
</cm-response>
```

Data about the original instances and instances referenced by them, can therefore be obtained in any depth with a single request. To do this, you encode a property to be queried-- as in the above example-- not as an empty element. Instead you designate as its content the template for the next query.

The following example shows how you obtain the names of folders and their file formats. The allowed formats for files contained in the folder are queried from the file formats and also from these the name and the title in English:

```
<cm-request ...>
  <obj-where>
    <objType>publication</objType>
  </obj-where>
  <obj-get>
    <name/>
    <objClass>
      <name/>
      <validSubObjClasses>
        <name/>
        <title lang="en"/>
      </validSubObjClasses>
    </objClass>
  </obj-get>
</cm-request>
```

This request creates a response according to the following pattern:

```
<cm-response ...>
  <cm-code numeric="0" phrase="ok">
    <obj>
      <name>ROOTPUB</name>
      <objClass>
        <name>publication<name/>
        <validSubObjClasses>
          <objClass>
            <name>publication</name>
            <title lang="en">Standard Publication</title>
          </objClass>
          <objClass>
            <name>document</name>
            <title lang="en">Standard Document</title>
          </objClass>
          ...
        </validSubObjClasses>
      </objClass>
    <obj>
  </cm-code>
  <cm-code numeric="0" phrase="ok"> ... </cm-code>
  <cm-code numeric="0" phrase="ok"> ... </cm-code>
  ...
</cm-response>
```

Please note that a property only becomes a reference to be subsequently followed by not leaving the corresponding element empty but encoding the properties to be obtained in its content. The following request supplies for `publication` files the ID of their respective parent folder and the IDs of the files in this folder (i.e. the original file and its siblings):

```
<cm-request ...>
  <obj-where>
    <objType>publication</objType>
  </obj-where>
  <obj-get>
    <parent/>
    <parent>
      <children/>
    </parent>
  </obj-get>
</cm-request>
```

Whereas the first `parent` element only retrieves the ID because it is empty, the second supplies the properties of the parent file specified in its content, i.e. the `children`.

15.6 Error handling

Error handling is not symmetric in the XML Interface protocol: A CMS server application can transfer error messages to clients in the response payload, however not vice versa. The server applications can thus inform the client that there is an error in the request payload or a request contained therein. A client, however, does not have the possibility to inform the server that a response payload contained an error.

A differentiation is made between protocol errors and those errors which can occur when the Content Manager performs an operation. All errors as well information on how the `cm-code` element is structured in the case of an error can be obtained from the Tcl Reference (currently only available in German). Therefore, only the `cm-code` element of a response to the failed attempt of committing a file is given here by means of an example:

```
<cm-code
  phrase="[100134] cannot commit object"
```

```
  numeric="100134">
  <errorStack>
    <error>
      <numeric>100134</numeric>
      <phrase>[100134] cannot commit object</phrase>
    </error>
    <error>
      <numeric>60029</numeric>
      <phrase>[060029] no edited content</phrase>
    </error>
  </errorStack>
</cm-code>
```

15.7 Examples

The following shows a request payload with two requests and a corresponding response payload. The subelements which may occur in `cm-request` and `cm-response` elements are described in section CRUL Payloads.

Request Payload

The ID and the name of all files with the file format `report` are queried with the first request. The second request creates the `color` field.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE request SYSTEM "http://www.infopark.com/cm.dtd">
<cm-payload payload-id="B42TE241"
            timestamp="20000705020223" cm.version="5.2">
  <cm-header>
    <cm-sender sender-id="FX45RTDT" name="HTMLUI"/>
    <cm-authentication login="holmes" password="apple"/>
  </cm-header>
  <cm-request request-id="H4BPBYE3">
    <obj-where>
      <objClass>report</objClass>
    </obj-where>
    <obj-get>
      <id/>
      <name/>
    </obj-get>
  </cm-request>
  <cm-request request-id="BH423MXA">
    <attribute-create>
      <name>color</name>
      <type>string</type>
    </attribute-create>
  </cm-request>
</cm-payload>
```

Response Payload

The first response in the response document contains the ID and the name of two files as a result. The second response contains the error message that the field already exists.

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCTYPE cm-payload SYSTEM "http://www.infopark.com/cm.dtd">
<cm-payload payload-id="B3BWPOIU"
          timestamp="20000705020224" cm.version="5.2">
  <cm-header>
    <cm-sender sender-id="G33Z4GZU" name="CM SERVER"/>
```

```
  </cm-header>
  <cm-response
            response-id="BR12TI5X"
            request-id="H4BPBYE3"
            success="true">
    <cm-code numeric="0" phrase="ok">
      <obj>
        <id>3123</id>
        <name>reportMay</name>
      </obj>
    </cm-code>
    <cm-code numeric="0" phrase="ok">
      <obj>
        <id>4831</id>
        <name>reportJune</name>
      </obj>
    </cm-code>
  </cm-response>
  <cm-response response-id="AQP3L24V"
               request-id="BH423MXA" success="false">
    <cm-code numeric="1743" phrase="attribute already exits">
      <attribute>color</attribute>
    </cm-code>
  </cm-response>
</cm-payload>
```

16

16 The CRUL DTD

This document contains the definition of CRUL as a DTD. If you need to know which elements and attributes CRUL supports, you can find them here. The DTD is subject to change.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT cm-payload (cm-header, (cm-request+ | cm-response+))>
<!ATTLIST cm-payload
        payload-id CDATA #REQUIRED
        timestamp CDATA #REQUIRED
        cm.version CDATA #REQUIRED
>
<!ENTITY % cm.atom "#PCDATA">
<!ENTITY % cm.dictitem "dictitem">
<!ENTITY % cm.listitem "listitem">
<!ELEMENT listitem (%cm.atom; | %cm.listitem; | %cm.dictitem;)*>
<!ELEMENT dictitem (key, value)>
<!ELEMENT key (%cm.atom;)>
<!ELEMENT value (%cm.atom; | %cm.listitem; | %cm.dictitem;)*>
<!ENTITY % cm.date "
 (%cm.atom; |
  isoDateTime |
   systemConfigFormattedTime |
  userConfigFormattedTime)*
">
<!ENTITY % cm.logEntry "
 (logEntryId,
  execResult,
  execStart,
  execEnd)
">
<!ELEMENT cm-header (cm-sender, cm-receiver?, cm-authentication?)>
<!ELEMENT cm-sender EMPTY>
<!ATTLIST cm-sender
        sender-id CDATA #REQUIRED
        name CDATA #REQUIRED
>
<!ELEMENT cm-receiver EMPTY>
<!ATTLIST cm-receiver
        receiver-id CDATA #REQUIRED
        name CDATA #REQUIRED
>
<!ELEMENT cm-authentication EMPTY>
<!ATTLIST cm-authentication
        login CDATA #REQUIRED
        password CDATA #REQUIRED
        session CDATA #REQUIRED
>

<!ELEMENT cm-response (cm-code*)>
<!ATTLIST cm-response
        response-id CDATA #REQUIRED
        request-id CDATA #IMPLIED
        payload-id CDATA #IMPLIED
        success (true | false) #REQUIRED
```

```
>
<!ELEMENT cm-code ANY>
<!ATTLIST cm-code
        numeric CDATA #REQUIRED
        phrase CDATA #REQUIRED
>
<!ENTITY % cm.cm-request "
 (app-get) |
 (app-execute) |
 (attribute-create) |
 (attribute-types) |
 (attribute-getValidEditFieldKeys) |
 (attribute-where+, attribute-addEnumValues) |
 (attribute-where+, attribute-delete) |
 (attribute-where+, attribute-get) |
 (attribute-where+, attribute-description) |
 (attribute-where+, attribute-removeEnumValues) |
 (attribute-where+, attribute-set) |
 (attributeGroup-create) |
 (attributeGroup-where+, attributeGroup-addAttributes)|
 (attributeGroup-where+, attributeGroup-delete) |
 (attributeGroup-where+, attributeGroup-get) |
 (attributeGroup-where+, attributeGroup-description) |
 (attributeGroup-where+, attributeGroup-moveAttribute) |
 (attributeGroup-where+, attributeGroup-moveToIndex) |
 (attributeGroup-where+, attributeGroup-removeAttributes) |
 (attributeGroup-where+, attributeGroup-set) |
 (channel-create) |
 (channel-where+, channel-delete) |
 (channel-where+, channel-description) |
 (channel-where+, channel-get) |
 (channel-where+, channel-set) |
 (content-where+, content-addLinkTo) |
 (content-where+, content-delete) |
 (content-where+, content-get) |
 (content-where+, content-description) |
 (content-where+, content-load) |
 (content-where+, content-resolveRefs) |
 (content-where+, content-set) |
 (content-where+, content-debugExport) |
 (contentService-editedOverview) |
 (contentService-releasedOverview) |
 (customCommand-execute) |
 (group-create) |
 (group-where+, group-addUsers) |
 (group-where+, group-delete) |
 (group-where+, group-get) |
 (group-where+, group-description) |
 (group-where+, group-grantGlobalPermissions) |
 (group-where+, group-removeUsers) |
 (group-where+, group-revokeGlobalPermissions) |
 (group-where+, group-set) |
 (groupProxy-where+, groupProxy-get) |
 (incrExport-get) |
 (incrExport-getUpdateData) |
 (incrExport-removeUpdateRecords) |
 (incrExport-reset) |
 (job-create) |
 (job-listQueue) |
 (job-where+, job-cancel) |
 (job-where+, job-delete) |
 (job-where+, job-exec) |
 (job-where+, job-get) |
 (job-where+, job-getLogEntry) |
 (job-where+, job-getOutput) |
 (job-where+, job-description) |
 (job-where+, job-set) |
 (licenseManager-license) |
 (licenseManager-licenseExpirationDate) |
 (licenseManager-licenseType) |
 (licenseManager-loginCount) |
 (licenseManager-logins) |
```

```
  (licenseManager-logout) |
  (licenseManager-maxConcurrentUsers) |
  (link-create) |
  (link-where+, link-delete) |
  (link-where+, link-get) |
  (link-where+, link-description) |
  (link-where+, link-set) |
  (logEntry-where+, logEntry-delete) |
  (logEntry-where+, logEntry-get) |
  (obj-contentTypesForObjType) |
  (obj-generatePreview) |
  (obj-search+, obj-addComment) |
  (obj-search+, obj-commit) |
  (obj-search+, obj-copy) |
  (obj-search+, obj-create) |
  (obj-search+, obj-delete) |
  (obj-search+, obj-edit) |
  (obj-search+, obj-exportSubtree) |
  (obj-search+, obj-forward) |
  (obj-search+, obj-get) |
  (obj-search+, obj-description) |
  (obj-search+, obj-give) |
  (obj-search+, obj-mirror) |
  (obj-search+, obj-permissionGrantTo) |
  (obj-search+, obj-permissionRevokeFrom) |
  (obj-search+, obj-reject) |
  (obj-search+, obj-release) |
  (obj-search+, obj-removeActiveContents) |
  (obj-search+, obj-removeArchivedContents) |
  (obj-search+, obj-revert) |
  (obj-search+, obj-set) |
  (obj-search+, obj-sign) |
  (obj-search+, obj-take) |
  (obj-search+, obj-unrelease) |
  (obj-types) |
  (obj-where+, obj-addComment) |
  (obj-where+, obj-commit) |
  (obj-where+, obj-copy) |
  (obj-where+, obj-create) |
  (obj-where+, obj-delete) |
  (obj-where+, obj-edit) |
  (obj-where+, obj-exportSubtree) |
  (obj-where+, obj-forward) |
  (obj-where+, obj-get) |
  (obj-where+, obj-description) |
  (obj-where+, obj-give) |
  (obj-where+, obj-mirror) |
  (obj-where+, obj-permissionGrantTo) |
  (obj-where+, obj-permissionRevokeFrom) |
  (obj-where+, obj-reject) |
  (obj-where+, obj-release) |
  (obj-where+, obj-removeActiveContents) |
  (obj-where+, obj-removeArchivedContents) |
  (obj-where+, obj-revert) |
  (obj-where+, obj-set) |
  (obj-where+, obj-sign) |
  (obj-where+, obj-take) |
  (obj-where+, obj-unrelease) |
  (objClass-create) |
  (objClass-goodAvailableBlobEditorsForObjType) |
  (objClass-validAttributes) |
  (objClass-where+, objClass-delete) |
  (objClass-where+, objClass-get) |
  (objClass-where+, objClass-description) |
  (objClass-where+, objClass-set) |
  (systemConfig-formatDate) |
  (systemConfig-formatDateTime) |
  (systemConfig-getAttributes) |
  (systemConfig-getCounts) |
  (systemConfig-getElements) |
  (systemConfig-getKeys) |
  (systemConfig-getTexts) |
```

```
  (systemConfig-parseInputDate) |
  (systemConfig-installedLanguages) |
  (systemConfig-validInputCharsets) |
  (systemConfig-validTimeZones) |
  (task-where+, task-get) |
  (task-where+, task-description) |
  (user-create) |
  (user-where+, user-addToGroups) |
  (user-where+, user-delete) |
  (user-where+, user-get) |
  (user-where+, user-description) |
  (user-where+, user-grantGlobalPermissions) |
  (user-where+, user-removeFromGroups) |
  (user-where+, user-revokeGlobalPermissions) |
  (user-where+, user-set) |
  (user-writeUserFile) |
  (userAttribute-create) |
  (userAttribute-types) |
  (userAttribute-where+, userAttribute-addEnumValues) |
  (userAttribute-where+, userAttribute-delete) |
  (userAttribute-where+, userAttribute-get) |
  (userAttribute-where+, userAttribute-description) |
  (userAttribute-where+, userAttribute-removeEnumValues) |
  (userAttribute-where+, userAttribute-set) |
  (userConfig-getAll) |
  (userConfig-formatDate) |
  (userConfig-formatDateTime) |
  (userConfig-getAttributes) |
  (userConfig-getCounts) |
  (userConfig-getElements) |
  (userConfig-getKeys) |
  (userConfig-getTexts) |
  (userConfig-parseInputDate) |
  (userConfig-removeAttributes) |
  (userConfig-removeKeys) |
  (userConfig-setAttributes) |
  (userConfig-setElements) |
  (userConfig-setTexts) |
  (userProxy-where+, userProxy-get) |
  (workflow-create) |
  (workflow-where+, workflow-delete) |
  (workflow-where+, workflow-get) |
  (workflow-where+, workflow-description) |
  (workflow-where+, workflow-set)
">

<!ELEMENT cm-request (%cm.cm-request;)>
<!ATTLIST cm-request
        request-id CDATA #REQUIRED
        preclusive (true | false) "false"
>

<!ENTITY % cm.app-get "
  (appName |
   baseDir |
   binDir |
   configDir |
   dataDir |
   debugChannels |
   debugLevel |
   getKeys |
   instanceDir |
   libDir |
   logDir |
   now |
   rootConfigPath |
   scriptDir |
   shareDir |
   timeZone |
   tmpDir |
   today |
   version)*
```

```
">
<!ENTITY % cm.app-execute "
 (command,
  arguments)
">
<!ENTITY % cm.attribute-addEnumValues "
 (listitem+)
">
<!ENTITY % cm.attribute-create "
 (callback |
  helpText |
  isSearchableInCM |
  isSearchableInTE |
  maxSize |
  minSize |
  name |
  title |
  type |
  values |
  wantedTags |
  editField)*
">
<!ENTITY % cm.attribute-delete "EMPTY">
<!ENTITY % cm.attribute-get "
 callback |
  displayTitle |
  displayValueCallback |
  editField |
  editFieldSpec |
  getKeys |
  helpText |
  isSearchableInCM |
  isSearchableInTE |
  localizedTitle |
  localizedHelpText |
  maxSize |
  minSize |
  name |
  setKeys |
  title |
  type |
  validEditFieldKeys |
  validEditFieldTypes |
  values |
  wantedTags
">
<!ENTITY % cm.attribute-getValidEditFieldKeys "(listitem)*">
<!ENTITY % cm.attribute-removeEnumValues "
 (listitem+)
">
<!ENTITY % cm.attribute-set "
 (callback |
  displayValueCallback |
  helpText |
  isSearchableInCM |
  isSearchableInTE |
  maxSize |
  minSize |
  title |
  type |
  values |
  wantedTags |
  editField)*
">
<!ENTITY % cm.attribute-types "(listitem)*">
<!ENTITY % cm.attribute-where "
 (name |
 (nameLike?,
  type?))?
">
<!ENTITY % cm.attributeGroup-addAttributes "(listitem)*">
<!ENTITY % cm.attributeGroup-create "
```

```
  (objClass,
   name,
   title?,
   index?)
">
<!ENTITY % cm.attributeGroup-delete "EMPTY">
<!ENTITY % cm.attributeGroup-get "
 attributes |
  displayTitle |
  localizedTitle |
  getKeys |
  index |
  isDefaultGroup |
  isEmpty |
  name |
  objClass |
  setKeys |
  title
">
<!ENTITY % cm.attributeGroup-moveAttribute "
 (attribute, index)
">
<!ENTITY % cm.attributeGroup-moveToIndex "
 (index)
">
<!ENTITY % cm.attributeGroup-removeAttributes "
 (listitem+)
">
<!ENTITY % cm.attributeGroup-set "
 (title)*
">
<!ENTITY % cm.attributeGroup-where "
 (objClass)
">
<!ENTITY % cm.channel-create "
 (name,
  title*)
">
<!ENTITY % cm.channel-delete "EMPTY">
<!ENTITY % cm.channel-get "
 (name |
  title)*
">
<!ENTITY % cm.channel-set "
 (name |
  title)*
">
<!ENTITY % cm.channel-where "
 (name |
  namePrefix)*
">
<!ENTITY % cm.content-addLinkTo "
 (attribute,
  destinationUrl)*
">
<!ENTITY % cm.content-delete "EMPTY">
<!ENTITY % cm.content-get "
  anchors |
  blob |
  blobLength |
  body |
  bodyTemplateName |
  codeForSourceView |
  contentType |
  displayTitle |
  editor |
  exportBlob |
  exportFiles |
  externalAttrNames |
  externalAttributes |
  frameNames |
  freeLinks |
```

```
   getKeys |
   id |
   isActive |
   isCommitted |
   isComplete |
   isEdited |
   isReleased |
   lastChanged |
   linkListAttributes |
   mimeType |
   nextEditGroup |
   nextSignGroup |
   objectId |
   reasonsForIncompleteState |
   relatedLinks |
   setKeys |
   signatureAttrNames |
   signatureAttributes |
   sortKey1 |
   sortKey2 |
   sortKey3 |
   sortKeyLength1 |
   sortKeyLength2 |
   sortKeyLength3 |
   sortOrder |
   sortType1 |
   sortType2 |
   sortType3 |
   subLinks |
   textLinks |
   thumbnail |
   title |
   validFrom |
   validSortKeys |
   validSortOrders |
   validSortTypes |
   validUntil |
   workflowComment |
   xmlBlob
">
<!ENTITY % cm.content-load "
 (blob |
   contentType |
   filter)*
">
<!ENTITY % cm.content-resolveRefs "EMPTY">
<!ENTITY % cm.content-set "
 (contentType |
   sortKey1 |
   sortKey2 |
   sortKey3 |
   sortKeyLength1 |
   sortKeyLength2 |
   sortKeyLength3 |
   sortOrder |
   sortType1 |
   sortType2 |
   sortType3 |
   title |
   validFrom |
   validUntil)*
">
<!ENTITY % cm.content-where "
 (id |
 (objectId , state))
">
<!ENTITY % cm.group-addUsers "
 (listitem+)
">
<!ENTITY % cm.group-create "
 (globalPermissions |
   name |
```

```
   owner |
   realName)*
">
<!ENTITY % cm.group-delete "EMPTY">
<!ENTITY % cm.group-get "
 displayTitle |
   getKeys |
   globalPermissions |
   hasGlobalPermission |
   users |
   name |
   owner |
   realName |
   setKeys
">
<!ENTITY % cm.group-grantGlobalPermissions "
 (listitem+)
">
<!ENTITY % cm.group-removeUsers "
 (listitem+)
">
<!ENTITY % cm.group-revokeGlobalPermissions "
 (listitem+)
">
<!ENTITY % cm.group-set "
 (globalPermissions |
   owner |
   realName |
   users)*
">
<!ENTITY % cm.group-where "
 (name |
   groupText)?
">
<!ENTITY % cm.incrExport-get "
 (getKeys |
   mode |
   updateRecords |
   updateRecordCount)*
">
<!ENTITY % cm.incrExport-getUpdateData "
 (updateData?)
">
<!ENTITY % cm.incrExport-removeUpdateRecords "
 (updateRecordId+ |
   all)
">
<!ENTITY % cm.incrExport-reset "EMPTY">
<!ENTITY % cm.job-cancel "EMPTY">
<!ENTITY % cm.job-create "
 (name,
   comment?,
   execLogin?,
   execPerm?,
   schedule?,
   script?,
   title?)
">
<!ENTITY % cm.job-delete "EMPTY">
<!ENTITY % cm.job-exec "EMPTY">
<!ENTITY % cm.job-get "
 (category |
   comment |
   displayTitle |
   execLogin |
   execPerm |
   getKeys |
   id |
   isActive |
   lastExecEnd |
   lastExecResult |
   lastExecStart |
```

```
   lastLogEntry |
   lastOutput |
   log |
   logEntries |
   name |
   nextExecStart |
   queuePos |
   schedule |
   script |
   setKeys |
   title)*
">
<!ENTITY % cm.job-getLogEntry "%cm.logEntry;">
<!ATTLIST job-getLogEntry
  id CDATA #REQUIRED
>
<!ENTITY % cm.job-getOutput "(%cm.atom;)">
<!ENTITY % cm.job-listQueue "(%cm.listitem;)*">
<!ATTLIST job-listQueue
  category CDATA #IMPLIED
>

<!ENTITY % cm.job-set "
 (comment |
  execLogin |
  execPerm |
  isActive |
  schedule |
  script |
  title)*
">
<!ENTITY % cm.job-where "
 (category |
  comment |
  execLogin |
  id |
  isActive |
  isQueued |
  name |
  title)*
">
<!ENTITY % cm.licenseManager-license "(%cm.atom;)">
<!ENTITY % cm.licenseManager-licenseExpirationDate "(%cm.atom;)">
<!ENTITY % cm.licenseManager-licenseType "(%cm.atom;)">
<!ENTITY % cm.licenseManager-loginCount "(%cm.atom;)">
<!ENTITY % cm.licenseManager-logins "(listitem*)">
<!ENTITY % cm.licenseManager-logout "EMPTY">
<!ENTITY % cm.licenseManager-maxConcurrentUsers "(%cm.atom;)">
<!ENTITY % cm.link-create "
 (attributeName |
  destinationUrl |
  sourceContent)*
">
<!ENTITY % cm.link-delete "EMPTY">
<!ENTITY % cm.link-get "
 (attributeName |
  canHaveAnchor |
  canHaveTarget |
  destination |
  destinationUrl |
  displayTitle |
  expectedPath |
  getKeys |
  id |
  isComplete |
  isExternalLink |
  isFreeLink |
  isIncludeLink |
  isInlineReferenceLink |
  isLinkFromCommittedContent |
  isLinkFromEditedContent |
  isLinkFromReleasedContent |
```

```
   isRelatedLink |
   isWritable |
   setKeys |
   source |
   sourceContent |
   sourceTagAttribute |
   sourceTagName |
   target |
   title)*
">
<!ENTITY % cm.link-set "
 (destinationUrl | attributeName |
   target |
   title)*
">
<!ENTITY % cm.link-where "
 (id?)
">
<!ENTITY % cm.logEntry-delete "(deleteLogEntriesCount?)">
<!ENTITY % cm.logEntry-get "
 (logTime |
   logText |
   logType |
   objectId |
   receiver |
   userLogin)*
">
<!ENTITY % cm.logEntry-where "
 (fromDate?,
   logText?,
   logTypes?,
   objectId?,
   receiver?,
   untilDate?,
   userLogin?)
">
<!ENTITY % cm.obj-commit "
 (comment)
">
<!ENTITY % cm.obj-contentTypesForObjType "(listitem*)">
<!ENTITY % cm.obj-copy "
 ((parent | name)+,
   recursive?)
">
<!ENTITY % cm.obj-create "
 (blob |
   contentType |
   file |
   filter |
   name |
   objClass |
   suppressExport |
   xmlBlob)*
">
<!ENTITY % cm.obj-deactivate "EMPTY">
<!ENTITY % cm.obj-delete "EMPTY">
<!ENTITY % cm.obj-edit "
 (comment,
   contentId?)
">
<!ENTITY % cm.obj-exportSubtree "
 (absoluteFsPrefix |
 absoluteUrlPrefix |
 exportCharset |
 filePrefix |
 templateName |
 purgeCollections)*
">
<!ENTITY % cm.obj-forward "
 (comment)
">
<!ENTITY % cm.obj-generatePreview "EMPTY">
```

```
<!ENTITY % cm.obj-get "
  archivedContents |
  children |
  committedContentId |
  contentIds |
  contents |
  editedContentId |
  editedContent |
  exportMimeType |
  getKeys |
  hasChildren |
  hasMirrors |
  hasSuperLinks |
  hierarchy |
  id |
  isCommitted |
  isEdited |
  isExportable |
  isMirror |
  isReleased |
  isRoot |
  isGoodDestination |
  isGoodParent |
  name |
  next |
  objClass |
  objType |
  objectsToRoot |
  parent |
  path |
  permission |
  permissionGrantedTo |
  prefixPath |
  previous |
  releasedContentId |
  releasedContent |
  releasedVersions |
  rootPermissionFor |
  setKeys |
  sortValue |
  superLinks |
  superObjects |
  suppressExport |
  toclist |
  validControlActionKeys |
  validCreateObjClasses |
  validObjClasses |
  validPermissions |
  version |
  visibleExportTemplates |
  visibleName |
  visiblePath |
  workflowName
">
<!ENTITY % cm.obj-give "(comment, (groupName | userLogin))">
<!ENTITY % cm.obj-mirror "
 (parent |
  name)*
">
<!ENTITY % cm.obj-permissionGrantTo "
 (group+)
">
<!ENTITY % cm.obj-permissionRevokeFrom "
 (group+)
">
<!ENTITY % cm.obj-reject "
 (comment)
">
<!ENTITY % cm.obj-release "
 (comment)
">
<!ENTITY % cm.obj-removeActiveContents "EMPTY">
```

```
<!ENTITY % cm.obj-removeArchivedContents "EMPTY">
<!ENTITY % cm.obj-revert "
 (comment)
">
<!ENTITY % cm.obj-search "
 (query |
  minRelevance |
  maxDocs |
  offsetStart |
  offsetLength |
  parser |
  collections)*
">
<!ENTITY % cm.obj-set "
 (suppressExport |
  name |
  objClass |
  parent |
  permission |
  workflowName)*
">
<!ENTITY % cm.obj-sign "
 (comment)
">
<!ENTITY % cm.obj-take "
 (comment)
">
<!ENTITY % cm.obj-types "(listitem)*">
<!ENTITY % cm.obj-unrelease "
 (comment)
">
<!ENTITY % cm.obj-where "
 (id |
  ids |
  path |
  parent |
  condition)*
">
<!ENTITY % cm.objClass-create "
 (attributes |
  bodyTemplateName |
  completionCheck |
  createPermission |
  externalEditExtension |
  externalEditMimeType |
  isEnabled |
  mandatoryAttributes |
  name |
  objType |
  presetAttributes |
  presetFromParentAttributes |
  recordSetCallback |
  title |
  validContentTypes |
  validSubObjClassCheck |
  validSubObjClasses |
  workflowModification)*
">
<!ENTITY % cm.objClass-delete "EMPTY">
<!ENTITY % cm.objClass-get "
 attributeGroups |
  attributes |
  availableBlobEditors |
  bodyTemplateName |
  completionCheck |
  createPermission |
  customBlobEditorUrl |
  defaultAttributeGroup |
  displayTitle |
  emptyAttributeGroups |
  externalEditExtension |
  externalEditMimeType |
```

```
   getKeys |
   goodAttributeGroupAttributes |
   goodAttributes |
   goodMandatoryAttributes |
   goodPresetAttributes |
   goodPresetFromParentAttributes |
   isEnabled |
   localizedTitle |
   mandatoryAttributes |
   name |
   objType |
   presetAttributes |
   presetFromParentAttributes |
   recordSetCallback |
   setKeys |
   title |
   validContentTypes |
   validSubObjClassCheck |
   validSubObjClasses |
   workflowModification |
   xmldtd
">
<!ENTITY % cm.objClass-goodAvailableBlobEditorsForObjType "(listitem*)">
<!ENTITY % cm.objClass-set "
 (attributes |
  availableBlobEditors |
  bodyTemplateName |
  completionCheck |
  createPermission |
  customBlobEditorUrl |
  externalEditExtension |
  externalEditMimeType |
  isEnabled |
  mandatoryAttributes |
  presetAttributes |
  presetFromParentAttributes |
  recordSetCallback |
  title |
  validContentTypes |
  validSubObjClassCheck |
  validSubObjClasses |
  workflowModification)*
">
<!ENTITY % cm.objClass-validAttributes "(listitem)*">
<!ENTITY % cm.objClass-where "
 (name |
 (nameLike?,
  isEnabled?,
  objType?))?
">
<!ENTITY % cm.systemConfig-getAttributes "(key, attributeNames?)">
<!ENTITY % cm.systemConfig-getCounts "(listitem+)">
<!ENTITY % cm.systemConfig-getElements "(listitem+)">
<!ENTITY % cm.systemConfig-getKeys "(listitem+)">
<!ENTITY % cm.systemConfig-getTexts "(listitem+)">
<!ENTITY % cm.systemConfig-installedLanguages "(listitem+)">
<!ENTITY % cm.systemConfig-validTimeZones "(listitem+)">
<!ENTITY % cm.systemConfig-validInputCharsets "(listitem+)">
<!ENTITY % cm.task-delete "EMPTY">
<!ENTITY % cm.task-get "
 (comment |
  displayTitle |
  getKeys |
  groupName |
  objectId |
  taskType |
  timeStamp |
  title |
  userLogin)*
">
<!ENTITY % cm.task-where "
 (id |
```

```
  (objectId,
   userLogin?,
   groupNames?,
   taskText?,
   taskType?))?
">
<!ENTITY % cm.user-addToGroups "
 (listitem+)
">
<!ENTITY % cm.user-create "
 (defaultGroup |
  encryptedPassword |
  email |
  globalPermissions |
  login |
  groups |
  owner |
  password |
  realName |
  userLocked)*
">
<!ENTITY % cm.user-delete "EMPTY">
<!ENTITY % cm.user-get "
 defaultGroup |
  displayTitle |
  encryptedPassword |
  email |
  externalUserAttrNames |
  getKeys |
  globalPermissions |
  groups |
  hasGlobalPermission |
  isSuperUser |
  isOwnerOf |
  hasPassword |
  login |
  owner |
  realName |
  setKeys |
  userLocked |
  externalAttrNames
">
<!ENTITY % cm.user-grantGlobalPermissions "
 (listitem+)
">
<!ENTITY % cm.user-removeFromGroups "
 (listitem+)
">
<!ENTITY % cm.user-revokeGlobalPermissions "
 (listitem+)
">
<!ENTITY % cm.user-set "
 (defaultGroup |
  encryptedPassword |
  email |
  globalPermissions |
  groups |
  owner |
  password |
  realName |
  userLocked)*
">
<!ENTITY % cm.user-where "
 (login |
  userText)
">
<!ENTITY % cm.user-writeUserFile "EMPTY">
<!ENTITY % cm.userAttribute-addEnumValues "
 (listitem+)
">
<!ENTITY % cm.userAttribute-create "
 (name |
```

```
  type)*
">
<!ENTITY % cm.userAttribute-delete "EMPTY">
<!ENTITY % cm.userAttribute-get "
 (displayTitle |
  getKeys |
  name |
  setKeys        |
  type |
  values)*
">
<!ENTITY % cm.userAttribute-removeEnumValues "
 (listitem+)
">
<!ENTITY % cm.userAttribute-set "
 (type |
  values)*
">
<!ENTITY % cm.userAttribute-types "(listitem)*">
<!ENTITY % cm.userAttribute-where "
 (name |
 (nameLike?,
  type?))?
">
<!ENTITY % cm.userConfig-getAll "(%cm.atom;)">
<!ENTITY % cm.userConfig-getAttributes "(key, attributeNames?)">
<!ENTITY % cm.userConfig-getCounts "(listitem+)">
<!ENTITY % cm.userConfig-getElements "(listitem+)">
<!ENTITY % cm.userConfig-getKeys "(listitem+)">
<!ENTITY % cm.userConfig-getTexts "(listitem+)">
<!ENTITY % cm.userConfig-removeAttributes "(key, attributeNames?)">
<!ENTITY % cm.userConfig-removeKeys "(listitem*)">
<!ENTITY % cm.userConfig-setAttributes "(key, attributes?)">
<!ENTITY % cm.userConfig-setElements "(dictitem+)">
<!ENTITY % cm.userConfig-setTexts "(dictitem+)">
<!ENTITY % cm.workflow-create "
 (name,
  editGroups?,
  signatureDefs?,
  allowsMultipleSignatures?,
  isEnabled?,
  title?)
">
<!ENTITY % cm.workflow-delete "EMPTY">
<!ENTITY % cm.workflow-get "
 name |
  displayTitle |
  editGroups |
  signatureDefs |
  allowsMultipleSignatures |
  getKeys |
  isEnabled |
  setKeys |
  title
">
<!ENTITY % cm.workflow-set "
 (editGroups |
  signatureDefs |
  allowsMultipleSignatures |
  getKeys |
  isEnabled |
  setKeys |
  title)*
">
<!ENTITY % cm.workflow-where "
 (name |
  nameLike)*
">

<!ELEMENT app-get %cm.app-get;>
<!ELEMENT app-execute %cm.app-execute;>
<!ELEMENT attribute-addEnumValues %cm.attribute-addEnumValues;>
```

```
<!ELEMENT attribute-create %cm.attribute-create;>
<!ELEMENT attribute-delete %cm.attribute-delete;>
<!ELEMENT attribute-get (%cm.attribute-get;)*>
<!ELEMENT attribute-description (%cm.atom;)>
<!ELEMENT attribute-getValidEditFieldKeys %cm.attribute-getValidEditFieldKeys;>
<!ATTLIST attribute-getValidEditFieldKeys
        type CDATA #REQUIRED
>
<!ELEMENT attribute-removeEnumValues %cm.attribute-removeEnumValues;>
<!ELEMENT attribute-set %cm.attribute-set;>
<!ELEMENT attribute-types %cm.attribute-types;>
<!ELEMENT attribute-where %cm.attribute-where;>
<!ATTLIST attribute-where
    maxResults CDATA #IMPLIED
>
<!ELEMENT attributeGroup-addAttributes %cm.attributeGroup-addAttributes;>
<!ELEMENT attributeGroup-create %cm.attributeGroup-create;>
<!ELEMENT attributeGroup-delete %cm.attributeGroup-delete;>
<!ELEMENT attributeGroup-get (%cm.attributeGroup-get;)*>
<!ELEMENT attributeGroup-description (%cm.atom;)>
<!ELEMENT attributeGroup-moveAttribute %cm.attributeGroup-moveAttribute;>
<!ELEMENT attributeGroup-moveToIndex %cm.attributeGroup-moveToIndex;>
<!ELEMENT attributeGroup-removeAttributes %cm.attributeGroup-removeAttributes;>
<!ELEMENT attributeGroup-set %cm.attributeGroup-set;>
<!ELEMENT attributeGroup-where %cm.attributeGroup-where;>
<!ELEMENT channel-create %cm.channel-create;>
<!ELEMENT channel-delete %cm.channel-delete;>
<!ELEMENT channel-description (%cm.atom;)>
<!ELEMENT channel-get %cm.channel-get;>
<!ELEMENT channel-set %cm.channel-set;>
<!ELEMENT channel-where %cm.channel-where;>
<!ATTLIST channel-where
    maxResults CDATA #IMPLIED
>
<!ELEMENT content-addLinkTo %cm.content-addLinkTo;>
<!ELEMENT content-delete %cm.content-delete;>
<!ELEMENT content-get (%cm.content-get;)+>
<!ELEMENT content-description (%cm.atom;)>
<!ELEMENT content-load %cm.content-load;>
<!ELEMENT content-resolveRefs %cm.content-resolveRefs;>
<!ELEMENT content-set %cm.content-set;>
<!ELEMENT content-where %cm.content-where;>
<!ELEMENT contentService-editedOverview (%cm.atom;)>
<!ATTLIST contentService-editedOverview
    encoding (plain | base64 | stream) #IMPLIED
>
<!ELEMENT contentService-releasedOverview (%cm.atom;)>
<!ATTLIST contentService-releasedOverview
    encoding (plain | base64 | stream) #IMPLIED
>
<!ELEMENT customCommand-execute (command, arguments?)>
<!ELEMENT content-debugExport ANY>
<!ATTLIST content-debugExport
  quoteHtml (true | false) "false"
  htmlPrefix CDATA #IMPLIED
  htmlSuffix CDATA #IMPLIED
  infoPrefix CDATA #IMPLIED
  infoSuffix CDATA #IMPLIED
  errorPrefix CDATA #IMPLIED
  errorSuffix CDATA #IMPLIED
  detailed (true | false) "false"
  preferEditedTemplates (true | false) #IMPLIED
  allowEditedContents (true | false) #IMPLIED
  templateName CDATA #IMPLIED
>
<!ELEMENT group-addUsers %cm.group-addUsers;>
<!ELEMENT group-create %cm.group-create;>
<!ELEMENT group-delete %cm.group-delete;>
<!ELEMENT group-get (%cm.group-get;)>
<!ELEMENT group-description (%cm.atom;)>
<!ELEMENT group-grantGlobalPermissions %cm.group-grantGlobalPermissions;>
<!ELEMENT group-removeUsers %cm.group-removeUsers;>
```

```
<!ELEMENT group-revokeGlobalPermissions %cm.group-revokeGlobalPermissions;>
<!ELEMENT group-set %cm.group-set;>
<!ELEMENT group-where %cm.group-where;>
<!ATTLIST group-where
    maxResults CDATA #IMPLIED
>
<!ELEMENT groupProxy-get (%cm.group-get;)>
<!ELEMENT groupProxy-where %cm.group-where;>
<!ELEMENT incrExport-get %cm.incrExport-get;>
<!ELEMENT incrExport-getUpdateData %cm.incrExport-getUpdateData;>
<!ATTLIST incrExport-getUpdateData
        updateRecordId CDATA #REQUIRED
>
<!ELEMENT incrExport-removeUpdateRecords %cm.incrExport-removeUpdateRecords;>
<!ELEMENT incrExport-reset %cm.incrExport-reset;>
<!ELEMENT job-cancel %cm.job-cancel;>
<!ELEMENT job-create %cm.job-create;>
<!ELEMENT job-delete %cm.job-delete;>
<!ELEMENT job-exec %cm.job-exec;>
<!ELEMENT job-get %cm.job-get;>
<!ELEMENT job-getLogEntry %cm.job-getLogEntry;>
<!ELEMENT job-getOutput %cm.job-getOutput;>
<!ELEMENT job-description (%cm.atom;)>
<!ELEMENT job-listQueue %cm.job-listQueue;>
<!ELEMENT job-set %cm.job-set;>
<!ELEMENT licenseManager-license %cm.licenseManager-license;>
<!ELEMENT licenseManager-licenseExpirationDate %cm.licenseManager-licenseExpirationDate;>
<!ELEMENT licenseManager-licenseType %cm.licenseManager-licenseType;>
<!ELEMENT licenseManager-loginCount %cm.licenseManager-loginCount;>
<!ELEMENT licenseManager-logins %cm.licenseManager-logins;>
<!ELEMENT licenseManager-logout %cm.licenseManager-logout;>
<!ATTLIST licenseManager-logout
        login CDATA #IMPLIED
>
<!ELEMENT licenseManager-maxConcurrentUsers %cm.licenseManager-maxConcurrentUsers;>
<!ELEMENT job-where %cm.job-where;>
<!ATTLIST job-where
    maxResults CDATA #IMPLIED
>
<!ELEMENT link-create %cm.link-create;>
<!ELEMENT link-delete %cm.link-delete;>
<!ELEMENT link-get %cm.link-get;>
<!ELEMENT link-description (%cm.atom;)>
<!ELEMENT link-set %cm.link-set;>
<!ELEMENT link-where %cm.link-where;>
<!ELEMENT logEntry-delete %cm.logEntry-delete;>
<!ELEMENT logEntry-get %cm.logEntry-get;>
<!ELEMENT logEntry-where %cm.logEntry-where;>
<!ELEMENT obj-addComment (%cm.atom;)>
<!ELEMENT obj-commit %cm.obj-commit;>
<!ELEMENT obj-contentTypesForObjType %cm.obj-contentTypesForObjType;>
<!ATTLIST obj-contentTypesForObjType
        objType CDATA #IMPLIED
>
<!ELEMENT obj-copy %cm.obj-copy;>
<!ELEMENT obj-create %cm.obj-create;>
<!ELEMENT obj-delete %cm.obj-delete;>
<!ELEMENT obj-edit %cm.obj-edit;>
<!ELEMENT obj-exportSubtree %cm.obj-exportSubtree;>
<!ELEMENT obj-forward %cm.obj-forward;>
<!ELEMENT obj-generatePreview %cm.obj-generatePreview;>
<!ATTLIST obj-generatePreview
        fsPrefix CDATA #REQUIRED
        masterTemplate CDATA #REQUIRED
        mode (0|1|2|3|4|5|6|7|8|9|10|11|12|13|14|15) #REQUIRED
        path CDATA #REQUIRED
        urlPrefix CDATA #REQUIRED
    editor CDATA #REQUIRED>
<!ELEMENT obj-get (%cm.obj-get;)*>
<!ELEMENT obj-description (%cm.atom;)>
<!ELEMENT obj-give %cm.obj-give;>
<!ELEMENT obj-mirror %cm.obj-mirror;>
```

```
<!ELEMENT obj-permissionGrantTo %cm.obj-permissionGrantTo;>
<!ATTLIST obj-permissionGrantTo
        permission (permissionCreateChildren | permissionWrite | permissionRead |
 permissionRoot) #REQUIRED
>
<!ELEMENT obj-permissionRevokeFrom %cm.obj-permissionRevokeFrom;>
<!ATTLIST obj-permissionRevokeFrom
        permission (permissionCreateChildren | permissionWrite | permissionRead |
 permissionRoot) #REQUIRED
>
<!ELEMENT obj-reject %cm.obj-reject;>
<!ELEMENT obj-release %cm.obj-release;>
<!ELEMENT obj-removeActiveContents %cm.obj-removeActiveContents;>
<!ELEMENT obj-removeArchivedContents %cm.obj-removeArchivedContents;>
<!ELEMENT obj-revert %cm.obj-revert;>
<!ELEMENT obj-search %cm.obj-search;>
<!ELEMENT obj-set %cm.obj-set;>
<!ELEMENT obj-sign %cm.obj-sign;>
<!ELEMENT obj-take %cm.obj-take;>
<!ELEMENT obj-types %cm.obj-types;>
<!ELEMENT obj-unrelease %cm.obj-unrelease;>
<!ELEMENT obj-where %cm.obj-where;>
<!ATTLIST obj-where
    maxResults CDATA #IMPLIED
>


<!ELEMENT objClass-create %cm.objClass-create;>
<!ELEMENT objClass-delete %cm.objClass-delete;>
<!ELEMENT objClass-get (%cm.objClass-get;)*>
<!ELEMENT objClass-description (%cm.atom;)>
<!ELEMENT objClass-goodAvailableBlobEditorsForObjType %cm.objClass-
goodAvailableBlobEditorsForObjType;>
<!ATTLIST objClass-goodAvailableBlobEditorsForObjType
        objType CDATA #REQUIRED
>
<!ELEMENT objClass-set %cm.objClass-set;>
<!ELEMENT objClass-validAttributes %cm.objClass-validAttributes;>
<!ELEMENT objClass-where %cm.objClass-where;>
<!ATTLIST objClass-where
    maxResults CDATA #IMPLIED
>
<!ELEMENT systemConfig-formatDate (%cm.atom;)>
<!ELEMENT systemConfig-formatDateTime (%cm.atom;)>
<!ELEMENT systemConfig-getAttributes %cm.systemConfig-getAttributes;>
<!ELEMENT systemConfig-getCounts %cm.systemConfig-getCounts;>
<!ELEMENT systemConfig-getElements %cm.systemConfig-getElements;>
<!ELEMENT systemConfig-getKeys %cm.systemConfig-getKeys;>
<!ELEMENT systemConfig-getTexts %cm.systemConfig-getTexts;>
<!ELEMENT systemConfig-parseInputDate %cm.date;>
<!ELEMENT systemConfig-installedLanguages %cm.systemConfig-installedLanguages;>
<!ELEMENT systemConfig-validTimeZones %cm.systemConfig-validTimeZones;>
<!ELEMENT systemConfig-validInputCharsets %cm.systemConfig-validInputCharsets;>
<!ELEMENT task-delete %cm.task-delete;>
<!ELEMENT task-get %cm.task-get;>
<!ELEMENT task-description (%cm.atom;)>
<!ELEMENT task-where %cm.task-where;>
<!ATTLIST task-where
    maxResults CDATA #IMPLIED
>
<!ELEMENT user-addToGroups %cm.user-addToGroups;>
<!ELEMENT user-create %cm.user-create;>
<!ELEMENT user-delete %cm.user-delete;>
<!ELEMENT user-get (%cm.user-get;)*>
<!ELEMENT user-description (%cm.atom;)>
<!ELEMENT user-grantGlobalPermissions %cm.user-grantGlobalPermissions;>
<!ELEMENT user-removeFromGroups %cm.user-removeFromGroups;>
<!ELEMENT user-revokeGlobalPermissions %cm.user-revokeGlobalPermissions;>
<!ELEMENT user-set %cm.user-set;>
<!ELEMENT user-where %cm.user-where;>
<!ATTLIST user-where
    maxResults CDATA #IMPLIED
```

```
>
<!ELEMENT user-writeUserFile %cm.user-writeUserFile;>
<!ELEMENT userAttribute-addEnumValues %cm.userAttribute-addEnumValues;>
<!ELEMENT userAttribute-create %cm.userAttribute-create;>
<!ELEMENT userAttribute-delete %cm.userAttribute-delete;>
<!ELEMENT userAttribute-get %cm.userAttribute-get;>
<!ELEMENT userAttribute-description (%cm.atom;)>
<!ELEMENT userAttribute-removeEnumValues %cm.userAttribute-removeEnumValues;>
<!ELEMENT userAttribute-set %cm.userAttribute-set;>
<!ELEMENT userAttribute-where %cm.userAttribute-where;>
<!ATTLIST userAttribute-where
    maxResults CDATA #IMPLIED
>
<!ELEMENT userAttribute-types %cm.userAttribute-types;>
<!ELEMENT userConfig-getAll %cm.userConfig-getAll;>
<!ATTLIST userConfig-getAll
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-formatDate (%cm.atom;)>
<!ATTLIST userConfig-formatDate
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-formatDateTime (%cm.atom;)>
<!ATTLIST userConfig-formatDateTime
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-getAttributes %cm.userConfig-getAttributes;>
<!ATTLIST userConfig-getAttributes
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-getCounts %cm.userConfig-getCounts;>
<!ATTLIST userConfig-getCounts
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-getElements %cm.userConfig-getElements;>
<!ATTLIST userConfig-getElements
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-getKeys %cm.userConfig-getKeys;>
<!ATTLIST userConfig-getKeys
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-getTexts %cm.userConfig-getTexts;>
<!ATTLIST userConfig-getTexts
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-initFromFile (%cm.atom;)>
<!ATTLIST userConfig-initFromFile
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-parseInputDate %cm.date;>
<!ATTLIST userConfig-parseInputDate
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-removeAttributes %cm.userConfig-removeAttributes;>
<!ATTLIST userConfig-removeAttributes
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-removeKeys %cm.userConfig-removeKeys;>
<!ATTLIST userConfig-removeKeys
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-setAttributes %cm.userConfig-setAttributes;>
<!ATTLIST userConfig-setAttributes
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-setElements %cm.userConfig-setElements;>
<!ATTLIST userConfig-setElements
        login CDATA #IMPLIED
>
<!ELEMENT userConfig-setTexts %cm.userConfig-setTexts;>
<!ATTLIST userConfig-setTexts
```

```
        login CDATA #IMPLIED
>
<!ELEMENT userProxy-get (%cm.user-get;)*>
<!ELEMENT userProxy-where %cm.user-where;>
<!ELEMENT workflow-create %cm.workflow-create;>
<!ELEMENT workflow-delete %cm.workflow-delete;>
<!ELEMENT workflow-get (%cm.workflow-get;)*>
<!ELEMENT workflow-description (%cm.atom;)>
<!ELEMENT workflow-set %cm.workflow-set;>
<!ELEMENT workflow-where %cm.workflow-where;>
<!ATTLIST workflow-where
    maxResults CDATA #IMPLIED
>

<!ELEMENT attribute (%cm.atom; | %cm.attribute-get;)*>
<!ELEMENT attributeGroup (attribute | %cm.attributeGroup-get;)*>
<!ELEMENT content (%cm.content-get;)*>
<!ELEMENT group (%cm.atom; | %cm.group-get;)*>
<!ELEMENT job (%cm.job-get;)>
<!ELEMENT link (%cm.link-get;)>
<!ELEMENT obj (%cm.obj-get;)*>
<!ELEMENT objClass (%cm.atom; | %cm.objClass-get;)*>
<!ELEMENT updateData ANY>
<!ELEMENT user (%cm.atom; | %cm.user-get;)*>
<!ELEMENT userAttribute (%cm.userAttribute-get;)>
<!ELEMENT arguments (%cm.listitem;)*>
<!ELEMENT all (%cm.atom;)>
<!ELEMENT allowsMultipleSignatures (%cm.atom;)>
<!ELEMENT anchors (%cm.listitem;)*>
<!ELEMENT archivedContents ((content)* | (%cm.listitem;)*)>
<!ELEMENT attributeName (%cm.atom; | %cm.attribute-get;)*>
<!ELEMENT attributeNames (%cm.listitem;)*>
<!ELEMENT attributeGroups ((attributeGroup)* | (%cm.listitem;)*)>
<!ELEMENT attributes ((attribute)* | (%cm.listitem;)* | (dictitem)+)>
<!ELEMENT availableBlobEditors (%cm.listitem;)*>
<!ELEMENT blob (%cm.atom;)>
<!ATTLIST blob
    encoding (plain | base64 | stream) #IMPLIED
>
<!ELEMENT blobLength (%cm.atom;)>
<!ELEMENT body (%cm.atom;)>
<!ELEMENT bodyTemplateName (%cm.atom;)>
<!ELEMENT callback (%cm.atom;)>
<!ELEMENT canHaveAnchor (%cm.atom;)>
<!ELEMENT canHaveTarget (%cm.atom;)>
<!ELEMENT category (%cm.atom;)>
<!ELEMENT children ((obj)* | (%cm.listitem;)*)>
<!ELEMENT codeForSourceView (%cm.atom;)>
<!ATTLIST codeForSourceView
        objectPageUrl CDATA #REQUIRED
        linkPageUrl CDATA #REQUIRED
        useJavaScript CDATA #IMPLIED
>
<!ELEMENT collections (%cm.listitem;)>
<!ELEMENT command (%cm.atom;)>
<!ELEMENT comment (%cm.atom;)>
<!ELEMENT committedContentId (%cm.atom; | %cm.content-get;)*>
<!ELEMENT completionCheck (%cm.atom;)>
<!ELEMENT contentId (%cm.atom;)>
<!ELEMENT contentIds ((content)* | (%cm.listitem;)*)>
<!ELEMENT contents ((content)* | (%cm.listitem;)*)>
<!ELEMENT contentType (%cm.atom;)>
<!ELEMENT count (%cm.atom;)>
<!ELEMENT createPermission (%cm.atom;)>
<!ELEMENT customBlobEditorUrl (%cm.atom;)>
<!ELEMENT defaultAttributeGroup (%cm.atom; | attributeGroup)*>
<!ELEMENT defaultGroup (%cm.atom; | group)*>
<!ELEMENT deleteLogEntriesCount (%cm.atom;)>
<!ELEMENT description (%cm.atom;)>
<!ELEMENT destination (%cm.atom; | %cm.obj-get;)*>
<!ELEMENT destinationUrl (%cm.atom;)>
<!ELEMENT displayTitle (%cm.atom;)>
```

```
<!ATTLIST displayTitle
        type CDATA #IMPLIED
>
<!ELEMENT displayValueCallback (%cm.atom;)>
<!ELEMENT editField (%cm.atom;)>
<!ATTLIST editField
        parameter (length | maxlength | nilAllowed | objClasses | rows | startPub| type)
 #REQUIRED
>
<!ELEMENT editFieldSpec (%cm.atom; | %cm.dictitem;)*>
<!ELEMENT editGroups ((group)* | (%cm.listitem;)*)>
<!ELEMENT editedContent (%cm.atom; | %cm.content-get;)*>
<!ELEMENT editedContentId (%cm.atom; | %cm.content-get;)*>
<!ELEMENT editor (%cm.atom; | %cm.user-get;)*>
<!ELEMENT email (%cm.atom;)>
<!ELEMENT emptyAttributeGroups ((%cm.listitem;)* | (attributeGroup)*)>
<!ELEMENT encryptedPassword (%cm.atom;)>
<!ELEMENT execLogin (%cm.atom;)>
<!ELEMENT execPerm (%cm.atom;)>
<!ELEMENT expectedPath (%cm.atom;)>
<!ELEMENT exportBlob (%cm.atom;)>
<!ATTLIST exportBlob
    encoding (plain | base64 | stream) #IMPLIED
>
<!ELEMENT exportFiles (%cm.listitem;)*>
<!ELEMENT exportMimeType (%cm.atom;)>
<!ELEMENT externalAttrNames ((attribute)* | (%cm.listitem;)*)>
<!ELEMENT externalAttributes (%cm.dictitem;)*>
<!ELEMENT externalEditExtension (%cm.atom;)>
<!ELEMENT externalEditMimeType (%cm.atom;)>
<!ELEMENT externalUserAttrNames ((userAttribute)* | (%cm.listitem;)*)>
<!ELEMENT extractToDirectory (%cm.atom;)>
<!ELEMENT filter (%cm.atom;)>
<!ELEMENT file (%cm.atom;)>
<!ELEMENT filePrefix (%cm.atom;)>
<!ELEMENT frameNames (%cm.listitem;)*>
<!ELEMENT freeLinks ((link)* | (%cm.listitem;)*)>
<!ELEMENT fromDate (%cm.atom;)>
<!ELEMENT getKeys (%cm.listitem;)*>
<!ELEMENT globalPermissions (%cm.listitem;)*>
<!ELEMENT goodAttributeGroupAttributes ((%cm.listitem;)* | (attribute)*)>
<!ELEMENT goodAttributes ((%cm.listitem;)* | (attribute)*)>
<!ELEMENT goodMandatoryAttributes ((attribute)* | (%cm.listitem;)*)>
<!ELEMENT goodPresetAttributes ((attribute)* | (%cm.listitem;)*)>
<!ELEMENT goodPresetFromParentAttributes ((attribute)* | (%cm.listitem;)*)>
<!ELEMENT groupName (%cm.atom; | %cm.group-get;)*>
<!ELEMENT groupNames (%cm.listitem;)*>
<!ELEMENT groupText (%cm.atom;)>
<!ELEMENT groups ((group)* | (%cm.listitem;)*)>
<!ELEMENT hasChildren (%cm.atom;)>
<!ELEMENT hasGlobalPermission (%cm.atom;)>
<!ATTLIST hasGlobalPermission
        permission CDATA #REQUIRED
>
<!ELEMENT hasPassword (%cm.atom;)>
<!ATTLIST hasPassword
        password CDATA #REQUIRED
>
<!ELEMENT hasSuperLinks (%cm.atom;)>
<!ELEMENT helpText (%cm.atom;)>
<!ATTLIST helpText
        lang (en | de | it | fr | es) #IMPLIED
>
<!ELEMENT hierarchy (%cm.listitem;)*>
<!ATTLIST hierarchy
        maxDepth CDATA #IMPLIED
        maxLines CDATA #IMPLIED
        document (0 | 1) #IMPLIED
        generic (0 | 1) #IMPLIED
        image (0 | 1) #IMPLIED
        publication (0 | 1) #IMPLIED
        template (0 | 1) #IMPLIED
```

```
>
<!ELEMENT condition (%cm.atom; | %cm.listitem;)*>
<!ATTLIST condition
    subject CDATA #REQUIRED
    verb CDATA #REQUIRED
>
<!ELEMENT id (%cm.atom;)>
<!ELEMENT ids (%cm.atom; | %cm.listitem;)*>
<!ELEMENT index (%cm.atom;)>
<!ELEMENT inputType (%cm.atom;)>
<!ELEMENT isActive (%cm.atom;)>
<!ELEMENT isCommitted (%cm.atom;)>
<!ELEMENT isComplete (%cm.atom;)>
<!ELEMENT isDefaultGroup (%cm.atom;)>
<!ELEMENT isEdited (%cm.atom;)>
<!ELEMENT isEmpty (%cm.atom;)>
<!ELEMENT isEnabled (%cm.atom;)>
<!ELEMENT isExternalLink (%cm.atom;)>
<!ELEMENT isFreeLink (%cm.atom;)>
<!ELEMENT isGoodDestination (%cm.atom;)>
<!ATTLIST isGoodDestination
        linkId CDATA #REQUIRED
>
<!ELEMENT isGoodParent (%cm.atom;)>
<!ATTLIST isGoodParent
        objectId CDATA #REQUIRED
>
<!ELEMENT isIncludeLink (%cm.atom;)>
<!ELEMENT isInlineReferenceLink (%cm.atom;)>
<!ELEMENT isLinkFromCommittedContent (%cm.atom;)>
<!ELEMENT isLinkFromEditedContent (%cm.atom;)>
<!ELEMENT isLinkFromReleasedContent (%cm.atom;)>
<!ELEMENT isoDateTime (%cm.atom;)>
<!ELEMENT isOwnerOf (%cm.atom;)>
<!ATTLIST isOwnerOf
        login CDATA #REQUIRED
>
<!ELEMENT isQueued (%cm.atom;)>
<!ELEMENT isRelatedLink (%cm.atom;)>
<!ELEMENT isReleased (%cm.atom;)>
<!ELEMENT isRoot (%cm.atom;)>
<!ELEMENT isSearchableInCM (%cm.atom;)>
<!ELEMENT isSearchableInTE (%cm.atom;)>
<!ELEMENT isSuperUser (%cm.atom;)>
<!ELEMENT isWritable (%cm.atom;)>
<!ELEMENT lastChanged %cm.date;>
<!ATTLIST lastChanged
        type CDATA #IMPLIED
>
<!ELEMENT lastExecEnd %cm.date;>
<!ATTLIST lastExecEnd
        type CDATA #IMPLIED
>
<!ELEMENT lastExecResult (%cm.atom;)>
<!ELEMENT lastExecStart %cm.date;>
<!ATTLIST lastExecStart
        type CDATA #IMPLIED
>
<!ELEMENT lastLogEntry %cm.logEntry;>
<!ELEMENT lastOutput (%cm.atom;)>
<!ELEMENT linkListAttributes ((attribute)* | (%cm.listitem;)*)>
<!ELEMENT localizedTitle (%cm.atom;)>
<!ELEMENT localizedHelpText (%cm.atom;)>
<!ELEMENT log (%cm.listitem;)*>
<!ELEMENT logEntries (%cm.listitem;)*>
<!ELEMENT logEntryId (%cm.atom;)>
<!ELEMENT execResult (%cm.atom;)>
<!ELEMENT execStart %cm.date;>
<!ATTLIST execStart
    type CDATA #IMPLIED
>
<!ELEMENT execEnd %cm.date;>
```

```
<!ATTLIST execEnd
    type CDATA #IMPLIED
>
<!ELEMENT logText (%cm.atom;)>
<!ELEMENT logTime %cm.date;>
<!ELEMENT logType (%cm.atom;)>
<!ELEMENT logTypes (%cm.listitem;)*>
<!ELEMENT login (%cm.atom;)>
<!ELEMENT maxDocs (%cm.atom;)>
<!ELEMENT maxSize (%cm.atom;)*>
<!ELEMENT mandatoryAttributes ((attribute)* | (%cm.listitem;)*)>
<!ELEMENT method (%cm.atom;)>
<!ELEMENT mimeType (%cm.atom;)>
<!ELEMENT minRelevance (%cm.atom;)>
<!ELEMENT minSize (%cm.atom;)*>
<!ELEMENT mode (%cm.atom;)>
<!ELEMENT name (%cm.atom;)>
<!ATTLIST name
        type CDATA #IMPLIED
>
<!ELEMENT nameLike (%cm.atom;)>
<!ELEMENT namevalue (name, value)>
<!ELEMENT next (%cm.atom; | %cm.obj-get;)*>
<!ELEMENT nextEditGroup (%cm.atom; | %cm.group-get;)*>
<!ELEMENT nextExecStart %cm.date;>
<!ATTLIST nextExecStart
        type CDATA #IMPLIED
>
<!ELEMENT nextSignGroup (%cm.atom; | %cm.group-get;)*>
<!ELEMENT now %cm.date;>
<!ATTLIST now
        type CDATA #IMPLIED
>
<!ELEMENT today %cm.date;>
<!ELEMENT objClasses ((%cm.listitem;)* | (objClass)*)>
<!ELEMENT objType (%cm.atom;)>
<!ELEMENT objectId (%cm.atom; | %cm.obj-get;)*>
<!ELEMENT objectsToRoot ((obj)* | (%cm.listitem;)*)>
<!ELEMENT offsetStart (%cm.atom;)>
<!ELEMENT offsetLength (%cm.atom;)>
<!ELEMENT outputType (%cm.atom;)>
<!ELEMENT owner (%cm.atom; | %cm.user-get;)*>
<!ELEMENT parent (%cm.atom; | %cm.obj-get;)*>
<!ELEMENT parser (%cm.atom;)>
<!ELEMENT password (%cm.atom;)>
<!ELEMENT path (%cm.atom;)>
<!ELEMENT permission ((group)* | (%cm.listitem;)*)>
<!ATTLIST permission
        permission (permissionCreateChildren | permissionWrite | permissionRead |
 permissionRoot) #REQUIRED
>
<!ELEMENT permissionGrantedTo (%cm.atom;)>
<!ATTLIST permissionGrantedTo
        permission (permissionCreateChildren | permissionWrite | permissionRead |
 permissionRoot) #REQUIRED
        user CDATA #IMPLIED
        group CDATA #IMPLIED
>
<!ELEMENT prefixPath (%cm.atom;)>
<!ELEMENT presetAttributes (dictitem)*>
<!ELEMENT presetFromParentAttributes ((attribute)* | (%cm.listitem;)*)>
<!ELEMENT previous (%cm.atom; | %cm.obj-get;)*>
<!ELEMENT queuePos (%cm.atom;)>
<!ELEMENT query (%cm.atom;)>
<!ELEMENT realName (%cm.atom;)>
<!ELEMENT reasonsForIncompleteState (%cm.listitem;)*>
<!ELEMENT receiver (%cm.atom;)>
<!ELEMENT recordSetCallback (%cm.atom;)>
<!ELEMENT recursive (%cm.atom;)>
<!ELEMENT relatedLinks ((link)* | (%cm.listitem;)*)>
<!ELEMENT releasedContent (%cm.atom; | %cm.content-get;)*>
<!ELEMENT releasedContentId (%cm.atom; | %cm.content-get;)*>
```

```
<!ELEMENT releasedVersions ((content)* | (%cm.listitem;)*)>
<!ELEMENT rootPermissionFor (%cm.atom;)>
<!ATTLIST rootPermissionFor
        user CDATA #IMPLIED
        group CDATA #IMPLIED
>
<!ELEMENT schedule (%cm.listitem;)*>
<!ELEMENT script (%cm.atom;)>
<!ELEMENT setKeys (%cm.listitem;)*>
<!ELEMENT signatureAttrNames ((attribute)* | (%cm.listitem;)*)>
<!ELEMENT signatureAttributes (namevalue)*>
<!ELEMENT signatureDefs (%cm.dictitem;)*>
<!ELEMENT sortKey1 (%cm.atom;)>
<!ELEMENT sortKey2 (%cm.atom;)>
<!ELEMENT sortKey3 (%cm.atom;)>
<!ELEMENT sortKeyLength1 (%cm.atom;)>
<!ELEMENT sortKeyLength2 (%cm.atom;)>
<!ELEMENT sortKeyLength3 (%cm.atom;)>
<!ELEMENT sortOrder (%cm.atom;)>
<!ELEMENT sortType1 (%cm.atom;)>
<!ELEMENT sortType2 (%cm.atom;)>
<!ELEMENT sortType3 (%cm.atom;)>
<!ELEMENT sortValue (%cm.atom;)>
<!ELEMENT source (%cm.atom; | %cm.obj-get;)*>
<!ELEMENT sourceContent (%cm.atom; | %cm.content-get;)*>
<!ELEMENT sourceTagAttribute (%cm.atom;)>
<!ELEMENT sourceTagName (%cm.atom;)>
<!ELEMENT state (%cm.atom;)>
<!ELEMENT subLinks ((link)* | (%cm.listitem;)*)>
<!ELEMENT superLinks ((link)* | (%cm.listitem;))>
<!ELEMENT superObjects ((obj)* | (%cm.listitem;))>
<!ELEMENT suppressExport (%cm.atom;)>
<!ELEMENT systemConfigFormattedTime (%cm.atom;)>
<!ELEMENT target (%cm.atom;)>
<!ELEMENT taskText (%cm.atom;)>
<!ELEMENT taskType (%cm.atom;)>
<!ELEMENT textLinks ((link)* | (%cm.listitem;)*)>
<!ELEMENT templateName (%cm.atom;)>
<!ELEMENT thumbnail (%cm.atom;)>
<!ELEMENT timeStamp (%cm.atom;)>
<!ELEMENT timeZone (%cm.atom;)>
<!ELEMENT title (%cm.atom;)>
<!ATTLIST title
        lang (en | de | it | fr | es) #IMPLIED
        type CDATA #IMPLIED
>
<!ELEMENT toclist ((obj)* | (%cm.listitem;)*)>
<!ELEMENT type (%cm.atom;)>
<!ELEMENT untilDate %cm.date;>
<!ELEMENT updateRecords (%cm.listitem;)*>
<!ELEMENT updateRecordCount (%cm.atom;)>
<!ELEMENT updateRecordId (%cm.atom;)>
<!ELEMENT userConfigFormattedTime (%cm.atom;)>
<!ELEMENT userLocked (%cm.atom;)>
<!ELEMENT userLogin (%cm.atom; | %cm.user-get;)*>
<!ELEMENT userText (%cm.atom;)>
<!ELEMENT users ((user)* | (%cm.listitem;)*)>
<!ELEMENT userProxy ANY>
<!ELEMENT validContentTypes (%cm.listitem;)*>
<!ELEMENT validControlActionKeys (%cm.listitem;)*>
<!ELEMENT validCreateObjClasses ((objClass)* | (%cm.listitem;)*)>
<!ELEMENT validEditFieldKeys (%cm.listitem;)*>
<!ELEMENT validEditFieldTypes (%cm.listitem;)*>
<!ELEMENT validFrom %cm.date;>
<!ATTLIST validFrom
        type CDATA #IMPLIED
>
<!ELEMENT validObjClasses ((objClass)* | (%cm.listitem;)*)>
<!ELEMENT validPermissions (%cm.listitem;)*>
<!ELEMENT validSortKeys (%cm.listitem;)*>
<!ELEMENT validSortOrders (%cm.listitem;)*>
<!ELEMENT validSortTypes (%cm.listitem;)*>
```

```
<!ELEMENT validSubObjClassCheck (%cm.atom;)>
<!ELEMENT validSubObjClasses ((objClass)* | (%cm.listitem;)*)>
<!ELEMENT validUntil %cm.date;>
<!ATTLIST validUntil
        type CDATA #IMPLIED
>
<!ELEMENT values (%cm.listitem;)*>
<!ELEMENT version (%cm.atom;)>
<!ELEMENT visibleExportTemplates ((obj)* | (%cm.listitem;)*)>
<!ELEMENT visibleName (%cm.atom;)>
<!ELEMENT visiblePath (%cm.atom;)>
<!ELEMENT wantedTags (%cm.listitem;)*>
<!ELEMENT workflowComment (%cm.atom;)>
<!ELEMENT workflowModification (%cm.atom;)>
<!ELEMENT workflowName (%cm.atom; | %cm.workflow-get;)*>
<!ELEMENT xmlBlob (%cm.atom;)>
<!ELEMENT xmldtd (%cm.atom;)>

<!ELEMENT appName (%cm.atom;)>
<!ELEMENT baseDir (%cm.atom;)>
<!ELEMENT binDir (%cm.atom;)>
<!ELEMENT configDir (%cm.atom;)>
<!ELEMENT dataDir (%cm.atom;)>
<!ELEMENT debugChannels (%cm.atom;)>
<!ELEMENT debugLevel (%cm.atom;)>
<!ELEMENT instanceDir (%cm.atom;)>
<!ELEMENT libDir (%cm.atom;)>
<!ELEMENT logDir (%cm.atom;)>
<!ELEMENT rootConfigPath (%cm.atom;)>
<!ELEMENT scriptDir (%cm.atom;)>
<!ELEMENT shareDir (%cm.atom;)>
<!ELEMENT tmpDir (%cm.atom;)>
```