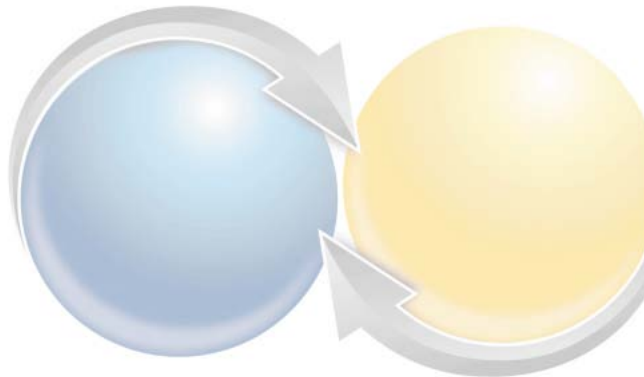


Livelink WCM Server

## **Portal Manager API Programmer's Manual**

This manual describes how to develop dynamic and personalized websites on the basis of the Portal Manager API. Highlighted topics include:

- basic concepts of the Portal Manager API
- configuring the Portal Manager API in the Admin client
- overview of the classes and interfaces of the Portal Manager API
- application examples of programming with the Portal Manager API



---

## Copyright 2005 by Open Text Corporation.

The copyright to these materials and any accompanying software is owned, without reservation, by Open Text. These materials and any accompanying software may not be copied in whole or part without the express, written permission of Open Text.

Open Text Corporation is the owner of the trademarks Open Text, 'Great Minds Working Together', Livelink, and MeetingZone among others. This list is not exhaustive. All other products or company names are used for identification purposes only, and are trademarks of their respective owners. All rights reserved.

Open Text Corporation provides certain warranties and limitations in connection with the software that this document describes. For information about these warranties and limitations, refer to the license agreement entered into between the licensee and Open Text Corporation.

### Contacting Us

Corporate Headquarters  
Open Text Corporation  
185 Columbia Street West,  
Waterloo, Ontario  
N2L 5Z5  
Canada  
(519) 888-7111

If you subscribe to our Software Maintenance Program or would like more information about additional support programs, visit Open Text Customer Support at <http://www.opentext.com/services/support.html>.

If you have suggestions for this publication, send an e-mail message to [documentation@opentext.com](mailto:documentation@opentext.com) to contact the Open Text Publications Group.

Visit our home page at <http://www.opentext.com> for more information about Open Text products and services.

### © 2005 IXOS SOFTWARE AG

Bretonischer Ring 12  
85630 Grasbrunn, Germany  
Tel.: +49 (89) 4629-0  
Fax: +49 (89) 4629-1199  
eMail: [<office@ixos.de>](mailto:office@ixos.de)  
Internet: <http://www.ixos.de>

All rights reserved, including those regarding reproduction, copying or other use or communication of the contents of this document or parts thereof. No part of this publication may be reproduced, transmitted to third parties, processed using electronic retrieval systems, copied, distributed or used for public demonstration in any form without the written consent of IXOS SOFTWARE AG. We reserve the right to update or modify the contents. Any and all information that appears within illustrations of screenshots is provided coincidentally to better demonstrate the functioning of the software. IXOS

---

---

SOFTWARE AG hereby declares that this information reflects no statistics of nor has any validity for any existing company. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (<http://www.openssl.org/>) and software developed by the Apache Software Foundation (<http://www.apache.org/>).

*Trademarks* IXOS: IXOS SOFTWARE AG.

SAP<sup>®</sup>, R/3<sup>®</sup> and SAP ArchiveLink<sup>®</sup> are registered trademarks of SAP AG.

Microsoft<sup>®</sup>, Microsoft Windows NT<sup>®</sup> and the names of further Microsoft products are registered trademarks of Microsoft Corporation.

Acrobat Reader Copyright © 1987 Adobe Systems Incorporated. All rights reserved. Adobe and Acrobat are trademarks of Adobe Systems Incorporated which may be registered in certain jurisdictions.

Siebel<sup>®</sup> is a registered trademark by Siebel Systems, Inc.

Other product names are used only to identify the products and they may be registered trademarks of the relevant manufacturers.

**Copyright © 2005 Gauss Interprise AG Hamburg, Gauss Interprise, Inc. Irvine, California. All rights reserved worldwide.**

This document and the related software are property of Gauss Interprise AG or its suppliers and protected by copyright and other laws. They are distributed under licenses restricting their use, copying, distribution, and decompilation. Neither receipt nor possession of this document confers or transfers any right to reproduce or disclose any part of the contents hereof. No part of this document may be reproduced in any form by any means without prior written authorization of Gauss Interprise AG or Gauss Interprise, Inc.

Moreover, the regulations of the software license agreement apply to this documentation.

All brand names and trademarks mentioned are the property of their respective owners.

<http://www.opentext.com/bridging/gauss.html>

Program version: Livelink Web Content Management Server<sup>™</sup> (Content Server) 9.5.0

Document version: En-01

Publication date: April 2005

---

---

# Table of Contents

<b>List of Figures</b>		<b>6</b>
<b>List of Tables</b>		<b>8</b>
<b>Chapter 1</b>	<b>Introduction</b>	<b>11</b>
	About this Manual	12
	Typographic Conventions	16
<b>Chapter 2</b>	<b>Concepts</b>	<b>19</b>
	The Architecture of the Portal Manager API	19
	Integrating the Portal Manager API in the WCM System	24
	Programming Concepts	32
	Integrating External Programs	46
<b>Chapter 3</b>	<b>Managing the Portal Manager API</b>	<b>51</b>
	Managing Repositories	52
	Managing applications	65
	LDAP	86
	Locating Errors	89
<b>Chapter 4</b>	<b>Classes and Interfaces of the Portal Manager API</b>	<b>95</b>
	Class Diagrams and Class Descriptions	95
	Basic Classes	97
	Application and Bean Classes	110

---

<b>Chapter 5</b>	<b>Notes on Programming with the Portal Manager API</b>	<b>127</b>
	Authentication and Session Management	127
	Application Examples for the Portal Manager API	133
	Framework for Applications	152
	Designing a JSP Page	157
	Security Aspects	159
	Internationalization	160
<b>Appendix A</b>	<b>Metadata Schemes</b>	<b>167</b>
<b>Appendix B</b>	<b>Caching</b>	<b>173</b>
	Accessing WCM Objects	173
	Deployment Cache	175
	How the Cache for WCM Objects Works	175
	Caching RepositoryEntry Objects	177
<b>Glossary</b>		<b>179</b>
<b>Index</b>		<b>183</b>

---

# List of Figures

Fig. 1 –	The layer model of the Portal Manager API	20
Fig. 2 –	The layer model of the Portal Manager API with session and application scope	23
Fig. 3 –	Structure of a minimum system	28
Fig. 4 –	Distributed WCM system with separate databases	30
Fig. 5 –	The life cycle state of a WCM object.	45
Fig. 6 –	Principal class design in the Portal Manager API	49
Fig. 7 –	The Configuration items	51
Fig. 8 –	Overview of available repositories	53
Fig. 9 –	Adding a repository	54
Fig. 10 –	Adding a parameter for a repository	56
Fig. 11 –	Applications assigned to a repository	63
Fig. 12 –	Structure below the tree item <i>Repositories</i>	64
Fig. 13 –	Overview of available applications	66
Fig. 14 –	Defining basic application data	68
Fig. 15 –	Quit wizard?	69
Fig. 16 –	Assigning a repository to an application during setup	70
Fig. 17 –	Assigning resources when adding an application	71
Fig. 18 –	Adding a parameter for an application	73
Fig. 19 –	Repositories assigned to an application	82
Fig. 20 –	Structure below the tree item <i>Applications</i>	83
Fig. 21 –	Applications assigned to a Content server	85
Fig. 22 –	The <code>DirectoryRepository</code> with its parameters	89
Fig. 23 –	Representation scheme for class diagrams	96
Fig. 24 –	Multi-level key value lists in the <code>TupleMap</code>	99
Fig. 25 –	The basic classes in the entire class diagram	100
Fig. 26 –	The filtering and sorting classes in the entire class diagram	102

---

Fig. 27 –	The RepositoryMap class in the entire class diagram	102
Fig. 28 –	Structural relations of complex data types	103
Fig. 29 –	Multi-level key value lists in the repository map	105
Fig. 30 –	The repository classes in the entire class diagram	109
Fig. 31 –	The application classes in the entire class diagram	112
Fig. 32 –	The bean classes in the entire class diagram	115
Fig. 33 –	Structure of VipProfile and the managed information	119
Fig. 34 –	Configuring the trusted login mode via the Admin client	129
Fig. 35 –	Authentication and session management	131
Fig. 36 –	Login Repositories	132
Fig. 37 –	Class diagram for applications	152
Fig. 38 –	General hierarchy for developing applications	154
Fig. 39 –	The RepositoryImpl class with all methods relevant to data sources	155
Fig. 40 –	JSP model 2 architecture	157
Fig. 41 –	Architecture of the Portal Manager API with internationalization aspects	161
Fig. 42 –	Structure of a website	164

---

# List of Tables

Table 1 –	Available functions of the Livelink WCM Server APIs	14
Table 2 –	Functions for configuring repositories and applications	52
Table 3 –	Repositories and their parameters	57
Table 4 –	WCM Applications and their parameters	73
Table 5 –	Start parameters for Content servers running in the context of a JSP engine	110
Table 6 –	Sample configuration of a DirectoryRepository	135







---

# CHAPTER 1

## Introduction

The Portal Manager API is part of Livelink Web Content Management Server™ (Livelink WCM Server for short). By purchasing Livelink WCM Server, you have acquired a powerful Content Management System, which you can use to build, maintain, and manage complex and distributed websites.

The Portal Manager API enables you to create and manage dynamic content for your website, thus allowing you to develop enterprise portals. Furthermore, the Portal Manager API forms the basis for personalizing websites.

The Portal Manager API can be integrated in application servers which conform to the J2EE specification and offers the possibility of dynamic load balancing and session fail over. It also functions as a platform for Enterprise Application Integration (EAI). The Portal Manager API is the basis for a multitude of connectors that you can use to establish a connection to third-party products.

The Portal Manager API is not a product that gives you a finished web portal after installation. You should consider it as a framework that helps you create web portals and that provides support in coming up with necessary solutions to problems. A seamless integration with Livelink WCM Server is guaranteed. The Portal Manager API allows you to quickly develop dynamic components that are internally consistent, in line with your own ideas, and that meet the demands of modern websites.

**Important!** Incorrect use of the programming interface described in this manual may lead to errors in the WCM system including system crashes and data loss. Incorrect programming can also cause problems concerning performance and system resources. For this reason it is essential to test the developed software with regard to correctness, stability, robustness, and performance before putting it to productive operation.

Gauss Interprise AG cannot assume any liability for the correct functionality of the developed software. Our Professional Services Group can help you plan and implement solutions. This may help you avoid problems right from the start.

## About this Manual

The Portal Manager API and hence this document are designed for individuals with sound technical know-how. The following requirements should be satisfied:

- knowledge of the product Livelink WCM Server, preferably not only as user, but also as administrator
- technical understanding of the processes involved in using JSP pages and servlets
- general programming experience, Java programming experience, and familiarity with HTML

The contents of this manual are organized as follows:

- Chapter 2 “Concepts” explains the basic concepts of the Portal Manager API and the integration of the Portal Manager API in the system architecture of a WCM system.

- Chapter 3 “Managing the Portal Manager API” describes how to configure the Portal Manager API by means of applications and repositories. The chapter also contains notes on LDAP.
- Chapter 4 “Classes and Interfaces of the Portal Manager API” contains a description of the programming interface that the programmer must be familiar with in order to create JSP pages.
- Chapter 5 “Notes on Programming with the Portal Manager API” offers an introduction to the Portal Manager API’s programming interfaces as well as a few application examples.
- Appendix A “Metadata Schemes” compares the metadata schemes of version 5e and version 8.
- Appendix B “Caching” describes the caching behavior of the Portal Manager API.

The programming interface described here is a component of Livelink WCM Server. In addition to this Programmer’s Manual, the following sources provide information:

- Livelink WCM Server Installation Manual: This document describes the installation of the WCM system and contains information on the configuration of web server and application server.
- Livelink WCM Server Administrator Manual: This document describes the configuration and administration of a WCM system from the point of view of an administrator.
- WCM Java API Programmer’s Manual: This document contains information on interfaces, classes, and methods of the Java programming interface (WCM Java API), which makes it possible to use the functionality of the WCM servers via external programs.
- Javadoc – Here you will find the complete documentation on the classes of the Portal Manager API.

Livelink WCM Server offers flexible programming interfaces for using the core functionality of Livelink WCM Server via external programs. The following table provides an overview of the different interfaces and the available functions.

**Table 1 – Available functions of the Livelink WCM Server APIs**

Functionality of the	WCM Java API	Remote API	Portal Manager API	WCM WebServices
AdminHandler	✓	✓	✓ read-only by means of VipUserBean	
AttributeHandler	✓	✓	✓ by means of VipWebsiteBean	✓ by means of VipWebService_ Port
EventDispatcher	✓	✓		
MailHandler	✓		✓ by means of VipEmailBean	
ObjectHandler	✓	✓	✓ by means of VipObject HandlerBean	✓ by means of VipWebService_ Port
ObjectLoader	✓	✓		
SystemHandler	✓	✓		

Functionality of the	WCM Java API	Remote API	Portal Manager API	WCM WebServices
WorkflowHandler	✓		✓ by means of VipWorkflow Bean	✓ by means of VipWebService_ Port
DeploymentHandler	✓	✓		
ContextHandler	✓	✓		
LivelihoodAdmin Handler	✓		✓ by means of LivelihoodUser Bean	
LivelihoodObject Handler	✓		✓ by means of LivelihoodObject Bean	

## Typographic Conventions

The following conventions are used in the text to draw attention to program elements, etc.:

Element	Font or symbol	Examples
Program interface such as menu commands, windows, dialog boxes, field and button names	<i>Menu → Entry</i>	<i>File → Create</i>
Paths to directories, file and directory names	<b>Drive:\Directory\File name</b>	<b>D:\WCM\ admin.bat</b>
Quotations from program code or configuration files	Code quotations	<code>&lt;head&gt; &lt;title&gt;heading &lt;/title&gt; &lt;/head&gt;</code>
Variables, i.e. placeholders for specific elements	{variable}	{WCM installation directory}

Important **information** and **warnings** are enclosed in gray boxes. Make sure to read such information to avoid losing data or making errors when using and managing WCM systems.







---

# CHAPTER 2

## Concepts

This chapter describes how the Portal Manager API is integrated in a WCM system. Moreover, the general concepts of programming with the Portal Manager API are explained.

### The Architecture of the Portal Manager API

The functionality of the Portal Manager API is a result of the collaboration of the various components of the WCM system. The most important are

- servers (e.g. master Admin server, master Content server, proxy Content server)
- websites managed in the WCM system
- session beans and other classes of the Portal Manager API
- JSP pages using these interfaces

The internal structure of the Portal Manager API can be described by a simple layer model (see the following figure).

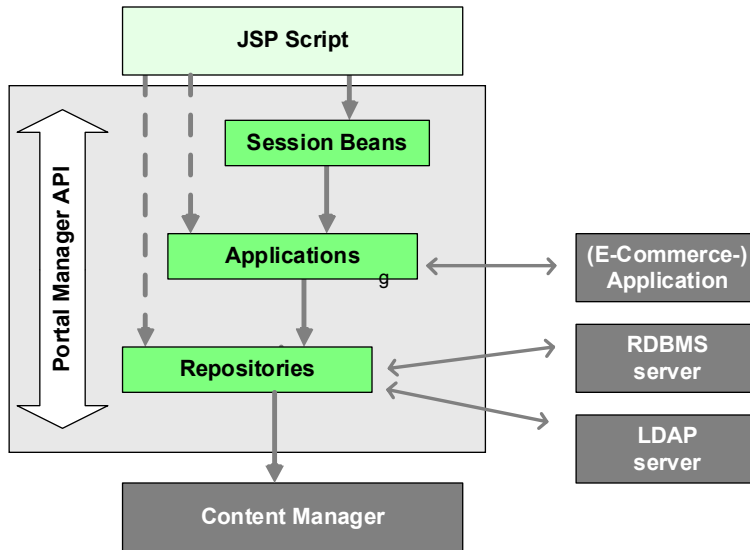


Fig. 1 – The layer model of the Portal Manager API

## JSP Scripts

JSP scripts are Java code embedded in HTML pages. Requests from clients (HTML browsers) to the web server result in the JSP engine executing the Java code in such HTML pages. The result is returned to the client as content. The Java code is thus executed on the server side. The Java code is converted into servlets by the JSP engine once before the initial execution. This is invisible to the client because the client only sees HTML pages.

Java code used in JSP pages must be enclosed in tags that correspond to the JSP syntax. Otherwise, the code cannot be processed by the JSP engine.

## Session Beans

Beans are classes whose instances can be used in JSP scripts. They are the starting point in the Portal Manager API. A bean is based on an application, depending on which tasks are supposed to be performed with the Portal Manager API. As with applications, external programs cannot be connected until individual bean classes have been developed.

Each of the beans contained in the Portal Manager API is linked to a JSP session. That is why they are also called *session beans*.

Sessions are units managed by the JSP engine. The purpose of a session is to be able to combine the actions of a web client. This is not possible with HTTP as the protocol between web client (HTML browser) and web server because HTTP is a stateless protocol. But a session exists beyond individual HTML requests. A JSP script can reference the current session. This means that a state for each user can be managed despite numerous requests. Session beans facilitate resource management considerably.

A session begins with the first contact of a web client and ends with the last contact. The JSP engine is technically responsible for managing the session. Generally, the JSP engine saves a session key in the respective browser. This means that the browser must allow (non-persistent) cookies. Each JSP engine has its own strategies to determine when and how the session key becomes invalid or when the last contact with a browser took place. Normally, this is determined empirically through time behavior. Different browser instances on the same client computer are normally associated with different session keys. The entire JSP engine complex is almost completely standardized.

# Applications

Applications are descriptions of any programs that are to be referenced in the Portal Manager API. The Portal Manager API already contains some applications that have been implemented to control internal access to Livelink WCM Server. However, additional individual implementations must be developed if external applications such as e-commerce solutions are intended to be controlled using the Portal Manager API.

Applications provide access to data sources. The concept provided in the Portal Manager API for processing the data of an application is based on the use of repositories. Repositories are modeled by the basic classes that are described in chapter 4 “Classes and Interfaces of the Portal Manager API”.

In the Portal Manager API, applications are defined, in the broadest sense, as a collection of abstract access processes which standardize the view of programs. New, concrete applications can be easily integrated by using the application interface. Using this interface is not mandatory, but it makes available the full functionality of the session beans.

In contrast to session beans, the lifetime of resources in applications is unlimited. Applications do not know sessions. There is only one single resource context that begins with the application’s runtime start and finishes when the runtime ends. If necessary, the application is instantiated by the bean and stopped by the server-side process (Java Virtual Machine).

The following figure shows where the borderline between session and application scope lies.

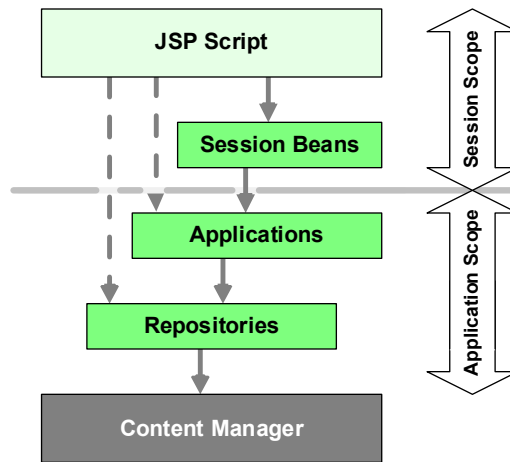


Fig. 2 – The layer model of the Portal Manager API with session and application scope

## Repositories

Similar to applications, repositories are an abstraction allowing uniform accesses to data. New applications that are created by using the application interface can access existing repositories or repositories that are yet to be developed.

## Integrating the Portal Manager API in the WCM System

The integration of the Portal Manager API in the WCM system involves different aspects:

- access of Portal Manager API solutions to WCM system data
- integration of Content servers on which the Portal Manager API is available into the WCM system

### Access to the Data of the WCM System

If you look at the solutions based on the Portal Manager API, the repository interface connects the Portal Manager API and the WCM system. When programming a solution with the Portal Manager API, you use beans to access the data of the WCM system. The bean requests the necessary repository from the application and establishes a connection to the WCM system via this repository. The repository provides the object data from the WCM system for the Portal Manager API solution. You can assign repositories and applications to each user via the Admin client, see “Assigning a Repository to an Application” on page 81.

There are numerous beans that access certain services in the WCM system. They allow access to the data of the managed objects as well as to administrative data. This enables access to object content, object metadata, user, group, role, and functional area data. Support is also provided for filtering and sorting the data of the WCM system.



## Portal Manager API and Deployment Systems

When a repository is used to access object data, the data of a certain deployment system is accessed. A deployment system is specified in the configuration of each repository that can be used to access object content and metadata. When you create a new website via the Admin client, the repositories for all deployment systems of the new website are generated automatically.

The Portal Manager API can access the data of any deployment system. The only requirement is that the respective deployment systems use the same data storage as the Content server running in the context of a JSP engine or as web application in an application server. For accessing objects with WebDAV-capable clients, a servlet is used that requires the Portal Manager API.

## Portal Manager API and Data Storage

The Portal Manager API requires access to both the data of deployment systems (e.g. the generated URLs of objects) and to the WCM objects themselves (via repositories). Access to the WCM objects is required for reading the metadata and for processing them for applications and programs. For accessing the data of deployment systems, the deployment system of the repository must be installed on a Content server that uses the same data storage as the Content server running in the context of a JSP engine or as web application in an application server.

In a distributed WCM system, it is, for example, possible that two Edit deployment systems for the same website have been installed on different servers. If these two servers have separate data storages, the deployment systems also use different data storages when generating pages. The Portal Manager API cannot access both Edit deployment systems because this would require simultaneous access to the data of *one* website via different JDBC pools. In such a case, two Content servers running in the context of a JSP engine or as web application in an application server are required.

## Integrating the Portal Manager API in a WCM System

The servers are divided into two main categories: **master** and **proxy**. In every WCM system, there is one master Administration server and one or several master Content servers.

The **master Administration server** handles the user administration and is responsible for configuration, system administration, and license management. The Administration server can be accessed using the Admin client.

The **master Content server** manages one or more websites. Each website is assigned to exactly one master Content server. Changes to the content and status of WCM objects can only be made on a master Content server. The master Content server always has all data storage views (Edit, QA, and Production).

In addition to the master Content server, you can set up **proxy Content servers** in a WCM system, which also provide access – by way of deployment systems – to the website data. Unlike a master Content server, however, proxy Content servers merely have read-only access to the content. If website data is to be edited using a proxy Content server, the proxy Content server consults the master Content server, which locks the object from further write accesses and, once editing is complete, stores the changed objects in the data storage. The master Content server then informs all proxy Content servers connected to the website that the WCM object has changed. This ensures that your website content remains consistent.

The **deployment systems** are responsible for the distribution of the pages in the WCM system.

The Portal Manager API is available on all Content servers running in the context of a JSP engine or as web application in an application server.

Such a Content server is required for using the Content client and the Content client (Classic).

Please note the following for configuring Content servers running in the context of a JSP engine or as web application in an application server.

- Deployment systems of a website that use different data storages must be accessed via different Content servers.
- The available view of a website's data and the possible actions depend on the routing settings of the Content server. If changes to a website's data are to be performed by the Content server, for example via the Content client, the Content server must receive the Edit view of the data (in which case, the Content server would have access to all three views of the website).

Due to the flexible system architecture of Livelink WCM Server, there are numerous possibilities of setting up a WCM system. The Livelink WCM Server Administrator Manual contains detailed information on this topic.

In the following sections, two scenarios for integrating the Portal Manager API in a WCM system are described.

## Minimum System

The WCM system consists of a master Admin server and a master Content server. The master Content server is executed in the context of a JSP engine or as web application in an application server. Thus, the Portal Manager API, which is required for editing websites with the Content client, is available on the master Content server.

The website "InternetSite" is created on the master Content server. The website objects are distributed to different directories via three deployment systems (Edit, QA, and Production). Thus, the complete staging is realized on the master Content server. In the figure, "DS" is used as abbreviation for "deployment system".

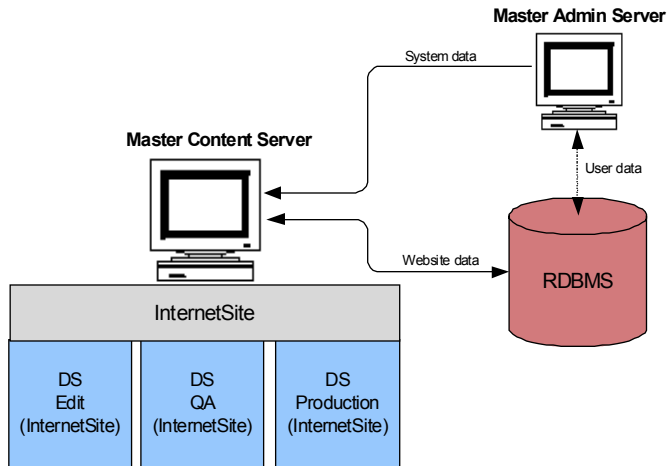


Fig. 3 – Structure of a minimum system

A minimum system is installed by selecting the relevant option in the WCM installation program (see Livelink WCM Server Installation Manual). In order to create a website in this scenario, start the new-website wizard in the Admin client and create the website with the option *Minimum* or *Minimum (dynamic)*.

### Distributed WCM System with Separate Data Storages

The flexible and scalable system architecture of Livelink WCM Server enables you to set up distributed WCM systems according to your corporate structure. To minimize data transmission between the individual servers, the proxy servers can access separate data storages.

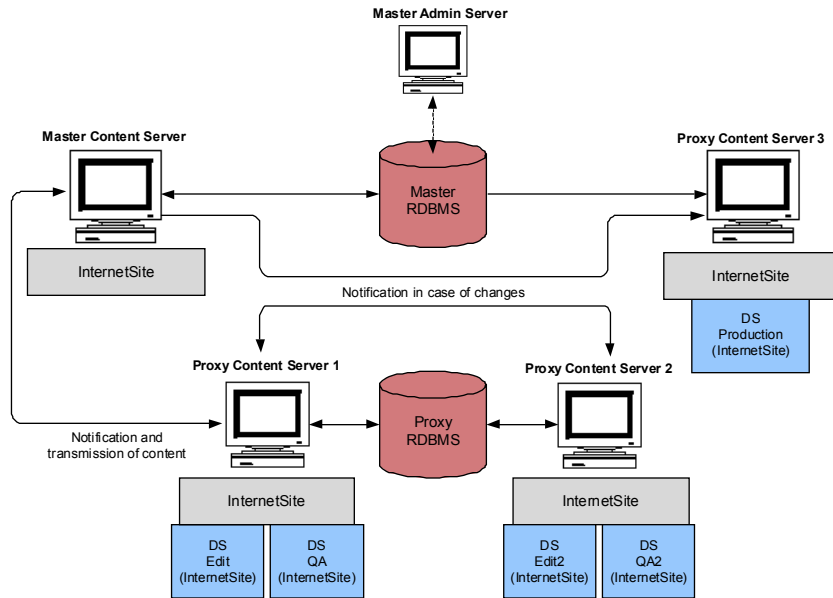
When using the Portal Manager API in such a scenario, please note that a Content server running in the context of a JSP engine or as web application in an application server cannot access different data storages at the same time (see also “Portal Manager API and Data Storage” on page 25). If the data of your website is managed by different proxy Content servers

that access separate data storages, these servers must run in the context of a JSP engine or as web application in an application server.

The following example illustrates such a scenario:

The WCM system consists of a master Admin server and a master Content server. The website “InternetSite” is created in the WCM system. The system is distributed among three additional proxy Content servers. Two of these proxy Content servers run in the context of a JSP engine or as web application in an application server. They are used for editing the WCM objects by means of the Content client. Thus, the respective Edit and QA deployment systems are installed on these two servers. The third proxy Content server is used for publishing the Production view of the website. On this server, a Production deployment system is installed.

To make the illustration more comprehensible, the connections from the master Admin server to the other servers are not contained in the following figure.



**Fig. 4 – Distributed WCM system with separate databases**

If a WCM object is changed, the master Content server informs the proxy Content servers 1 and 3. Proxy Content server 1 informs proxy Content server 2 about the changes. Deployment for the changed objects is carried out on all the notified servers, so that the corresponding pages are updated.

A distributed system of this kind is created using of the option *User-defined installation* in the WCM installation program (see Livelink WCM Server Installation Manual).

In order to create a website in this scenario, start the new-website wizard in the Admin client and create the website with the option *User-defined*.

## Starting a Content Server in the Application Server

Both the Content client and the Content client (Classic) use the Portal Manager API. For making this API available (also for custom JSP pages), a Content server must be started in the context of the application server.

### Content server as web application

If a web application has been generated for the Content server, the Content server runs in an application server. If the application server used supports the servlet standard 2.3, the Content server is started and stopped automatically together with the respective web application.

If the application server supports the servlet standard 2.2, the web application and the Content server are started and stopped separately. In this case, use the respective scripts for the Content server (see the following section). Make sure that the web application is started first.

### Starting the Content server in the JSP engine

If the Content server runs in the context of a JSP engine that does not support web applications, the scripts located in the directory **{WCM installation directory}\tools\** must be used for starting the server.

The following steps are required:

1. To set the class path required for the Content server, call the script **setPomaClasspath.bat** or **.sh**. This script is located in the directory **{WCM installation directory}\tools\**.
2. Add the class path that was created as the script was called to the class path of the JSP engine.
3. Copy the supplied script **portalmanager.bat** or **.sh**, which is located in the directory **{WCM installation directory}\tools\**, to the root directory of the WCM installation. Rename the script **{name of the Content server}.bat** or **.sh**.

4. In the script **{name of the Content server}.bat** or **.sh**, replace all placeholders `SERVERNAME` with the name of the Content server.

In order to start the Content server via the script, the JSP engine must already be running. In the configuration of the JSP engine used, the servlet mapping `'/servlet/*'` must be entered.

**Note:** If you set up two or more Content servers on the same computer, you must use different instances for your JSP engine for executing the servers. After the installation of the WCM system, modify the default URL in the scripts for starting the Content servers according to the configuration of the JSP engine used.

For stopping the Content server, use the supplied script **shutdown\_{name of the Content server}.bat** or **.sh**. Alternately, the server can be shut down via the Admin client.

## Programming Concepts

In order to use the Portal Manager API to create dynamic and personalized websites, you must first understand its basic programming concepts. The purpose of this section is to introduce you to these basic concepts.

The Portal Manager API enables you to develop programs for the JSP engine, so-called JSP scripts. JSP scripts implement the part of a website or portal that the user sees, similar to the role of the GUI in a conventional application. Furthermore, the Portal Manager API provides extensive support when integrating new solutions.



Interfaces in the form of completed Java classes are provided for creating JSP scripts as well as applications. The Java classes and JavaBeans implement the concepts introduced in this chapter. Chapter 4 “Classes and Interfaces of the Portal Manager API” gives you an overview of the classes and beans. For more detailed information, please refer to the Javadoc on the Portal Manager API.

## Dynamic Content

The Portal Manager API can be used to create and manage dynamic web *content*, which is characteristic of corporate portals. In contrast to the static content that the writer or editor specifies when a page is created, dynamic content is generated entirely or in part at runtime. Dynamic content thus allows you to react to the current situation and the current context while the website is being used.

The template concept already implemented in Livelink WCM Server is basically a step in the direction of dynamic content, because content is created by connecting two or more content objects. The result, however, is static and cannot adjust in real-time to the context in which the website is used.

In a “really” dynamic website, the content can be checked or updated at runtime, when a page is requested by a user. Its power and flexibility can be seen from the look at the typical requirements of modern websites, which are introduced below on the basis of a selection of typical application cases of the Portal Manager API. These scenarios serve as examples only; the use of the Portal Manager API is in no way limited to these examples.

### Navigation Elements

First of all, navigation is defined as the possibility to display the context of the current page so that the user is able to understand the context quickly. In addition, navigation provides the user with hyperlinks for quickly and easily switching to associated content, for example, to pages on related topics or neighboring pages in the hierarchic structure. *Sitemaps* also belong to the navigation elements, although they generally represent information concerning the entire website and not the context of the current page.

Navigation elements can appear on a page in many ways. Animated components that display navigation elements as menus or as a expandable tree structure are possible. Navigation elements can be designed with the “look and feel” of a website, fitting in with the total website design.

In the WCM system, the fact that metadata are stored for each content object provides sufficient information for navigating. For each content object, the program stores details of those objects that can be reached by hyperlink and those objects from which the active content object can be reached by hyperlink. Further information for navigation is provided by the topic tree and by linking a content object to a template object.

If the underlying metadata change, the content changes automatically without any need for manual modification. As a result, these navigation elements do not get out of date when objects are deleted, copied, or moved.

### Personalization

Personalized websites are websites that are individually prepared and displayed, depending on the user. Such user-specific content may be desirable for several reasons.

It may be necessary to withhold certain information from some users. A static system such as Livelink WCM Server can prevent users without sufficient rights from accessing content, but this does not prevent users from learning of the existence of objects for which they have no access rights.

Personalized pages can also be used to provide user-specific, individual settings, preferences, etc. or the user's current context. An e-commerce shopping cart system is a typical example: users only see their personal shopping carts.

Personalization and navigation should always go hand in hand. Navigation should not display content that the user does not want to see or is not supposed to see.

## **Integrating External Programs**

Website applications are always dynamic components. When integrating an application in a website, it is important to adapt the user interface to the website with regard to data and the "look and feel". Ideally, program data is interpreted as content or as elements of content so that the source of the content no longer plays a role when the website is displayed. How to perform this task strongly depends on which interfaces the program offers. The Portal Manager API gives you programming support and makes such an integration possible on the functional and data levels.

## **Placing Several Content Objects on one Page (Multi-Content Page)**

Using the Portal Manager API, any number of content objects can be presented on a page. Of course, limiting aspects – as mentioned under the keyword personalization – must be taken into account. The content can consist of objects from WCM-managed websites or data from programs. Even inter-website and inter-program content links are no problem at all.

# Forms

Thanks to the options provided by the Portal Manager API, you can easily develop form-based solutions. You can use form-based interfaces, for example, to create and edit the objects of a website managed with Livelink WCM Server.

A *form* is a JSP page that provides a form-based interface to the user. This type of form is used to retrieve and process data for a form instance.

The possibilities of form-based work are represented by two special object types: *Form template* and *Form instance*.

- Form template – template defining the type “Form”
- Form instance – content of a form (form template) as XML data

In order to offer a form-based interface on the basis of these object types, the actual forms must be developed as JSP pages with the corresponding selection and entry options.

Basically, the form template serves only to identify and typify the different form instances. The form instance (a filled-out form) is an XML file that can have a specific form template as its WCM template.

**Note:** For editing XML data structures, the four object types “XML document”, “XML template”, “XSLT document”, and “XSLT template” are available. For detailed information on these object types, refer to the Content Client User Manual.

## Processing Form Data

In a form instance, the data of the individual form fields is stored in an XML notation. To generate these XML data from an HTML form, the methods of the `FormData` class are used.

## The FormData class

Full name: `de.gauss.vip.portalmanager.util.forms.FormData`

The `FormData` class provides methods for processing form data, in particular for reading data from an HTML form and saving the form data as XML document.

The `processFilledForm` method of the `FormData` class is of particular importance in this context. This method generates an XML document and extracts the metadata for the WCM object from the form data supplied. For more information on this method, please refer to the Javadoc on the Portal Manager API.

An additional utility class for processing forms and their data is the `XmlContentConverter` class, which allows to directly separate fields from the XML document. For further information on this class and its methods, please refer to the Javadoc on the Portal Manager API.

## Creating an Object

After the content and the metadata for a form instance have been extracted and converted using the `processFilledForm` method, the form instance must be created with the relevant data. The `VipObjectHandlerBean` class (see section “The `VipObjectHandlerBean` class” on page 121) establishes the connection to the functions of Livelink WCM Server. For more information on the methods of this class, please refer to the Javadoc on the Portal Manager API.

The `VipObjectHandlerBean` is also the basis for form-supported work with objects of a WCM-managed website. In principle, it is possible to develop an HTML-based user interface for Livelink WCM Server. Using the methods of the `VipObjectHandlerBean`, all functions from creating a new object, to submitting it to Quality Assurance, and to releasing the object can be realized via this type of form-based user interface.

# Statification

In order to speed up access to dynamic content, it might be useful to convert the code of the dynamic page into static code. The term *statification* refers to the representation of dynamic content on static pages. When statification is performed, the code of the dynamic page is already executed during the (asynchronous) generation of the page and the result – usually HTML code – is saved.

Statifying pages means freezing the object state and publishing it in the Production view of a website. For example, in the case of a page containing navigation elements, the current navigation status at the time of publication is recorded in the form of static elements on the page. Changes in the topic structure released subsequently are not reflected in the statified navigation. The dynamic page elements are updated only when the page is regenerated or republished. Statification is generally unsuitable for personalized pages, as these pages have to react dynamically, i.e. at runtime, to the user data entered.

Statification of pages depends on various conditions.

### Configuration of the deployment system

- In the Admin client, you determine whether statification is to be performed for all objects of the website, for no objects or depending on the object.
- In the Admin client, the option *Analyze statified pages* can be activated. During this analysis, the OIDs for the WCM URLs used on the pages are saved in the database. If the URL of a referenced object changes, the respective pages are automatically regenerated and statified.

For detailed information on the configuration of deployment systems, refer to the online help of the Admin client.

## Object type

Only objects whose object type is assigned to the attribute set “dynamic” can be statified. These are the following object types:

- “JSP page”, “JSP topic”, and “JSP template”
- “Form instance” and “Form template”.
- “XML document”, “XML template”, “XSLT document”, and “XSLT template”

## Special Attributes

The following attributes are important for statification of objects:

- **generate\_static**

The special attribute `generate_static` is used to control the object-dependent statification (see section “Object-Dependent Statification” on page 40).

In the Content client, this attribute is called “Statify page”.

- **timeout**

If the code of a dynamic page is to be converted into static code, the respective page is first generated as temporary page. This temporary page is executed via a URL connection. The result of this execution is stored as final static page.

When configuring the deployment system in the Admin client, you set the waiting time (in milliseconds) for establishing the URL connection to the temporarily generated dynamic page. This value can be overwritten by means of the special attribute `timeout` in the metadata dialog of the Content client.

The timeout value that applies to a page is determined as follows:

1. The value of the special attribute `timeout` is determined for the object to be statified.
2. The value of the special attribute `timeout` is determined for the template assigned to the object to be statified (if necessary, the template cascade is searched).
3. The timeout value is read from the configuration of the deployment system.

Thus, the timeout value specified in the Admin client is only used if neither the object to be statified nor one of its templates has a valid value for the special attribute `timeout`.

**Note:** The statification is performed by means of the Content server running in the context of the JSP engine or as web application in the application server. Especially if this Content server is overloaded, the valid timeout value may be exceeded. This results in the abortion of the statification attempt and in error messages. Accordingly, select generous timeout values.

### ■ `absolute_urls`

The special attribute `absolute_urls` plays, among other things, a special role when statifying form instances (see section “Statification of form instances” on page 41).

## Object-Dependent Statification

An object whose type is assigned to the attribute set “dynamic” possesses the special attribute `generate_static`. You can use this attribute to determine in the Metadata dialog of the Content client how the code of the object and – if existing – of its template(s) is to be statified. This requires that object-dependent statification is set for the deployment system in the Admin client.



### Statification of JSP objects

Statification of JSP objects is object-dependent if the special attribute `generate_static` has the value `on` or `local`:

- `on`: This value causes the content of the object to be converted into static code. The same applies to all templates assigned to the object, provided their content is inserted into the page which the deployment system generates for the object.
- `local`: Regarding the body element, this value has the effect that only the content of the object is converted into static code. Additionally, the content of the entire head element – which may consist of the merged content of the head elements of the template cascade – is converted into static code.

Variables defined in a JSP template can only be used by objects that have the same “statification status” as the JSP template. For the object to be able to use the variable from the JSP template, the following conditions must be met in the case of a not completely statified template cascade:

- either the code of both the JSP template and the object must be statified
- or neither the code of the JSP template nor the code of the object may be statified.

### Statification of form instances

Statification of form instances is always object-dependent. If in the Admin client statification was set to “always” for a deployment system, it is not possible to create, edit, or statify form instances by means of a JSP page in this deployment system. In a deployment system with such a configuration, you cannot work with forms in a useful way.

Although the special attribute `generate_static` is also analyzed for the statification of form instances, a different algorithm is used than with JSP objects: form instances are statified by means of a JSP object that must be specified. The OID of the JSP object that takes over the actual statification is located in the special attribute `generate_static`. If this special attribute does not have a value in a form instance, the system checks the instance's form template to see if such a value exists. If this is the case, the value is used for statification. The OID of the form instance is sent to the determined JSP object for statification. Afterwards, the result (usually an HTML page) is generated by the deployment system. References from other objects to the form instance are automatically mapped to this HTML page.

If the option *Analyze statified pages* is activated for the deployment system, all form instances to which this JSP object is assigned are regenerated and statified each time the JSP object is changed.

The JSP object determines the layout and presentation of the data from the form instance. If no OID is stored in the special attribute `generate_static`, the deployment system generates the unchanged content of the form instance in the form of XML code.

When a deployment system is created in the Admin client, you determine how the references in the WCM objects are converted into URLs. The following options exist:

- *absolute*: Absolute paths are used in the links.  
Example: `<a href="http://www.company.example/products/NewProducts.htm">`
- *relative*: Relative paths are used in the links.  
Example: `<a href="NewProducts.htm">`
- *server-relative*: Relative references without specification of protocol and base URL are used in the links.

Example: `<a href="/products/NewProducts.htm">`

In the metadata dialog of the Content client, you can overwrite the setting in the Admin client by means of the special attribute `absolute_urls`.

Deployment systems created with the option *relative* calculate the references that the JSP object contains relatively to the path of the JSP object. When the JSP object is executed for statifying a form instance, the relative URLs are written to the static code of the form instance, where they are no longer adapted. This causes problems if the form instance is not located in the same directory as the JSP object used for statification. In this case, the relative URLs are wrong. To avoid this problem, set the special attribute `absolute_urls` for the JSP object to `on` (or to any value other than `SERVER_RELATIVE`). This generates absolute URLs on the executed JSP page. When the form instance is statified, these URLs are converted into relative URLs – relatively to the newly generated form instance.

### **Statification of XML objects**

XML objects can be statified. Two cases are differentiated for statifying XML objects:

- The XML object is assigned a form template **or** the special attribute `generate_static` of the XML object has a purely numeric value.

Statification is performed in the same way as for form instances, i.e. by means of a JSP page (see section “Statification of form instances” on page 41). For each XML object, the following files are generated:

- the unstatified page
- the statified page
- if required: the surrogate page with a link to the XML document

**Note:** When a form template is used, the content of the unstatified page is distributed as HTML code (file extension `.html`) because the form template is taken into account. This means that the JSP page responsible for statification cannot directly evaluate the XML code of the XML object. For this reason, it is not recommendable to assign a form instance to an XML object.

- The special attribute `generate_static` of the XML object does not have a numeric value. No form template is assigned to the XML object. At least one of the templates assigned to the XML object is a JSP template. In this case, the template directly assigned to the XML object should be an XSLT template.

Statification is performed in the same way as for JSP objects (see section “Statification of JSP objects” on page 41). For each XML object, the following files are generated:

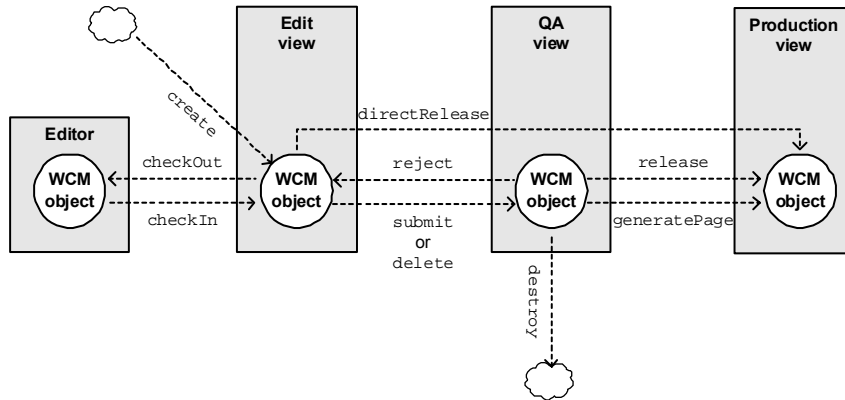
- the statified page
- if required: the surrogate page with a link to the XML document

## Staging

All objects of the WCM-managed website pass through the following stages: editing, quality assurance, and publication for productive operation. Editing and quality assurance of the objects is carried out in the Content client. After publication, the objects can be accessed via a browser. Passing through these stages is called *staging*.

Objects of a WCM-managed website are edited using methods of the Portal Manager API, especially the methods of the `VipObjectHandlerBean`. Some of these methods change the content or the metadata of an object, others cause a change in object status. This represents the multi-stage staging of Livelink WCM Server. This staging

corresponds to the life cycle of a WCM object, which is shown in the following diagram.



**Fig. 5 – The life cycle state of a WCM object.**

The life cycle of a WCM object may include the following transitions, which are represented by the `VipObjectHandlerBean` methods:

- The object is created in the Edit view (`create`).
- For editing an object's content, the object can be checked out in the Edit view and changed in an editor program (`checkOut`).
- After changes have been made, the object must be checked in again in the Edit view (`checkIn`).
- The object becomes available in the QA view by being submitting to Quality Assurance in the Edit view (`submit`).
- In the QA view, the object can be released and thus moved to the Production view (`release`).
- In the Edit view, the object can be released if the metadata item "Direct release" is set for the object and if the object has already been released once with this setting (`directRelease`).

- The object can also be rejected in the QA view and sent back to the Edit view (`reject`).
- Regenerating an already released dynamic object that has been statified in the Production view updates the object in the Production view (`generatePage`).
- Objects are deleted in the Edit view (`delete`). If the respective object has already been released, the object must be destroyed in the QA view afterwards (`destroy`). However, if the deletion is rejected in the QA view (`reject`), the object becomes available again in the Edit view.

Please refer to the Content Client User Manual for more information on the staging. For further information on the `VipObjectHandlerBean`, see section “The `VipObjectHandlerBean` class” on page 121. The Javadoc on the Portal Manager API provides a detailed description of the methods of this class.

## Integrating External Programs

The Portal Manager API is basically a framework that allows the integration of external programs such as e-commerce applications. In this context, subjects such as dynamic content, personalization, and correlation with website content play an important role.

Integrating an external program is relatively easy and very well prepared for in the Portal Manager API. If you want to implement external programs within the given framework, you should observe several design principles. The easiest way to implement external applications is to use a bottom-up strategy.

- In most cases, an external program functions exclusively as data source and data target. For this purpose, it is necessary to implement appropriate repositories or use already existing repository implementations. Repositories are important (generally, the most important or sometimes even the only) components of an application.

New repositories can be implemented by extending existing repositories or by developing them on the basis of the abstract basic repository (`RepositoryImpl`).

- For each new program, it is necessary to develop an application class. This class describes the basic functions and behavior of the program. On this level, however, no functions should be combined to form function units.

You can create a new application class very quickly if you derive it from the abstract basic implementation (`ApplicationImpl`) supplied.

One or more repositories are an essential component of every application. For most programs, you do not have to develop additional functions because the repository already sufficiently describes the program. For example, the applications that implement the various aspects of Livelink WCM Server do not have methods apart from those of the abstract basic implementation for applications.

- If necessary, each application is encapsulated by one or more beans. You must also develop this bean. Here, too, it is recommended to use a given abstract basic implementation as a starting point.

The bean carries out two tasks. First, it creates the more or less formal interface to the JSP script and to the JSP engine's session handling. The basic implementation of the bean performs this task. The second, more demanding function of the beans is to make available simple but powerful access methods at the highest possible abstraction level for an application and its repositories. The bean combines the basic functionality made available by an application with the access possibilities of the repositories, thus creating consistent complex methods. Furthermore, these methods should be simple and reflect the possibilities of the application on which they are based as "naturally" as possible.

Finally, you should use the bean methods in a JSP script whenever possible, although access to the application and even the repositories is, of course, always possible.

The basic implementation is designed in such a way that the user interface can be presented in different languages.

- In order to be able to control an external program via the browser, you must design an HTML page with a JSP script. The HTML page must use the bean that originated previously and must be an WCM-managed object. It is therefore necessary to integrate the HTML page in the website to be used for reaching the underlying external program.

Following the described design specifications automatically results in the architecture typical of the Portal Manager API, which has already been described. The following figure illustrates the relevant section slightly modified.



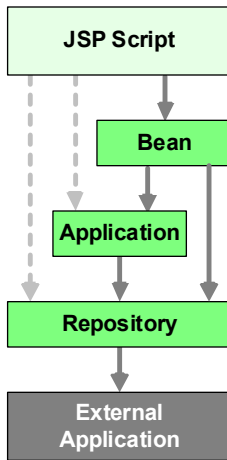


Fig. 6 – Principal class design in the Portal Manager API

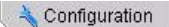


---

# CHAPTER 3

## Managing the Portal Manager API

This chapter provides information on the Portal Manager API configuration. It also contains information on using an LDAP directory service and locating errors.

The entire configuration of your WCM system is performed via the Admin client. This is also the central point for administering and configuring the Portal Manager API. To open the configuration, launch the Admin client and click the  Configuration tab.

The items you can manage in this view are displayed in a tree structure.



**Fig. 7 – The Configuration items**

In this chapter the following configuration items are described:

**Table 2 – Functions for configuring repositories and applications**

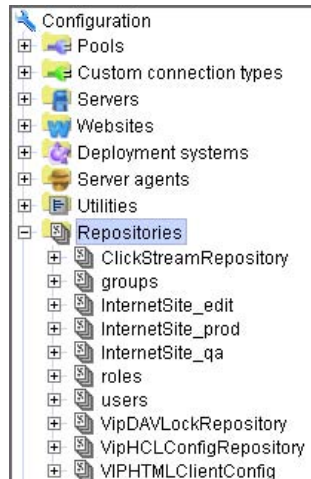
Item	Available functions
<i>Repositories</i>	Adding, editing, and deleting repositories See section “Managing Repositories” starting on page 52
<i>Applications</i>	Adding, editing, and deleting applications See section “Managing applications” starting on page 65

The administration of the other items and the use of the Admin client is described in detail in the Livelink WCM Server Administrator Manual.

## Managing Repositories

For general information on repositories, refer to section “Repositories” on page 106.

If you open the *Repositories* item, you will see all available repositories – both in the tree on the left and in a list in the right window pane.



**Fig. 8 – Overview of available repositories**

The displayed repositories `ClickStreamRepository`, `groups`, `roles`, `users`, `VipDAVLockRepository`, `VipHCLConfigRepository`, and `VIPHTMLClientConfig` are created during the installation of the WCM system. These repositories are supplied with the Portal Manager API and should not be deleted. The repositories `InternetSite_edit`, `InternetSite_prod`, and `InternetSite_qa` were automatically added when the website “InternetSite” was created.

The following functions are available to you for repositories:

- adding a repository (see the following section)
- configuring repositories by means of parameters (see section “Setting the Parameters for a Repository” on page 55)
- editing repository settings (see section “General Settings of Repositories” on page 62)

- overview of assigned applications, assign new applications, and remove assignments (see section “Assigning an Application to a Repository” on page 62)
- removing a repository (see section “Deleting a Repository” on page 65)

## Adding a Repository

To add a repository:

1. Select *Configuration* → *Repositories*.
2. Choose *New repository* on the context menu or click the corresponding icon.



Icon for adding a repository

The dialog box for adding a repository opens.



**Fig. 9 – Adding a repository**

3. In the *Name* field, enter a name for the new repository. You have a free choice of name, but it must be unique within the WCM system.

For repositories of type `VipObjectRepository` and `VipObjectHandlerRepository`, this may be the name of the deployment system.

You cannot change the name of the repository subsequently.

4. In the *Class name* field, enter the name of the repository's Java class.

The class loader uses this name to load the class. It is expected that the class implements the `Repository` interface and has an empty public constructor.

5. Click the *Finish* button.

## Setting the Parameters for a Repository

Repositories are configured by means of parameters. Type and number of parameters depend on the functions of the repository.

Parameters may be defined as nodes or as individual parameters. Nodes group parameters of the same type. Nodes may contain other nodes. Unlike parameters, they do not have a value.

To define parameters for a repository:

1. Select *Configuration* → *Repositories* → *{repository name}* → *Parameters*.
2. Choose *New parameter* on the context menu or click the corresponding icon.



Icon for adding a parameter

The *New node* dialog box opens.



**Fig. 10 – Adding a parameter for a repository**

3. Select whether you want to add a *Node* or a *Parameter*.

If you want to assign numerous parameters to a repository, we recommend that you group the parameters. First, add the required nodes and then one or more parameters below the nodes.

4. If you clicked the *Node* radio button, enter a unique name for the new node.

If you clicked the *Parameter* radio button, enter a name and a value for the parameter.

5. Click the *OK* button.

**Note:** The repositories `ClickStreamRepository` and `VipDAVLockRepository` save their data in a database table. They can only be used if a JDBC pool is assigned to them. This JDBC pool manages the connections to the database. To assign a JDBC pool, specify the name of the respective pool as the value for the parameter `store/{table name}/poolname/default`.

The following table gives you an overview of the parameters expected by the repositories supplied with the Portal Manager API.



Table 3 – Repositories and their parameters

Repository	
Parameter	Description
<b>VipHclConfigRepository</b>	
ProfileDataRefresh-Milliseconds	The time (in milliseconds) that an unchanged profile is maintained in the Content client until it is reloaded from the Administration server
SaveWaitMilli-seconds	The interval (in milliseconds) for saving changes to the profile data
<b>DirectoryRepository</b>	
authentication	The authentication method. The values SIMPLE (no encryption) and NONE are possible. NONE should not be used because in this case no authorization is performed and no data can be read or written.
base	The start node under which the entries are searched or entered Example: ou=software, o=company.example
build-profile-filter_expr	The search expression that is used when searching for groups and roles. The expression *DN* is replaced by the “distinguished name”. The expression *UID* is replaced by the user ID; or, if no user ID has been supplied to the method, by the “distinguished name”.
context-factory	The “Initial Context Factory” for the LDAP classes. Currently, only the class com.sun.jndi.ldap.LdapCtxFactory is supported.
credentials	The password of the user entered under principal.
credentials-changes	The password of the user entered under principal-changes.

Repository	
Parameter	Description
credentials-read	The password of the user entered under principal-read.
dn-pattern	The pattern for the structure of the “distinguished name” (dn). * is replaced by the key value. Example: uid=*, ou=software, o=company.example becomes uid=smith, ou=software, o=company.example, if the current key value is smith.
group-base	The start node below which the entries for the user’s groups are searched. If the attribute has not been set, the value of base is used for this attribute.
mappings	The mapping between the LDAP parameters and the WCM parameters. Possible mappings are, for example, name and language. These have the attributes ldapattr and pomaattr. The attribute pomaattr must not be changed (see figure 22 “The DirectoryRepository with its parameters” on page 89).
member-key-attr	The name of the LDAP attribute under which the users of a group or role are stored
objectclass	The name of the object class that should be inserted when creating a new entry, provided that no object class has been specified (example: vipUser). The entries are separated by spaces.
password-attr	The name of the LDAP attribute to be used as the password in Livelink WCM Server
principal	An LDAP user with the right to add entries Example: uid=admin, ou=software, o=company.example

<b>Repository</b>	
<b>Parameter</b>	<b>Description</b>
principal-changes	An LDAP user with write access to the entries in the LDAP directory service. If this parameter has not been set, entries are changed and deleted in the context of the logged-in user.
principal-read	An LDAP user with read access to the entries in the LDAP directory service. If the parameter has not been set, entries are read in the context of the logged-in user.
role-base	The start node below which the entries for the user's roles are searched. If the attribute has not been set, the value of base is used for this attribute.
search-scope	Describes the search scope below the start node specified in dn-pattern. SUBTREE_SCOPE searches the entire sub-tree beginning with BASE, ONELEVEL_SCOPE only searches in the level directly below BASE.
url	The list of URLs to directory servers separated by spaces. Initially, the first LDAP server is addressed. If this fails, the second will be addressed, etc.  Example: ldap://ldap.company.example:389 ldap://ldap2.company.example:389
<b>DoorwaysRepository</b>	
alias	An alias for the DoorwaysRepository. If no alias has been specified, the name of the DoorwayRepository is used in order to establish the connection to the Doorways server.  This parameter is optional.

<b>Repository</b>	
<b>Parameter</b>	<b>Description</b>
authentication	<p>The authentication method. Possible values are LOGIN, SESSION, and CONFIG.</p> <p>LOGIN: For the DoorwaysBean class, an explicit login must be made using the DoorwaysBean.setLogin method.</p> <p>SESSION: The login information from the SessionBean class will be used.</p> <p>CONFIG: All users are logged in with the user information of a default user. The user information is specified by means of the user and pwd parameters.</p>
hostname	The host name of the Doorways server to which a connection is to be established
port	The port of the Doorway server for the connection
pwd	The password of the default user entered under user
user	The default user whose user account is used for the CONFIG authentication method.

Repository	
Parameter	Description
<b>VipObjectHandlerRepository (for deployment systems)</b>	
deployment-name	<p>The name of the deployment system, e.g. {website name}_edit, whose objects are accessed via the repository</p> <p><b>Notes:</b></p> <p>Alternately, you can also use the <code>view</code> and <code>website</code> parameters to configure the repository. In this way, you can access the website objects without having to set up deployment systems. Repositories configured in this way cannot return deployment-specific object data, such as an URL or path.</p> <p>The <code>deployment-name</code> parameter takes priority over the <code>view</code> and <code>website</code> parameters. That means that these two parameters are ignored, if the <code>deployment-name</code> parameter is specified.</p>
metadata-style	<p>The metadata scheme used. In this parameter, you specify whether the repository uses the attribute names and values according to the version 5e or 8. Possible values are 5e and 8. For a comparison of the respective values and attributes, refer to appendix A “Metadata Schemes”.</p>

### Deleting repository parameters

In order to remove a parameter from the configuration of a repository, mark the parameter to be deleted in the right window pane. Choose *Delete parameter* on the context menu or click the corresponding icon.



Icon for deleting a parameter

## General Settings of Repositories

To edit the settings of a repository:

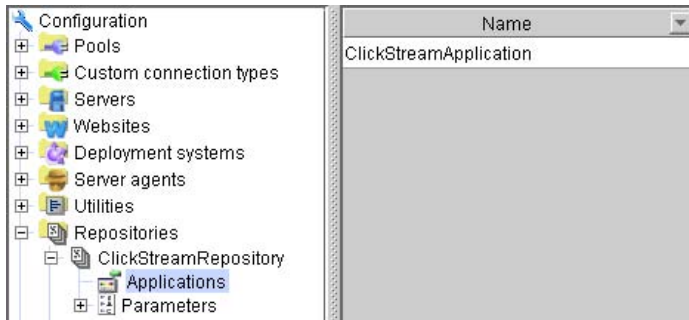
1. Select *Configuration* → *Repositories*.
2. Mark the desired repository in the tree on the left.

The settings are displayed in the right window pane. You can subsequently change the name of the Java class of the repository. The name of the repository itself is defined when the repository is added. You cannot change it subsequently.

3. Click the *Apply* button.

## Assigning an Application to a Repository

Select *Configuration* → *Repositories* → *{repository name}* → *Applications* to display an overview of the applications that have already been assigned to the repository. The assigned applications are displayed in a list in the right window pane.



**Fig. 11 – Applications assigned to a repository**

### Assigning an application to a repository

To assign an application to a repository:

1. Select *Configuration* → *Repositories* → *{repository name}* → *Applications*.
2. Choose *Assign application* on the context menu or click the corresponding icon.



Icon for assigning an application to a repository

The *Select application* dialog box opens. It shows a list of the applications not yet assigned.

3. Mark the desired application(s).
4. Click the *OK* button.

### Removing the assignment of an application to a repository

To remove the assignment of an application to a repository, mark the respective application in the right window pane. Choose *Remove assignment of application* on the context menu or click the corresponding icon.



Icon for removing the assignment of an application to a repository

## Structure Below the Tree Item *Repositories*

Each repository has the nodes *Applications* and *Parameters*. If you have assigned applications or parameters to a repository, these entries will appear below the respective node (in the illustration, the parameters max-records, store, table, and tosql have been assigned to the ClickStreamRepository).

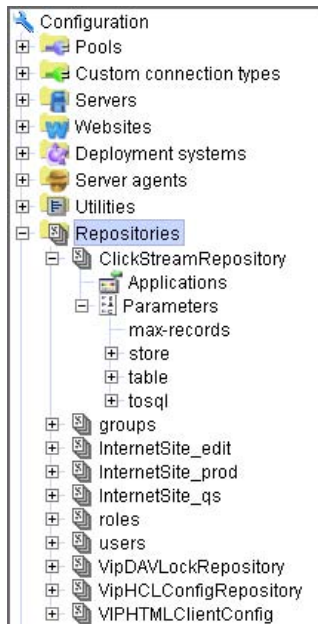


Fig. 12 – Structure below the tree item *Repositories*



## Deleting a Repository

To delete a repository:

1. Select *Configuration* → *Repositories*.
2. Mark the desired repository in the tree on the left.
3. Choose *Delete repository* on the context menu or click the corresponding icon.



Icon for deleting a repository

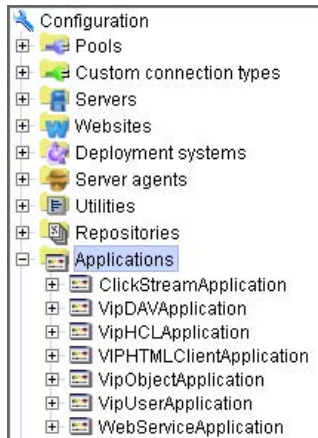
4. Confirm the security prompt by clicking the Yes button.

The configuration of the repository is deleted from the WCM system, the associated Java class is, however, not deleted.

## Managing applications

For general information on applications, see section “Applications” on page 111.

If you open the *Applications* item, you will see all available applications – both in the tree on the left and in a list in the right window pane.



**Fig. 13 – Overview of available applications**

The applications ClickStreamApplication, VipDAVApplication, VipHCLApplication, VIPHTMLClientApplication, VipObjectApplication, VipUserApplication, and WebServiceApplication, which are displayed here, are created during the installation of the WCM system. These applications are supplied with the Portal Manager API and should neither be changed nor deleted. However, you can add and edit any of your own applications.

The following functions are available for applications:

- adding an application (see the following section)
- configuring applications by means of parameters (see section “Setting the Parameters of an Application” on page 72)
- editing application settings (see section “General Settings of Applications” on page 81)

- overview of assigned repositories, assigning new repositories, and removing assignments (see section “Assigning a Repository to an Application” on page 81)
- deleting an application (see section “Deleting an Application” on page 84)
- assigning an application to a Content server (see section “Assigning an Application to a Content Server” on page 84)

## Adding an Application

To add an application:

1. Select *Configuration* → *Applications*.
2. Choose *New application* on the context menu or click the corresponding icon.



Icon for adding an application

3. Follow the instructions of the wizard.

## Defining Basic Data

In the first dialog boxer you define the basic data of the application.



Fig. 14 – Defining basic application data

- **Name:** The name of the application must be unique and start with a letter.

You cannot change the name subsequently.

- **Class name:** name of the application's Java class. The class loader uses this name to load the class. It is expected that the class implements the application interface and has an empty public constructor.
- **Logical name:** The logical name of the application is the name the Portal Manager API uses to address the application. If, for example, two Content servers are to possess differently configured VipUserApplications, two VipUserApplications can be created with different names that both have the logical name "VipUserApplication".

- *Language*: default language of the application. The language is used, among other things, to read localized resources. An entry of type {language code}\_{country code} (e.g. en\_US) is expected. For more information, refer to the Java SDK API documentation of the `java.util.Locale` class.

Confirm your entries by clicking the *Next* button.

## Quit Wizard?



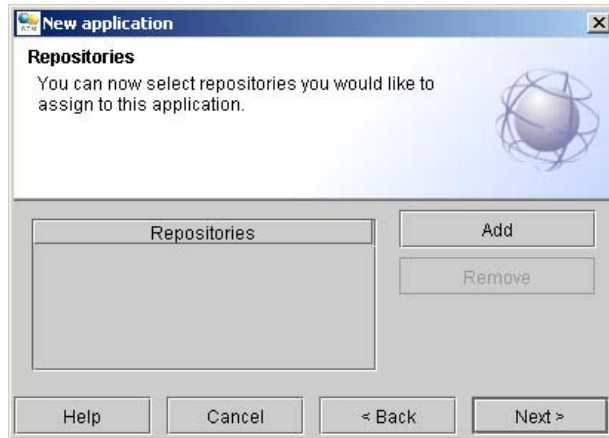
**Fig. 15 – Quit wizard?**

The information that you have specified so far is sufficient for adding the application. You can, however, optionally assign repositories and resources to the application.

Click the desired radio button and confirm by clicking the *Finish* or *Next* button.

## Assigning Repositories

In this dialog box, you can assign repositories to the application already during the setup.



**Fig. 16 – Assigning a repository to an application during setup**

Proceed as follows:

1. Click the *Add* button.
2. In the list displayed, mark the repositories that you want to assign.
3. Confirm by clicking the *OK* button.

The repositories are included in the wizard.

4. Click the *Next* button.

## Assigning Resources

The resources form the basis for internationalization in the beans. In this dialog box, you can configure access to the Java resources.



**Fig. 17 – Assigning resources when adding an application**

Proceed as follows:

1. Click the *Add* button.
2. A dialog box opens in which you enter the name of the resource.

The resources specified here are loaded via the class path and must correspond to the Java resources (for detailed information, refer to the Java SDK API documentation of the `java.util.ResourceBundle` class).

3. Confirm the entries by clicking the *OK* button.

The resource is included in the wizard.

4. Click the *Finish* button.

## Setting the Parameters of an Application

Applications are configured by means of parameters. Type and number of parameters depend on the functions of the application.

Parameters may be defined as nodes or as individual parameters. Nodes group parameters of the same type. Nodes may contain other nodes. Unlike parameters, they do not have a value.

**Note:** Do not delete the applications that are supplied with the Portal Manager API (ClickStreamApplication, VipDAVApplication, VipHCLApplication, VIPHTMLClientApplication, VipObjectApplication, VipUserApplication, and WebserviceApplication). The basic data of these applications should not be changed either.

To define parameters for an application:

1. Select *Configuration* → *Applications* → {application name} → *Parameters*.
2. Choose *New parameter* on the context menu or click the corresponding icon.



Icon for adding a parameter

The *New node* dialog box opens.



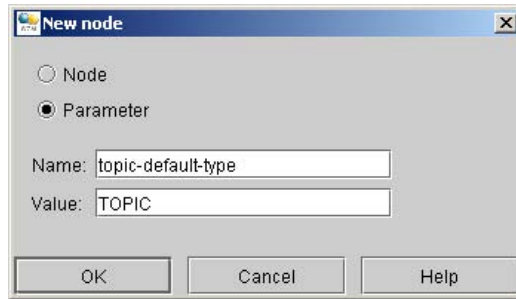


Fig. 18 – Adding a parameter for an application

3. Select whether you want to add a *Node* or a *Parameter*.

If you want to assign numerous parameters to an application, we recommend that you group the parameters. First, add the required nodes and then one or more parameters below the nodes.

4. If you clicked the *Node* radio button, enter a unique name for the node.

If you clicked the *Parameter* radio button, enter a name and a value for the parameter.

5. Click the *OK* button.

The following table gives you an overview of the parameters expected by the applications supplied with the Portal Manager API.

Table 4 – WCM Applications and their parameters

Application	
Parameter	Description
<b>ClickStreamApplication</b>	
none	none

<b>Application</b>	
<b>Parameter</b>	<b>Description</b>
<b>VipDAVApplication</b>	
import-indexfile-name	<p>Specifies the name of index files in directories. When new resources are created, files with this file name are interpreted as index files – independent of the file extension. The content of these files is integrated in the index file of the respective collection.</p> <p>Default value: index</p>
lock-cleanup-interval	<p>Specifies the interval (in milliseconds) for searching for expired locks</p> <p>Default value: 600000 (corresponds to 10 minutes)</p>
lock-infinite-timeout	<p>Specifies the maximum expiration interval for a lock</p> <p>Default value: 86400000 (corresponds to 24 hours)</p>
topic-default-type	<p>WebDAV makes it possible to add collections (directories, WCM objects of a certain topic type). This parameter specifies the topic type of objects that are created by the WebDAV command MKCOL.</p> <p>Default value: TOPIC</p>
vip-default-attributes	<p>Specifies the name of the WCM metadata that can be queried via the WebDAV interface.</p> <p>Default value: oid, title, subtitle, description, target_group, keywords</p>

Application	
Parameter	Description
<b>VipHCLApplication</b>	
default-profile-user	<p>Specifies the ID of the WCM user whose default profile is to be used as the template for the default profile of new users.</p> <p>Default value: empty (i.e. there is no template for default profiles)</p>
embedded-type-names	<p>Contains a comma-separated list of object types. When a WCM object is added, objects of this type are regarded as embedded objects.</p> <p>If this parameter is not set, the following list is used as the default value: ETC , GIF , JAVASCRIPT , JPG , PNG</p>
hcl-absolute-servlet	<p>Specifies the absolute URL for addressing the controller servlet of the Content client. The controller servlet is responsible for controlling the Content client.</p> <p>Example: <code>http://wcmserver.company.example/hclservlet</code></p>
hcl-root	<p>Specifies the absolute URL for addressing the start page of the Content client</p> <p>Example: <code>http://wcmserver.company.example/cmsclient</code></p>

<b>Application</b>	
<b>Parameter</b>	<b>Description</b>
hcl-servlet-url	<p>Specifies the name for addressing the controller servlet of the Content client. The controller servlet is responsible for controlling the Content client.</p> <p>Default value: hclServlet</p>
{name of an InSite Editing deployment system}	<p>Configures the window with the object information for the respective InSite Editing deployment system. The following parameters can be added below this parameter node:</p> <p>keys: space-separated list of attributes displayed for each object. By default, oid, title, type, state, and version are displayed.</p> <p>width: width of the window in pixels</p> <p>height: height of the window in pixels</p>
<b>VIPHTMLClientApplication</b>	
DEFAULT_VALUE_FOR_APPLET_WORK_DIRECTORY	<p>Indicates the local working directory for the Download applet of the Content client (Classic). The value must not be empty and has to be a valid directory.</p> <p>Default value: c:\temp</p>
DEFAULT_VALUE_FOR_AUTO_REFRESH_TIME	<p>Indicates the period of time in seconds after which the action list of the Content client (Classic) is automatically updated. The value must be a positive integer or 0.</p> <p>Default value: 0</p>

Application	
Parameter	Description
DEFAULT_VALUE_FOR_NAVIGATION_APPLET_OBJECTLIST_POS	Specifies where in the Content client (Classic) to display the object list when the Navigation applet is used. Possible values are: 0 = below the topic structure 1 = to the right of the topic structure 2 = within the topic structure Default value: 0
DEFAULT_VALUE_FOR_NO_AUTO_TASKMANAGER_WINDOW	Specifies whether the action list of the Content client (Classic) is to be opened automatically when starting an asynchronous action. Possible values are false (open) or true (do not open). Default value: false
DEFAULT_VALUE_FOR_NO_OBJECT_PREVIEW_MODE	Determines whether the object preview of the Content client (Classic) is to be switched off. Possible values are false (do not switch off) or true (switch off). Default value: false
DEFAULT_VALUE_FOR_USE_DOWNLOAD_APPLET	Determines whether the Download applet of the Content client (Classic) is to be used. Possible values are false (do not use) or true (use). Default value: true
DEFAULT_VALUE_FOR_USE_INTEGRATED_HTML_EDITOR	Determines whether the integrated HTML editor of the Content client (Classic) is to be used. Possible values are false (do not use) or true (use). Default value: false

Application	
Parameter	Description
DEFAULT_VALUE_FOR_USE_NAVIGATION_APPLET	Determines whether the Navigation applet of the Content client (Classic) is to be used. Possible values are <code>false</code> (do not use) or <code>true</code> (use). Default value: <code>false</code>
DEFAULT_VALUE_FOR_USE_SEPARATE_PREVIEW_WINDOW	Determines whether a separate browser window is to be opened for the object preview of the Content client (Classic). Possible values are <code>false</code> (no separate window) or <code>true</code> (separate window). Default value: <code>false</code>
MAX_SHOWN_FILTER_RESULTS	Specifies the maximum number of objects to be displayed in the hitlist of the Content client (Classic) after applying the object filter. Default value: 1000
MAX_SHOWN_LOG_ENTRIES	Specifies the maximum number of entries to be displayed in the log of the Content client (Classic). Default value: 100
MAX_SHOWN_PRINCIPALS	Specifies the maximum number of entries to be displayed in the selection lists for adding principals to an access control list (see dialog box for changing the access control list). Default value: 300

<b>Application</b>	
<b>Parameter</b>	<b>Description</b>
MAX_SHOWN_REFERENCES	Specifies the maximum number of entries to be displayed in the lists in the References dialog box of the Content client (Classic). Default value: 300
MAX_SHOWN_TEMPLATES	Specifies the maximum number of entries to be displayed in the selection list <i>Template</i> of the Content client (Classic) (see Metadata dialog box and dialog box for adding an object). Default value: 300
MAX_SHOWN_VERSIONS	Specifies the maximum number of versions to be displayed in the <i>Restore old version</i> dialog box of the Content client (Classic). The most current object versions are displayed. Default value: 300
OPTION_SHOW_DEBUG_INFO	If the value of this parameter is <code>true</code> , additional information is displayed in some dialog boxes of the Content client (Classic). This information can be used for locating errors. Default value: <code>false</code>
<b>VipObjectApplication</b>	
none	none
<b>VipUserApplication</b>	
extend-profile	A list of repositories separated by spaces. These repositories are used for the extension of the profile.

Application	
Parameter	Description
<b>WebServiceApplication</b>	
allowedWebsites	A comma-separated list of website names. The specified websites can be accessed via WCM WebServices. Entering an asterisk (*) allows access to all WCM-managed websites.  Default value: empty (access via WCM WebServices not possible)
deniedWebsites	A comma-separated list of website names. The specified websites cannot be accessed via WCM WebServices. Entering an asterisk (*) blocks access to all WCM-managed websites. A prohibition takes priority over a permission.  Default value: empty

### Deleting an application parameter

In order to remove a parameter from the configuration of an application, mark the desired parameter in the right window pane. Choose *Delete parameter* on the context menu or click the corresponding icon.



Icon for deleting a parameter



## General Settings of Applications

To edit the settings of an application:

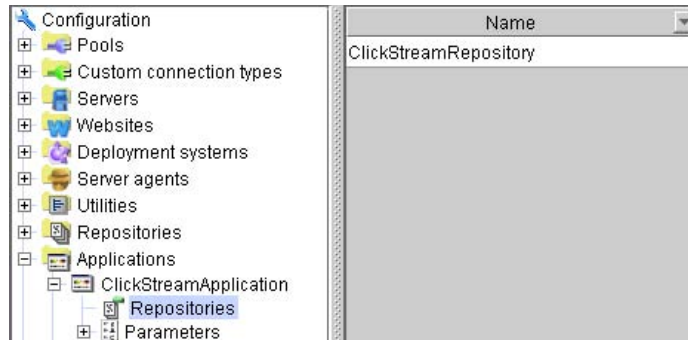
1. Select *Configuration* → *Applications*.
2. Mark the desired application in the tree on the left.

In the right window pane, the settings are displayed on two tabs.

- The settings on the *Basic data* tab are made when the application is added. You can change all settings except the name subsequently.
  - On the *Resources* tab, you can change the settings for the access to Java resources.
3. Make the desired changes.
  4. Click the *Apply* button.

## Assigning a Repository to an Application

Select *Configuration* → *Applications* → *{application name}* → *Repositories* to display an overview of the repositories that have already been assigned to the application. The assigned repositories are displayed in a list in the right window pane.



**Fig. 19 – Repositories assigned to an application**

### Assigning a repository to an application

How to assign a repository to an application:

1. Select *Configuration* → *Applications* → *{application name}* → *Repositories*.
2. Choose *Assign repository* on the context menu or click the corresponding icon.



Icon for assigning a repository to an application

The *Select repository* dialog box opens. It shows a list of the repositories not yet assigned.

3. Mark the desired repository.
4. Click the *OK* button.

## Removing the assignment of a repository to an application

To remove the assignment of a repository to an application, mark the respective repository in the right window pane. Choose *Remove assignment of repository* on the context menu or click the corresponding icon.



Icon for removing the assignment of a repository to an application

## Structure Below the Tree Item *Applications*

Each application has the nodes *Repositories* and *Parameters*. If you have assigned repositories or parameters to an application, these entries will appear below the respective node (in the figure below, a parameter was assigned to the *VipHCLApplication*).

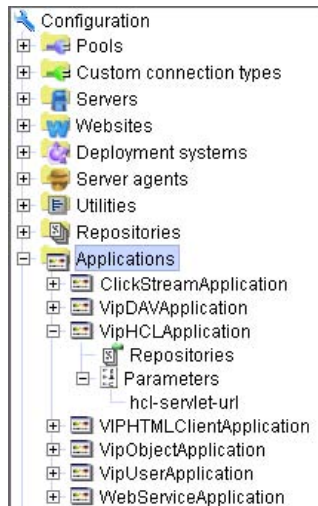


Fig. 20 – Structure below the tree item *Applications*

## Deleting an Application

To delete an application:

1. Select *Configuration* → *Applications*.
2. Mark the desired application in the tree on the left.
3. Choose *Delete application* on the context menu or click the corresponding icon.



Icon for deleting an application

4. Confirm the deletion with *Yes*.

The configuration of the application is deleted from the WCM system, the associated Java class is, however, not deleted.

## Assigning an Application to a Content Server

A Content server running in the context of a JSP engine or as a web application in an application server can only access applications that have been assigned to it. Select *Configuration* → *Servers* → *{name of the Content server}* → *Applications* to display an overview of the applications that have already been assigned to the server. The applications are displayed in a list in the right window pane.

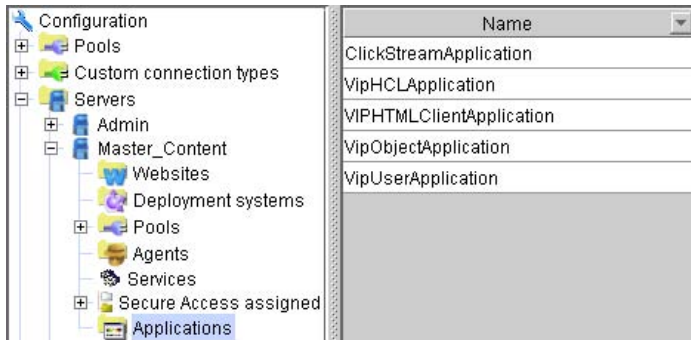


Fig. 21 – Applications assigned to a Content server

### Assigning an application to a Content server

To assign an application to a Content server:

1. Select *Configuration* → *Servers* → {name of the Content server} → *Applications*.
2. Choose *Assign application* on the context menu or click the corresponding icon.



Icon for assigning an application to a Content server

The *Select application* dialog box opens. It shows a list of the applications not yet assigned.

3. Mark the desired application.
4. Click the *OK* button.

**Note:** You cannot assign two applications with the same logical name to one Content server.

### Removing assignment of an application to a Content server

To remove the assignment of an application to a Content server, mark the application in the right window pane. Choose *Remove assignment of application* on the context menu or click the corresponding icon.



Icon for removing the assignment of an application to a Content server

## LDAP

By using an LDAP directory service, you can considerably reduce administration efforts connected with the configuration and maintenance of user data. The Portal Manager API is available on all Content servers running in the context of a JSP engine or as web applications in an application server. These Content servers access the general data storage of the WCM system. As a result, the user, group and role information of an LDAP directory service can be adopted and used by the Portal Manager API.

For detailed information on configuring the LDAP integration, refer to the Livelink WCM Server Installation Manual. The Portal Manager API also offers the possibility to access the LDAP directory service directly. The `DirectoryRepository` class of the Portal Manager API provides LDAP support.

## The DirectoryRepository class

The `DirectoryRepository` class implements the access to an LDAP directory server. This is an alternative way of accessing LDAP, independent of the LDAP connection of a master Administration server. The `DirectoryRepository` can be used to load data from the LDAP server and to store data there. For an application example, refer to section “Creating Users in an LDAP System via a Portal” on page 133.

The `DirectoryRepository` can be configured by means of parameters. Configuration involves, for example, specifying a search node (base) for searching and loading existing entries, and adding new entries. Each access to the LDAP directory service requires an authentication of the respective user. To be able to add new entries, the user must have administrator rights for the LDAP directory service.

The `DirectoryRepository` class can be used for the authentication of users for the Portal Manager API, since it implements the `LoginRepository` interface. If write access to the WCM system is required or if the requested objects cannot be read by all users, the user must log in using the `VipUserRepository`. For this login, the user is assigned a context ID which the WCM system requires for checking the rights.

Using a `DirectoryRepository` for user administration in the Portal Manager API requires the integration of additional object classes and attributes in the LDAP directory service. These settings must also be made when Livelink WCM Server accesses the LDAP directory service.

It is possible to add further object classes and attributes in the LDAP directory service. When adding an entry to LDAP, these object classes can be specified. They must be entered as `Value` in the `RepositoryEntry` under the key `AttributeName.OBJECTCLASS` as a `RepositoryMap`.

The identifiers for the attribute types in the LDAP directory service differ from those used internally in the Administration server. The `AttributeName` class contains constants for the valid attribute names and methods that help to map the internal attribute identifiers to the LDAP directory service spelling.

The mapping of identifiers is only necessary for individual implementations of a repository class that accesses the LDAP directory service. The internal attribute identifiers are automatically transformed into the identifiers for the LDAP directory service when the `DirectoryRepository` class writes data. When reading from the LDAP directory service, this process is automatically performed in the opposite direction.

**Note:** Microsoft Active Directory does not support overwriting an entry. Thus, when using Microsoft Active Directory, the `putEntry` method results in an exception if the entry already exists in the LDAP directory service. It is, however, possible to create a new entry. All method calls take place in the context of the user currently logged in to the session. Only getting the profile at login and creating new entries is performed in the context of `principal`.

Like any other repository, the `DirectoryRepository` must be configured using the Admin client.



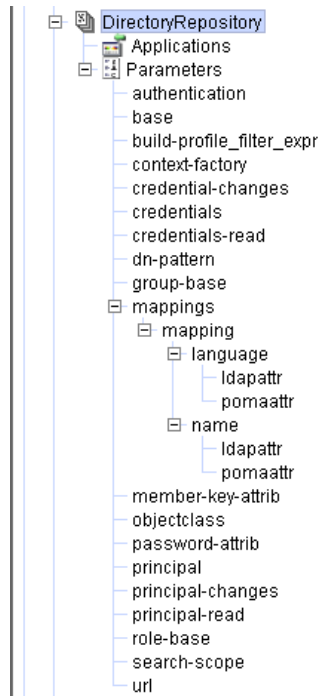


Fig. 22 – The DirectoryRepository with its parameters

## Locating Errors

Locating errors within the methods and classes that derive directly or indirectly from the Base class is called tracing. The tracing behavior can be configured in the **trace.properties** file. For the changes made in the file to take effect, the servers must be restarted. You can also configure the tracing behavior in the Admin client. In this case, it is not necessary to restart the servers.

The file **trace.properties** contains the following entries:

**Entries in the trace.properties file**

<b>Entry</b>	<b>Meaning</b>
<code>output.policy</code>	This entry is used to determine the output type. The following values are possible: Stdout: default output channel Stderr: default error channel Process: one file for all outputs (default) Thread: one file per thread Class: one file per class Thread+Class: one file per thread and class
<code>output.overwrite</code>	This entry is used to overwrite existing log files (value <code>true</code> ) or add the data to existing files (value <code>false</code> ).
<code>output.limit</code>	The maximum size of the output file in KB
<code>file.path</code>	The path to the directory in which the log files are located
<code>file.prefix</code>	The prefix of the file names
<code>file.suffix</code>	The suffix of the file names (usually the file extension)
<code>clearer.delay</code>	The time in seconds between two runs in order to clean the thread table of no longer running threads. If the value is 0, no clean-up thread is started (only for tests in a productive installation, a clean-up thread must always be running).  The time should amount to at least 10 seconds, a duration of 60 to 300 seconds is recommended.

Entry	Meaning
clearer.entries	<p>The minimum number of entries in the thread table, so that the clean-up of this table can be performed at all. This entry is only considered if the clean-up thread is running.</p> <p>There should be 50 to 200 entries in this table.</p>
trace.filter	<p>The created thread objects use the filter value indicated here to decide which trace outputs are written to the logging channel. The filter value is interpreted as a bit field. Each bit stands for a trace information type (for example: entering a method, exiting a method, warning message, error). The value 0 has the effect that no tracing is performed. The filter value can also be indicated as a hex number by using 0x as a prefix (the setting of all bits for bit-oriented filter implementations in the logging channel can be achieved by using the value 0x7F).</p> <p>For detailed information, refer to the comments of the <b>trace.properties</b> file.</p>
trace.indentspaces	<p>The number of spaces that should be output per indentation depth behind the date stamp.</p>
trace.width	<p>The width of the output lines. The output is interrupted at word boundaries. The width must have a value of at least 40. If no word-wrap is to be used, the value 0 should be entered.</p>
trace.timestamp	<p>The type of time stamp at the beginning of each line. One of the four following values is allowed:</p> <p>NO: no time stamp</p> <p>INTERNAL: time stamp in milliseconds</p> <p>SHORT: time stamp hour to millisecond</p> <p>LONG: time stamp year to millisecond</p>

Entry	Meaning
<code>trace.format</code>	<p>The format string for the trace output. The following placeholders are possible:</p> <ul style="list-style-type: none"><li>{0}: the complete name of the class calling the trace</li><li>{1}: the name of the class calling the trace without package name</li><li>{2}: the name of the method from which the trace is called</li><li>{3}: the name of the object that uses the trace</li><li>{4}: the name of the current thread</li><li>{5}: the indentation depth as a number</li><li>{6}: the trace text</li></ul>
<code>trace.acceptclasses</code>	<p>As a rule, the traces are filtered by checking the entry <code>trace.filter</code>. Exceptions exist for certain classes: explicitly named classes have filter values that deviate from the standard filter values. This behavior is very time-consuming and can be (de)activated. The value <code>true</code> activates exception treatment of the appropriate classes; the value <code>false</code> deactivates it.</p> <p>For detailed information, refer to the comments of the <b><code>trace.properties</code></b> file.</p>
<code>class1 and class2</code>	<p>This entry is used to determine for which class(es) tracing is to be enabled. Possible values are</p> <ul style="list-style-type: none"><li><code>com.org.pkg.Class1</code>: tracing for the class</li><li><code>com.org.pkg.*</code>: tracing for the package</li><li><code>com.org.pkg.**</code>: tracing for the package with all sub-packages</li></ul>





---

# CHAPTER 4

## Classes and Interfaces of the Portal Manager API

This chapter describes the classes that are necessary for understanding the concepts of the Portal Manager API.

### Class Diagrams and Class Descriptions

#### Class Diagrams

To give an overview, class diagrams are provided with the individual sections of this chapter. The entire diagram is available as a file in the directory `{WCM installation directory}\documentation\manual\portalManagerClassDiagram.pdf`. The diagrams in the text each represent the relevant sections. In the entire diagram, the classes belonging to one topic are each marked in a specific color.

For the class diagrams, the following scheme – derived from UML diagrams – is used.

- *White boxes* indicate classes or interfaces that belong to other sections, for example to other related topics of the Portal Manager API, or to the Java basic libraries (Java Foundation classes or Java extensions). These classes are not explained in the text. They are included in the diagram as contextual references.

- *Light boxes* symbolize interfaces. The interface name is always written in italics.
- *Dark boxes* represent classes. Only the names of abstract classes are written in italics.

The relations of the classes and interfaces are described by arrows. A filled arrow represents the inheritance of a class (keyword extends), an empty arrow represents the implementation of an interface (keyword implements). Dashed arrows represent associative relations between classes. In this case, the relationship is explicitly stated.

The following figure illustrates the conventions used in this scheme:

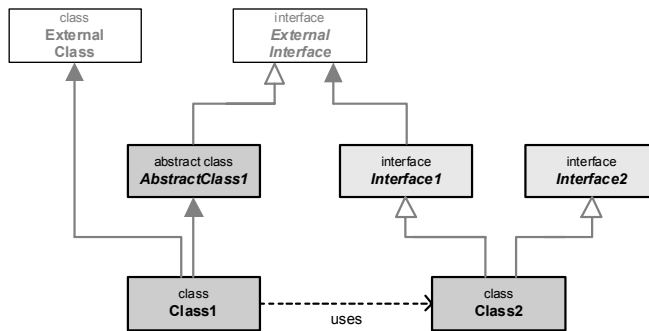


Fig. 23 – Representation scheme for class diagrams



## Class Description

During installation of the WCM system, the complete class documentation of the Portal Manager API is installed as Javadoc pages. The Javadoc documentation is located in the directory **{WCM installation directory}\documentation\javadoc\**. The class descriptions in this manual differ from the Javadoc documentation. The Javadoc on the Portal Manager API offers detailed information on all the classes and interfaces and their associated methods.

## Basic Classes

All classes supplied with the Portal Manager API use a number of basic classes, which can be categorized in four groups:

- basic data types
- filtering and sorting
- RepositoryMap and associated classes
- repositories

These four groups are described in this section. For each group, the respective section of the class diagram is shown.

The basic classes supply the important algorithmic and data-related foundations of the Portal Manager API, including important properties such as comparability, filter capability, and sort capability. The highest level of abstraction is attained by the repositories. Since all basic classes are used directly or indirectly in session beans, knowledge of these classes and the related concepts is essential.

## Basic Data Types

All attribute keys of a WCM object implement the `Key` interface. The attribute values implement the `Value` interface. This interface enhances the `Comparable`, `Cloneable`, and `Serializable` interfaces (as shown in figure 25 “The basic classes in the entire class diagram” on page 100).

Object data can be represented as attribute-value pairs and can thus be managed by the Portal Manager API. For further information on this topic, please refer to the WCM Java API Programmer’s Manual.

### Key Values (Keys)

Since both the class `TupleMap` and the class `RepositoryMap` basically describe the assignment of keys to data values, a `Key` interface was introduced. However, any class can be used as a key in the repository, as long as they implement the `Key` interface.

### Data Values (Values)

The values to be stored must correspond to the `Value` interface. The basic data implementation supplied with the Portal Manager API contains, except in one case, the description “Value” as a suffix in the class name. Classes other than those supplied can also be used here, as long as they consider the `Value` interface.

#### The `Value` interface

Full name: `de.gauss.lang.Value`

The `Value` interface represents a data value. Data values are used in the `TupleMap` class, but not as a key (for this, the `Key` interface is available, which extends the `Value` interface even further). A data value must be comparable, cloneable, and serializable. Thus, the `Value` interface is based on the three corresponding interfaces.

A specific implementation of the Value interface must be mentioned in particular: the TupleMap class. A TupleMap stores pairs of Key and Value. Since it is itself of the type Value, it can appear as a value in another TupleMap. This is how multi-level (recursive) structures are developed. The following figure demonstrates this construction.

Key	Value														
<i>key1</i>	<i>value1</i>														
<i>key2</i>	<i>value2</i>														
<i>key3</i>	<i>value3</i>														
	<table border="1"> <thead> <tr> <th>Key</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td><i>key1</i></td> <td><i>value1</i></td> </tr> <tr> <td><i>key2</i></td> <td><i>value2</i></td> </tr> <tr> <td><i>key3</i></td> <td><i>value3</i></td> </tr> <tr> <td><i>key4</i></td> <td><i>value4</i></td> </tr> <tr> <td>...</td> <td>...</td> </tr> <tr> <td><i>keyn</i></td> <td><i>valuen</i></td> </tr> </tbody> </table>	Key	Value	<i>key1</i>	<i>value1</i>	<i>key2</i>	<i>value2</i>	<i>key3</i>	<i>value3</i>	<i>key4</i>	<i>value4</i>	...	...	<i>keyn</i>	<i>valuen</i>
	Key	Value													
	<i>key1</i>	<i>value1</i>													
	<i>key2</i>	<i>value2</i>													
<i>key3</i>	<i>value3</i>														
<i>key4</i>	<i>value4</i>														
...	...														
<i>keyn</i>	<i>valuen</i>														
<i>key4</i>	<i>value4</i>														
...	...														
<i>keyn</i>	<i>valuen</i>														

Fig. 24 – Multi-level key value lists in the TupleMap

It is also possible that a TupleMap contains itself, even if this is not really a sensible or desirable constellation. In this case, some algorithms in the Portal Manager API classes that operate on a TupleMap, would assume the state of an endless recursion.

## Class Diagram

The following figure shows the relevant section of the entire class diagram.

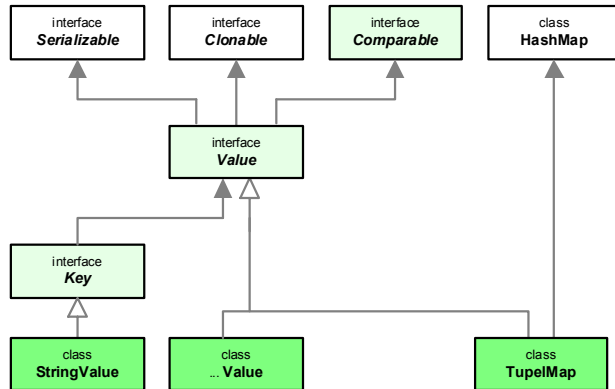


Fig. 25 – The basic classes in the entire class diagram

## Filtering and Sorting

Filtering and sorting data are fundamental functions that are frequently required for processing data.

### Filtering

Filtering is generally described by the `Filter` interface (full name: `de.gauss.vip.lang.filter.Filter`), for which several implementations are available. Complex filters can be combined from simple filters, as in a construction set. For more detailed information on this topic, please refer to the WCM Java API Programmer's Manual.

### Sorting

Sorting is generally described by the `Sort` interface (full name: `de.gauss.lang.Sort`). The `KeySort` class is the only implementation of the `Sort` interface. This class is used to define a sort criterion for the `RepositoryEntry` objects contained in a `RepositoryMap` (see section “`RepositoryMap` and associated classes” starting on page 102). The `KeySort` class contains the appropriate key and sort order. The key determines the sort criterion of the `RepositoryEntry` objects. The `RepositoryEntry` objects are compared on the basis of the value they contain for the specified key.

### Example

A `RepositoryMap` contains three objects of the `RepositoryEntry` class. For the key “A”, the first `RepositoryEntry` contains the value “Benjamin”, the second `RepositoryEntry` contains the value “Charlie”, the third contains the value “Alfred”. When the key “A” is applied, the `KeySort` class sorts the `RepositoryEntry` objects in the following order: third `RepositoryEntry`, first `RepositoryEntry`, second `RepositoryEntry`.

An example of how the `KeySort` class is used can be found in the code in section “Generating Dynamic Navigation Elements” starting on page 142.

### Class Diagram

The following figure shows the relevant section of the entire class diagram.

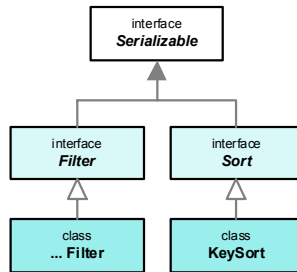


Fig. 26 – The filtering and sorting classes in the entire class diagram

## RepositoryMap and associated classes

The RepositoryMap class is among the most universal of the basic data types in the Portal Manager API. The classes belonging to RepositoryMap must not be confused with the classes that implement the Repository interface. Even though they are closely linked thematically, there is no data inheritance relationship between them.

### Class Diagram

The following figure shows the relevant section of the entire class diagram.

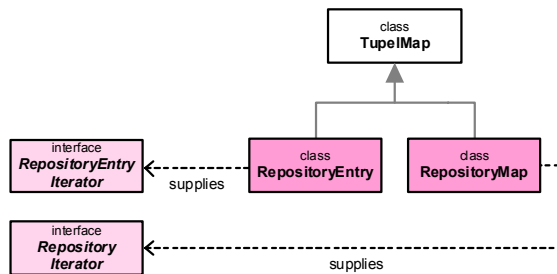


Fig. 27 – The RepositoryMap class in the entire class diagram

The following figure shows the classes that are used in the Portal Manager API to store data. The diagram does not emphasize the data attribute hierarchy, but the structural relationships.

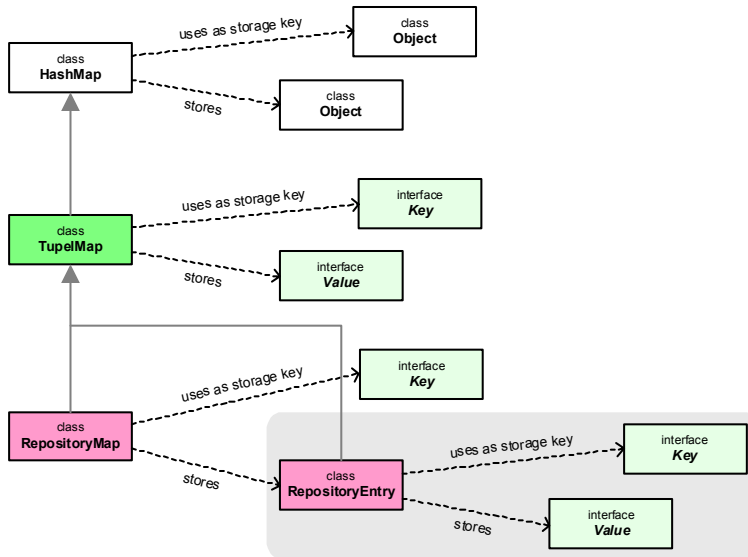


Fig. 28 – Structural relations of complex data types

### The Repository Entry class

Full name: `de.gauss.vip.RepositoryEntry`

The objects of the `RepositoryEntry` class are the values that are stored in a `RepositoryMap`. In principle, a `RepositoryEntry` is also a map, since the class is derived from the `TupleMap` class. Storing an object of this class in a `RepositoryMap` results in a two-level data structure.

An important function of the class `RepositoryEntry` compared with the class `TupleMap` is to supply an interface to a directory service via JNDI. A `RepositoryEntry` can be initialized with JNDI data and can also present itself as JNDI data. The applicable interface is the JNDI `attributes` interface.

### The `RepositoryEntryIterator` interface

Full name: `de.gauss.vip.repository.RepositoryEntryIterator`

A `RepositoryEntryIterator` is a listing of the key-value pairs in a `RepositoryEntry`. By definition, a new iterator is positioned *before* the first pair. In order to access the pairs, the iterator must be incremented with one of the `next` methods for the first pair and the following pairs.

### The `RepositoryMap` class

Full name: `de.gauss.vip.repository.RepositoryMap`

The class `RepositoryMap` extends the class `TupleMap`, especially by the possibility of filtering, sorting, and listing data. However, instead of storing values of the type `Value`, values of the type `RepositoryEntry` are stored. Since a `RepositoryEntry` is also derived from `TupleMap`, multi-level lists are inevitably developed. This is represented in the following figure.



Key	RepositoryEntry														
key1	entry1														
key2	entry2														
key3	entry3														
	<table border="1"> <thead> <tr> <th>Key</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>key1</td> <td>value1</td> </tr> <tr> <td>key2</td> <td>value2</td> </tr> <tr> <td>key3</td> <td>value3</td> </tr> <tr> <td>key4</td> <td>value4</td> </tr> <tr> <td>...</td> <td>...</td> </tr> <tr> <td>keyn</td> <td>valuen</td> </tr> </tbody> </table>	Key	Value	key1	value1	key2	value2	key3	value3	key4	value4	...	...	keyn	valuen
	Key	Value													
	key1	value1													
	key2	value2													
	key3	value3													
key4	value4														
...	...														
keyn	valuen														
key4	entry4														
...	...														
keyn	entryn														

Fig. 29 – Multi-level key value lists in the repository map

To filter, sort, and list the data of a `RepositoryMap`, there is an iterator (`RepositoryIterator`), which can be used to run through previously filtered or sorted data.

### The `RepositoryIterator` interface

Full name: `de.gauss.vip.repository.RepositoryIterator`

A `RepositoryIterator` is an enumeration of the key-value pairs in a `RepositoryMap`, in which the data values are always of type `RepositoryEntry`. By definition, a new iterator is positioned *before* the first pair. In order to access the pairs, the iterator must be incremented with one of the next methods for the first pair and the following pairs.

## Repositories

### Access to External Resources

Each access to external resources (*application data*) requires, on the one hand, the definition of the resources (such as host name, port number, program name) and, on the other hand, the authorization information for access (user name and password).

In the Portal Manager API, the application data is modeled by repositories. The initialization data for defining the resource (URL, user ID, password, and several configuration parameters) is supplied to a repository as parameters. The expected entries are different for each application.

### Options for Authorization

If an application requires an authorization, two different ways can be chosen: authorization data can either be *directly* supplied or *centrally* managed.

As previously described, authorization data can be directly supplied in the parameters. The other possibility is to perform the management centrally, by means of account information (login information) that is stored in the Portal Manager API. The account information is central and can – in case there are more repositories – be used by all repositories. This creates a single sign-on mechanism.

Systems that store account information can also be interpreted as applications. Therefore, it is obvious to make the repositories accessible. This is realized in the Portal Manager API.

## Repository Types

Repositories are used for two purposes:

- representation of an application or the data of an application (primary usage in the Portal Manager API)
- retrieving and checking user-specific account information (login repositories)

In the special case that just the account information represents the data of an application, a repository implementation can, of course, be used for both purposes.

### Login Repositories

Login repositories are repositories used for processing account information. The Portal Manager API has two login repositories that support the following data sources:

- account information in an LDAP server (`DirectoryRepository`)
- account information in the Admin client (`VipUserRepository`); the Administration server can also access the LDAP server for this purpose.

Each login repository uses the user name as a key. The account information for the respective users serves as data value. These are filed in a `RepositoryEntry`. In this respect, the login repositories are equal to the “normal” repositories.

After successful authentication of a user, the user's password is saved. Thus, other repositories can use the password again without the user needing to reenter it (single sign-on support).

### Class Hierarchy

A repository is represented by the `Repository` interface. The `Repository` interface basically contains access methods that suggest (but do not force) an implementation by means of a `RepositoryMap`. Some implementations of the `Repository` interface actually store the repository data as `RepositoryMap`, others forward the method calls to the underlying external data source.

The `Repository` interface is implemented by the abstract basic class `RepositoryImpl`. This class is the basis for all `Repository` implementations supplied with the Portal Manager API.

#### The `RepositoryImpl` class

Full name:

```
de.gauss.vip.portalmanager.repository.RepositoryImpl
```

This `Repository` implementation is the abstract basic implementation of a repository for all repository classes supplied with Livelink WCM Server.

A login repository is modeled as a `LoginRepository` interface. It extends the `Repository` interface by methods for checking the login.

### Sessions

For each access to repository contents, a session must be indicated. The information contained in the session can be used by the repository.

A session describes the current processing context between the browser and web server. In addition, the session contains an ID. With this ID, the name of the user who is logged in to this session is available. This user name should be used to query account information from a login repository.

The interface `Session` is represented in connection with the session-related classes (see figure 32 “The bean classes in the entire class diagram” on page 115).

## Class Diagram

The following figure shows the relevant section of the entire class diagram.

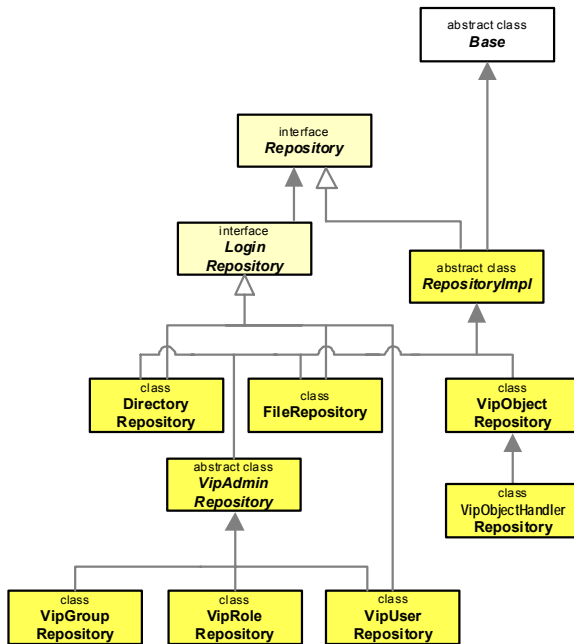


Fig. 30 – The repository classes in the entire class diagram

## Application and Bean Classes

### Basic Class

Bean classes, application classes, and repository classes are derived from the basic class `Base`. This class implements several interfaces and extends an external `CoreObject` class.

#### The Base class

Full name: `de.gauss.vip.portalmanager.core.Base`

`Base` is an abstract basic class of all applications and repositories. In addition, the `Base` class contains the methods for loading properties and multi-properties files.

`Base` also contains the static method `startPMServer`, which you can use to start Content servers running in the context of a JSP engine or as web application in an application server. If these Content servers have already been started, a call of the method remains without effect. The following parameters must be set as system properties.

**Table 5 – Start parameters for Content servers running in the context of a JSP engine**

Parameter	Meaning
<code>vip.server.name</code>	The name of the Content server running in the context of a JSP engine or as web application in an application server
<code>vip.admin.host</code>	The name of the computer on which the master Admin server is running
<code>vip.admin.socket</code>	The socket port of the master Admin server
<code>vip.admin.http</code>	The HTTP port of the master Admin server

<b>Parameter</b>	<b>Meaning</b>
<code>vip.admin.secure</code>	Indicates whether the communication with the master Admin server is to be encrypted. Possible values are: true – encrypted communication false – non-encrypted communication
<code>vip.installdir</code>	The installation directory of the WCM system

## **Applications**

Programs are described by the `Application` interface. The `Application` interface is implemented by the abstract basic class `ApplicationImpl`. On this basis, two concrete implementations of the `Application` interface are provided for the integration of Livelink WCM Server. However, these do not contain any further methods. New classes that implement the `Application` interface for other external programs should also be derived from the abstract basic class.

### **Class diagram**

The following figure shows the relevant section of the entire class diagram.

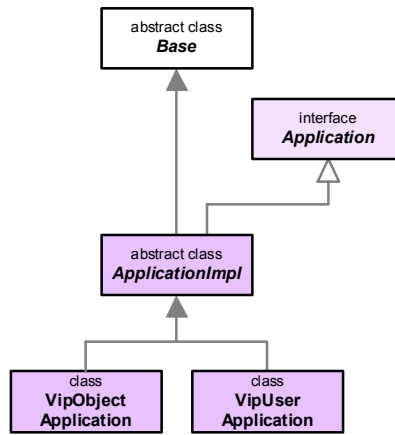


Fig. 31 – The application classes in the entire class diagram

### The Application interface

Full name:

`de.gauss.vip.portalmanager.application.Application`

The Application interface represents a program in the Portal Manager API. This interface determines, in particular, that a program is based on one or more repositories that describe the status and data of a program.

Each Application implementation must support at least the following properties:

- An Application should, as far as possible, fulfill the factory design pattern: each Application class checks which instances exist.
- An Application can have any number of resources that are used for configuration (for example, for setting the language). The purpose of a resource depends on the concrete Application. Resources are entered as strings that describe resource bundle files in the file system or in JAR files identified



via the class path. Resources are loaded via the standard class loader.

- An `Application` uses any number of repositories, which differ by name.
- An `Application` has a default locale that can be used for localization and internationalization purposes.

The listed properties of each `Application` implementation are realized as methods in the interface declaration. However, there are only methods to determine the property values (get methods), the definition of the properties (by set methods) is not required by an `Application`. Usually, the properties are set implicitly by conventions or by properties that are entered at initialization by the `open` method.

### The `ApplicationImpl` class

Full name:

`de.gauss.vip.portalmanager.application.ApplicationImpl`

An `ApplicationImpl` is an abstract implementation of the `Application` interface on which basis concrete `Application` classes can be developed without great effort. `ApplicationImpl` does not have a public constructor. Beyond the methods defined in `Application`, only the method `getInstance` exists as a surrogate for a constructor, whereby the factory pattern is implemented.

At the first call of the method `getInstance(String)`, `ApplicationImpl` automatically starts the Content server running in the context of a JSP engine or as web application in an application server. If you want to start the Content server explicitly, use the method `startPMServer` of the Base class.

The class `ApplicationImpl` uses the following conventions for its own initialization:

- The settings of the application are read from the configuration file **application.xml**. The configuration file **server.xml** contains a mapping of the logical application names to existing application names.
- Each repository, which needs an `ApplicationImpl`, is read through the settings of the configuration file **repository.xml**.

These configuration files contain all information that allows creating `Application` objects as factory. In this way, supplying concrete `Application` implementations with initialization data is considerably facilitated.

## Beans

The beans made available in the Portal Manager API can be used as instances in JSP scripts. They make it possible to access any program through HTML pages. As scope for the beans, a “session scope” should be set. Beans use an application, which represents the actual program. For this reason, the highest-level bean class is called `SessionApplication`.

**Note:** All beans of the Portal Manager API were developed for use with session scope. When using other scopes – especially “request” and “page” – there is the danger of using up many CA licenses (CA = concurrent author), which can only be used again after the respective context IDs in Livelink WCM Server have become invalid.

Beans contain high-level methods based on the `Application` and `Repository` classes. These methods summarize typical use cases of a program in an easy-to-understand and efficient manner.

The abstract implementation in `SessionApplication` provides, in particular, simple facilities for internationalizing programs. In this way, users are offered a program with country-specific standards and language. This is only possible if localization information on the respective user is available.

### Class diagram

The following figure shows the relevant section of the entire class diagram.

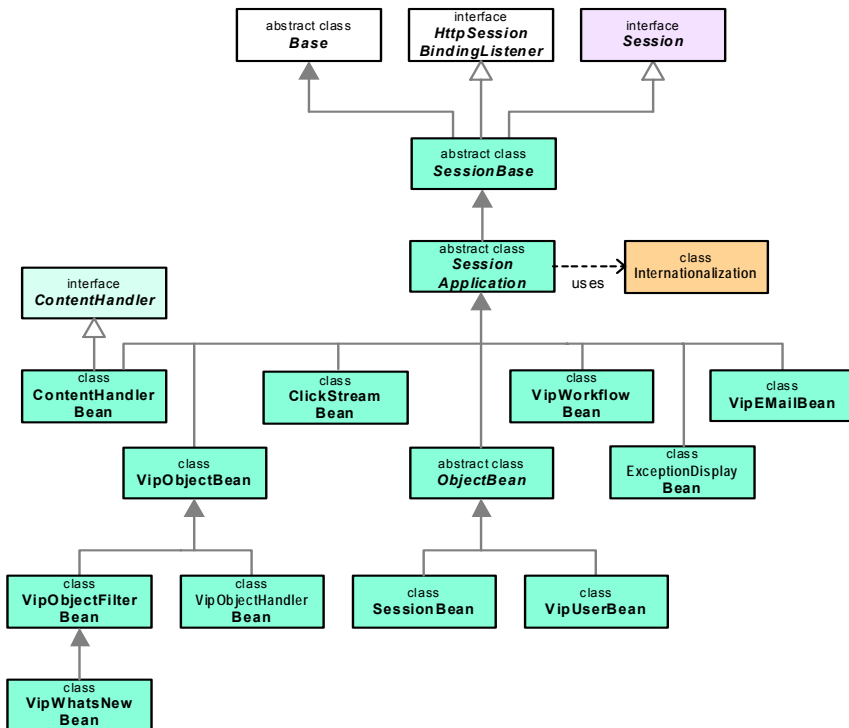


Fig. 32 – The bean classes in the entire class diagram

### The `SessionApplication` class

Full name:

`de.gauss.vip.portalmanager.application.SessionApplication`

The `SessionApplication` class represents the abstract basic class for all beans in the Portal Manager API. The class provides the following functionality:

- assignment of the bean to a session
- method for retrieving the assigned application
- localization information
- methods for receiving and setting the repository name

The default locale and the resources for the respective application are retrieved from the configuration file **application.xml**.

### The `Internationalization` class

Full name: `de.gauss.vip.i18n.Internationalization`

The `Internationalization` class is responsible for internationalizing messages. Internationalization is controlled by a `Locale` object.

Resource bundles are the basis for internationalization. An `Internationalization` object can use any number of resource bundles on the basis of any `Locale` object. Access to resources is performed, as usual, using keys. The selection of the `Locale` objects is controlled by the current `Locale` of the session. A key is checked for each resource bundle that is possibly associated with the `Locale`, until one of the resource bundles returns a value. If resource bundles have the same key, the resource bundle which is entered first in the file **application.xml** is used.

The `Internationalization` class makes methods available for

- retrieving localized messages for the key
- retrieving Locale-specific date and number formats

### The `SessionBean` class

Full name: `de.gauss.vip.portalmanager.SessionBean`

A `SessionBean` manages user profiles on the basis of the unique session ID. (The term session bean refers to a bean that realizes a session-related user administration. However, it should not be confused with the session beans that are part of the Enterprise JavaBeans.) A session ID can either be specified as a string or as an HTTP request.

Before a user has been authenticated by a login process, the user will be treated as an anonymous user with very limited rights. Users are not automatically logged in by the `SessionBean` class. It is possible to log out a user explicitly.

### The `VipProfile` interface

Full name: `de.gauss.vip.portalmanager.VipProfile`

You can use the interface `VipProfile` to store and manage additional information on users. The Content client uses `VipProfile` to permanently store a user's configuration information. `VipProfile` also offers the possibility to store and manage non-WCM information such as a user's address.

To retrieve a user's `VipProfile`, call the method `SessionBean.getCurrentUserProfile(String)` from the `SessionBean`. The user ID must be supplied with the call. Within `VipProfile`, user information is managed and (if necessary) stored as key-value pairs. That means that a `Value` can be stored with a certain key.

VipProfile provides three sections for user information:

- **transient profile information** (VipProfile.TRANSIENT)

This section is used to store profile information which must be available during a user session, but which is not to be stored persistently. Transient profile data from VipProfile can be read and manipulated even if the corresponding user is not logged in to the WCM system.

- **client-related profile information** (VipProfile.KNOWN\_USER)

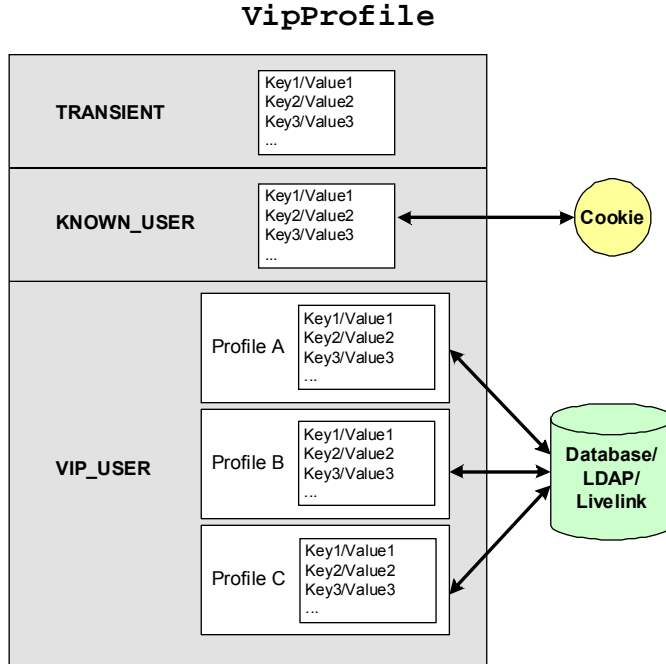
This section is used to manage and store user information which is to be stored persistently. The data is saved in a cookie. Thus, the data in this section is only available on the client (browser) which stores the cookie. By default, the life span of the cookie is limited to 90 days. You can, however, change this default by means of the KNOWNUSER\_COOKIE\_DURATION parameter of the VipUserApplication class.

- **user-specific profile information** (Vip.Profile.VIP\_USER)

This section is used to persistently store and manage user information which is to be available in the whole system. Depending on the kind of user administration, this profile information is stored in a database, an LDAP directory service, or in Livelink. To read and manipulate data from this section, the corresponding user must be logged in to the WCM system.

Within this section, the user information must be stored and managed in sub-profiles which must have unique names. The methods `VipProfile.getProfileValue(String profileName, String attribute)` and `VipProfile.setProfileValue(String profileName, String attribute, Value value)` allow you to access and manipulate the information in this section. A profile name must be specified for this purpose. Alternately, you can use the method

`VipProfile.setCurrentProfileName(String name)` to set the profile name which is to be used for the `VIP_USER` section. To retrieve the name of the profile currently used for the user-specific profile information, you can use the method `VipProfile.getCurrentProfileName`.



**Fig. 33 – Structure of VipProfile and the managed information**

The methods `VipProfile.getValue(String attribute, int type)` and `VipProfile.setValue(String attribute, Value value, int type)` are provided for accessing and manipulating individual profile values. This applies to all `VipProfile` sections. The section to be used must be specified as type (`TRANSIENT`, `KNOWN_USER`, or `VIP_USER`).

### The VipUserBean class

Full name: `de.gauss.vip.portalmanager.VipUserBean`

The logical name of the underlying application of this class is `VipUserApplication`. The `VipUserBean` class is an interface to the user administration of Livelink WCM Server. It offers methods for storing and loading the `Value` objects of different repositories, as well as convenience methods for dealing with bookmark entries. The `VipUserBean` class continues to offer methods that make it possible to represent an individual user. These methods have been deprecated. Instead, the class `VipProfile` should be used. In order to get a reference to an object of the class `VipProfile`, use the class `SessionBean`.

### The VipObjectBean class

Full name: `de.gauss.vip.portalmanager.VipObjectBean`

The `VipObjectBean` class provides (read-only) access to all objects of the websites managed with Livelink WCM Server. In particular, there are several methods for retrieving the website's hierarchical structure, which is determined by topic objects. This can, for example, be used to construct navigation.

The `VipObjectBean` uses repositories of the class `VipObjectRepository` to query data from the WCM system. These repositories can be configured in two ways:

- by specifying the name of a deployment system

In this case, each repository represents the data of a deployment system. (For each deployment system, a `VipObjectRepository` or `VipObjectHandlerRepository` with the same name is created automatically; however, this name can also be chosen in such a way that the name of the repository differs from the name of the deployment system.) A deployment system is responsible for exactly one website and one data storage view (Edit, QA,



Production). By choosing a `VipObjectRepository`, a specific website and a specific view are always specified at the same time. The logical name of the underlying application is `VipUserApplication`.

- by specifying the website name and the data storage view of the website (Edit, QA, Production)

This makes it possible to access the objects of a website without setting up deployment systems. Repositories configured in such a way cannot return deployment-specific object data (e.g. URL, path).

**Note:** For further information on the configuration of a `VipObjectRepository`, refer to table 3 “Repositories and their parameters” on page 57.

Individual WCM objects are entries in these repositories. All metadata of the WCM objects are keys of these entries. The descriptors of this metadata and the values of some metadata are described in the WCM Java API Programmer’s Manual or in the Javadoc on the Portal Manager API.

### The `VipObjectHandlerBean` class

Full name: `de.gauss.vip.portalmanager.VipObjectHandlerBean`

The `VipObjectHandlerBean` extends the `VipObjectBean` by the write access to the website objects managed with Livelink WCM Server. The `VipObjectHandlerBean` is the interface of the Portal Manager API to the WCM Java API.

The available methods are almost identical to the methods of the `ObjectHandler` interface of the WCM Java API. The `ObjectHandler` is the central instance of the WCM Java API for manipulating a website via the program. All techniques, limitations, constants, etc. described in the WCM Java API Programmer's Manual are therefore also valid for the `VipObjectHandlerBean`. Therefore, knowledge of the `ObjectHandler` methods is essential.

The name of the underlying application of this bean is `VipObjectApplication`. The repositories that use this application are entered in the configuration files **`application.xml`** and **`repository.xml`**.

These repositories can be configured in two ways:

- by specifying the name of a deployment system

A deployment system is responsible for exactly one website and one data storage view (Edit, QA, Production). Therefore, by selecting a deployment system for a repository, you specify exactly which data will be returned and which objects are affected. With the method `setRepositoryName(String)`, you can specify which `VipObjectHandlerRepository` is to be used by this bean to carry out operations in the WCM system.

- by specifying the website name and the data storage view of the website (Edit, QA, Production)

This makes it possible to access the objects of a website without setting up deployment systems. Repositories configured in such a way cannot return deployment-specific object data (e.g. URL, path).

The data storage view, which is assigned directly to the `VipObjectHandlerRepository` or indirectly via the deployment system, determines which objects are visible. The data storage view also specifies which actions can be performed. For a detailed description of which actions can be performed in which view, please refer to the description of the `ObjectHandler` class in the WCM Java API Programmer's Manual.

The most methods of the `VipObjectHandlerBean` can only be used if the user is logged in. The logged-in user must have sufficient rights to carry out the actions. Each user who performs actions on the WCM objects uses up one license. This license can be used again when the user logs off or the user login becomes invalid in the server.

**Note:** Every user with write access to the WCM system uses up a license. This license is released after one minute, even if the user logs out before this minute has elapsed. When the minute has elapsed, the license will be released together with the user logout.

### The `VipObjectFilterBean` class.

Full name: `de.gauss.vip.portalmanager.VipObjectFilterBean`

The class `VipObjectFilterBean` makes it possible to filter the WCM objects of a website according to specific criteria. All methods inherited from the super class `VipObjectBean` that return certain object sets are additionally restricted according to the filter conditions defined here.

Filtering results are stored temporarily in the `VipObjectFilterBean` in a cache. The cache is filled using the method call `applyFilter`. Subsequent filtering with identical filter conditions is not actually performed, but is read from the cache. The cache can be deleted explicitly (method `invalidateCache`) to force a new filtering.

### The VipWhatsNewBean class

Full name: `de.gauss.vip.portalmanager.VipWhatsNewBean`

The VipWhatsNewBean extends the filter function of the VipObjectFilterBean by the “What's New” feature. “What's New” returns pages (WCM objects) that have changed recently or are new. The release date is compared to a specific date. This date determines how “changed recently” and “new” is defined.

The “What's New” feature can be extended by the following properties:

- The retrieved set of changed or new WCM objects can be sorted.
- Individual WCM objects can be excluded explicitly.
- Object types can be excluded explicitly.
- File extensions can be excluded explicitly.

The following properties are inherited from the super class VipObjectFilterBean:

- In addition to the “What's New” condition, another filter condition can be entered for restriction purposes.
- Only WCM objects starting from a specific root object can be considered.
- The maximum number of WCM objects can be specified.

It is possible to enter a sorting sequence for the retrieved objects. Without an explicit entry, the WCM objects are sorted by release date. The WCM objects with the most recent release dates are at the top of the hitlist.

### **The ContentHandler interface**

Full name: `de.gauss.vip.portalmanager.ContentHandler`

The `ContentHandler` interface allows access to the content of an object that is specified by a URL.

The content can be read as a string, stream, or reader. In all cases, a converter can be specified (for example: `XML ContentConverter`), which converts the content. Converters interpret the content and only return the relevant parts.



---

# CHAPTER 5

## Notes on Programming with the Portal Manager API

This chapter gives you a brief introduction to the programming interfaces of the Portal Manager API. It also provides some application examples.

### Authentication and Session Management

To be able to use the functionality of the Portal Manager API, a proper authentication as a WCM user is necessary due to the system's access and security mechanism. The HTTP protocol used for communication between user and the Portal Manager API is stateless. This means that authentication is necessary for each HTTP request. This problem is remedied by the JSP specification, which provides a mechanism for HTTP sessions.

An HTTP session has the following features:

- Initialization – an HTTP session is initialized when the connection between a client and a web server is established for the first time.
- Session ID – a session that takes place in the context of a JSP engine is identified by a session ID that is unique for the individual installation. The session ID is managed by the JSP engine (server).

- HTTP session timeout – an HTTP session is only valid if the client sends a request within a certain period of time. Otherwise, a timeout occurs in the JSP engine.

The HTTP session mechanism is used by the Portal Manager API for session management of the WCM JavaBeans and at the same time guarantees WCM authentication for a specific session.

## Authentication

Authentication in the Portal Manager API is based on the concept of the WCM user, who is characterized by the following attributes:

- unique user name (for a specific installation or for the LDAP server that is used for user administration)
- password

The user information for authorization in the Portal Manager API is stored in a `LoginRepository` implementation. There are various `LoginRepository` implementations, e.g. `VipUserRepository`. Depending on the use of the Portal Manager API, there are various authentication modes.

- **Authentication via HTTP request parameters**

The WCM user parameters are defined using the HTTP request parameters `userName` and `userPassword` in the HTTP request of the JSP engine. In this mode, the authentication parameters are made available by using an HTML form that supplies the required parameters.

- **Authentication via HTTP standard authentication**

The WCM user parameters are provided via the authentication mechanism of the (web) server. The *Authorization* field in the header of the HTTP request contains the user name and the password, which are separated by a colon and Base 64-coded.



- **Context ID login from request or cookie**

The context ID is read and the user is logged in. This ID is assigned by Secure Access or written to the request during statification.

- **Trusted login (authentication via the session attribute)**

For enabling the Portal Manager API to work with other e-business applications and to support single sign-on, it is possible to request the WCM user's name from an attribute that is associated with the HTTP session. The trusted login mode is a function of the LoginRepository interface. The following figure shows how to configure trusted login in the Admin client.

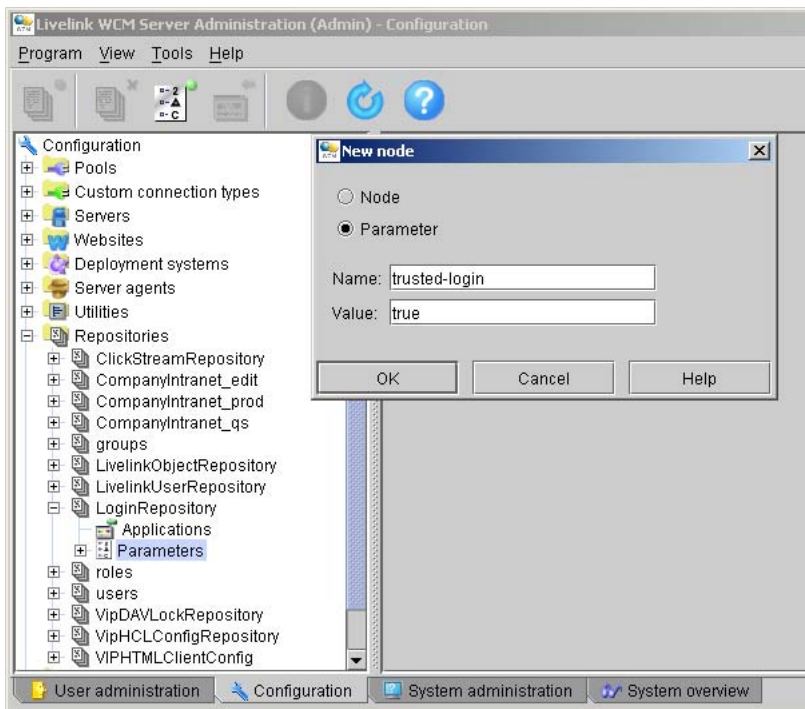


Fig. 34 – Configuring the trusted login mode via the Admin client

### ■ Trusted login in the WCM system

(`checkLogin(request, "uid", null)`), in the user's settings in the Admin client, the option *Trusted login* must be activated)

If the user does not log in to the Portal Manager API, the user is treated as an anonymous user with minimum rights. If the trusted login mode is used for user authentication, it is not possible to use the class `DirectoryRepository`, as this repository demands a password that is not available in this case.

If the WCM system is to be accessed (including write access) or the requested objects cannot be read by all users, the user must log in using the `VipUserRepository`. For this login, the user is assigned a context ID which the WCM system requires for checking the rights.

Each user writing uses one CA license (CA = concurrent author), which will be returned when the user logs out or the used context ID times out.

**Note:** The license used is released after one minute, even if the user logs out before this minute has elapsed. When the minute has elapsed, the license will be released together with the user logout.

## Session Management

The following class diagram shows the HTTP session management in connection with authentication.

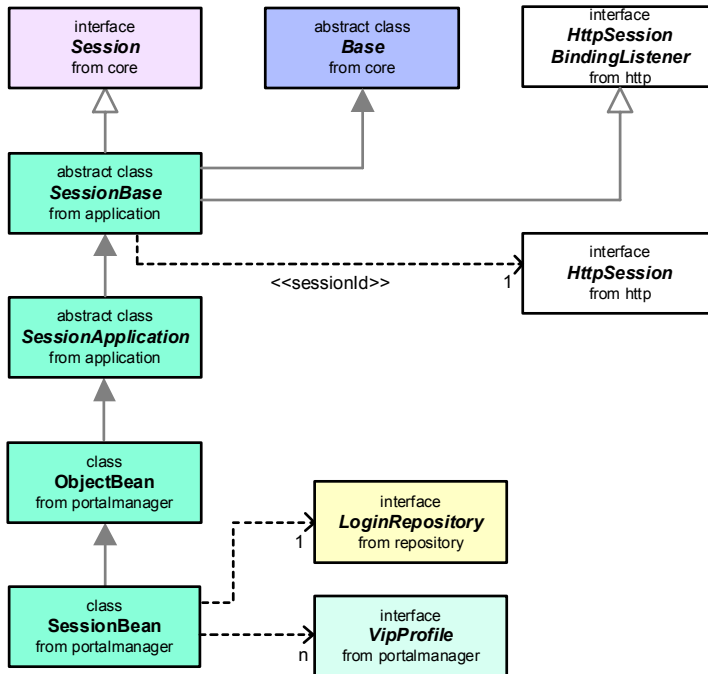


Fig. 35 – Authentication and session management

The following main components are used for session management and authentication:

- HttpSession – makes it possible to retrieve a unique session ID for an HTTP request
- HttpSessionBindingListener – informs the SessionBase class when an HTTP session has ended or started
- LoginRepository – contains the user information for checking the login request

- `SessionBean` – contains all session information in the Portal Manager API
- `VipProfile` – contains all information about the WCM user associated with an HTTP session

## Login Repositories

The following diagram shows the repositories that can be used for a login.

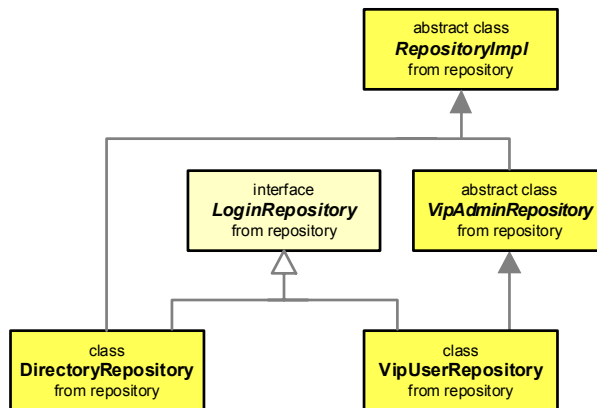


Fig. 36 – Login Repositories

Sometimes it is necessary to write your own login repositories. For example, if you have a database with all the user information and want to use the data source for authentication purposes, you must develop a login repository that accesses the database. This class must implement the `LoginRepository` interface and must be derived from the abstract basic class `RepositoryImpl` or from one of its subclasses. For further information, see section “Deriving from the `RepositoryImpl` class” on page 154. The default login repository is the users repository.

## Application Examples for the Portal Manager API

This section contains application examples of programming with the Portal Manager API for the following topics:

- creating users via a portal (see the following section)
- creating dynamic navigation elements (see section “Generating Dynamic Navigation Elements” on page 142)
- multi-content pages (see section “Placing Several Content Objects on one Page (Multi-Content Page)” on page 146)
- form-based creation of objects (see section “Creating WCM Objects via an HTML Form” on page 149)

### Creating Users in an LDAP System via a Portal

Internet portals usually require users to specify their identity. For this purpose, the users create a user accounts with their personal data in the portal. By means of this account, users can tailor the layout of the portal to their wishes and needs. Moreover, it is possible to control which content provided by the portal the users may access.

The user data can be stored in a database or an LDAP directory service. This section describes how to create new user accounts in an LDAP directory service via a portal by means of the `DirectoryRepository` of the Portal Manager API. The required data can be taken from login forms. This way, portal users can create and manage their accounts themselves.

It is also possible to develop JSP applications for managing user accounts. Such web applications potentially offer more functionality than the Admin client because any data can be stored and managed in them.

### Requirements

- To enable a user whose account is created in the LDAP directory service to log in to the WCM system and thus become part of the access control, the WCM user administration must be based on LDAP. The newly created user accounts must be available from the Administration server (master or proxy) that performs the login procedure for the Content server running in the context of the JSP engine or as web application in an application server.
- The `DirectoryRepository` can use all LDAP directory services that are supported by Livelink WCM Server (for detailed information on the supported LDAP directory services, refer to the Release Notes for Livelink WCM Server).

### Configuring the `DirectoryRepository`

For accessing an LDAP directory service, a repository of the type `DirectoryRepository` must be configured. Proceed as follows:

1. In the Admin client, create a new repository (see section “Adding a Repository” on page 54).

As class name, enter `de.gauss.vip.portalmanager.repository.DirectoryRepository`.

2. Configure the new repository by means of parameters (see section “Setting the Parameters for a Repository” on page 55).

The following table contains a sample configuration. For basic information on the `DirectoryRepository`, refer to section “The `DirectoryRepository` class” on page 87. The individual parameters are described in table 3 “Repositories and their parameters” on page 57.

**Notes:**

If necessary, the values of the `ldapattr` parameters must be modified according to the configuration of the LDAP directory service.

Mappings must only be made for those parameters that have different values for the parameters `pomaattr` and `ldapattr`. A distinction is made between upper and lower cases, i.e. the values “userpassword” and “userPassword” are **not** identical.

**Table 6 – Sample configuration of a DirectoryRepository**

Parameter	Value
<code>url</code>	<code>ldap://ldapsrvr.company.example:389</code>
<code>authentication</code>	<code>simple</code>
<code>principal</code>	{LDAP user with the right to add entries}
<code>credentials</code>	{password of the user entered under <code>principal</code> }
<code>principal-changes</code>	{LDAP user with the right to change and delete entries} <b>Note:</b> This parameter is used to assign users to groups and roles. This also applies if the logged-in user does not have the right to change and delete LDAP entries.
<code>credentials-changes</code>	{password of the user entered under <code>principal-changes</code> }
<code>search-scope</code>	<code>SUBTREE_SCOPE</code>
<code>base</code>	<code>ou=software, o=company.example</code>
<code>group-base</code>	<code>ou=groups, ou=software, o=company.example</code>

Parameter	Value
role-base	ou=roles, ou=software, o=company.example
dn-pattern	uid=*, ou=software, o=company.example
password-attr	userPassword
<input type="checkbox"/> mappings	
<input type="checkbox"/> mapping	
<input type="checkbox"/> LDAP_USER_ID	
pomaattr	uid
ldapattr	uid
<input type="checkbox"/> CN	
pomaattr	cn
ldapattr	cn
<input type="checkbox"/> INIT_PASSWORD	
pomaattr	pwdChange
ldapattr	initPassword
<input type="checkbox"/> VIP_ACCESS	
pomaattr	account
ldapattr	vipAccess
<input type="checkbox"/> LDAP_OBJECTCLASS	
pomaattr	objectclass
ldapattr	objectClass



Parameter	Value
<input type="checkbox"/> TRUSTED_LOGIN	
pomaattr	trustedLogin
ldapattr	trustedLogin
<input type="checkbox"/> VIP_FUNCAREAS	
pomaattr	vipFuncarea
ldapattr	vipFuncarea
<input type="checkbox"/> VIP_RIGHTS	
pomaattr	vipRights
ldapattr	vipRights
<input type="checkbox"/> LANGUAGE	
pomaattr	language
ldapattr	preferredLocale
<input type="checkbox"/> LDAP_VIP_TYPE	
pomaattr	vipType
ldapattr	vipType
<input type="checkbox"/> VIP_WEBSITES	
pomaattr	website
ldapattr	vipWebsite
<input type="checkbox"/> HCL_PROFILES	
pomaattr	hclProfiles
ldapattr	hclProfiles

Parameter	Value
<input type="checkbox"/> MAIL	
pomaattr	email
ldapattr	mail
<input type="checkbox"/> VIP_MEMBERS	
pomaattr	user-name
ldapattr	member
<input type="checkbox"/> VIP_SUBSTITUTE	
pomaattr	vipSubstitute
ldapattr	vipSubstitute
<input type="checkbox"/> USER_PASSWORD	
pomaattr	userpassword
ldapattr	userPassword

3. Assign the new repository to the application `VipUserApplication` (see section “Assigning a Repository to an Application” on page 81).

### Methods for using the `DirectoryRepository`

For accessing the data of the LDAP directory service and changing them, the following methods are available. The class `VipUserBean` inherits them from the class `ObjectBean`:

- `containsKey("<repositoryName>", key)`
- `createEntry("<repositoryName>", key, data)`
- `loadEntry("<repositoryName>", key)`

- `removeEntry("<repositoryName>", key)`
- `storeEntry("<repositoryName>", key, data)`

The parameter `key` of these methods must be a `String` or a `StringValue`. The value of the parameter must be the fully qualified “distinguished name” (DN), the name of a user, group, or role. If no DN has been specified as parameter `key`, the `DirectoryRepository` searches for a principal in the following order:

1. for a user whose DN contains the value of the parameter `dn-pattern` of the repository.
2. for a group with the retrieved DN `cn={key}, {group-base}`
3. for a role with the retrieved DN `cn={key}, {role-base}`

The parameter `data` of the methods and the return values of the most methods are `RepositoryEntry` objects. These `RepositoryEntry` objects contain the attributes of the LDAP entry referenced by `key`. According to the mapping specified in the configuration of the repository, these attributes are converted when they are written in the LDAP directory service or read from there.

The methods `containsKey`, `createEntry`, and `loadEntry` are executed in the context of the repository binding profile, which is specified by the parameters `principal` and `credential`. Thus, these method can be executed independently of the currently logged-in user. As a precondition, the binding profile user must have sufficient rights in the LDAP directory service.

The methods `removeEntry` and `storeEntry` are executed in the context of the currently logged-in user provided that the parameters `principal-changes` and `credential-changes` have been configured for the repository.

The method `createNewUser("{repository name}", key, data)` of the class `VipUserBean` makes it possible to create new users and assign them to groups and roles in just one step. After a user has been assigned to a group or role, the data of the group or role must be updated. For this reason, either the currently logged-in user must have sufficient rights for this operation or the parameters `principal-changes` and `credential-changes` must be used together.

### Example of creating a new user account

In this JSP example, the entries from an HTML form are used to create a new user account.

---

```
<html>
<head></head>
<body>
<%@ page
    import="java.util.*,
           de.gauss.lang.*,
           de.gauss.vip.util.*,
           de.gauss.vip.repository.*"
%>
<jsp:useBean id="vub"
             class="de.gauss.vip.portalmanager.VipUserBean"
             scope="session" />
<%
    // read the parameters from the request
    String name = request.getParameter("name");
    if (name == null) name = "";
    String uid = request.getParameter("uid");
    if (uid == null) uid = "";
    String mail = request.getParameter("mail");
    if (mail == null) mail = "";
    String pwd = request.getParameter("pwd");
    if (pwd == null) pwd = "";
%>
<p><form action="{VIPURL}">
    <table>
        <tr><td>Name</td><td><input name="name"
            value="{%=name%}"></td></tr>
        <tr><td>User-Id</td><td><input name="uid"
            value="{%=uid%}"></td></tr>
        <tr><td>E-Mail</td><td><input name="mail"
            value="{%=mail%}"></td></tr>
```

```
<tr><td>Password</td><td><input name="pwd" type="password"
      value="<%=pwd%>"></td></tr>
<tr><td>&nbsp;</td><td><input type="submit"
      value="create user"></td></tr>
</table>
</form></p>
<hr>
<%
    if ((name.length() > 0) && (uid.length() > 0) && (pwd.length()
        > 0))
    {
        RepositoryEntry newUserEntry = new RepositoryEntry();
        // add the common name
        newUserEntry.putValue(AttributeName.CN,
                               new StringValue(name));

        // add the user id
        newUserEntry.putValue(AttributeName.UID,
                               new StringValue(uid));

        // add the mail address
        newUserEntry.putValue(AttributeName.MAIL,
                               new StringValue(mail));

        // add the password
        newUserEntry.putValue(AttributeName.PWD,
                               new StringValue(pwd));

        // allow WCM access
        newUserEntry.putValue(AttributeName.ACCOUNT, new
                               BooleanValue(true));

        // define a language
        newUserEntry.putValue(AttributeName.LANGUAGE,
                               new LocaleValue(Locale.getDefault()));
        // the user does not have to change his password
        newUserEntry.putValue(AttributeName.PWD_CHANGE,
                               new BooleanValue(false));

        // no trusted login allowed
        newUserEntry.putValue("trustedLogin",
                               new BooleanValue(false));

        // no default object rights are defined
        newUserEntry.putValue("vipRights", new StringValue(" "));
        // the user is assigned two object classes: vip and vipUser
        RepositoryMap objectclassMap = new RepositoryMap();
        objectclassMap.putEntry("vip", null);
        objectclassMap.putEntry("vipUser", null);
        newUserEntry.putValue(AttributeName.OBJECTCLASS,
                               objectclassMap);

        // assign the user to the website
        RepositoryMap websiteMap = new RepositoryMap();
        websiteMap.putEntry("{VIPSITE}", null);
        newUserEntry.putValue(AttributeName.WEBSITES, websiteMap);
```

```
// assign the user to a group
RepositoryMap groupMap = new RepositoryMap();
groupMap.putEntry("testgroup", null);
newUserEntry.putValue(AttributeName.GROUPS, groupMap);
// create the user
vub.createNewUser("ldap", uid, newUserEntry);

%><p>Created new user account: <%=vub.loadEntry("ldap",
                                                    uid)%></p><%
}
else
{
    %><p>Please insert data into all fields.</p><%
}
%>
</html>
```

---

## Generating Dynamic Navigation Elements

The purpose of consistent navigation in a website is to make the content accessible to the user in an easy way. In addition, navigation provides the user with hyperlinks for quickly and easily switching to associated content, for example, to pages on related topics or neighboring pages in the hierarchic structure. Sitemaps are also navigation elements. Generally speaking, however, they are not confined to the context of the current page, but provide information about the website as a whole.

In Livelink WCM Server, the fact that metadata are stored for each content object provides sufficient information for navigating. For each content object, the program stores details of those objects that can be reached by hyperlink and those objects from which the active content object can be reached by hyperlink. It also stores the references to the parent WCM object and to all WCM objects that are located on the same or a lower level in the topic structure. Further information for navigation is provided by the topic tree and by linking a content object to a template object.

If the underlying metadata are changed, the content is automatically modified – in the case of a respective implementation – without any need for manual modification. As a result, these navigation elements do not get out of date when objects are deleted, copied, or moved.

The Portal Manager API makes dynamic generation of navigation elements possible. For example, the subtopics of a main topic could be automatically output as subheadings in a menu, or sitemaps could give the user an overview of the structure of the website. In both cases, changes in the structure of the website are taken into account without any need to revise the navigation.

When navigation elements are dynamically generated by means of the methods of the Portal Manager API, the users' access rights can be taken into account. Users have only those navigation options for which they have the necessary rights. For example, an anonymous user sees different elements than an authorized user. More far-reaching personalization concepts would filter object information on the basis of attributes in the user profile.

### **Design**

This section outlines how to design dynamic navigation elements. Proceed as follows:

1. Sketch the layout of the page and consider what HTML elements can be used to implement it.
2. On the basis of this plan, write the JSP code. Consider which parts of the HTML code must be generated dynamically and which beans provide the appropriate methods for obtaining this information.

Normally, hyperlinks to WCM objects within a certain topic are offered as a table or list. The number and names of the objects are not known, i.e. the names and/or titles and the URLs of all WCM objects below the selected topic are needed.

By means of the `VIPObjectBean` class, a `RepositoryIterator` can be supplied to all objects below the topic. It is also possible to only consider objects of a certain type.

The `RepositoryIterator` runs through the list of objects. Each entry is a `RepositoryEntry` containing metadata (name, title, etc.) that can be read and output in HTML.

It is also possible to obtain the URL of an object via the `RepositoryEntry`, though it is only possible to specify the URL from the applicable deployment system. However, it makes sense to show the hyperlinks for the system in which the user is already located. A suitable method is provided by the class `VipObjectBean`.

### Implementation

The following section outlines how to build the JSP page.

1. Import the required classes. Instantiate beans with a special JSP statement.
2. Make sure that all WCM beans used are initialized.

It must be known to the `VipObjectBean` class which repository is to be used. The repository specifies the deployment system. The WCM tag `{VIPDEPLOYMENT_NAME}`, which references the deployment system of the object, is usually used for the repository. The WCM tag `{VIPOID}`, however, references the OID of the WCM object.

The following example contains the JSP code used to read and run through the `RepositoryIterator` (applicable to all JSP topics below the WCM OID). The commands are coded in HTML code in such a way that, for example, an entry in a list or a row in a table is output for each `RepositoryEntry`. A hyperlink can then be constructed from the name of the entry and the URL of the WCM object.



```
<html>

<head>
<TITLE>getSubElements</TITLE>
</head>

<body>

<!-- This should be part of the Template -- Begin -->
<%@ page
    import="de.gauss.vip.repository.RepositoryEntry,
           de.gauss.vip.repository.RepositoryIterator,
           de.gauss.lang.KeySort,
           de.gauss.lang.StringValue"
%>

<jsp:useBean id="sessionBean"
class="de.gauss.vip.portalmanager.SessionBean"    scope="session" />
<jsp:useBean id="objBean"
class="de.gauss.vip.portalmanager.VipObjectBean"  scope="session" />

<%
    objBean.setRepositoryName("{VIPDEPLOYMENT_NAME}");

    // A user has to be logged in, otherwise the operations are
    // executed in the content of "Anonymous".
    if( !sessionBean.isLoggedIn(request) )
    // The request contains the necessary attributes for the
    // login.
        sessionBean.checkLogin( request );
%>
<!-- This should be part of the Template -- End -->
<%
    // get the meta data of the object
    RepositoryEntry entry = objBean.getEntry( "{VIPOID}" );
    if( entry != null )
    {
        KeySort sort = new KeySort(new StringValue("title"));
        // get all subelements of the current object which are
        // jsp topics
        RepositoryIterator it = objBean.getSubElements( entry,
            "JSPTOPIC", sort);
        %><table><%
        while( it.hasNext() )
        {
            RepositoryEntry e = it.nextEntry();
            %>
            <tr><td>
```

```
        <a href="<%= e.getValue("url") %>">
        <%=e.getValue("title") %>
        </a>
        </td></tr>
        <%
        }
        %></table><%
    }
%>
</body>
</html>
```

---

## Placing Several Content Objects on one Page (Multi-Content Page)

With the Portal Manager API, you can present as many content objects (texts, tables, graphics, etc.) as you like on one page. It is of course necessary to take account of restrictive aspects such as access rights. The objects you use as content may be objects of the WCM-managed website or data from external programs. Even inter-website and inter-program content links are no problem at all.

The multi-content page approach is particularly suitable for sophisticated websites where responsibility for content creation and content provision is usually divided between different staff. For example, product descriptions may be written by copy writers, the illustrations are supplied by a photo agency, and product data such as name, article number, price, and delivery period is entered in a database by the purchasing department.

### Design

A multi-content page is created as a JSP page by means of the methods of the Portal Manager API. This requires – even at this early stage – a division of labor between web designer and programmer.

Sketch the layout of the multi-content page on a sheet of paper. Since it is advisable to arrange the content objects with the aid of HTML tables, you should determine the exact position and size of the tables before going any further.

### Implementation

When you have finished creating the layout, proceed as follows.

1. Create a JSP object.
2. Execute your layout with HTML elements, and then check the result in the browser. Make any changes needed until the result is in line with your ideas. You may have to check the page out and in again several times.
3. Insert the JSP code in the page. Make sure to separate the JSP elements from the HTML code as far as possible. This makes the structure of the JSP page considerably clearer and makes it easier for you to implement changes necessary at a later stage.

The following sample code contains a JSP code for a multi-content page, in which different possibilities for loading the content are listed.

---

```
<html>

<head>
<TITLE>multiContent</TITLE>
<body>

<!-- This should be part of the Template -- Begin -->
<%@ page
    import="de.gauss.vip.repository.RepositoryEntry,
           de.gauss.vip.portalmanager.converter.HTMLContentConverter,
           de.gauss.vip.portalmanager.ContentHandler"
```

```
%>
<jsp:useBean id="sessionBean"
  class="de.gauss.vip.portalmanager.SessionBean"
  scope="session" />
<jsp:useBean id="objBean"
  class="de.gauss.vip.portalmanager.VipObjectBean"
  scope="session" />

<%
    objBean.setRepositoryName("{VIPDEPLOYMENT_NAME}");

    // A user has to be logged in, otherwise the operations are
    // executed in the content of "Anonymous".
    if( !sessionBean.isLoggedIn(request) )
    // The request contains the necessary attributes for the
    // login.
        sessionBean.checkLogin( request );
    %>
<!-- This should be part of the Template -- End -->
<%

    // In this example is no check, if the retrieved entries
    // are null in my website is object "215" a jsp-object
    RepositoryEntry jsp  = objBean.getEntry("215");
    ContentHandler chJsp = objBean.getContent(jsp);

    // in my website is object "255" a picture object
    RepositoryEntry pic  = objBean.getEntry("255");
    ContentHandler chPic  = objBean.getContent(pic);

%>
<table>
<tr>
    <td><!-- Gets the whole content, i. e. content plus
        template. -->
        <%= chJsp.getString() %>
    </td>
    <td><!-- Returns the content, in this case the <img> tag -->
        <%= chPic.getString(new HTMLContentConverter()) %>
    </td>
</tr>
<tr>
    <td><!-- This is faster, because it is not necessary to load
        the content -->
        ">
    </td>
    <td><!-- Gets only the content of the object, i. e. content
        without template. -->
        <%= objBean.getVipContent(jsp) %>
    </td>
</tr>
</table>
```

```
</tr>
</table>

</body>
</html>
```

---

Using the method `getVipContent` of the class `Vip0bjectBean`, you can access the HTML code of the content objects. This bean also provides the method `getContent`, which returns a `ContentHandler`. This class can be used to access specific items in the content (e.g. specific attributes in form instances). The content is loaded via a URL connection, i.e. dynamic pages are executed before they are returned. The methods require a `RepositoryEntry` as parameter. The framework of the Portal Manager API provides you with numerous methods of determining a `RepositoryEntry` indirectly. If you know the object ID of a content object, however, you can determine a `RepositoryEntry` directly with the method `getEntry` of the class `Vip0bjectBean`.

## Creating WCM Objects via an HTML Form

Using the Portal Manager API, you can develop HTML forms that can be used for form-based content creation. A form-based entry is always advantageous if large numbers of similar content objects are to be created and the responsible users need a simplified user interface for implementation.

Although the Content client, for example, is very easy to use, it provides a wide scope of functionality. Only a small proportion of these functions are needed for creating simple, identically structured documents such as article pages and press releases. Making these functions available through a form considerably reduces familiarization time for the user.

This is important, for example, if external resources are to be integrated in the website creation process. A professional web content management system such as Livelink WCM Server makes it possible for external staff to work on a distributed basis at their individual locations. In this way, advertising copy writers and photo agencies can be integrated in the creation of article pages.

The structure of the content object created in this way is determined by a form. The objects created are XML objects – known as *form instances* – with a structure corresponding to the structure of the form. The elements of these XML objects can then be arranged and formatted as desired on an HTML page.

**Note:** It is of course possible to create other object types.

### Design

A form-based entry must always be implemented as a clearly defined program. The user interface consists of several components that are integrated in a workflow and tailored to the program in question. As a rule, the following components are needed:

- Overview page – lists the form instances created and provides buttons for the actions that can be performed on these WCM objects
- Input form – is used for creating new form instances, and provides entry fields, selection fields, etc. for creating the XML objects
- Edit form – is used for changing existing form instances. The field contents and the status of the form instances are shown in editable form.
- View page – is used for preparing the field contents and the status of form instances for display.
- `VipObjectHandlerBean` – performs, among other things, the following actions on the form instances:

- Create
- Change
- Submit
- Reject
- Release
- Delete

The input form is called from the overview page. The user enters the form data and sends off the form. The contents of the form are passed to the JSP page for creation. The JSP page extracts the form entries from the HTTP request and generates the metadata and the content for the form instance. The form instance must then be created as a WCM object and its content must be written.

If an existing form instance is to be edited, it is loaded into the edit form from the overview form. The user edits the data and sends the form off again. Unlike creation of form instances, the contents of the form are passed to the JSP page for editing. This extracts the form data and updates the metadata and content of the form instance that is to be edited.

The remaining actions that can be performed from the overview page are less spectacular. They merely involve calling the relevant `ObjectHandler` for the form instance and updating the overview page.

## Framework for Applications

This section describes the framework used to implement user-specific applications.

The Portal Manager API provides a number of Java classes that can be used to integrate user-specific applications in the Portal Manager API. The interaction of the classes is shown in the following class diagram:

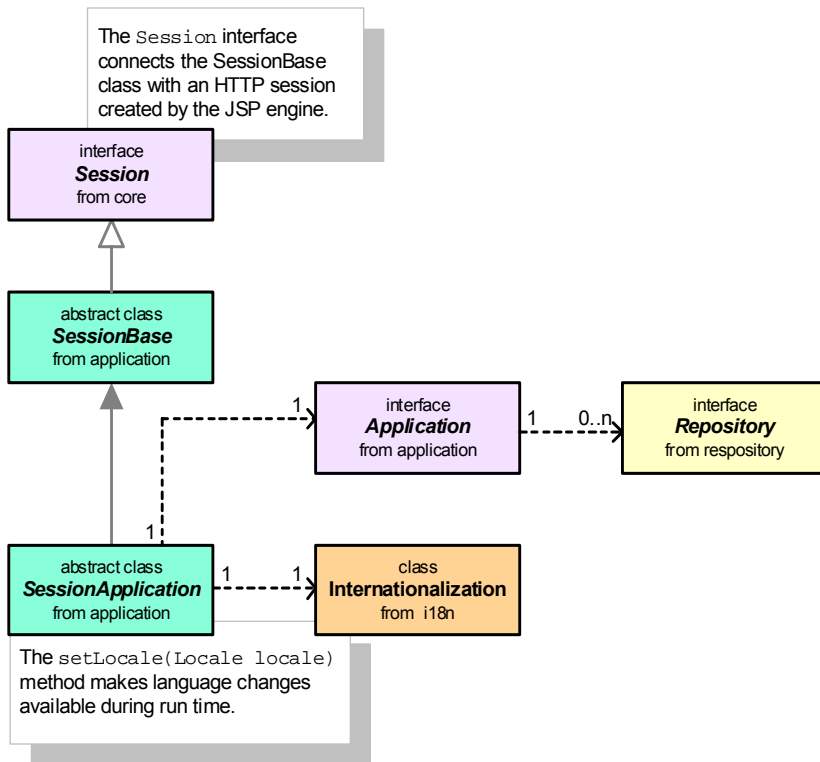


Fig. 37 – Class diagram for applications

The following elements are needed to implement an application:



- **Repository** – An implementation of the interface `Repository` is used as a wrapper class to make an external data source WCM-capable. An application may be associated with 0–n repositories
- **Application** – An implementation of the interface `Application` is used as a wrapper class to make the logic of an application WCM-capable. The application uses the associated repositories to maintain and edit the business objects.
- **JavaBean** – A class deriving from the class `SessionApplication` packs the `Application` in a JavaBean, which performs the instantiation of the `Application` in the context of a JSP page. The bean provides the following functionality:
  - linking the bean with a `javax.servlet.http.HttpSession` – The HTTP session provides state support for the stateless HTTP protocol. This makes it possible to associate a specific user with an HTTP session for authentication purposes.
  - internationalization support – switchable at runtime – for an application via the class `Internationalization`
  - additional application-related functionality

## Implementing an Application

The actual process of implementing an application which is derived from the Portal Manager API comprises the following steps:

1. Implementing the repositories, applications, `SessionApplication` objects
2. Configuring the Portal Manager API to use the new application (see section “Managing applications” starting on page 65)

## Deriving the Application-Specific Classes

To make the procedure easier for the user, the Portal Manager API provides abstract basic classes with the basic implementations of all functionalities connected with the Portal Manager API. A user-specific application merely has to implement a certain basic class. The following diagram shows the general hierarchy for developing applications:

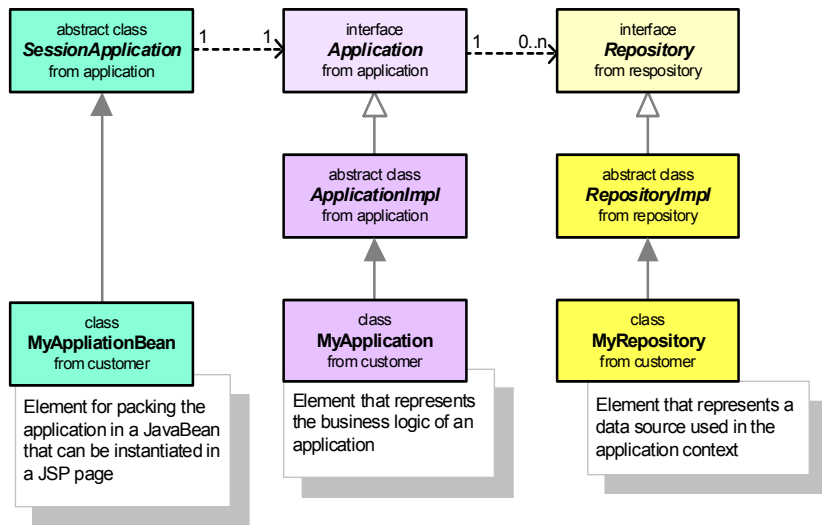


Fig. 38 – General hierarchy for developing applications

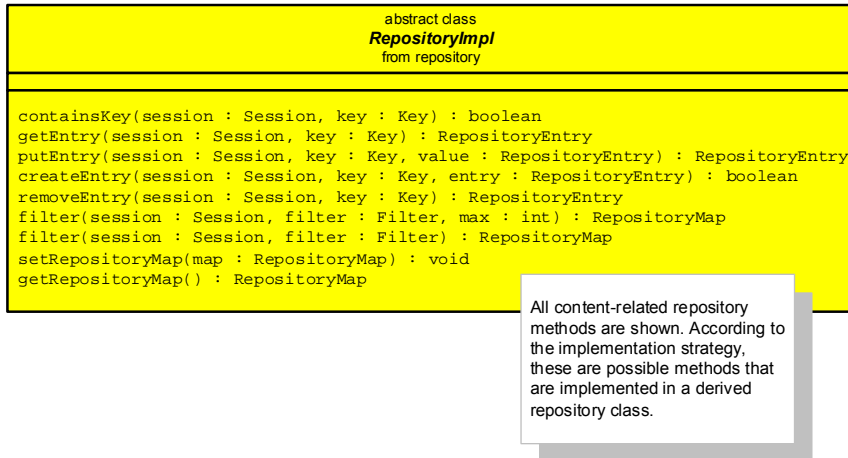
### Deriving from the RepositoryImpl class

As already described, the interface `Repository` functions as a wrapper class for preparing data sources so that the Portal Manager API can access them.

The abstract basic class `RepositoryImpl` provides standard implementations for methods which are related to the Portal Manager API. This means that only methods that are relevant to the data sources must be implemented in the derived `Repository` classes. A repository in the

context of the Portal Manager API is based on data structures used within the WCM system (see section “RepositoryMap and associated classes” on page 102).

The following figure shows all methods that are related to data sources and that are provided by the class `RepositoryImpl`.



**Fig. 39 – The `RepositoryImpl` class with all methods relevant to data sources**

The class derived from the `RepositoryImpl` class has its own persistency mechanism. It passes the persistency task to an LDAP or DBMS system (e.g. via a JDBC connection to an RDBMS).

For a `RepositoryImpl`-capable implementation, it is necessary to implement all specified methods. For all `Repository` classes that only support certain methods, the non-supported methods should throw a meaningful exception.

### **Deriving from the `ApplicationImpl` class**

The `ApplicationImpl` class and the classes derived from it have the following functions:

- Managing all repositories associated with an application
- Providing the business logic for an application

All methods required to support the Portal Manager API are provided via the basic class `ApplicationImpl` and do not have to be implemented by means of a derived `Application` class. Applications are configured via the Admin client. There are no general rules for implementing the business logic for a specific application.

### **Deriving from the `SessionApplication` class**

The `SessionApplication` class (and the classes derived from it) have the following functions:

- Encapsulating an `Application` as a JavaBean to support the JSP application
- Associating an `Application` with a specific HTTP session (and the HTTP session with a WCM user)
- Maintaining a runtime locale for the `Application`
- Returning messages and resources in accordance with the current locale setting. This is done via the internationalization object that is associated with the `SessionApplication`.

All methods required to support the Portal Manager API are provided by the basic class `SessionApplication` and do not have to be implemented by the derived bean classes.

## Designing a JSP Page

Various models are available to you for designing a JavaServer page (JSP page). In the JSP model 1 architecture, the JSP page is responsible for processing a request and sending a reply to the client. Access to the data is performed by the JavaBeans. This model results in a large number of scriptlets and/or large amounts of Java code in the JSP page. This makes it more difficult to maintain these sources, eliminate errors in them, or reuse them. Moreover, there is no clear separation of logic and presentation, which makes it difficult to split jobs between Java developers and web designers.

The JSP model 2 uses the MVC model design (Model View Controller). Here, a *controller* JSP page or a servlet is used that is responsible for processing the request and generating the beans. The JavaBeans (Portal Manager API classes) are the *model* that has access to the data sources. The controller decides – depending on the request parameters – which JSP page to forward the request to. The *View* JSP page has no processing logic. It uses the beans to build up the dynamic content.

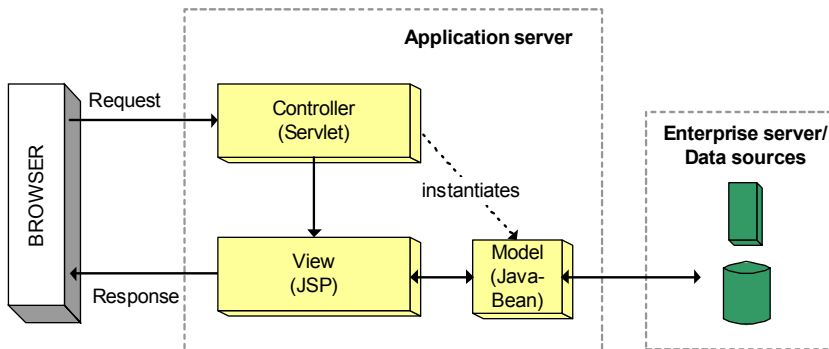


Fig. 40 – JSP model 2 architecture

## Tips for Developing JSP Pages with the Portal Manager API

Livelihood WCM Server offers two possible ways of implementing the controller in the JSP model 2 architecture. First, you can use the JSP page or the servlet as a controller. Since JSP pages are easier to handle in Livelihood WCM Server than servlets, we recommend using JSP pages. You can also work with WCM templates that may contain the following functionality:

- Login check to see whether the session user has already logged in to the Portal Manager API
- Creating and initializing Portal Manager API beans that are needed in the view
- Forwarding the request to a JSP page, depending on the request parameters
- Error handling by means of a try/catch block

---

```
<%
    try {
%>
{VIPCONTENT}
<%
    } catch (Exception ex) {
        //Error handling
%>
Error: <%=ex.getMessage()%>
<%
    }
%>
```

---

Although the generated page contains the template and the content of the view (as in JSP model 1), the web designers cannot see the content of the template. The templates should therefore be written by experienced Java developers.

The web designers are responsible for writing the view (of the WCM object). When they create a WCM object, they can select a template that may subsequently be modified. The web designers should use as little Java as possible in the JSP page.

If web designers cannot avoid using Java, e.g. because of logical or arithmetical operations that do not relate to the design, they should offload their code into JavaBeans or classes.

## Security Aspects

When designing a web application that operates with sensitive data, the security aspect is particularly important. Apart from aspects such as authentication, encryption etc., which must be implemented in the application, it is also important to be aware of the access for the Java Virtual Machine (JVM), which executes the server-side software. When using JSP pages, it is necessary to take a closer look at the security mechanisms of the JVM that are used within the JSP engine.

## JVM Without SecurityManager

If no SecurityManager is explicitly set in the context of a program, the JVM possesses the access rights of the user who started the program. Depending on the functionality of the program, this can result in serious security loopholes. Possible scenarios:

- The application and hence also the client can navigate in the server-side file system and view or modify content.
- The application and hence also the client may establish network connections with computers in the server-side environment.

In such an environment, you are recommended to give the above-mentioned user only those rights that are needed to ensure a correct functioning of the application. Moreover, access should be prohibited to other resources, especially security-relevant or sensitive data.

## JVM with SecurityManager

The Java 2 platform provides a revised security architecture with finely structured and easily configured security mechanisms. To be able to use these functions, perform the following steps:

1. Integrate a SecurityManager in the server process.
2. Create a security mechanism file that provides the program with the rights required for a correct functioning. This file can be used to impose considerable restrictions on access by a program compared with the rights granted to a user by the operating system.

A detailed description of the SecurityManager can be found in the documentation on Java 2.

## Internationalization

The Internet and intranet applications are increasingly used globally and designed for users in different countries. Thus, there is potentially a need to internationalize a solution based on the Portal Manager API.

The solutions are based on Java, so they automatically inherit the internationalization support offered by Java. This includes:

- Locale – the Locale object as a simple identifier for a language (or region)
- localized resources – the ResourceBundle as container for all Locale-specific commands and objects



- calendar and time zone support
- Locale-specific formatting of numbers, dates, etc.
- Locale-specific string methods (e.g. Locale-specific sorting)

All functions are described in detail in the Java 2 specification.

As a web application, the Portal Manager API uses other software components in addition to the Java programming language. The architecture and the implications of the architecture for internationalization are explained in the following sections.

## Architecture of the Portal Manager API and Internationalization

The following diagram illustrates the architecture of the Portal Manager API with all aspects relevant to internationalization.

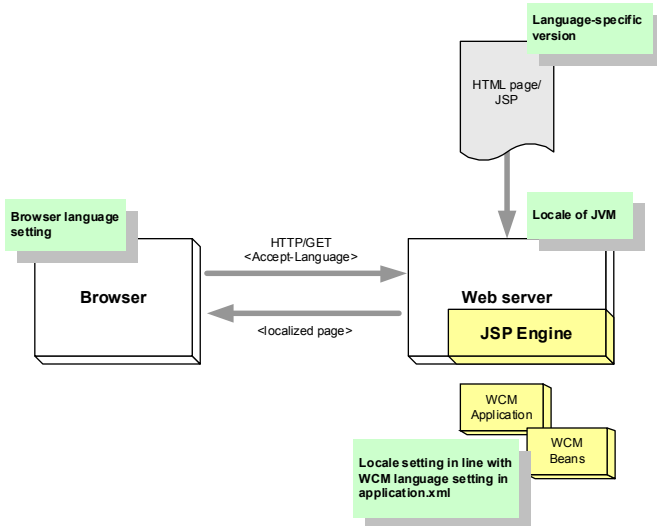


Fig. 41 – Architecture of the Portal Manager API with internationalization aspects

The architecture is based on the following components:

- a browser as client (for the HTTP protocol)
- a web server as server (for the HTTP protocol)
- a JSP engine as adapter between web server and Java applications
- components of the Portal Manager API (e.g. JavaBeans) that are executed in a JSP engine

As shown in the diagram, each of these components has its own settings relevant to language and internationalization:

- The browser can request a specific language via the HTTP header.
- The web server has to keep different document versions (HTML, JSP page, graphics, etc.) available.
- The JVM that runs the JSP engine has a default `Locale` setting.
- The WCM beans and WCM applications are `Locale`-specific. The default language setting for an application and the respective bean is specified in the Admin client (see section “Managing applications” starting on page 65).

**Note:** The default `Locale` for the `SessionBean` is determined by the default language of the logged-in WCM user.

## Recommendation for Internationalizing a Portal Manager API Solution

On the basis of the architecture described in section “Architecture of the Portal Manager API and Internationalization” on page 161, internationalization can be carried out according to the following strategy: the client that uses the solution is the recipient of all information and the entire content. It therefore specifies its preferred language or localization. All

other system components must modify their behavior to suit the client's choice.

This strategy results in the following technical recommendations:

- Users can define their preferred language in the browser. This preference is sent in the header of every HTTP request and can be retrieved by the web server and by the JSP engine. The choice of language according to RFC 1766 and ISO 639 can be retrieved from the servlet request.
- The URL can be dynamically created and reference the applicable content.
- For internationalizing dynamic content made available by applications, the following must apply:
  - The JavaBeans or applications must support internationalization
  - A JSP page that instantiates a specific bean or component must define the `Locale` in accordance with the user settings. This is done by obtaining the value `accept-language` from the header of the `HttpRequest` object and creating a corresponding Java locale object.

The following example shows a JSP page that uses internationalization with a session bean.

---

```
...  
  
<body>  
  
<%@ page import="java.util.Locale" %>  
<h1>Setting the Locale for a WCM bean from the HTTP header</h1>  
<jsp:useBean id="sessionBean"  
    class="de.gauss.vip.portalmanager.SessionBean" />  
<%  
    sessionBean.init(request);  
    Locale locale = request.getLocale();  
    sessionBean.setLocale(locale);
```

```
        out.println("sessionBean locale=" + sessionBean.getLocale()
            + "<br>");
        out.println(sessionBean.getI18n().getString("MY_MESSAGE_KEY"));
    %>

</body>

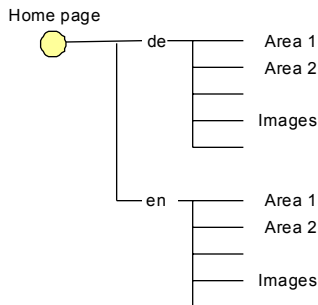
...

```

---

## Supporting Several Languages by Using one Central Template

The website structure can be presented as in the following figure:



**Fig. 42 – Structure of a website**

When users reach the start page of the website, they can choose a language, for example English. After selecting the language, a JSP page is called up, which will set all the necessary OIDs such as the start files of the individual areas. Once an area has been selected, the template will receive the OID from the session and will display the applicable objects.

## Applications and Internationalization

The Portal Manager API offers the possibility of integrating user-specific applications in the API. For being able to use the internationalization support of the Portal Manager API, you must perform the following steps.

1. Localize all resources in accordance with the standards laid down in Java 2 (`PropertyResourceBundle`, etc.).
2. Assign the resources and the parameters to the user-specific application (see section “Managing applications” starting on page 65).

## WCM Beans and Internationalization

All WCM beans have integrated support for internationalization. Its use is made possible by the following methods:

- The current setting of the `Locale` valid for a specific bean is provided by `public java.util.Locale getLocale`.
- The `Locale` for a specific bean can be set by `public void setLocale(java.util.Locale locale)`.

## WCM Internationalization Object

To support internationalization, Livelink WCM Server provides a specific internationalization object that is used in the WCM beans. Access to this object is ensured by means of the method `getI18n`. You can also use this object to internationalize your own beans and applications.

The class `de.gauss.vip.i18n.Internationalization` makes the following functionality available:

- Loading `ResourceBundle` objects
- Obtaining keys from a specific `ResourceBundle`
- Obtaining `Locale`-specific date and number formats

---

---

# APPENDIX A

## Metadata Schemes

Compared with version 5e, the metadata names and data types have remained unchanged in order to ensure downward compatibility with previous versions of the Portal Manager API.

When configuring a `VipObjectRepository` or a `VipObjectHandlerRepository`, you can enter which metadata scheme is to be used (see table 3 “Repositories and their parameters” on page 57).

The following table compares the metadata schemes of version 5e and version 8.

Metadata scheme in version 5e		Metadata scheme in version 8	
Attribute	Value	Attribute	Value
acl	VipAclValue	acl	Acl
author	StringValue	created_by	User
date-created	DateValue	date_created	DateValue
date-expire	DateValue	date_expire	DateValue
date-release	DateValue	date_released_at	DateValue
date-released	DateValue	date_released	DateValue
direct-release	StringValue	direct_release	BooleanValue

Metadata scheme in version 5e		Metadata scheme in version 8	
Attribute	Value	Attribute	Value
edit-url	StringValue	url	StringValue
		Note: in repositories of type EDIT	
elements	RepositoryMap	all_children	ListValue
email-edit	StringValue	email_edit	SetValue
email-qa	StringValue	email_qa	SetValue
email-release	StringValue	email_release	SetValue
filename	StringValue	filename	StringValue
		Note: Portal Manager API-specific	
generic1	StringValue	keywords	SetValue
generic2	StringValue	description	StringValue
generic3	StringValue	target_group	StringValue
language	StringValue	language	LocaleValue
level	IntegerValue	level	IntegerValue
		Note: Portal Manager API-specific	
linked-from	RepositoryMap	linked_from	SetValue
links-to	RepositoryMap	links_to	SetValue
name	StringValue	title	StringValue
object-icon	StringValue		
oid	StringValue	oid	ObjectId



Metadata scheme in version 5e		Metadata scheme in version 8	
Attribute	Value	Attribute	Value
pathname	StringValue	pathname	StringValue
prod-url	StringValue	url	StringValue
		Note: in repositories of type PROD	
qs-url	StringValue	url	StringValue
		Note: in repositories of type QA	
status	StringValue	state	ObjectState
status-icon	StringValue		
		Remark: accessible via ObjectState	
subtitle	StringValue	subtitle	StringValue
suffix	StringValue	suffix	StringValue
		Note: Portal Manager API-specific	
template	StringValue	template	ObjectId
topic	StringValue	topic	ObjectId
topics	RepositoryMap		
type	StringValue	type	ObjectType
url	StringValue	url	StringValue
version	IntegerValue	version	Version
vip-edit-url	StringValue	surrogate_url	StringValue
		Note: in repositories of type EDIT	

Metadata scheme in version 5e		Metadata scheme in version 8	
Attribute	Value	Attribute	Value
vip-qs-url	StringValue	surrogate_url	StringValue
		Note: in repositories of type QA	
website	StringValue	website	StringValue
@@codebase	StringValue		
		category	StringValue
		date_modified	DateValue
		deployment_hint	StringValue
		Note: contains file name with extension	
		modified_by	User
		released_by	User





---

# APPENDIX B

## Caching

For operating Livelink WCM Server, a relational database management system is required. In the database, the metadata of the WCM objects are saved, among other things. This appendix describes the caching behavior of the Portal Manager API with regard to metadata.

### Accessing WCM Objects

When an object is accessed for the first time, the saved metadata are loaded from the database. The object is created from the metadata and then stored temporarily in the server-side cache. When an object that has already been loaded is requested again, the object contained in the cache is returned.

If an object changes as the result of a staging action, it is removed from the cache. Consequently, the updated metadata must be reloaded from the database the next time the object is accessed. As an object may have different metadata in each view (Edit, QA, Production), a separate object is stored in the database and the cache for each view.

Creation of some metadata is quite time-consuming. As the Content client usually does not require all metadata, only the *basic data* is read from the database when the object is loaded. Thus, the server requires less memory and the load process is sped up. The metadata that are not loaded – the so-called *data to be loaded with delay* – are read from the database on demand.

In Livelink WCM Server, all special attributes as well as the following data are loaded with delay:

- `pathname`
- `filename`
- `suffix`
- `url`
- `surrogate_url`
- `links_to`
- `linked_from`
- `all_children` (for Oracle only)

### Notes

- If you do not want the data to be loaded from the database during operation, you should access the data to be loaded with delay directly after loading the object. Thus, the data can be read from the cache later on.
- The metadata `links_to` and `linked_from` should only be loaded in exceptional cases as they can only be retrieved by means of a time-consuming database request.
- For the metadata item `linked_from`, the rule applies that the object is not removed from the cache if a reference to the object is added to another object or if a reference to the object is removed.

## Deployment Cache

In Livelink WCM Server, deployment systems have an integrated cache for the metadata provided by the deployment. Thus, this data can be supplied directly the second time it is accessed. This presupposes that the deployment system is set up on the accessing server (usually a Content server running in the context of a JSP engine or as web application in an application server). This constellation is recommendable if during normal operation only a few changes are made that have a great impact on the deployment. Otherwise, the additional effort required for the deployment may result in bottlenecks on the Content server.

It must be evaluated individually whether faster access to the data of the deployment system compensates for the additional load caused by the separate deployment system. Example: in your website, you use a template cascade that is assigned to more than 10,000 objects. Some of the templates are changed regularly once a week. After the changed templates have been released, all objects affected must be regenerated. This causes a higher load on the server. Whether a higher load is acceptable for your system, can be determined by means of performance tests.

## How the Cache for WCM Objects Works

The cache for WCM objects is configured in the Admin client in the settings of the website (*Configuration* → *Websites* → *{website name}* → *General* tab → *Caching*). The following parameters can be set:

- *Minimum size*: number of objects to which the cache is reduced in server idle times. The default value is 20000.
- *Maximum size*: maximum number of objects in the cache. The default value is 45000.

- *Cache reduce interval*: interval in milliseconds. The default value is 7200000 (corresponds to 2 hours). If the objects in the cache are not accessed during this period, objects are removed from the cache until the minimum cache size is reached again.

When an object is accessed, the system first looks for the object in the cache. If the object is not in the cache, it is loaded from the database. Afterwards, the object is stored in the cache. Three cases are differentiated:

- **The minimum size has not been reached yet.**

The newly loaded object is stored at a free position in the cache. The cache grows.

- **The minimum size has been reached, the maximum size has not been reached yet.**

The newly loaded WCM object is stored in the cache. As a result, another object may be removed from the cache. This means that the size of the cache does not necessarily increase.

- **The maximum size has been reached.**

The newly loaded WCM object is stored in the cache. On the basis of a certain algorithm (How often has the object been accessed/How long has it been in the cache?), the cache determines which object is removed from the cache.

If – when the configured cache reduce interval is reached – the size of the cache is greater than specified in the minimum size parameter, the cache automatically removes objects until the minimum size is reached. This process is repeated at the configured interval.



If you want to ensure that many objects are stored in the cache, set the minimum size to a high value. This prevents the objects from being removed from the cache too early. The maximum size must be adapted accordingly. On Content servers with Production view in particular, it is recommendable to store the majority of objects in the cache during operation.

## Caching RepositoryEntry Objects

The representation of metadata on the Content server running in the context of a JSP engine or as web application in an application server, the so-called *external* representation, is realized by RepositoryEntry objects. The external representation is partly different from the internal representation of the data. To keep the conversion effort at a minimum level, each object in the cache has its own cache for storing RepositoryEntry objects. If an object is removed from the cache, the corresponding RepositoryEntry objects are removed as well.

When the Content server requests a RepositoryEntry, the WCM object checks whether the desired attribute combination already exists in its cache and provides the data directly from there. Otherwise, the requested data is prepared accordingly. For this purpose, they may have to be loaded from the database. Afterwards, the attribute combination with its values is stored in the cache.

When programming JSP pages, you should therefore specify – if possible – all attributes required on the page when accessing the RepositoryEntry for the first time. Thus, you do not have to access the database and the database does not have to return values for different attributes.



---

# Glossary

**Application** – Interface for a uniform description of an external application

**Context ID** – Object that is assigned to a user after successfully logging in to the WCM system. A context ID is always unique throughout the entire system. It thus precisely identifies a user. If a context ID is not used over a certain period of time, it expires.

**Deployment system** – The deployment systems generate pages from the WCM objects and distribute the generated files to the appropriate directories. From there, the files become visible for the users via an HTTP server. Deployment systems may be of various types and categories.

**Dynamization** – All or part of the content of a website is not generated until runtime.

**JSP engine** – A module, integrated in the web server, for running JSP scripts embedded in HTML pages. JSP engines generally contain Java compilers.

**JSP script** – HTML page in which Java code has been embedded which is run on the server side.

**LoginRepository** – Repository used to work with user accounts

**Master server** – Only master servers have read and write access to the data of a WCM system. The master Content server manages website data, while the master Administration server manages the configuration and system data of the WCM system. See also *Server category*.

**ObjectHandler** – This instance accepts actions and thus manipulates the objects of the website.

**Personalization** – Dynamization of the website tailored to the current user

**Portal** – A portal is a website that serves the user as a central point of access – as a gate – to certain Internet services. A portal often offers topic-specific and personalized offers and information.

**Properties file** – A file containing resource information in a defined format (key-value pairs)

**Proxy server** – A proxy server is used to intercept requests from a client application, e.g. a browser, to one or more other servers. If the proxy server can meet the request, it sends the requested data back to the client. Otherwise, it forwards the request to the specified server.

In the context of Livelink WCM Server, WCM servers of the category “proxy” do not have write access, but only read access to the WCM objects or the configuration. Changes to the WCM objects are only possible via the master Content server, changes to the configuration of the WCM system are made only via the master Administration server. See also *Server category*.

**Repository** – A repository is an abstraction that allows uniform access to data.

**Server category** – In a WCM system, a distinction is made between master and proxy servers. Master servers have write access to the data of the WCM system, while proxy servers have only read access. The master Content server manages the website data, the master Administration server manages the configuration and system data. In addition to this, any number of proxy servers can be set up.

**Server type** – According to the tasks of the servers, there are two server types: Content servers for managing website data and Administration servers for managing the user, configuration, and system data of the WCM system. Basically, every Content server is able to provide all views of the data of the managed websites – Edit, QA, and Production. The available views may be limited by the fact that the Content server only receives the data of certain views.

**Session** – Unit managed by the JSP engine so that logically related actions (in terms of resources) can be combined

**Session bean** – A JavaBean used in a JSP script in connection with the Portal Manager API. The life span of a session bean lasts throughout the entire session.

**Statification** – During statification, the dynamic components of, for example, a JSP page are converted into static components. The result is pure HTML without Java code.



---

# Index

## A

- absolute\_urls (special attribute) 40
- add
  - application 67
  - repository 54
- Application (interface) 112
- application examples
  - creating users in LDAP via a portal 133
  - dynamic navigation elements 142
  - HTML form 149
  - multi-content page 146
- ApplicationImpl (class) 113
- applications 111
  - add 67
  - assign repository 81
  - assign repository (during setup) 70
  - assign to Content server 84
  - class name 68
  - delete 84
  - framework 152
  - implement 153
  - internationalization 165
  - introduction 22
  - language 69
  - logical name 68
  - manage 65
  - parameters of supplied applications 73
  - resources 71
  - set parameters 72
- architecture
  - Portal Manager API 19
- authentication 127

## B

- Base (class) 110
- basic classes 97
  - basic data types 98
  - filtering and sorting 100
  - repositories 106
  - RepositoryMap 102
- basic data types 98
- beans 114

## C

- class description 97
- class diagrams 95
- classes
  - ApplicationImpl 113
  - Base 110
  - FormData 37
  - Internationalization 116
  - RepositoryEntry 103
  - RepositoryImpl 108
  - RepositoryMap 104
  - SessionApplication 116
  - SessionBean 117
  - VipObjectBean 120
  - VipObjectFilterBean 123
  - VipObjectHandlerBean 45, 121
  - VipUserBean 120
  - VipWhatsNewBean 124
- content
  - dynamic 33
- content objects
  - placing several 35
- Content server in application server
  - start 31
- ContentHandler (interface) 125

### D

- data values 98
- delete
  - application 84
  - repository 65
- DirectoryRepository
  - class 87
- dynamic content 33
- dynamic navigation elements 142

### E

- errors
  - locate 89

### F

- filtering 100
- FormData (class) 37
- forms
  - creating an object 37
  - working with 36
- framework
  - applications 152

### G

- generate\_static (special attribute) 39

### H

- HTML form 149

### I

- implement
  - applications 153
- integrating external programs 35
- integration
  - external programs 46
  - in the WCM system 24
- Interfaces
  - VipProfile 117

### interfaces

- Application 112
- ContentHandler 125
- RepositoryEntryIterator 104
- RepositoryIterator 105
- Value 98

### internationalization 160

- architecture of the Portal Manager API 161
- Portal Manager API solution 162
- WCM internationalization object 165

- Internationalization (class) 116

### J

- Javadoc 97
- JavaServer page
  - <Italic>see JSP page
- JSP page
  - design 157
  - model 1 architecture 157
  - model 2 architecture 157
- JSP script 20
- JVM
  - with SecurityManager 160
  - without SecurityManager 159

### K

- key values (keys) 98

### L

- language
  - application 69
- LDAP
  - create new users 133
  - DirectoryRepository (class) 87
- life cycle of a WCM object 45
- locale 160
- login repositories 107



**M**

manage  
  applications 65  
  repositories 52  
metadata schemes 167  
multi-content page 146

**N**

navigation elements 34  
new  
  application 67  
  repository 54

**P**

parameters  
  define for a repository 55  
  define for an application 72  
  supplied applications 73  
  VipDAVApplication 74  
  VipHCLApplication 75  
  VipHTMLClientApplication 76  
  VipObjectApplication 79  
  VipUserApplication 79  
  WebServiceApplication 80  
personalization 34  
placing several content objects 35  
portal  
  create users in LDAP 133  
Portal Manager API  
  application examples 133  
  architecture 19  
  integration 24  
  staging 44  
profile 117

**R**

repositories 106  
  add 54  
  assign application 62

  class name 55  
  define parameters 55  
  delete 65  
  introduction 23  
  login repositories 107  
  manage 52  
  name 55  
  types 107

RepositoryEntry (class) 103  
RepositoryEntryIterator (interface) 104  
RepositoryImpl (class) 108  
RepositoryIterator (interface) 105  
RepositoryMap 102  
RepositoryMap (class) 104

**S**

security aspects 159  
SecurityManager  
  JVM with 160  
  JVM without 159  
session 21, 108  
session beans 21  
session management 127, 130  
SessionApplication (class) 116  
SessionBean (class) 117  
sorting 101  
special attributes  
  statification 38  
staging 44  
start  
  Content server in application server  
  31  
statification 38  
  absolute\_urls (special attribute) 40  
  generate\_static (special attribute)  
  39  
  of form instances 41  
  of JSP objects 41  
  of XML objects 43

- timeout (special attribute) 39
- system architecture
  - master Administration server 26
  - master Content server 26
  - minimum system 27
  - proxy server 26
  - separate data storage 28

### T

- timeout (special attribute) 39
- trace.properties 89

### U

- users
  - create in LDAP via a portal 133

### V

- Value (interface) 98
- VipDAVApplication
  - parameters 74
- VipHCLApplication
  - parameters 75
- VipHTMLClientApplication
  - parameters 76
- VipObjectApplication
  - parameters 79
- VipObjectBean (class) 120
- VipObjectFilterBean (class) 123
- VipObjectHandlerBean (class) 45, 121
- VipProfile (Interface) 117
- VipUserApplication
  - parameters 79
- VipUserBean (class) 120
- VipWhatsNewBean (class) 124

### W

- WCM object
  - life cycle 45

- WebServiceApplication
  - parameters 80

