Programmers guide

Welcome to the programmers guide of cmsWorks.

This guide is for programmers of the general purpose content management cmsWorks and will lead you - step by step - on how to program your own sites and pages in cmsWorks.

For whom this guide is for

This guide describes how programming your own site(s) may be managed by using the cmsWorks ecosystem. It explains the hardware / OS / additional needed software for developer workstations and productive / testing servers.

Additionally it provides example programming code and best practices on how to get your site up and running. In case questions arise regarding the installation, administration or configuration please consult the administrators guide to cmsWorks.

Programming cmsWorks - needed hardware components (server / client)

To get cmsWorks up and running, and therfore program and deploy your own site you need

On a programmers workstation (on the "client" side):

- A Java JDK with the correct version installed
- A connection to a cmsWorks database like MySQL, ... (optional but recommended)
- An IDE which helps to create your programs (optional)

On a production system (on the "server" side):

- A server with root access or, at least, access to a shell running on that very server
- A Java JDK installed
- An cmsWorks internal database, optionally an extra database like MySQL (recommended)
- Optionally a http-server like Apache, nginx or IIS and the ability to configure it (recommended)

Eventually, having a bigger team developing / testing your site a staging system may be advised which means that the same needs as for a production system can apply.

Meeting these requirements and having knowledge of HTML, CSS, JavaScript and (a little bit of) Java, you are ready to go.

What is cmsWorks from a programmers point of view?

cmsWorks is

- a general purpose content management system, being agnostic to input and output formats or even your workflows
- a "full stack" implementation which means that you could run cmsWorks without any other software needed (if desired)
- an internal search engine based on big data, providing online ("on-site") and internal search capabilities for programming
- an "any OS" system only needing a JDK regarding your live / staging systems or your programmer workstations
- driven by microservices which could be started, stopped, added or removed any time, even during runtime in live environments
- therefore "hot-redeployable", which means that programming parts may be re-deployed without stopping the whole system
- a ZAK ("zero administration kit") system warning you also in regards of third-party-misbehavior (i.e. database failures)
- only using Java/OpenJDK JVM and HTML, CSS, JavaScript, no other dependencies required, in case you do not want/need them
- a way to simplify your configurations by simple, self-explaining properties instead of XML, applicable for different environments
- a straight-forward, simple visual BPM (business process management) engine on your fingertips called ScriptEngine
- a configurable graphical user interface (GUI) everyone can access with a "modern" web browser, no client installations needed
- an ecosystem for developers including cronjobs, CLI commands, batches, services, this all built in from scratch

Table of Contents

1 cmsWorks programmers guide - an overview

Programming a content management system, making sure that the users visiting the site and the editorial staff posting the content are all satisfied, that got a little bit of a science in itself.

Programming a content management system

The efforts of technically creating (or programming) a site does, normally, not matter to non-technical people. That is quite understandable because users want to consume content and therefore are not interested in the underlying techniques - all what they want is a better content experience. Furthermore the editorial staff wants to spend less time in "technical issues" while editing content. They rather want easy ways to publish their content fast and in an according way.

Regarding the technical view a content management system has to fulfill three basic tasks in first place:

- It provides the possibility to create, insert, update or delete contents
- It stores these contents reliably for later use (i.e. versioned in a database)
- It generates an output of these contents into any desired formats (i.e. HTML, PDF, JSON, ...)

cmsWorks solves the needs of the users consuming content and the editorial staff by providing best-of-breed tools and methods. And generating content in different formats is made easy, too. This is because cmsWorks is a "general purpose content management system".

What this programming guide is for

This guide is about the generation of that very content, meaning how to access, process and output it using cmsWorks and its features to present content or program own modules (or even workflows) using cmsWorks.

Advantages in programming cmsWorks

cmsWorks was built to make the programmer's life easier by

- providing standardized APIs and Frameworks and their documentation
- easily create visual or non-visual workflows
- utilizing a built-in big data search engine, thereby
- helping to programmatically create and weave content rather than administer content
- avoiding broken links at any means
- providing an internal CLI for further control
- using sophisticated (programming-agnostic transparent) caching mechanisms for speedup
- having a standardized logging (i.e. accessible via telnet)
- a zero administration kit (proactive warning for programming faults or i.e. for out-of-memory errors)

and a powerful editorial desktop meaning less time spending for editorial needs.

Working with documents (resources)

An - editorial - document (in programming a document is called a resource) is the uppermost "information unit" of a singluar content in cmsWorks. A document stores it's name, path and other meta data, additionally it contains properties which represent the real contents of the document like titles, texts or keywords.

The appearance and capabilities of cmsWorks documents are described in the users guide, the chapter Working with document properties shows how to get, read and handle different resources from cmsWorks.

2 Working with IDEs

Introduction

As like with most other tools, frameworks, APIs and systems you may use a variety of editing/programming/compiling tools with cmsWorks too. Stripped down, you might even program and extend cmsWorks yet alone only using a JDK and a text editor. This, of course, is not the most desirable way to create a modern website.

cmsWorks is tool / tool chain agnostic

cmsWorks thrives to be agnostic to your tools or your preferred tool chain because we do not know which your preferred tools are. In other words, you may use whatever tools are suitable for you.

Anyway, modern IDEs help a lot getting the workload done using tools for programming and evaluation. This chapter describes the work with cmsWorks using Eclipse in special and how to integrate cmsWorks in your favorite tool in general.

How cmsWorks classes and jars are organized

cmsWorks follows a straight-forward file system policy. The needed files for cmsWorks are stored in their respective folders using a so called "exploded deployment", meaning that cmsWorks and it's associated files are available in single class files (this is necessary i.e. for hot redeployment scenarios). These files, moreover their folders have to be configured to your IDE so that it can help you in your development process.

But not everything in cmsWorks is developed by itechWorks on it's own. We rather highly rely on well-tested, bullet-proof open source software (OSS with moderate licenses like MIT, Apache 2.0, LGPL etc.) having their own JAR files delivered with cmsWorks. Using a modern IDE and benefiting from it means that you will have to "announce" them to your IDE, too.

Using Eclipse with cmsWorks

Exemplarily, the next chapter explains a fast-to-setup way for the Eclipse-IDE.

2.1 Working with eclipse

Beside the root of the cmsWorks installation we create a project folder for Eclipse. The project configuration only relies on the two configuration files .classpath and .project. Also there have to be two folders which declare a folder for sources and a folder for the compile target.

A download with an example structure is in the download center of www.cmsworks.app and ready to be imported as project in your Eclipse IDE.

The .project file contains:

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
	<name>cmsWorks</name>
	<comment>cmsWorks</comment>
	<projects>
	</projects>
	<buildSpec>
		<buildCommand>
			<name>org.eclipse.jdt.core.javabuilder</name>
			<arguments>
			</arguments>
		</buildCommand>
	</buildSpec>
	<natures>
		<nature>org.eclipse.jdt.core.javanature</nature>
	</natures>
	<linkedResources>
		<link>
			<name>cmsworks</name>
			<type>2</type>
			<locationURI>PARENT-1-PROJECT_LOC/cmsworks</locationURI>
		</link>
	</linkedResources>
</projectDescription>
```

Example of the content of the .project file from the Eclipse project configuration

The entry <name> currently contains "cmsWorks". This is the unique project name for Eclipse. This name should be tailored if more than one cmsWorks project will be maintained by Eclipse.

The only linked folder for the project is the "cmsworks" folder beside the Eclipse project folder.

The .classpath file contains:

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

  <classpathentry kind="src" path="src"/>

  <classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER">
    <attributes>
      <attribute name="module" value="true"/>
      <attribute name="add-exports" value="java.base/com.sun.crypto.provider=ALL-UNNAMED"/>
      <attribute name="limit-modules" value="java.rmi,jdk.management.jfr,jdk.jdi,jdk.charsets,java.xml,jdk.xml.dom,java.datat
ransfer,jdk.jstatd,jdk.httpserver,jdk.jcmd,java.desktop,java.se,java.security.sasl,jdk.zipfs,java.base,jdk.crypto.ec,jdk.java
doc,jdk.management.agent,jdk.jshell,jdk.editpad,jdk.sctp,jdk.jsobject,jdk.unsupported,java.smartcardio,jdk.jlink,java.securit
y.jgss,java.compiler,jdk.nio.mapmode,jdk.dynalink,jdk.unsupported.desktop,jdk.accessibility,jdk.security.jgss,jdk.incubator.v
ector,jdk.hotspot.agent,java.xml.crypto,java.logging,jdk.jfr,jdk.internal.vm.ci,jdk.crypto.cryptoki,jdk.net,jdk.random,java.n
aming,jdk.internal.ed,java.prefs,java.net.http,jdk.compiler,jdk.naming.rmi,jdk.internal.opt,jdk.jconsole,jdk.attach,jdk.crypt
o.mscapi,jdk.internal.le,java.management,jdk.jdwp.agent,jdk.incubator.foreign,jdk.internal.jvmstat,java.instrument,jdk.intern
al.vm.compiler,jdk.internal.vm.compiler.management,jdk.management,jdk.security.auth,java.scripting,jdk.jdeps,jdk.jartool,jav
a.management.rmi,jdk.jpackage,jdk.naming.dns,jdk.localedata"/>
    </attributes>
  </classpathentry>

  <classpathentry kind="lib" path="cmsworks/run/lib/itechworks/commons-logging/b003/topas-commons-logging.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/ext/concurrent.jar"/>

  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/apache-jsp/org.eclipse.jdt.ecj-3.27.0.jar" sourcepath="lib-s
rc/jetty.project-jetty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/apache-jsp/org.eclipse.jetty.apache-jsp-11.0.8.jar" sourcepa
th="lib-src/jetty.project-jetty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/apache-jsp/org.mortbay.jasper.apache-el-10.0.14.jar" sourcep
ath="lib-src/jetty.project-jetty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/apache-jsp/org.mortbay.jasper.apache-jsp-10.0.14.jar" source
path="lib-src/jetty.project-jetty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/logging/jetty-slf4j-impl-11.0.8.jar" sourcepath="lib-src/jet
ty.project-jetty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/logging/slf4j-api-2.0.0-alpha5.jar" sourcepath="lib-src/jett
y.project-jetty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/jetty-http-11.0.8.jar" sourcepath="lib-src/jetty.project-jet
ty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/jetty-io-11.0.8.jar" sourcepath="lib-src/jetty.project-jetty
-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/jetty-security-11.0.8.jar" sourcepath="lib-src/jetty.project
-jetty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/jetty-server-11.0.8.jar" sourcepath="lib-src/jetty.project-j
etty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/jetty-servlet-11.0.8.jar" sourcepath="lib-src/jetty.project-
jetty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/jetty/11.0.8/jetty-util-11.0.8.jar" sourcepath="lib-src/jetty.project-jet
ty-11.0.8.zip"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/eclipse/jakarta-ee/jakarta.servlet-api-5.0.0.jar" sourcepath="lib-src/jak
arta-servlet-api-master.zip"/>

  <classpathentry kind="lib" path="cmsworks/run/lib/mysql/5.1.7/mysql-connector-java-5.1.7-bin.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/hsqldb/2.7.1/hsqldb.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/postgresql/postgresql-42.5.1.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/oracle/ojdbc11.jar"/>

  <classpathentry kind="lib" path="cmsworks/run/lib/eclipse/jakarta-ee/jakarta.mail-api-2.1.1.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/eclipse/jakarta-ee/jakarta.activation-api-2.1.1.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/eclipse/angus-mail-2.0.1.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/eclipse/angus-activation-2.0.0.jar"/>

  <classpathentry kind="lib" path="cmsworks/run/lib/apache/derby/derbynet.jar"/>

  <classpathentry kind="lib" path="cmsworks/run/lib/apache/avalon/avalon-framework-4.2.0.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/apache/avalon/logkit-1.2.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/apache/bcel/bcel.jar"/>

  <classpathentry kind="lib" path="cmsworks/run/lib/apache/ftp/ftplet-api-1.2.0.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/apache/ftp/ftpserver-core-1.2.0.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/apache/ftp/mina-core-2.1.6.jar"/>

  <!--  ================================================ -->

  <classpathentry kind="lib" path="cmsworks/run/lib/minimal-json-0.9.5.jar"/>

  <classpathentry kind="lib" path="cmsworks/run/lib/apache/lucene-3.6.0/lucene-analyzers-3.6.0.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/apache/lucene-3.6.0/lucene-core-3.6.0.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/apache/lucene-3.6.0/lucene-queries-3.6.0.jar"/>
  <classpathentry kind="lib" path="cmsworks/run/lib/apache/lucene-3.6.0/lucene-queryparser-3.6.0.jar"/>

  <classpathentry kind="lib" path="cmsworks/ext"/>
```

```
  <classpathentry kind="output" path="compiled"/>
</classpath>
```

Example of the content of the .classpath file from the Eclipse project configuration

This file declares the source path "src", the compilation target path "compiled" as output entry and all libraries the project needs.

After this, when the project folder is filled in Eclipse open "File" -> "Import..." -> "General" -> "Existing Projects into Workspace".

Choose "Select root directory" and use the Eclipse project folder for the cmsWorks project. Finally "Finish" and the Project is available.

With the running cmsWorks project and a Preview page open just edit the corresponding JSP and refresh the Browser preview page to see the changes.

3 Website generation

The main purpose using cmsWorks is to create websites. Users login to the cmsWorks editors desktop and create documents storing the content. The generator service gets requests via URLs to read the contents from the CMSCore service and produce and return a website (in HTML or even any other format).

In a cmsWorks generator service JSPs are used to create the (website) content.

The behavior of JSPs

After sending a Request by passing an URL to the Generator in a browser the Generator service accepts the request (because it listens to the port), analyzes the URL and finds the JSP to execute. This JSP is compiled into a Servlet and will then be executed with the parameters Request and Response. The Servlet uses the Request to access the calling URL, parameters, request headers and attributes with information about the Generator service and the therefore requested CMS document. It produces the content, writing it into the Response and can add some response header information. The browser then is, finally, showing the created website.

The first request for a JSP compiles the Servlet. After this compilation, the Servlet ist loaded into the server as single instance. This Servlet then will serve every request handled by it. If the JSP file behind this Servlet is changed it will be recompiled and the old Servlet is replaced with the new one. So without restartarting any cmsWorks service, a simple change in the JSP file changes the website.

The content of JSP files

A JSP-File contains

- JSP configuration
- JSF includes
- JSP methods
- Inline java code
- Simple response output

```
<%@page import="java.util.HashMap, java.text.SimpleDateFormat"
        session="false"
        contentType="text/html;charset=UTF-8"
%>

<%!
private String translate(String s) {
  return "interesting.";
}
%>
<!DOCTYPE html>
<html>
<head>
  <title>TextFilterExample-1</title>
</head>
<body>
<%@include file="header.jsf"%>
<%
String s = "JSPs are " + translate("great!");
out.println(s);
%>
You called the URL <%= request.getRequestURI() %>
</body>
</html>
```

Basic elements of a JSP

JSP configuration

```
<%@page import="java.util.HashMap, java.text.SimpleDateFormat"
        session="false"
        contentType="text/html;charset=UTF-8"
%>
```

The <%@page tag contains the configuration.

The property import contains all imports of classes used by this JSP. In the example above none of the classes is used but it's declared to show the syntax.

The property session is set false since the default is true and session management of the Generator service is not need most of the time.

The property contentType contains the value for the response header field Content-Type. Also the IDE Eclipse uses the information while developing the JSP.

JSP methods

```
<%!
private String translate(String s) {
  return "interesting.";
}
%>
```

The tag <%! ... %> contains Methods and inner classes of the Servlet to be compiled.

Simple response output

The code <!DOCTYPE html>... is simple response output.

JSF includes

```
<%@include file="header.jsf"%>
```

The code <%@include file="header.jsf"%> includes the file header.jsf of the current directory into the JSP. The root folder for referencing JSPs, JSFs or any static files is the configured htdocs folder of the Generator service. The file content of header.jsf is "simply" copied into the JSP at the include position before compiling.

Inline Java code

```
<%
String s = "JSPs are " + translate("great!");
out.println(s);
%>
```

The code between <% and %> is inline Java code.

and also

```
<%= request.getRequestURI() %>
```

The code <%= value %> writes the Java value into the response output just like out.print() would do.

Within the Java code the properties out (Response-Outputstream), request (HttpRequest), response (HttpResponse) can be used since they are declared when the Servlet is compiled.

3.1 Working with document properties

With an example of an article document this chapter shows how a JSP gets access to the document and document properties to produce an article website.

The following conditions for this JSP apply

- There is a document type **article.** It has a String property named headline.
- There is a document of type **article** in a folder. (i.e. /en/news/news1)
- The Generators generatorconfig.txt has an entry to use the page-article.jsp if a request targets a document of type **article** with the ending .html. The config entry looks as follows:

```
doctype=article, .html, page-article.jsp
```

- The Generator service configuration allows generating of documents in the folder /en.
- An URL http://myproject:[generator port]/en/news/news1.html is called.

With these preconditions applying the page-article.jsp is accessing the article document, reading and producing the articles headline into a web page.

```
<%@page import="
                app.cmsworks.cms.document.ErrorView,
                app.cmsworks.cms.document.DocumentModel,
```

```
                app.cmsworks.util.uilink.UILink,
                app.cmsworks.cms.document.HTMLErrorView
               "
        session="false"
        contentType="text/html;charset=UTF-8"
%><%@include file="includes/documentmodel.jsf"
%><%
DocumentModel dmPage = null;
ErrorView errors = new HTMLErrorView();
UILink uiLink = new UILink(request);
String htmlMetaTitle = "";
String htmlHeadline = "";
// fetch the sensible data
try {
  dmPage = new DocumentModel(request, new Types());
  errors.setPreview(dmPage);

  htmlHeadline = dmPage.getString(Types.PT_ARTICLE_HEADLINE, errors);
  htmlMetaTitle = htmlHeadline;
}
catch (Throwable t) {
  if (!errors.exit(response, t, dmPage, this.getClass().getName())) {
    %><%= errors.render() %><%
    return;
  }
}
//now start the HTML-Output
%><!DOCTYPE html>
<html lang="en">
  <head>
    <title><%= htmlMetaTitle %></title>
    <%= uiLink.getIncludes() %>
  </head>
  <body class="article">
    <%= uiLink.getPageLink() %>
    <h1><%= htmlHeadline %></h1>
    <%= errors.render() %>
  </body>
</html>
```

This example just shows the access to the article document when requesting an URL to a Generator JSP for an article.

The first line to access the article is this one:

```
dmPage = new DocumentModel(request, new Types());
```

With the request object from the Servlet the DocumentModel object will provide access to the Generator service (the service the JSP was executed in) as well as to the CMS service. The document is requested from the CMS service and is available too.

To get the field content from the String property headline the following line is used:

```
htmlHeadline = dmPage.getString(Types.PT_ARTICLE_HEADLINE, errors);
```

DocumentModel

This class represents a document and provides access to the CMS and to the interface of an UrlCreator. It wraps the document to extend the basic Resource implementation by adding an advanced error support accessing wrong or empty fields and supporting URL creation.

```
int documentId = 30;
DocumentModel dmAny = new DocumentModel(documentId, dmPage);
dmAny.getCMS(); // returning the CMS
dmAny.getMyService(); // returning the service the JSP is executed in
dmAny.getId(); // returning the document id
```

A DocumentModel can be created

- from a Request-Object of the Servlet
- from a document id and another DocumentModel
- form a document id and singularly the CMS service, the service implementing the UrlCreatorable interface and the Types object

Document property contents can be fetched with the following Methods:

```
dmAny.getString("propertyname");
dmAny.getText("propertyname");
dmAny.getInt("propertyname");
dmAny.getBlob("propertyname");
dmAny.getDate("propertyname");
dmAny.getLinkedResources("propertyname");
```

Types

The Types object is an implementation of the DocumentModelConstants interface adapted by the DocumentModelConstantsAdapter.

Primarily this class contains static constants defining the document types and properties of the projects document model.

Overriding the methods of the Adapter returning real values for special document types and property names enables the usage of special features like categories or teaser functions.

This types implementation is put into the documentmodel.jsf which is used in the example above.

```
<%@page import="app.cmsworks.cms.document.DocumentModelConstantsAdapter"
        pageEncoding="UTF-8"
%><%!
public class Types extends DocumentModelConstantsAdapter {
  /*
  Constant prefix RT means ResourceType
  Constant prefix PT means Property of Type
  */

  // ====================================================================
  public static final int RT_MEDIUM = 1;

  public static final String PT_MEDIUM_DATA = "data";
  public static final String PT_MEDIUM_WIDTH = "width";
  public static final String PT_MEDIUM_HEIGHT = "height";

  // ====================================================================
  public static final int RT_ARTICLE = 2;

  public static final String PT_ARTICLE_HEADLINE = "headline";
  public static final String PT_ARTICLE_TEXT = "text";
}

%>
```

Defining document type model constants in documentmodel.jsf

The constants about document type ids and property names combining the document type and field name secure the usage of those properties in the ongoing development. Always adjust the type definitions after changing the document model of the project. And all Errors while compiling the JSPs will show up to be adjusted to the new document model. Also no property name can be misspelled when using the constants.

ErrorView

The ErrorView is simply created with

```
ErrorView errors = new HTMLErrorView();
```

After initializing the DocumentModel from the request it has access to the Generator configuration and knows if the Generator creates a preview or a production (live) page.

This info has to be passed on to the ErrorView with

```
errors.setPreview(dmPage);
```

Every access to a document property could be on a wrong or empty field. In programming it would always be needed to check manually if the content is properly available. The ErrorView is handling those errors.

When accessing a document field content the ErrorView object can be added like this:

```
htmlHeadline = dmPage.getString(Types.PT_ARTICLE_HEADLINE, errors);
```

If the property content is missing, the ErrorView gets a new entry to be shown in the web page of the PreviewGenerator. Now the editor of the article document knows that for this web page the field content of the headline must be filled.

If the programmer used a wrong field a CMSPropertyAccessException arises which stops any further content processing. No exception is thrown if the field is just empty.

Further checking if the content is acceptable depends on the usage of the field content and must be implemented individually. So for example if the content is not empty but too long a self created Error message can be added:

```
errors.add("The headline is too long!");
```

To also create a reference to the missing field content a link can be added:

```
errors.add(errors.asUILink(dmPage) + "The headline is too long!");
```

If a missing content should lead to not producing the website and should abort any further processing, the following construct can be used:

```
if (htmlHeadline.length() > 200) {
  errors.abort(errors.asUILink(dmPage) + "The headline is too long!");
}
```

If a field content is optional do not use the ErrorView. The following may apply

```
if (dmPage.has(Types.PT_MY_PROPERTY)) {
  sbContent.append("<p>" + dmPage.getString(Types.PT_MY_PROPERTY) + "</p>");
}
```

For finally rendering the errors into the web page the following code is placed at the end of the body tag:

```
<%= errors.render() %>
```

Those error are only produced if the generator is configured as preview generator. In live/productive generators the render-call will produce nothing.

UILink

```
UILink uiLink = new UILink(request);
```

UILink is a class to provide connections to the documents used to create the web page. Mostly multiple documents are used in one website and those documents contents will be displayed. The UILink is a helper for editors to directly open such a used document for editing. Press Shift + Control to activate the Links in a preview website.

To create such a link the JSP has to contain:

```
<%= uiLink.getPageLink() %>
```

The document called by the request (dmPage) is referenced by this UILink.

When using other documents contents within the website those documents can also be accessible using:

```
<%= uiLink.getLink(dmAny.getResource()) %>
```

The document dmAny is some other document that is used in the page because it's a linked document or a search result or something.

To enable the engine of UILinks (activating the shortcut listener) the following code has to be placed at the head of the website:

```
<%= uiLink.getIncludes() %>
```

The following chapter will now deal with the access to documents properties in detail.

3.1.1 Document properties in detail

Accessing a document in a JSP producing a website in a Generator service is explained in the previous chapter. All about the several types of document properties is explained within the admisistrators guide and the users guide. Now we look into accessing content from the several types of properties.

Assuming the object dmPage is of Type DocumentModel created from the JSP request like:

```
DocumentModel dmPage = new DocumentModel(request, new Types());
```

Accessing a String property

Assuming the constant PT_STRING_PROP_NAME is a valid property of type *[String]* within the document dmPage.

```
String s = dmPage.getString(PT_STRING_PROP_NAME);
```

The result s is filled with the valid value or an empty String. The String value is never null so it is empty if the document field is not filled.

```
String s = dmPage.getString(PT_STRING_PROP_NAME, errors);
```

The result s is filled with the valid value or an empty String. If the value was empty, an error message appears in the website preview indicating what field of what document is not filled.

```
String defaultValue = "Fallbackstring";
String s = dmPage.getString(PT_STRING_PROP_NAME, defaultValue);
```

If using a defaultValue, this value is returned if the property is empty. In this case no warning is shown in the preview website. If necessary an error message could be added manually.

Accessing a Date property

Assuming the constant PT_DATE_PROP_NAME is a valid property of type *[Date]* within the document dmPage.

```
Date d = dmPage.getDate(PT_DATE_PROP_NAME);
```

The result d is filled with the valid date object or null.

```
Date d = dmPage.getDate(PT_DATE_PROP_NAME, errors);
```

The result d is filled with the valid date object or null. If the value was empty, an error message appears in the website preview indicating what field of what document is not filled.

```
// Fetching the changed date of the resource (document)
Date defaultValue = dmPage.getResource().getChanged();

// and passing that date as default
Date d = dmPage.getDate(PT_DATE_PROP_NAME, defaultValue);
```

If using a defaultValue, this value is returned if the property is empty. In the example the date falls back to the date of the latest change of the document (date of last editing or publishing)

Accessing an Integer property

Assuming the constant PT_INT_PROP_NAME is a valid property of type *[Integer]* within the document dmPage.

```
int i = mrPage.getInt(PT_INT_PROP_NAME);
```

The result i is filled with the valid int value or -1 if the property is empty.

```
int i = mrPage.getInt(PT_INT_PROP_NAME, errors);
```

The result i is filled with the valid int value or -1 if the property is empty. If the value was empty, an error message appears in the website preview indicating what field of what document is not filled.

```
int defaultValue = 0;
int s = mrPage.getInt(PT_INT_PROP_NAME, defaultValue);
```

If using a defaultValue, this value is returned if the property is empty. In this case no warning is shown in the preview website. If necessary add a warning manually.

Accessing a Blob property

Assuming the constant PT_BLOB_PROP_NAME is a valid property of type *[Blob]* within the document dmPage.

```
byte[] blobData = mrPage.getBlob(PT_BLOB_PROP_NAME);
```

The result blobData is filled with the valid byte array or null if the property is empty.

```
byte[] blobData = mrPage.getBlob(PT_BLOB_PROP_NAME, errors);
```

The result blobData is filled with the valid byte array or null if the property is empty. If the value was empty, an error message appears in the website preview indicating what field of what document is not filled.

At the mimetype of the blob property it's decided how to handle the binary data. To get the MimeType use:

```
MimeType mimetype = dmPage.getBlobMimeType(PT_BLOB_PROP_NAME);
```

The MimeType object combines the internal mimetype ID with the name (text/plain, image/jpg, ...) and the ending of a file (.txt, .jpg, ...). The property itself only stores the internal minetype ID.

Performance issues

Reading and processing blob data is an expensive action. It is an extra load for the database and the memory management. So it is advised to read the blob only if preconditions apply. For instance if only an text blob content can be used at this point the code should look like:

```
String blobText = "";
if (dmPage.has(Types.PT_BLOB_PROP_NAME) && dmPage.getBlobMimeType(Types.PT_BLOB_PROP_NAME).getName().startsWith("text/")) {
  blobText = new String(dmPage.getBlob(Types.PT_BLOB_PROP_NAME), "UTF-8");
}
```

dmPage.has() returns true if the property is not empty and this is a property access without the expensive reading of the blob.

Examining the property access examples above:

```
byte[] blobData = mrPage.getBlob(PT_BLOB_PROP_NAME);
```

This is an expensive access especially if the property is empty.

```
byte[] blobData = mrPage.getBlob(PT_BLOB_PROP_NAME, errors);
```

This is not expensive because internally the emptiness is checked before access. But it's still unknown, which mimetype the content has. And maybe the access was necessary because the mimetype does not fit.

Accessing a Text property

The text property value is a String containing valid SGML code. The text is almost HTML containing non standard links and text components that have to be transformed into either standard HTML for a website or other formats for other content types.

Assuming the constant PT_TEXT_PROP_NAME is a valid property of type *[Text]* within the document dmPage.

```
String text = mrPage.getString(PT_TEXT_PROP_NAME);
```

The result text is filled with the valid HTML code or an empty String if the property is empty.

If not links or components are to be handled this way of accessing the text is sufficient for this purpose. This may be the case in simple teaser text fields or in meta description fields for page document types. Also within the configuration of these fields for the editors desktop inserting links and components can be disabled.

To transform links and handle components use TinyRichtextParts as described in a separate chapter.

Accessing a Linklist property

Assuming the constant PT_LINKLIST_PROP_NAME is a valid property of type *[Linklist]* within the document dmPage.

To get all documents from a linklist property use:

```
DocumentModel[] arLinkedDocuments = dmPage.getLinkedResources(PT_LINKLIST_PROP_NAME);
```

If no document is linked, the Array is simply empty but never null.

To only get linked documents of special resource types use:

```
ModelResource[] arLinkedDocuments = mrPage.getLinkedResources(PT_LINKLIST_PROP_NAME, RT_ARTICLE);
```

```
ModelResource[] arLinkedDocuments = mrPage.getLinkedResources(PT_LINKLIST_PROP_NAME, new int[] {RT_ARTICLE, RT_MEDIUM});
```

The array arLinkedDocuments contains the ModelResource objects for the documents of the requested types or the list is empty.

To only get the first linked document of the linklist property of any document type use:

```
ModelResource mrLinkedDocument = mrPage.getLinkedResource(PT_LINKLIST_PROP_NAME, 0);
```

If no document is available, a null value is returned.

```
ModelResource mrLinkedDocument = mrPage.getLinkedResource(PT_LINKLIST_PROP_NAME, RT_ARTICLE, 0);
```

This is the way to get the first document from the linklist property that is of document type article.

Using the ErrorView in these Linklist property access methods is also possible. But creating individual Error messages if linked resources are missing is recommended to clarify the reason for the document relation.

3.1.2 Special document properties

Based on the standard property types there can be used some additional input types within the cmsWorks editors desktop.

An int property can be used for simple int values but also for a checkbox or a dropdown input.

A text property can be used as a grid input element.

Following is how the special inputs are used while generating websites.

Checkbox

If a checkbox is checked, the value 1 is stored, if not the value 0 is stored. To fetch the document property value use simply:

```
boolean checked = 1 == mrPage.getInt(PT_CHECKBOX_PROP_NAME, 0);
```

DropDown

The property type for this input element is *[Integer.]*

As an example there is a property articletype of type *[Integer]* within the document type **article**. To change the input field to a dropdown the configuration has to be placed within the q-custom.js of the editors desktop configuration.

```
"document": {
  "article": {
    "articletype_prop": {
      "xtype": "itwselectbox",
      "options": [
        [   0,'--- no type ---'],
        [   1,'News' ],
        [   2,'Interview' ],
        [   3,'Background story'],
        [   4,'Gossip']
      ]
    }
  }
}
```

In the example articles can be of type News, Interview, Background story or Gossip. To use the type names and IDs within JSPs to generate websites creating a helper class in an JSF include is recommended to map this "feature set"

and to be able to reuse it in a second JSP if necessary.

```
<%@page pageEncoding="UTF-8"
%><%!
/** Defines a list of constants to name the types of articles
 */
public static class ArticleType {

  final static int NO_TYPE = 0;
  final static int NEWS = 1;
  final static int INTERVIEW = 2;
  final static int BACKGROUND_STORY = 3;
  final static int GOSSIP = 4;

  public static String getName(int id) {
    switch (id) {
      case NO_TYPE          : return "no type";
      case NEWS             : return "News";
      case INTERVIEW        : return "Interview";
      case BACKGROUND_STORY : return "Background story";
      case GOSSIP           : return "Gossip";
    }
    return "unknown";
  }

}

%>
```

articletype.jsf to use articletypes by name

Reading the article type property value from the DocumentModel dmArticle is processed:

```
int articleType = dmArticle.getInt(Types.PT_ARTICLE_TYPE);
String topLine = ArticleType.getName(articleType);
if (articleType == ArticleType.GOSSIP) {
  headlineColor = "green";
}
```

Grid

The property type for this input element is *[Text.]*

As an example there is a property somegrid of type *[Text]* within the document type **article**. To change the input field to a Grid the configuration has to be placed within the q-custom.js of the editors desktop configuration.

```
"document": {
  "article": {
    "somegrid_prop": {
      "xtype": "itwdatagrid",
      "info": "Some information about the purpose and the structure of the grid"
    }
  }
}
```

The input field content is structured by rows, columns and column titles. Cell contents can be either strings or links to documents. Those links are encoded just like the links produced in the standard rich text input element.

In the JSP reading the Grid information the text has to be fetched not from getText() but vom getString(). This is to avoid using the StandardTextFilter changing the links. A textfilter class DataGrid supports the access to the grid contents:

```
DataGrid dg = new DataGrid(dmPage.getString(PT_SOMEGRID_PROP_NAME));
StringBuffer sbTable = new StringBuffer();

// Create a HTML table
sbTable.append("<table>");

// Create the  HEADER
sbTable.append("<tr>");
for (int colIdx = 0; colIdx < dg.getColumnCount(); colIdx++) {
  sbTable.append("<th>");
  sbTable.append(dg.getTitle(colIdx));
  sbTable.append("</th>");
}
sbTable.append("</tr>");

// Create the Body grid
for (int rowIdx = 0; rowIdx < dg.getRowCount(); rowIdx++) {
  sbTable.append("<tr>");
  for (int colIdx = 0; colIdx < dg.getColumnCount(); colIdx++) {
    sbTable.append("<td>");
    if (colIdx == 2) {// only the third column has links to documents
      try {
        sbTable.append(new DocumentModel(dg.getDocumentId(colIdx, rowIdx), dmPage).toLink().getUrl());
      }
      catch (Throwable t) {
        // ignore not existing links to documents
      }
    }
    else {
      sbTable.append(dg.getField(colIdx, rowIdx));
    }
    sbTable.append("</td>");
  } // end of for columns
  sbTable.append("</tr>");
} // end of for rows
sbTable.append("</table>");
```

GridMap

If a Text field in a document is configured as in the example Grid, but the purpose is to create just two columns to enter key value pairs and the keys should be unique, there is a helper class that reads the map values into a HashMap.

The DataGridMap textfilter class can be used like this:

```
String MAP_KEY_COMPONENT_TITLE   = "Component Title";
String MAP_KEY_COMPONENT_PICTURE = "Component Some Picture";

DataGridMap dgMap = new DataGridMap(dmPage.getString(PT_SOME_GRID_PROP_NAME));
String componentTitle = dgMap.get(MAP_KEY_COMPONENT_TITLE);
ModelResource dmMed = null;
try {
  dmMed = new DocumentModel(dgMap.getDocumentId(MAP_KEY_COMPONENT_PICTURE), dmPage);
}
catch (Throwable t) {}
```

This Grid should have at least 2 columns. If not, the internal Map of the DataGridMap will be empty.

If no entry for the key Component Title is available the componentTitle value is null.

If no entry for the key Component Some Picture is available, the Exception ResourceNotFoundException will be thrown because the document id is -1 (default for "no value"). The effect after the try-catch block is that the mrMed reference is null.

3.2 Creating URLs

There are different types of documents that can be created in the CMS. When using page types like an article or a news page the HTML page will have links to to pages filled from those documents. But also there will be document types like teaser documents or navigation documents which will be used to structure informations but not be produced as pages by themself.

The Generators generatorconfig.txt declares which document types are page types where URLs can be created to for instance with

```
doctype=news, .html, page-news.jsp
doctype=article, .html, page-article.jsp
```

Also it declares all document types which are allowed for includes like

```
doctype=teaser, .html
doctype=container, .html
```

Page types are also automaticly includeable types.

And finally if a medium document type contains a blob property which should be addressed by a URL the declaration should be

```
blobtype=medium, data, blob.jsp
```

The Generator service is in charge of the strategy to create URLs. The DocumentModel gives a simple access to the default URLs.

String url = dmPage.toLink().getUrl();

Referenceables

If the editor should be enabled to create more advanced links there is the Referenceable interface containing the following fields

Internal link
: of type Linklist

External link
: of type String

Link type
: an integer from a dropdown selection

Link paramter
: of type String for additional parameters

A navigation document or a teaser document would have all those fields beside others like a navi title or a teaser picture.

These fields enable to link not only to another document but also to an external URL. It gives the option to open the link in a new window or with other link types to trigger any tracking.

To use one central interface to create an URL from this informations from different document types the DocumentModelConstantsAdapter can be configured within the documentmodel.jsf.

```
public int[] getRT_REFERENCEABLE() { return new int[] { RT_REFERENCE, RT_NAVIGATION };}
public String getPT_REFERENCE_LINK_INTERNAL() { return PT_REFERENCE_LINK_INTERNAL; }
public String getPT_REFERENCE_LINK_EXTERNAL() { return PT_REFERENCE_LINK_EXTERNAL; }
public String getPT_REFERENCE_LINK_TYPE() { return PT_REFERENCE_LINK_TYPE; }
public String getPT_REFERENCE_LINK_PARAMETER() { return PT_REFERENCE_LINK_PARAMETER; }
```

The field names of the Reference fields have to be named the same in every referenceable document type.

After that a RefernceModel can be used to create URLs either from page documents or from Referenceable documents.

String url = new app.cmsworks.cms.document.ReferenceModel(dmAny).toLink().getUrl().

If the link is optional, the Method toLink() will return null if the link fields are not filled or if the generator has no configuration to create a link to the targeted document. So this should be checked.

```
String text = "I am a link";
app.cmsworks.cms.document.Link link = new ReferenceModel(dmAny).toLink();
if (link != null) {
  text = "<a " + link.createAnchorTarget() + ">" + text + "</a>";
}
```

But if the link is mandatory, use the ErrorView to handle this.

```
String text = "<a " + new ReferenceModel(dmAny).toLink(errors).createAnchorTarget() + ">I am a link</a>";
```

3.3 Handle text with TinyRichtextParts

The richtext in a document is stored in a property of type *[text]*. The text property value is a String beeing almost normal HTML that could be processed by a DOM parser. Only links and components are not standard HTML and have to be transformed into either standard HTML for a website or other formats for other content types.

The href attribute of the A (anchor) tag has encoded either an external URL or a document ID as link target. Also a link type and optional additional parameters can be found there. From the document ID the Generator can produce an URL.

TinyRichtextParts

The app.cmsworks.util.text.tiny.TinyRichtextParts provide a lightweight but powerful way of handling various text manipulations.

Constructor

After giving the the text into the constructor the TinyRichtextParts creates a list of TinyRichtextPart objects representing a text part, a tag part (opening or closing) or a text component part.

The structure is not a tree, just a list referencing the different text parts in order. This list is not to be changed by removing, adding or exchanging elements.

An opening tag part is referencing it's closing part, contains a level.

Common functions of all TinyRichtextPart types

setText(String)
: To replace the origin text simply set a new text to be produced in the final product

setTextBefore(String), setTextAfter(String), addTextBefore(String), addTextAfter(String)
: Add some text before/after a text part.

setDoProduce(boolean)
: To ignore this text part for the final product call setDoProduce(false). In this case also before and after Strings of this part will not be produced.

TinyRichtextTextPart

It does not add any functionality to the common functions.

TinyRichtextTagPart

- It contains the state if it's the opening or closing tag part or for example in case of a BR tag beeing both at the same time.
- If it's the opening part it references its closing part
- It contains a level info about the depth in the hierarchy of tags
- It manages an Attribute list.

getName()
: Returns the tag name

setName(String)
: For setting a new tag name

isNamed(String)
: Helper for searching tags with special names

isOpeningTagPart(), isClosingTagPart(), getLevel()
: Returning the state

setDoProduce(boolean)
: not only for this tag part but also for the closing tag part if it is referenced

getClosingTagPart()

Returns the closing text part or null if not existing

getAttributeNames(), getAttribute(String), setAttribute(String, String), removeAttribute(String)

Managing attributes of the tag

TinyRichtextComponentPart

getDocumentId()

If a component is to be rendered from a Document

getPosition()

The positioning of the component in the text (left/right/inline/centered)

setText(String)

After producing the component the component code (HTML in most cases) is inserted into the text here.

TinyRichtextParts

getTextParts()

Returns the full list of text parts as TinyRichtextPart objects.

getTextParts(TinyRichtextPart)

returns a sublist of this text part. If the given TinyRichtextPart is actually an opening TinyRichtextTagPart all text parts are returned beginning at the opening tag and ending at the closing tag.

build(), build(TinyRichtextPart), build(idxStart, idxEnd)

Joining the text parts to a new text as a result after handling links, text components and any other creative text manipulations.

getUnTaggedText(), getUnTaggedText(idxStart, idxEnd)

It returns the collected text from each TinyRichtextPartText making sure that a space is added between block element texts.

hasOnlyEmptyText()

It tests if a tag has no visible text - whitespaces including   are ignored.

To use the TinyRichtextParts for website texts more text handling is necessary especially to handle links created from the Generator. In this case a sub class is created for example in a util-textparts.jsf.

```
<%@page import="
                app.cmsworks.cms.document.DocumentModel,

                app.cmsworks.util.text.tiny.TinyRichtextParts,
                app.cmsworks.util.text.tiny.TinyRichtextPart,
                app.cmsworks.util.text.tiny.TinyRichtextTagPart,
                app.cmsworks.util.text.tiny.TinyRichtextTextPart,
                app.cmsworks.util.text.tiny.TinyRichtextComponentPart,
                app.cmsworks.util.text.tiny.TinyRichtextLinkTransformerHTML,

                java.util.ArrayList
               "
        pageEncoding="UTF-8"
%><%!

/** The subclass of TinyRichtextParts will handle links and components
 */
public class MyTinyRichtextParts extends TinyRichtextParts {

  /** Constructor parsing the given text
   */
  public MyTinyRichtextParts(String text) {
    super(text);
  }

  /** Find all p elements with no component and no text inside to deaktivate them
   */
  public void deleteEmptyP() {
    // for each opening tag part named p
    for (TinyRichtextTagPart trpTag : getOpeningTagsNamed("p")) {
      // having no class attribute
      if (trpTag.getAttribute("class") == null) {
        // having no component and no text
        if (hasNoComponentAndOnlyEmptyText(trpTag)) {
          // set doProduce=false for this tag and all content text parts
          removeAll(trpTag);
        }
      }
    }
  }

  /** Collecting and returning the component parts of the text
   */
  public TinyRichtextComponentPart[] getComponentParts() {
    // create a list to collect the component parts in
```

```
    ArrayList<TinyRichtextComponentPart> list = new ArrayList<TinyRichtextComponentPart>();
    // for each text part
    for (TinyRichtextPart trp : getTextParts()) {
      // if the text part is still to be produced
      if (trp.doProduce()) {
        // if the text part is a component part
        if (trp instanceof TinyRichtextComponentPart) {
          // add it into the list
          list.add((TinyRichtextComponentPart) trp);
        }
      }
    }
    // return the array from the list
    return list.toArray(new TinyRichtextComponentPart[list.size()]);
  }

  /** Working the links of the text
   */
  public void handleLinks(DocumentModel dmAny) throws Exception {
    // for each link
    for (TinyRichtextTagPart trpA : getOpeningTagsNamed("a")) {
      // create a Transformer
      TinyRichtextLinkTransformerHTML tran = new TinyRichtextLinkTransformerHTML(trpA);
      // set a resolved URL if it's an internal link
      tran.resolveUrl(dmAny.getUrlCreator());
      // do produce the link in propert HTML coding
      tran.transform();
    }
  }

}

%>
```

Exteding the TinyRichtextParts to handle links, removing empty lines and provide access to text components.

Within a page-xxx.jsp the MyRichtextParts can be used to render text into a HTML page.

```
<%@page import="
                app.cmsworks.cms.document.ErrorView,
                app.cmsworks.cms.document.HTMLErrorView,
                app.cmsworks.util.uilink.UILink,
                app.cmsworks.cms.document.ContentInclude
               "
        session="false"
        contentType="text/html;charset=UTF-8"
%><%@include file="includes/documentmodel.jsf"
%><%@include file="includes/util-texthandler.jsf"
%><%

ErrorView errors = new HTMLErrorView();
DocumentModel dmPage = null;
UILink uiLink = new UILink(request);

String htmlText = "";

// fetch the sensible data
try {
  // get the DocumentModel from request
  dmPage = new DocumentModel(request, new Types());
  // init the errorview
  errors.setPreview(dmPage);

  // use the created MyTinyRichtextParts with the text to render
  MyTinyRichtextParts textPartsText = new MyTinyRichtextParts(dmPage.getString(Types.PT_TEXT));

  // transform the links
  textPartsText.handleLinks(dmPage);

  // remove empty lines
  textPartsText.deleteEmptyP();

  // fill components
  for (TinyRichtextComponentPart compPart : textPartsText.getComponentParts()) {
    DocumentModel mrComponent = new DocumentModel(compPart.getDocumentId(), dmPage);
    ContentInclude include = null;
    if (mrComponent.isType(Types.RT_MEDIUM)) {
      include = new ContentInclude(dmPage, mrComponent, "cmp-medium.jsp", errors);
    }
    else if (mrComponent.isType(Types.RT_INFOBOX)) {
      include = new ContentInclude(dmPage, mrComponent, "cmp-infobox.jsp", errors);
    }
    ...
    String htmlComponent = null;
    if (include != null) {
      htmlComponent = include.getIncludeContent();
    }
    // if the include has errors or was not assigned by the conditions before the compPart will simply
    // be deactivated
    if (htmlComponent != null) {
      htmlComponent = "<span class=\"txtcomp " + compPart.getPosition() + "\">" + htmlComponent + "</span>";
      compPart.setText(htmlComponent);
```

```
    }
    else {
      compPart.setDoProduce(false);
      errors.add(errors.asUILink(mrComponent) + "This document is not usable as text component.");
    }
  }

  htmlText = textPartsText.build();
}
catch (Throwable t) {
  if (errors.exit(response, t, dmPage, this.getClass().getName())) {
    return;
  }
}
//now start the HTML-Output
%><!DOCTYPE html>
<html>
<head>
...
  <%= uiLink.getIncludes() %>
</head>
<body>
...
  <!-- placing the text and editable markers into the html code -->
  <div class="richtext"><%= htmlText %></div>
...
  <%= errors.render() %>
</body>
</html>
```

Using MyRichtextParts to render richtext into the HTML page.

3.4 Text components and cmsWorks includes

Text components are special tags within a rich text property of a cmsWorks document. They are non standard HTML tags containing the ID of a component cmsWorks document and an information about the positioning of the component.

Allowing text components in a single text property of a document is configured in the q-custom.js of the cmsWorks editors desktop. Also the types of positioning options are configured there.

Why text components

Text components appear to be simple at the first glance, but they are not. Almost every component comes with an additional feature (magnify the picture) or with a structure of information (add a title and a copyright to the picture). The strategy is to separate the content management from presentation. Components could be rendered differently for various products (HTML website / PDF / XML / JSON). Components of the same type should appear always in the same manner. Decide in your programming which information are mandatory and which are optional. Components can be reused in several locations of the website. Change the component content once and all appearances will update immediately. Avoid to copy those contents except for special reasons.

Example of text components

The following code shows the integration of text components using the TinyRichtextParts.

```
<%@page import="
                app.cmsworks.cms.document.ErrorView,
                app.cmsworks.cms.document.HTMLErrorView,
                app.cmsworks.util.uilink.UILink,
                app.cmsworks.cms.document.ContentInclude
               "
        session="false"
        contentType="text/html;charset=UTF-8"
%><%@include file="includes/documentmodel.jsf"
%><%@include file="includes/util-texthandler.jsf"
%><%

ErrorView errors = new HTMLErrorView();
DocumentModel dmPage = null;
UILink uiLink = new UILink(request);

String htmlText = "";

// fetch the sensible data
try {
  // get the DocumentModel from request
  dmPage = new DocumentModel(request, new Types());
  // init the errorview
  errors.setPreview(dmPage);
```

```
  // use the created MyTinyRichtextParts with the text to render
  MyTinyRichtextParts textPartsText = new MyTinyRichtextParts(dmPage.getString(Types.PT_TEXT));

  // transform the links
  textPartsText.handleLinks(dmPage);

  // remove empty lines
  textPartsText.deleteEmptyP();

  // fill components
  for (TinyRichtextComponentPart compPart : textPartsText.getComponentParts()) {
    DocumentModel mrComponent = new DocumentModel(compPart.getDocumentId(), dmPage);
    ContentInclude include = null;
    if (mrComponent.isType(Types.RT_MEDIUM)) {
      include = new ContentInclude(dmPage, mrComponent, "cmp-medium.jsp", errors);
    }
    else if (mrComponent.isType(Types.RT_INFOBOX)) {
      include = new ContentInclude(dmPage, mrComponent, "cmp-infobox.jsp", errors);
    }
    ...
    String htmlComponent = null;
    if (include != null) {
      htmlComponent = include.getIncludeContent();
    }
    // if the include has errors or was not assigned by the conditions before the compPart will simply
    // be deactivated
    if (htmlComponent != null) {
      htmlComponent = "<span class=\"txtcomp " + compPart.getPosition() + "\">" + htmlComponent + "</span>";
      compPart.setText(htmlComponent);
    }
    else {
      compPart.setDoProduce(false);
      errors.add(errors.asUILink(mrComponent) + "This document is not usable as text component.");
    }
  }

  htmlText = textPartsText.build();
}
catch (Throwable t) {
  if (errors.exit(response, t, dmPage, this.getClass().getName())) {
    return;
  }
}
//now start the HTML-Output
%><!DOCTYPE html>
<html>
<head>
...
  <%= uiLink.getIncludes() %>
</head>
<body>
...
  <!-- placing the text and editable markers into the html code -->
  <div class="richtext"><%= htmlText %></div>
...
  <%= errors.render() %>
</body>
</html>
```

Using MyRichtextParts to render richtext into the HTML page.

The loop

```
 for (TinyRichtextComponentPart compPart : textPartsText.getComponentParts()) {
...
}
```

iterates over the component parts. The document is fetched from the ComponentParts document ID. And after identifying the document type a component include is performed.

In the example either a document of type **Medium** or **Infobox** is handled. Any other document type produces an Error that is shown in the preview page but not in the production/live page.

```
htmlComponent = "<span class=\"txtcomp " + compPart.getPosition() + "\">" + htmlComponent + "</span>";
compPart.setText(htmlComponent);
```

With these two lines the created component is written into the component text part. Adding the position as a class to a span tag surrounding the component code enables the positioning using CSS within the HTML page.

ContentInclude

```
ContentInclude include = new ContentInclude([calling document], [called document], [JSP], [ErrorView]);
String htmlComponent = include.getIncludeContent();
```

The actual include is done with these two lines.

This executes an HTTP call to the own Generator targeting the URL of the called document, using the named JSP to be executed and showing an error if the call was not successful. Reasons may be: The called document is not allowed in the Generator config. The JSP does not exist. The JSP answered with an error code.

An example of a JSP producing a component:

```
<%@page import="
                app.cmsworks.cms.document.ErrorView,
                app.cmsworks.cms.document.HTMLErrorView,
                app.cmsworks.util.uilink.UILink,
                app.cmsworks.util.uilink.PreviewEditable
               "
        session="false"
        contentType="text/html;charset=UTF-8"
%><%@include file="includes/documentmodel.jsf"
%><%@include file="includes/util-texthandler.jsf"
%><%
ErrorView errors = new HTMLErrorView();
DocumentModel dmCmp = null;
UILink uiLink = new UILink(request);

try {
  dmCmp = new DocumentModel(request, new Types());
  errors.setPreview(dmCmp);

  PreviewEditable previewEditable = new PreviewEditable(request);

  StringBuffer sbContent = new StringBuffer();

  String htmlTitle = dmCmp.getString(Types.PT_INFOBOX_TITLE);
  if (htmlTitle.length() > 0) {
    htmlTitle = previewEditable.createEditableMarker(dmCmp, Types.PT_INFOBOX_TITLE) +
        "<div class=\"inf-title\">" + htmlTitle + "</div>";
    sbContent.append(htmlTitle);
  }
  MyTinyRichtextParts textParts = new MyTinyRichtextParts(dmCmp.getString(Types.PT_INFOBOX_TEXT));
  String htmlTextEditable = previewEditable.createEditableMarkerForText(dmCmp, Types.PT_INFOBOX_TEXT, textParts);

  textParts.handleLinks(dmCmp);
  textParts.deleteEmptyP();

  String htmlText = textParts.build();
  if (htmlText.length() > 0) {
    htmlText = "<div class=\"inf-body\">" + htmlTextEditable + htmlText + "</div>";
    sbContent.append(htmlText);
  }
  // now start the HTML-Output!
  if (errors.isEmpty()) {
    %>
<%= uiLink.getPageLink() %>
<%= sbContent.toString() %>
    <%
  }
}
catch (Throwable t) {
  if (errors.exit(response, t, dmCmp, this.getClass().getName())) {
    return;
  }
}
%><%= errors.render() %>
```

The cmp-infobox.jsp produces a component to be used in richtext but also in any other possible context.

In this example the info box headline and text are optional. If no field is filled the returning code is simply empty.

The general strategy for components should be: If an error arises the component should not be produced at all. This is why all production is executed within the try {} catch () {} block. If errors are produced they are presented in the page calling this include (always only in preview).

3.5 Preview Text Editing

The Preview editing allows the Editor to edit Texts from String or Text fields directly within the preview HTML page.

When an editor is checking the preview he may be tempted to correct a spelling error directly in the text or even rewrite individual text passages. This feature enables this action. With the shortcut Shift + Control not only UILinks but also Edit buttons appear if the programming supports it.

The following example shows how the article headline and the text are marked Preview editable.

```
<%@page import="
                app.cmsworks.cms.document.ErrorView,
                app.cmsworks.cms.document.HTMLErrorView,
                app.cmsworks.util.uilink.UILink,
                app.cmsworks.util.uilink.PreviewEditable,
                app.cmsworks.cms.document.ContentInclude
               "
        session="false"
        contentType="text/html;charset=UTF-8"
%><%@include file="includes/documentmodel.jsf"
%><%@include file="includes/util-texthandler.jsf"
%><%

ErrorView errors = new HTMLErrorView();
DocumentModel dmPage = null;
UILink uiLink = new UILink(request);
PreviewEditable previewEditable = new PreviewEditable(request);

String htmlHeadlineEditable = "";
String htmlHeadline = "";

String htmlText = "";
String htmlTextEditable = "";

// fetch the sensible data
try {
  // get the DocumentModel from request
  dmPage = new DocumentModel(request, new Types());
  // init the errorview
  errors.setPreview(dmPage);

  // use previewEditable to create the marker for headline text to be editable
  htmlHeadlineEditable = previewEditable.createEditableMarker(dmPage, Types.PT_ARTICLE_HEADLINE);
  htmlHeadline = dmPage.getString(Types.PT_ARTICLE_HEADLINE, errors);

  // use the created MyTinyRichtextParts with the text to render
  MyTinyRichtextParts textPartsText = new MyTinyRichtextParts(dmPage.getString(Types.PT_TEXT));
  htmlTextEditable = previewEditable.createEditableMarkerForText(dmPage, Types.PT_TEXT, textPartsText);

  // transform the links
  textPartsText.handleLinks(dmPage);

  // remove empty lines
  textPartsText.deleteEmptyP();

  // fill components
  for (TinyRichtextComponentPart compPart : textPartsText.getComponentParts()) {
    DocumentModel mrComponent = new DocumentModel(compPart.getDocumentId(), dmPage);
    ContentInclude include = null;
    if (mrComponent.isType(Types.RT_MEDIUM)) {
      include = new ContentInclude(dmPage, mrComponent, "cmp-medium.jsp", errors);
    }
    else if (mrComponent.isType(Types.RT_INFOBOX)) {
      include = new ContentInclude(dmPage, mrComponent, "cmp-infobox.jsp", errors);
    }
    ...
    String htmlComponent = null;
    if (include != null) {
      htmlComponent = include.getIncludeContent();
    }
    // if the include has errors or was not assigned by the conditions before the compPart will simply
    // be deactivated
    if (htmlComponent != null) {
      htmlComponent = "<span class=\"txtcomp " + compPart.getPosition() + "\">" + htmlComponent + "</span>";
      compPart.setText(htmlComponent);
    }
    else {
      compPart.setDoProduce(false);
      errors.add(errors.asUILink(mrComponent) + "This document is not usable as text component.");
    }
  }

  htmlText = textPartsText.build();
}
catch (Throwable t) {
  if (errors.exit(response, t, dmPage, this.getClass().getName())) {
    return;
  }
}
//now start the HTML-Output
%><!DOCTYPE html>
<html>
<head>
...
  <%= uiLink.getIncludes() %>
</head>
<body>
...
  <%= uiLink.getPageLink() %>
...
  <%= htmlHeadlineEditable %><h1><%= htmlHeadline %></h1>
...
  <!-- placing the text and editable markers into the html code -->
  <div class="richtext"><%= htmlTextEditable %><%= htmlText %></div>
```

```
...
  <%= errors.render() %>
</body>
</html>
```

Using MyRichtextParts to render richtext into the HTML page, also create editable markers for headline and text.

With the following code the Preview editing feature is initialized:

```
PreviewEditable previewEditable = new PreviewEditable(request);
```

The PreviewEditable reads the Generator service from the request and only creates code if the Generator is in preview mode.

Using the feature for the article headline is prepared as follows:

```
htmlHeadlineEditable = previewEditable.createEditableMarker(dmPage, Types.PT_ARTICLE_HEADLINE);
```

The method reads the property from the given document and the property name and creates a marker code. This code has to be placed before the HTML element containing the article headline. This is how *[string]* property editor markers have to be placed.

```
<%= htmlHeadlineEditable %><h1><%= htmlHeadline %></h1>
```

Do not place the marker into the h1 element except if the headline text is additionally surrounded by another tag (i.e. span)

```
<!-- this will work -->    <h1><%= htmlHeadlineEditable %><span><%= htmlHeadline %><span></h1>
<!-- this will NOT work --><h1><%= htmlHeadlineEditable %><%= htmlHeadline %></h1>
```

For properties of type *[text]* the marker has to be placed right at the beginning of the text:

```
<div class="text"><%= htmlTextEditable %><%= htmlText %></div>
```

When editing

When starting the property editing in the preview website the document is locked for the CMS user just like it would be if the document is edited in the document window of the CMS editors desktop. Only after aborting or storing the change the document will be unlocked. So it's necessary to be logged in into the editors desktop (in the same cookie context - meaning same browser, same incognito or normal mode).

Editing is split at components

The rendered version of the text in the website may be transformed and filled with included component renderings. To avoid editing those component renderings the preview editing is split into the section before the component and after. The text to be edited is not the transformed Text but the original Text from the document content. This splitting may also be happening if the transformation of the text handles tables by surrounding them with a scrollable container working mostly in mobile presentation where tables are larger than the page with.

Links are editable too

An internal link to another document can be placed in the editors desktop by copy and paste of a document into the Linklist property of the link dialog. This cannot be performed within the website preview. But. Use the preview link of the link target or an online link or simply the document ID. After editing the link the server tries to find the targeted document from the id or the path and rewrites the link to an internal link (if you allow this). Also the presentation of the existing internal link to another document is transformed into the generated relative URL.

4 Using the cmsWorks document search

cmsWorks comes with a built in big data search engine based on Lucene. On any change of cmsWorks documents the contents are updated in the search index of the search service. The search service can be called to query results for search keys and as result a list of document IDs is returned.

Usually there is more than one search index configured in cmsWorks.

An index indexintern is used to contain all information about all documents in all states and is used within the cmsWorks desktop for all search documents features.

An index indexonline can be configured that only published versions of documents are contained. Every type of document is to be named and everty property is to declared the contains relevant content for the online search. So if the updated document not declared, the indexonline is not updated. In this configuration it's also possible to declare: If an article title is not filled, the article is not valid and therefor not to be found in the indexonline of the search service. All details about the configuration is contained in the administrators guide.

The search service is accessible via HTTP requests. The Generator service has a property configured listing remote hosts. So within a JSP of the Generator service producing a website the host information of the search service can be fetched to send a query for documents to be found.

The search example

In the example using the search service a component JSP is created. The component shall search for the latest News and produce a list of Links using the articles titles to be linked.

```
<%@page import="
                app.cmsworks.service.generator.Generator,
                app.cmsworks.cms.document.DocumentModel,
                app.cmsworks.cms.document.ErrorView,
                app.cmsworks.cms.document.Link,
                app.cmsworks.cms.document.HTMLErrorView,
                app.cmsworks.util.uilink.UILink,

                app.cmsworks.util.search.SearchResultIdIterator,
                app.cmsworks.util.search.term.SearchUtil,
                app.cmsworks.util.search.term.SearchTerm
               "
        session="false"
        contentType="text/html;charset=UTF-8"
%><%@include file="includes/documentmodel.jsf"
%><%@include file="includes/articletype.jsf"
%><%
DocumentModel dmCmp = null;
ErrorView errors = new HTMLErrorView();
UILink uiLink = new UILink(request);
// fetch the sensible data
try {
  dmCmp = new DocumentModel(request, new Types());
  Generator generator = (Generator) dmCmp.getMyService();
  errors.setPreview(dmCmp);

  // create a Search utitlity object
  SearchUtil searchUtil = new SearchUtil();
  // set the search host
  searchUtil.setHost(generator.getHostData("search"));
  // set the name of the index to search in
  searchUtil.setIndex(SearchUtil.INDEX_ONLINE);
  // one search call will produce a maximum of 50 results
  searchUtil.setIteratorPageSize(50);

  // only find documents of type article
  searchUtil.filterDocumentType(Types.RT_ARTICLE);

  // add the search keywords
  SearchTerm searchTerm = searchUtil.createSearchTermAnd();
  // filter all articles by the article type news
  searchUtil.addIDs(searchTerm, new int[]{ArticleType.NEWS}, Types.PT_ARTICLE_TYPE);

  StringBuffer sb = new StringBuffer();
  // execute the search
  SearchResultIdIterator srii = searchUtil.search(searchTerm);
  int cnt = 0;
  // only produce 10 valid news links
  while(srii.hasNextId() && cnt < 10) {
    int id = srii.getNextId();
    DocumentModel dmNews = new DocumentModel(id, dmCmp);
    if (dmNews.isType(Types.RT_ARTICLE)) {
      int articleType = dmNews.getInt(Types.PT_ARTICLE_TYPE);
      String headline = dmNews.getString(Types.PT_ARTICLE_HEADLINE);
      Link link = dmNews.toLink();
      if (articleType == ArticleType.NEWS && headline.length() > 0 && link != null) {
```

```
        sb.append("<li><a " + link.createAnchorTarget() + ">" + headline + "</a></li>");
        cnt++;
      }
    }
  }

  String htmlNewsList = sb.toString();

%>
<div class="news">
  <%= uiLink.getPageLink() %>
  <h2>News</h2>
  <ul><%= htmlNewsList %></ul>
</div>
<%
}
catch (Throwable t) {
  if (!errors.exit(response, t, dmCmp, this.getClass().getName())) {
    return;
  }
}
%><%= errors.render() %>
```

Using the Search service to find news for a news component in cmp-news.jsp

This example relies on different includes we created beforehand:

- documentmodel.jsf - containing constants for all document types and property names of the project
- articletype.jsf - containing constants for types of articles

Single steps to walk through the search creating ten news links on articles

The search support object is created:

```
SearchUtil searchUtil = new SearchUtil();
```

SearchUtil is the main object retrieving information on where and what to search. At first the information about the HTTP-Request to the search service will be filled in. Therefore the host and port of the configured host (Generator service configuration) is needed (here: "search"). Additionally the name of the search index has to be announced:

```
searchUtil.setHost(dmCmp.getGenerator().getIncludeHost("search"));
searchUtil.setIndex(SearchUtil.INDEX_ONLINE);
```

To limit the count of search results a maximum of results to be returned can be set. Otherwise default value is 100:

```
searchUtil.setIteratorPageSize(50);
```

Restricting the search in this way doesn't mean that the iteration will stop after 50 results; it merely is a method to keep the memory usage footprint as low as possible.

Next up is to declare that only documents to the document type **article** should be found:

```
searchUtil.filterDocumentType(Types.RT_ARTICLE);
```

The search term in our case is just an expression saying that only articles of type *[News]* should be returned. No further restrictions are needed.

The SearchTerm is used to create query information. The search uses a query language that is wrapped by the SearchTerm object. There can be conditions created like the following examples:

- word1 - find all documents containing word1
- word1 and word2 - find all documents containing word1 and word2
- word1 or word2 - find all documents with either word1 containing or word2 containing or both
- (word1 or word2) and word3 - find all documents containing word3 and word1 or containing word3 and word2

So a tree can be created of words to be found and conditions AND and OR. The next code line creates a Term with the condition AND.

```
SearchTerm searchTerm = searchUtil.createSearchTermAnd();
```

The SearchTerm now can collect words under the condition as well as other SearchTerms with other conditions.

To create the condition to find only articles of type news:

```
searchUtil.addIDs(searchTerm, new int[]{ArticleType.NEWS}, Types.PT_ARTICLE_TYPE);
```

This method internally creates a new SearchTerm with the condition OR adding all ID-words and adding the created SearchTerm to the given searchTerm. Meaning: if more than one ID would be added:

```
utilSearch.addIDs(searchTerm, new int[]{ArticleType.NEWS, ArticleType.BACKGROUND_STORY, ArticleType.GOSSIP}, Types.PT_ARTICLE_T
YPE);
```

the search will find articles of type News or of type background or ...

Now the search is prepared and the search keywords are defined. The execution of the search can be started and the search results can be fetched.

```
SearchResultIdIterator srii = searchUtil.search(searchTerm);
while(srii.hasNextId()) {
  int id = srii.getNextId();
}
```

This loop will request all search results there are. Assuming there would be 63 hits, after the first 50 hits a second search request would have been fired to fetch search results after the first 50 hits which would return the last 13 hits. But this is done in the engine and the user has not to worry about that.

The rest of the code within the loop is some defensive way of reading the property of the search results. Is the document really an article? Is it of the correct type? Does it have a filled headline? Ok, than produce a link. 10 valid articles should be found in the first 50 hits of the search.

Conditions of the search

The content of a document is basically not typed in the search index.

When searching for a word and a document is returned as a hit it's still unknown which property this word contains.

Hits are not weighted.

Meaning if the word exists multiple times in one document and only one time in another document it's no difference to the order of hits.

The order of hits is always based on the configured date configuration

Either a document property of type date or the date of the latest change of the document is the criteria to the order of results.

There is no configuration, that a word found in a keyword field is more important than a word found in a headline field or in a text field. If a search is need where only hits in a keyword field is acceptable, create a special keyword search index for that reason.

5 Categories

Categories contain content, configuration or references that are used in several pages or content elements. Categories are organised as a tree where at each level information can be inherited or overridden. A document can have a category input element creating a reference to the category document chosen from the category tree.

Use cases

A configuration task is a document of a special type that defined by its type is assigned to a category document. This configuration task declares a special configuration type for instance a navi reference, an AD configuration, a header/footer layout/content/config, ... So maybe in the production of a page header the category is requested for the header logo task and the internal logic will find a configuration at this category level or at any category level above.

While several pages or content entities reference categories the search is able to filter content by category for instance when the task is "Find all articles from category X".

Document structure and Additional CMS properties

The documents of the category tree are placed at a special folder in the CMS. A category document type has to be created containing a string field for a view name and at least a linklist property containing configuration tasks. The category document must be named as configured in the additional properties. Only one document of the category document type correctly named is placed in a folder. To create a sub category create a subfolder with a category document. The category and subcategory are not linked. The connection is identified only as a category document in the subfolder.

The following additional properties have to be set:

category.folderpath
: Root of the category tree

category.documentname
: The name of the category document

category.documenttypeid
: The ID of the document type of the category document

category.propertyname.viewname
: The name of the property containing the view name for the category

In the cmsWorks administration tools these properties can simply be set under the Entry Additional CMS properties.

In programming there is a direct access to these configuration properties:

```
app.cmsworks.service.cms.util.CategoryConfig.getCategoryFolderPath(cms);
app.cmsworks.service.cms.util.CategoryConfig.getCategoryDocumentName(cms);
app.cmsworks.service.cms.util.CategoryConfig.getCategoryDocumentTypeId(cms);
app.cmsworks.service.cms.util.CategoryConfig.getCategoryPropertyNameViewname(cms);
```

If there document types for several different pages or any other document types containing a category reference, the category field name should be always the same. In that case a simple CategoryAccess object can be used to access the category of a document to find and use a configured task.

The DocumentModelConstants declare Methods to be implemented to configure the project to the existing document model. This can be added into the documentmodel.jsf

```
// declare the resource type id of the resource type "category"
public int    getRT_CATEGORY() { return RT_CATEGORY; }
// declare the field name in a document referencing the category
public String getPT_CATEGORY() { return "category"; }
// declare the name of the filed in the category document containing the view name of this category
public String getPT_CATEGORY_VIEWNAME() { return PT_CATEGORY_VIEWNAME; }

// declare the name of the field containing the config tasks in the category document
public String getPT_CATEGORY_CONFIGS() { return "tasks"; }
// declare the document type id of the document type for config tasks
public int    getRT_CONFIGTASK() { return RT_CONFIGTASK; }
// declare the field name of the task type in the config task document type
public String getPT_CONFIGTASK_TYPE() { return "type" };
```

To get the category model from a document having a category field simply use

```
app.cmsworks.cms.document.CategoryModel dmCategory = new app.cmsworks.cms.document.CategoryModelAccess(dmAny).getCategory();
```

This returns either a referenced category or null if the category is not set or a CMSPropertyAccessException if the document has no category property.

With

```
app.cmsworks.cms.document.CategoryModel dmCategory = new app.cmsworks.cms.document.CategoryModelAccess(dmAny).getCategory(error
s);
```

an Error appears in the generated preview giving the hint that the category in the document was not set.

Accessing a category configuration task

Preparing the access of a configuration task the editors desktop configuration declares the types of the configuration task:

```
"document": { // in the document section of q-custom.js
  "configtask": { // for the document of type Configtask
    "type_prop": { // the field type
      xtype: "itwselectbox", // show a dropdown
      options: [
        [   0,"--- no type ---" ],
        [   1,"Header logo" ],
        [   2,"Navi root" ],
        [   3,"Footer" ]
      ]
    }
  }
}
```

According to this list it's recommended to create a Constant collection representing this task type list (much alike the document types and document field contents in configtask.jsf)

```
/** Defines a list of constants to name the types of config tasks
 */
public static class ConfigTask {
  final static int NO_TYPE = 0;
  final static int HEADER_LOGO = 1;
  final static int NAVI_ROOT = 2;
  final static int FOOTER = 3;
  ...

public static String getName(int id) {
    switch (id) {
      case NO_TYPE : return "no type";
      case HEADER_LOGO : return "Header logo";
      case NAVI_ROOT : return "Navi root";
      case FOOTER : return "Footer";
      ...
    }
    return "unknown";
  }
}
```

And finally for the actual access:

```
// accessing the category from a document dmAny
app.cmsworks.cms.document.CategoryModel dmCategory = new app.cmsworks.cms.document.CategoryModelAccess(dmAny).getCategory();
// accessing the task of the type HEADER_LOGO
app.cmsworks.cms.document.DocumentModel dmTask = dmCategory.getTask(ConfigTask.HEADER_LOGO);
// accessing the image document in the content list of the task document (looking for the first document of type Medium)
app.cmsworks.cms.document.DocumentModel dmLogo = dmTask.getLinkedResource(Types.PT_CONFIG_TASK_CONTENT, Types.RT_MEDIUM, 0);
```

If the configuration task was not found in this category and also not in any of its parent categories, a Nullpointer is returnd. A more transparent way to access the task is to add an ErrorView and a description message:

```
// adding the errorview and an error message if the task cannot be found
app.cmsworks.cms.document.DocumentModel dmTask = dmCategory.getTask(ConfigTask.HEADER_LOGO, errors, "Error accessing category c
onfiguration task of type " + ConfigTask.getName(ConfigTask.HEADER_LOGO));
```

Category features

The category model also provides methods to access several category related data:

String getViewName()

returns the content of the view name field from the category document

String getBaseName()

returns the name of the folder containing this category document

int getLevel()

returns the level of the category

CategoryModel getParentCategory()

retuns the parent category model

String getCategoryNameForLevel(int level)

returns the viewname for a category of a special level regarding only this category and parent categories

String getCategoryNamesForLevels(int levelStart, int levelEnd, String glue)

returns the joined category names

DocumentModel getTask(int taskId)

returns a task of a type in this category document and if not found searching in the parent category document

6 Navigation

This chapter shows how to create a document structure which represents a Navigation tree, how to assign it to the category tree and finally how to process it using a NaviItemBuilder to create a Navigation for the website.

Document structure

A new document type navigation is needed also to make sure that the document category handles category config tasks. The document types should look like this:

```
### Category #########

DocumentType;10244;category;Category
  viewname   ; string ; Viewname
  configs    ; list   ; Config tasks

### Category task #########

DocumentType;10245;configtask;Configtask
  type    ; int    ; Type
  info    ; string ; Info
  content ; list   ; Content

### Navigation #########

DocumentType;10246;navigation;Navigation
  title         ; string ; Title
  linkinternal  ; list   ; Internal Link
  linkexternal  ; string ; External URL
  linktype      ; int    ; Linktype
  linkparameter ; string ; Link parameter
  children      ; list   ; Children
```

Document types for navigation/category tasks/category from ADM-tools

Also declare the constants in the documentmodel.jsf (.../generator.std/includes/documentmodel.jsf)

```
  // ========================================================================
  public static final int RT_CATEGORY = 10244;

  public static final String PT_CATEGORY_VIEWNAME = "viewname";
  public static final String PT_CATEGORY_TASKS = "configs";

  // ========================================================================
  public static final int RT_CONFIGTASK = 10245;

  public static final String PT_CONFIGTASK_TYPE = "type";
  public static final String PT_CONFIGTASK_INFO = "info";
  public static final String PT_CONFIGTASK_CONTENT = "content";

  // ========================================================================
  public static final int RT_NAVIGATION = 10246;

  public static final String PT_NAVIGATION_TITLE = "title";
  public static final String PT_NAVIGATION_CHILDREN = "children";
  public static final String PT_REFERENCE_LINK_INTERNAL = "linkinternal";
  public static final String PT_REFERENCE_LINK_EXTERNAL = "linkexternal";
  public static final String PT_REFERENCE_LINK_TYPE = "linktype";
  public static final String PT_REFERENCE_LINK_PARAMETER = "linkparameter";

  // Also make sure that all Interface methods are properly filled

  // ========================================================================
  // from the ModelResourceConstantsAdapter to override
  // ========================================================================
  public int    getRT_CATEGORY() { return RT_CATEGORY; }
  public int    getRT_CONFIGTASK() { return RT_CONFIGTASK; }
  public int    getRT_NAVIGATION() { return RT_NAVIGATION; }

  // supports CategoryModelAccess
  public String getPT_CATEGORY() { return PT_CATEGORY; }

  // supports category
  public String getPT_CATEGORY_VIEWNAME() { return PT_CATEGORY_VIEWNAME; }
  public String getPT_CATEGORY_CONFIGS() { return PT_CATEGORY_TASKS; }

  // supports category tasks (config tasks of the catgory)
  public String getPT_CONFIGTASK_TYPE() { return PT_CONFIGTASK_TYPE; }
  public String getPT_CONFIGTASK_DOCUMENTS() { return PT_CONFIGTASK_CONTENT; }

  // supports referenceables
  public int[]  getRT_REFERENCEABLE() { return new int[] {RT_NAVIGATION}; }
  public String getPT_REFERENCE_LINK_INTERNAL() { return PT_REFERENCE_LINK_INTERNAL; }
  public String getPT_REFERENCE_LINK_EXTERNAL() { return PT_REFERENCE_LINK_EXTERNAL; }
  public String getPT_REFERENCE_LINK_TYPE() { return PT_REFERENCE_LINK_TYPE; }
  public String getPT_REFERENCE_LINK_PARAMETER() { return PT_REFERENCE_LINK_PARAMETER; }
```

```
  // supports NaviItemBuilder
  public String getPT_NAVIGATION_ELEMENTS() { return PT_NAVIGATION_CHILDREN; }
  public String getPT_NAVIGATION_HEADLINE() { return PT_NAVIGATION_TITLE; }
```

Navigation, Category and Config task constants in the documentmodel.jsf

Configtask configuration

Create a configtask.jsf to define constants for configuration task types (.../generator.std/includes/configtask.jsf).

```
<%@page import="app.cmsworks.cms.document.ErrorView,
                app.cmsworks.cms.document.DocumentModel,
                app.cmsworks.cms.document.CategoryModel
               "
        pageEncoding="UTF-8"
%><%!

/** Defines a list of constants to name the types of config tasks for categories
 */
public static class ConfigTask {

  final static int NO_TYPE = 0;
  final static int NAVI_ROOT = 10;
  final static int NAVI = 11;

  public static String getName(int id) {
    switch (id) {
      case NO_TYPE : return "no type";
      case NAVI_ROOT : return "Navi root";
      case NAVI : return "Navi";
    }
    return "unknown";
  }

  public static DocumentModel getTask(int type, CategoryModel cmCategory, ErrorView errors) throws Exception {
    return cmCategory.getTask(type, errors, getName(type));
  }

}

%>
```

Configtask types constant definition.

Declare the configuration task types in the editors desktop configuration q-custom.js (.../webui/cmsdesk.custom/<customer>/q-custom.js)

```
"document": {

  "configtask": {
    "type_prop": {
      xtype: "itwselectbox",
      options: [
        ["", "-- not selected --"],
        [ 10,"Navi root","Fill the one navigation root document in here."],
        [ 11,"Navi","Use a document of type navigation here."]
      ]
    }
  },

  "*": {
    "category_prop" : { // properties named category of type link shall be category input elements
      xtype: "itwcategory"
    },
    "linktype_prop": {  // a linktype property of a referenceable document type should be a dropdown from linktypes declared
in an object sharedDefs
      xtype: "itwselectbox",
      options: sharedDefs.linkTypes
    }
  }

} // end document modification
```

Configtask type declarations in the editors desktop configuration q-custom.js

Documents in cmsWorks

The documents for the category tree, the navigation documents and the configtask document should be created and linked to each other like the graph in the adminstrators guide shows. We would suggest to place the documents **Configtask** right beside the category documents in the category tree within the editors desktop. It is advisable to

create a dedicated folder for all navigation documents that is not part of the folder containing the content but rather created aside that folder.

The root category references a configtask "Navi root" containing the root of the navigation tree.

A category references a configtask "Navi" containing a navigation document representing this category.

A Navigation document references a content page which then is the (leaf) content page of the navigation.

Benefits of Navigation documents

At each level of the navigation tree the order of the children is manually set by the editor. It is possible to also add entries to target external URLs or even skip to assign a category to a navigation.

Maybe add a linklist medium to the document type to produce icons for navi entries.

Programming

```
<%@page import="
                app.cmsworks.cms.document.ErrorView,
                app.cmsworks.cms.document.HTMLErrorView,
                app.cmsworks.cms.document.DocumentModel,
                app.cmsworks.cms.document.CategoryModel,
                app.cmsworks.cms.document.CategoryModelAccess,

                app.cmsworks.cms.document.NaviItem,
                app.cmsworks.cms.document.NaviItemBuilder

               "
        session="false"
        contentType="text/html;charset=UTF-8"
%><%@include file="includes/documentmodel.jsf"
%><%@include file="includes/configtask.jsf"
%><%!

// creats a list of navi entries from the chidren of a given NaviItem
public String getNaviEntries(NaviItem niCur) throws Exception {
  if (niCur == null) return "";
  StringBuffer sb = new StringBuffer();
  for (NaviItem ni : niCur.getChildren()) {
    String cls = "";
    if (ni.isSelected()) {
      cls = " sel";
    }
    String entry = "<a class=\"nv" + cls + "\" " + ni.getLink().createAnchorTarget() + ">" + ni.getTitle() + "</a>";
    sb.append(entry);
  }
  return sb.toString();
}

%><%

ErrorView errors = new HTMLErrorView();
DocumentModel dmPage = null;
String htmlNavi = "";

//Fetch the sensible data
try {
  dmPage = new DocumentModel(request, new Types());
  errors.setPreview(dmPage);

  // Get the category from the page document
  CategoryModel cmCategory = new CategoryModelAccess(dmPage).getCategory(errors);

  // Get the navi root task from the category (or any parent category because this task is only defined in the category root)
  DocumentModel dmNaviRootTask = ConfigTask.getTask(ConfigTask.NAVI_ROOT, cmCategory, errors);
  // Get the document Navigation representing the root of the navigation which itself is not part of the navigation
  DocumentModel dmNaviRoot = dmNaviRootTask.getLinkedResource(Types.PT_CONFIGTASK_CONTENT, Types.RT_NAVIGATION, 0, errors);

  // Use the NaviItemBuilder to create NaviItem objects in a tree
  // where one item is selected and its parent and its grand parent and so on
  NaviItem niRoot = new NaviItemBuilder().createSelectedNaviItems(dmNaviRoot, dmPage, ConfigTask.NAVI, errors);

  StringBuffer sb = new StringBuffer();

  NaviItem niCur = niRoot;
  String htmlNaviItems = getNaviEntries(niCur);
  while (niCur != null && htmlNaviItems.length() > 0) {
    String cls = " lvl" + niCur.getLevel();
    sb.append("<div class=\"nvline" + cls + "\">");
    sb.append(htmlNaviItems);
    sb.append("</div>");
    // get the child that has the isSelected flag or null if no such child exist
    niCur = niCur.getSelectedChild();
    htmlNaviItems = getNaviEntries(niCur);
  }
```

```
  htmlNavi = sb.toString();
}
catch (Throwable t) {
  // In preview the errors will be shown from errors.render()
  // but in live state this include returns an error code so in the page including this component is missing
  if (errors.exit(response, t, dmPage, this.getClass().getName())) {
    return;
  }
}
//Now start the HTML-Output
%><nav><%= htmlNavi %></nav><%= errors.render() %>
```

A Navigation is produces using NaviItemBuilder in a inc-navi.jsp

To include the navi into a page within a JSP "page-xyz.jsp" to render this page use

```
htmlNavi = new ContentInclude(dmPage, dmPage, "inc-navi.jsp", errors).getIncludeContent();
```

To select a NaviItem the NaviItemBuilder first tries to find the document (page) which currently rendered linked by any navigation document of the navigation tree.

If this is not found here, the NaviItemBuilder looks into the category to get a configuration task "Navi" and tries to find this navigation document in the navigation tree.

In case there is none, no selection is used and rendered.

The NaviItemBuilder uses the Referenceable interface to create a link from the four input fields (linkintern/linkextern/linktype/linkparameter).