# Table of Contents

1 Introduction ................................................................................................................................. 5
1.1 Audience and Organization of Sections .................................................................................. 5
1.2 Motivation: The RF - What is it all about? ............................................................................. 6
1.3 Scope of this Document: Views on the RF ............................................................................ 7
2 RF Concepts .................................................................................................................................. 8
2.1 Basic Aspects – Namespace, Content and Properties ............................................................. 8
  2.1.1 RF Objects .......................................................................................................................... 8
  2.1.2 RID and Resource Context ............................................................................................... 10
  2.1.3 Creating, Copying, and Deleting ....................................................................................... 11
  2.1.4 Renaming and Moving ....................................................................................................... 12
  2.1.5 Links .................................................................................................................................. 13
  2.1.6 Unstructured Data: Content ............................................................................................ 14
  2.1.7 Structured Data: Properties ............................................................................................. 14
2.2 Advanced Aspects – Ordering, Property Search, ID-Mapping, Locking, Security, Events...... 17
  2.2.1 Ordered Collections and Collators ................................................................................... 17
  2.2.2 Property Queries .............................................................................................................. 18
  2.2.3 Unique ID ........................................................................................................................ 19
  2.2.4 Security ........................................................................................................................... 20
  2.2.5 Locking ............................................................................................................................. 21
  2.2.6 Events ............................................................................................................................... 22
2.3 Expert Aspects – Semantic Objects and Versioning ................................................................. 25
  2.3.1 Type Handling .................................................................................................................. 25
  2.3.2 Versioning ........................................................................................................................ 26
3 RF Extensions, API Layers and Concepts ..................................................................................... 34
3.1 RF Extensions ............................................................................................................................ 34
3.2 API Layers, Current and New API .......................................................................................... 36
3.3 (New) API Concepts ................................................................................................................ 38
  3.3.1 Aspects of RF objects in the API ....................................................................................... 38
  3.3.2 Supported Options, Read-Only and “Mutable” Interfaces ............................................... 38
  3.3.3 Exceptions ....................................................................................................................... 39
  3.3.4 Descriptors ...................................................................................................................... 40
  3.3.5 Typed vs. Un-Typed Collections and Iterators ................................................................. 41
  3.3.6 Mass Calls and Multi Status ......................................................................................... 41
  3.3.7 Atomicity and Mass Calls ............................................................................................... 42
  3.3.8 Interfaces vs. Classes ....................................................................................................... 42

© 2003 SAP AG
Utility Packages .................................................................................................................................. 43
  4.1 Current API: com.sapportals.wcm.util .................................................................................... 43
  4.2 New API: com.sap.netweaver.bc.rf.util .................................................................................. 46
Using the RF .................................................................................................................................. 47
  5.1 Basic Aspects .......................................................................................................................... 47
    5.1.1 RF Objects, RID, Resource Context and the Resource Factory ........................................... 47
    5.1.2 Namespace Operations ....................................................................................................... 50
    5.1.3 Content .............................................................................................................................. 52
    5.1.4 Properties ........................................................................................................................... 53
  5.2 Advanced Aspects .................................................................................................................. 57
    5.2.1 Ordered Collections .......................................................................................................... 57
    5.2.2 Property Queries ............................................................................................................... 58
    5.2.3 Unique ID .......................................................................................................................... 60
    5.2.4 Security ............................................................................................................................. 60
    5.2.5 Locking .............................................................................................................................. 63
    5.2.6 Events ............................................................................................................................... 64
  5.3 Expert Aspects ........................................................................................................................ 70
    5.3.1 Type Handling .................................................................................................................... 70
    5.3.2 Versioning .......................................................................................................................... 73
  5.4 Built-In Extensions and Additional Aspects ......................................................................... 77
    5.4.1 Repository Services ............................................................................................................ 77
    5.4.2 Global Services ................................................................................................................... 78
    5.4.3 Basic Services ..................................................................................................................... 80

© 2003 SAP AG
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Extending the RF</td>
<td>..........................................................................................................................</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>6.1</td>
<td>Runtime of RF Components</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.1.1</td>
<td>Configuration Framework</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.1.2</td>
<td>Component Runtime</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.1.3</td>
<td>Logging and Tracing</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.1.4</td>
<td>Clustered Installation</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2</td>
<td>Repository Manager</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2.1</td>
<td>Implementation Considerations</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2.2</td>
<td>Components of a Repository Manager</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2.3</td>
<td>Mapping RF Objects to Backend Objects</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2.4</td>
<td>RIDs, Lookup and Resource Context</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2.5</td>
<td>Namespace Manager</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2.6</td>
<td>Content Manager</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2.7</td>
<td>Property Manager</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2.8</td>
<td>Advanced Aspects</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.2.9</td>
<td>Expert Aspects</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.3</td>
<td>Repository Filter and Filter Manager</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.3.1</td>
<td>Namespace Filter</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.3.2</td>
<td>Property Filter</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.3.3</td>
<td>Content Filter</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.4</td>
<td>Repository Service</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>6.5</td>
<td>Global Service</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>References</td>
<td>..........................................................................................................................</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Index</td>
<td>..........................................................................................................................</td>
</tr>
</tbody>
</table>
1 Introduction

This section briefly describes the Repository Framework (RF) and it gives a brief roadmap of how the other sections are organized.

1.1 Audience and Organization of Sections

This documentation is targeted at developers – either developers who want to use the RF as basis for an application or developers who want to develop new extensions for the RF.

Good Java knowledge and a basic knowledge of Web technologies, SQL, and normal development processes is required. Knowledge of UML and SAP’s block diagrams is helpful. The reader should be familiar with SAP Enterprise Portal and the PDK, and know how to develop and deploy SAP Enterprise Portal services (see [EP Help], [PDK-Start] and [PDK-Services]).

Section 2 Explains the overall concepts of the RF and describes the metaphors used in the RF’s APIs.

Section 3 Depicts the main RF building blocks and how the RF’s API relates to this architecture. It also explains some general concepts of the RF-API and how the API relates to the concepts introduced in section 2.

Chapter 4 Gives a short overview of the utility interfaces included in the RF’s API.

Chapter 5 Shows how to use the RF API from a RF client’s perspective – that is usually how to implement applications on top of the RF.

Chapter 6 Shows how to use the RF API from a plug-in’s perspective – that is, how to implement extensions for the RF.

Application developers should read sections 2, 3 and 5, for an introduction to the RF and to using the RF client APIs.

Extension developers might read sections 2, 3 and 6 for an introduction to the RF and to see what kind of extensions can be developed for the RF and how they are implemented. Extension developers might also read section 5, if they want to use other parts of the RF from within an extension.
1.2 Motivation: The RF - What is it all about?

The Repository Framework (RF) is an extensible framework that offers applications unified access to objects provided by various information sources.

An information source (or repository) can be a “store” (for example, a document management system or a file system) that typically stores unstructured data such as text or graphics. It might also be a “backend system” (“backend” for short) (for example, a database or an ERP system) which typically contains structured data, like data records or business objects.

A repository is connected to the RF using its Repository Manager (RM). The repository manager is responsible for converting the repository’s internal representation of the stored information into the unified aspects of the RF and vice versa.

The unified access eases application development, because applications don’t need to care about the specific behaviors of the different information sources. For example, a workflow application based on RF will instantly be extended to any other kind of document or business objects, as soon as the RF is extended with a new repository – without requiring changes to any lines of code in the workflow application.

In turn, objects from information sources benefit from the applications on top of the RF. For example, when the RF is extended with a new repository, all objects exposed by this new repository can instantly be searched using the generic search engine – without the need for re-implementing such functionality in the repository itself.
1.3 **Scope of this Document: Views on the RF**

There are several ways of looking at the RF:

From an **end user**’s point of view, the RF is more or less transparent, since a user usually only uses the applications on top of the RF within SAP Enterprise Portal 6.0 to access the objects of the various information stores. The user’s guide can be found at [User Guide].

From an **administrator**’s point of view the RF along with its extensions and some of the applications using it, is a component of SAP Enterprise Portal 6.0 and requires some configuration in order to work properly. The administrator’s guide can be found at [Admin Guide].

**Application developer** use the RF client APIs or APIs of other applications, built on top of the RF, to build their applications.

**Extension developer** use the RF extension APIs to extend the RF with new repository managers or additional services. To build their extension, they might use existing features of the RF or existing applications by using the RF client API or other applications’ APIs as well.
2 RF Concepts

This section describes basic concepts, but doesn’t explain the corresponding APIs – the APIs will be described in section 5. It explains the metaphors used within the RF when dealing with documents and business objects on an abstract level.

A lot of the concepts described here are similar to, but not the same as, those of [WebDAV].

2.1 Basic Aspects – Namespace, Content and Properties

The basic aspects concern objects handled by the RF, how they are accessed, and what data they contain or provide.

2.1.1 RF Objects

Although the RF’s main task is now to provide unified access to all kind of documents and business objects, it might be a little bit interesting to see how the RF has evolved through time:

Originally, the RF was intended as a document access layer. The basic concepts were very similar to those of WebDAV as described in the relevant RFCs (see [RFC2291], [RFC2518], and [RFC3253] for more details about WebDAV). Its main task was to unify the attributes of documents, such as author or creation date. Unlike JNDI, where only naming is unified and all objects are treated as java.lang.Objects, some basic attributes of the documents were considered to be so essential, that they became part of the RF’s unified objects (for example, display name). Therefore, the RF needs and provides more knowledge about its objects than JNDI does.

This introduced the need for a distinction between content (unstructured data, the internal structure of which is not known to the RF), and (meta-) attributes (which are structured data and can be inspected by the RF) such as content size, author, or creation date.

Furthermore, it turned out that even business objects modelled as plain java.lang.Objects are not sufficient for unified access, since too much knowledge about those business objects had to be incorporated into the applications, thereby preventing generic functionalities from being implemented at framework level.

The RF was therefore extended to support business objects as well, as documents. To achieve this, “type casting” was introduced in order to convert RF objects into business objects and back again (see 2.3.1 for details).

The characteristics of the RF’s unified objects (resources and collections) are:

- Each resource represents a unified data object within the RF (like a file or directory in the file system or like a WebDAV resource).
- A collection is a special kind of resource that “contains” other resources (like a directory contains files or subdirectories in the file system, or like a WebDAV collection resource).
- Each resource has a unique, hierarchical name (like a file has a filename or like a WebDAV resource has a URI).
- Each resource can contain unstructured data (content), although a collection usually doesn’t have content (see below)
- Each resource can contain structured data (attributes or properties).
- Each resource can be converted to or from a business object (also called semantic object in the RF).
The term "resource" is used here in two ways: It refers to an RF object and the most basic unified aspect of an RF object. Most of the time it should be obvious from the context, whether "resource" refers to the entire object or just the unified aspect. Sometimes "plain resource" is used, to distinguish a resource that is not a collection (a "leaf resource") from a collection (a "node resource").

A "collection" refers to the aspect of being able to have children of RF objects. Other unified aspects of RF objects will be introduced in the following sections.

On the other hand, "semantic object" refers to the non-unified aspect of an RF object.

For readers unfamiliar with WebDAV, the easiest approach to RF objects (resources) is to consider them as files and directories in a file system.

Resources are stored in and retrieved from the repository they belong to. A repository can be seen as a mount point in a Unix file system, but unlike a mount point, repositories cannot be "mounted" anywhere in the hierarchy – they can only be "mounted" beneath the RF’s root node.

Like files, resources have a unique, hierarchical name (see 2.1.2), and might have content and additional attributes. However, unlike a file system, nodes might also have content (in a file system, directories do not have content). Most applications and repositories are not able to deal with this feature anyway.

A special feature of resources in the RF that distinguishes them from WebDAV resources or files is that they can be "casted" onto other objects. This is where the "file system" view is a little bit too document-centric for some applications. It is convenient for WebDAV-like scenarios, but it might not be sufficient for business objects. It might be easier for a business-object scenario to consider RF objects (resources) as database entries: The RID is the primary key, and the property values are similar to the column values of a result table. While the property names are the column titles, the content can be regarded as one special BLOB column (Note: the RID is not really a primary key, since it might change, although it does identify the object uniquely – see 2.1.2 and 2.2.3 on resource IDs).

Remember that as with the "file system" metaphor, the "database" metaphor highlights some features of the RF but disregards others.

→ see also: 5.1.1 and 5.3.1
2.1.2 RID and Resource Context

As mentioned, each unified RF object (resource) is identified by its unique, hierarchical ID. This Resource IDentifier (or RID for short) is something like a hybrid between a relative URI (the Unified Resource Identifier, as defined in [RFC2396], but without a “net_path” component and it is never escaped) and a filename (but with an URI-like “query” component).

More formally:

\[
\text{<RID>} ::= \text{empty} | \text{<pathsegment>} [ '?' <query> ]
\]
\[
\text{<pathsegment>} ::= \text{<name>} [ '/' <pathsegment> ]
\]
\[
\text{<name>} ::= \text{any text, not containing ‘/’ and ‘?’}
\]
\[
\text{<query>} ::= \text{empty} | \text{<parameter>} ['=' <value> ]
\]
\[
\text{<parameter>} ::= \text{<varname>} [ '=' <value> ]
\]
\[
\text{<varname>} ::= \text{any text, not containing ‘=’ and ‘?’}
\]
\[
\text{<value>} ::= \text{any text, not containing ‘?’}
\]

The root resource (denoted by ‘/’) is reserved for the RF itself. This folder contains all the repositories known in the RF, and all these repositories have their own unique ID (called “repository prefix”). All the top-level resources are in turn root resources of the repositories, for example, ‘/etc’ is the RID of the top-level folder of the etc-repository (which contains some system data).

Beyond such a top-level resource (a repository’s root resource), it is up to the particular repository to decide how the RIDs are built and hierarchically organized.

For example, if a fileserver share (\server\share) is configured within the RF as a repository with the prefix fileserver, a file on the fileserver share (for example, \server\share\folder\myDoc.doc) might be mapped by the repository using the following RID: ‘/fileserver/folder/myDoc.doc’.

The hierarchical namespace structure implies a parent-child relation between the resources identified by a RID’s path segments. In the figure above, /etc/oth is a child of /etc, while /etc is parent of /etc/oth.

The hierarchical namespace implies that the RIDs of the parents of a resource along the hierarchy’s way up to the root resource can be derived from the resource’s RID, because they are separated by the ‘/’.

The part of the RID before the last ‘/’ is therefore called the path of the resource, while the part after the last ‘/’ (without the optional query) is called the name of the resource.
The namespace of a specific repository might be restricted in the sense that paths or names have a limited length. Therefore repositories have to provide name info including, which restrictions apply to the RIDs and which characters are not allowed to be used in names.

RF objects that are able to have references to child resources (can be parents) are collections. Each "pathsegment" of a RID’s path must identify a collection.

RIDs can either be absolute (start with an '/') or relative. The ‘.’ denotes the current collection and ‘..’ denotes the parent collection, just like in a file system. Relative RIDs can only be used for certain operations (like copy or move). RIDs returned by the RF are usually absolute, except for link targets (see 2.1.5).

Each resource is identified by its “pathsegment” only, without the “query”. The query part should only be used for adding context information (see below) or to select variants in the future. If an RID passed to the RF for retrieving a resource had query parameters, the retrieved resource’s RID may differ from the RID that was initially passed to the RF.

The RID is only one part of information required for retrieving resources. The other part is the access context, which provides additional information: While the RID specifies which resource to retrieve the access context contains further information about who wants to retrieve the resource (which in turn might affect the how).

The access context contains at least the user (or, more generally, a principal) on whose behalf the application wants to retrieve the resource. Based on this additional information, a repository might carry out additional selections. For example, a repository that supports access control might check the user against the access control lists for the requested resource in order to determine whether access is allowed. Alternatively, the access context might contain information about the preferred language for the user, so the repository can select the proper version of an object according to the preferred language.

The access context provided by the RF’s client is also referred to as the resource context.

→ see also: 5.1.1

### 2.1.3 Creating, Copying, and Deleting

Like files and folders in a file system, resources and collections can be created, copied and deleted. Resources can only be created in a collection, which then becomes the parent for the new resource, while the newly created resource becomes a child of the collection in which it was created.

Resources can be copied to another RID, which then duplicates the data of the copied resource to the given RID.

Finally, resources can be deleted. Once a resource is deleted it is no longer accessible, and neither is its content and meta-data. If a code still holds a reference to the RF’s resource object and tries to use the resource after it has been deleted, the RF raises an error.

→ see also: 5.1.2
2.1.4 Renaming and Moving

Both, renaming and moving change the RID of a resource. From a client’s point of view, both operations are identical, but from a repository’s point of view, there is a slight difference between the two operations.

Renaming a resource only changes its name (the RID’s “name” part). It does not change the parent-child relation – the resource remains in its parent collection.

Moving a resource can change any part of its RID.

Moving and renaming might be considered a bit like copying the resource to a new resource RID and then deleting the resource with the old RID. However, there are some major differences between moving or renaming, and a copying/deleting: Copying/deleting creates a new resource with a new unique ID, while moving leaves the unique ID unchanged (see 2.2.3 for details about unique IDs). Copying/deleting also causes two events to be sent, which are not related to each other. Renaming or moving produces only one event (see 2.2.6 on events).

From an object oriented point of view, moving only changes a business object’s place in the hierarchy, while copying/deleting creates a new instance of a business object and releases the old business object instance.

Although moving a resource is usually faster than copying and deleting, it is usually slower than renaming, because additional steps have to be taken, especially if a resource is moved from one repository to another (which can not happen when renaming a resource).

→ see also: 5.1.2 and 5.2.3
2.1.5 Links

As almost every file system has some concept for (soft) links or shortcuts, this is also true for the RF. A link is just a reference to a real resource, the link target. A link that refers to a target that no longer exists is called a broken link.

The graphic above shows one link /repository/link1 that points to the collection /repository/folder/subfolder, and another link /repository/link2 that points to the resource /repository/folder/subfolder/file.

The third link /repository/link3 points to a resource in another repository – /other_repository/folder/other_file.

All these links point to resources within the RF, so their link type is “internal link”.

The fourth link, /other_repository/link4, points to a URL that refers to an external (web) resource. Because URLs (which are not RIDs) are not within the RF’s namespace this link’s link type is “external link”.

Note: Although URLs are not within the RF’s namespace, they can be mapped by the KM’s “Web Repository Manager” to the RF’s namespace.

With EP 6.0 SP1, flexible links were introduced as a new link-type. A flexible link is an internal link that “follows” its target dynamically. While a “normal” internal link becomes broken, if its target is moved or renamed, a “flexible” link does not. Flexible links are also referred to as dynamic links, while normal links are called static links. A flexible link’s target has to reside in the same repository as the link, the flexible link is deleted if the resource is moved to another repository.

→ see also: 5.1.1
2.1.6 Unstructured Data: Content

Just as a file might contain raw data, a resource can have **content** as its data. The content of a resource is represented as a `java.io.InputStream`. The RF does not consider the contents of this input stream – it is simply passed to the responsible repository. Because the RF doesn’t know anything about the content itself, the content is also referred to as **unstructured data** within the RF. As with files, additional information about the content exists. This content **meta-data**, which describes the content data, is similar to an HTTP MIME header:

- The **content length** that indicates how many bytes of data are in the content; or its set to –1 not known by the repository (see [RFC2616] - section 14.13).
- The **content type** that returns the MIME content type of the data, if available to the repository (for example, 'text/html' or text/plain; see [RFC2616] - section 14.17)
- The **content encoding** that was used to encode the content, if known to the repository (e.g. 'UTF8').
- The **content language** as specified in [RFC2616] - section 14.12 if known to the repository (for example, 'en')
- The **entity tag** (ETag), as specified in [RFC2616] - section 14.19, if known to the repository. It may either be a **strong ETag**, which changes, whenever the content of the resource changes; or a **weak ETag** (prefixed by an 'W/'), which only changes when the resource’s content changes in a "semantically significant" way (see [RFC2616], section 13.3.3).
- An **expires** field that specifies – if set or known by the repository – the timestamp at which the content will expire (for example, 'Thu, 10 Jul 2003 10:00:00'. See [RFC2616] - section 14.21).
- The **last modified** field that gives the last timestamp at which content or properties were modified (for example, 'Fri, 11 Jul 2003 11:00:00'. See [RFC2616] section 14.29).

→ see also: 5.1.3

2.1.7 Structured Data: Properties

Unstructured content and its meta-attributes, which describe only basic aspects of the content, are only useful for bulk data, as used in documents. Any application dealing with the content has to "know" about the content’s structure itself, because the RF doesn’t know about it and can not provide any additional information.

To support generic applications that retrieve information about a business object’s data structure dynamically and do not use "hard-coded" knowledge about these business objects, a mechanism for retrieving and storing **structured data** and some possibilities for introspection on these structures had to be introduced: The **properties**.

These properties should not be confused with the `java.util.Properties`. Properties in the RF are properties as defined by WebDAV (see [RFC2518]). Properties are name/value pairs, where the name of a property defines its identity.

A **property name** must use the XML namespace mechanism, which provides a **namespace** and a **local name**. Thus, a property name is a tuple (namespace, local name), where the namespace is optional and is similar to 'http://sapportals.com/xmlns/cm' or empty.
By convention, property names are usually displayed in the following form: `namespace`local name, for example, `{'http://sapportals.com/xmlns/cm}displayname`.

Property names are not hierarchical, so if there are two properties, for example, `'{x}A'` and `'{x}A/B'`, no relationship between these two properties is recognized.

Some properties contain data about data. This is why properties sometimes are sometimes referred to as resource meta-data. Within the RF, properties may be used, not only to provide data about the content, but also to provide additional data about the object itself. This is especially relevant for business objects.

Because properties are typed (see below), they are also referred to as structured data within the RF.

Properties are either “live” or “dead”:

- The syntax and semantics of **live properties** are checked by the repository. For example, the values of the “content length” property `'{http://sapportals.com/xmlns/cm}contentlength` for resources have to be long integers and contain the length of the resource’s content in bytes. Most of the live properties are computed by the repository and cannot be changed by the client application. When a client changes a live property, there are some side effects in the repository.
- **Dead properties** are not checked by the repository, other than for their type (see below). The repository simply stores them.

As mentioned, properties are typed, as variables are typed in most computer languages. The following **property types** are defined:

- **String**: Any sequence of characters. The RF enforces no limit on the length of such a string, but most repositories do have a limit.
- **Integer**: An integer value.
- **Long**: A long integer value.
- **Dateonly**: A date value, such as `java.sql.Date`.
- **Timeonly**: A time value, such as `java.sql.Time`.
- **Datetime**: A timestamp value (with date and time), such as `java.sql.Timestamp`.
- **Boolean**: A boolean value (either `false` or `true`).
- **XML**: Should be a character string, representing a well-formed XML fragment. However the RF does not currently check XML values to see whether they are well formed – in fact they are simply treated as large strings.

Like content has meta-data, a property might also have meta-data: The **Property attributes**. Like properties, some of these attributes are “live” and are maintained by the repositories. The live attributes known to the RF are the following flags:

- **Multivalued**: `true`, if the property value is a list of values with the given type.
- **Required**: `true`, if the property value is required to remain, if set. That is, if the property was once set, it cannot be deleted, although its value can be changed and new resources might be created without the property being set.
- **Readonly**: `true`, if the property value can not be changed.
- **Hidden**: `true`, if the property is for system use only and not intended to be displayed to users.
Another attribute is the **property description**, which is a localized description of the property or at least the property’s name, if no description exists.

Additional “dead” attributes can be added to a property (only additional attributes are be added to an XML element). They are simple key/value pairs, where both, key and value, are strings – like `java.util.Properties`.

Within the RF, there are two special properties for a resource:

One special property is the **display name** of a resource: It provides a “user friendly” name for the resource, whereas the RID’s name part is a technical name. While the RID’s main task is to identify the resource uniquely within the system, the purpose of the display name is to make a resource in a collection identifiable for users.

For example, a news item might have `/news/20030708/110907003.1.xml` as its RID, but could have “latest news from the RF (11:09:07)” as its display name.

The resource’s display name is optional and might depend on the language of the resource context. If the display name does not exist, the RID’s name part is usually taken as a fallback option. Display names might not be unique, even amongst the children of a collection.

The other special property is the **resource type**: Like the content’s MIME type, which specifies the content type, the resource specifies the type of resource. It provides a string value that specifies the resource type.

This type is used, for example, in the type handling mechanism (see 5.3.1) and the object type handler (see 5.4.2.3).

The resource type is optional and might not be set.

→ see also: 5.1.4
2.2 Advanced Aspects – Ordering, Property Search, ID-Mapping, Locking, Security, Events

2.2.1 Ordered Collections and Collators

When inserting or retrieving children to/from a collection, it is usually desirable to have them sorted in a certain order, especially when displaying them.

The RF offers methods to for retrieving the order mechanism of a collection, in order to enable client applications to find out, if and how a collection’s children will be sorted within the repository’s persistency.

There are three different order mechanism types in the RF:

- **None**: No order mechanism is supported by the repository for the list of children.
- **Server**: An internal order mechanism is applied by the repository itself to the list of children. A repository can support several server order mechanisms, for example, “by creation date” or “by content size”.
- **Manual**: The order of the collection’s children can be rearranged freely by the client.

Collections that support this mechanism are called ordered collections. A collection’s server order mechanism is not changeable – the sequence of children is always defined by the order mechanism’s specification. However, it might be possible to choose another server order mechanism for a collection.

Whenever a new child is created in the collection, its new position is automatically determined with respect to the ordered collection’s order mechanism.

An ordered collection with a manual order mechanism preserves the position of its child resources as defined by an application. These positions can be changed by the client application as desired, and new child resources can be inserted anywhere in the sequence of children.

In addition to these order mechanisms, which have to be supported by the repository’s persistency itself, another way of sorting has been introduced with EP 6.0 SP1: Collators allow the specification of an ordering clause, similar to a SQL ORDER BY clause on properties. Collators define a list of collator entries that in turn specify the name of the property to order by and whether to sort in ascending or descending order.

Unlike an order mechanism for a collection, collators affect the order for a retrieved list of resources they are applied to (for example, the children of a collection or the result of a property query). An order mechanism affects the order of children as they are created or changed in the repository’s persistency.

Note: Although the RF is now able to deal with collators, their functionality has not been made available to client applications yet. This functionality will be released at a later date.

→ see also: 5.2.1
2.2.2 Property Queries

In 2.1.1, the parent-child relation of collections and resources has been described as a way of navigating through the hierarchy of objects in a repository. For large hierarchies, especially when the client application is only interested in specific resources, it might be not very efficient, to retrieve all these parent-child relations and then select the relevant resources on the client’s side. Therefore, two mechanisms have been introduced within the RF:

The first is the property search mechanism, which works on the properties of a resource and is similar to a SQL WHERE clause on properties for resources. This search mechanism must not be confused with the TREX search engine, which works on entire resources and even enables content to be searched for embedded words, and so on. The property search relies on mechanisms provided by the repositories that hold the resources. The result of a property search is always up-to-date, while the update intervals of TREX might vary depending on how TREX is configured. However, with a property search, only properties can be used as selectors.

As with SQL, where a SELECT expression returns a list of records that match the SELECT expression’s WHERE clause, a RF property query expression returns a list of resources that match the given query expression.

A query expression is constructed with the help of a query builder and can look like the following:

```
( '{http://sapportals.com/xmlns/cm}contenttype' like '/%jpg'
  or '{http://sapportals.com/xmlns/cm}contenttype' like '/%gif'
) and '{http://sapportals.com/xmlns/cm}createdby' like 'admin'
```

The example above would return all resources, where the content type ends with ...jpg or ...gif and that were created by the admin user.

The sub-tree of a hierarchy to be searched in can be restricted by the following parameters:

- The depth specifies how many levels of parent-child relations are to be followed:
  - 0 returns only the start resource, if it matches the given query expression
  - 1 returns the start resource and all its children that match the given query expression
  - Infinite returns a list of matching resources for the entire hierarchy beyond the given start resource

- The size specifies the maximum size of the list returned. Only the first <size> matching resources are searched.

A property search does not follow links. This ensures that only resources in the specified sub-tree and from the same repository are found.

→ see also: 5.2.2
2.2.3 Unique ID

As explained in 2.1.2, RIDs belong to a hierarchical namespace. Due to this, some operations that change the hierarchy, especially moving and renaming, also change the RID of a resource.

If an application needs to access resources regardless of their new location, there are two options:

- Each application tracks all relevant actions within the RF and updates its information (for example, it updates the reference to the resource it is interested in) whenever a specific action (for example, a move) is performed (see 2.2.6 on Events).
- Another kind of ID is available for resources. This ID, unlike the RID, does not change when the resource is moved, but also uniquely identifies each resource. The same ID is not assigned to two resources, even if a new resource is created with the same RID as an already deleted resource. These IDs are therefore unique IDs.

The RF supports the second option in two ways:

- A repository might support unique IDs – this feature is optional. An application should be aware, that unique IDs might not be supported by a repository. The unique IDs of a repository are only unique within the repository. This means that only the tuple (repository, unique ID) is globally unique within the RF.
  
  Note: Although the RF defines the interface for dealing with unique IDs for repositories, this functionality is not currently available to client applications. It will be released at a later date.

- An application might use the URI-mapping service (see 5.4.2.6).

→ see also: 5.2.3
2.2.4 Security

To prevent sensitive information being shown to unauthorized users, some kind of security mechanism had to be made available.

When a user tries to carry out any action on a resource, the repository checks if he or she has the appropriate permissions for performing the requested action on the given resource.

The RF distinguishes the following permissions:

- For resources (leafs in the hierarchy) and collections:
  - Read content: Permission to read the content of the resource
  - Read properties: Permission to read the properties of the resource
  - Write content: Permission to write or update the content of the resource
  - Write properties: Permission to write, update, or delete properties of the resource
  - Delete: Permission to delete the resource

- For collections (nodes in the hierarchy) only:
  - List child nodes: Permission to retrieve the children of a collection
  - Create child node: Permission to create a child in a collection
  - Read node properties: Permission to read the collection’s properties or names
  - Write node properties: Permission to write the collection’s properties or names
  - Delete child node: Permission to delete a child from a collection

Usually, an application does not have to concern itself with the security of the RF, because each time it tries to access a resource (read or modify it), the repository manager or service in question checks the relevant permissions.

Only RF extensions, or those applications that offer users the possibility of changing the permissions, need further knowledge on how these permissions are checked by the relevant resource’s repository.

A widely used mechanism for assigning permissions to resources and users is the Access Control List (ACL). An ACL is associated to a resource and has at least one owner. Because owners are allowed to modify the ACL, they are also granted full control for the resource, since they would be able to assign that permission to themselves anyway.

In addition, an ACL can contain one or more Access Control Entries (ACEs). Each ACE stores at least a tuple (principal, permission), where “principal” is, for example a user, group, or role. ACEs can be positive or negative, where positive ACEs grant permissions to a user and negative ACEs deny permissions to a user. This extends the tuple to a quadruple (principal, permission, negativeflag, priority), since mixing negative and positive ACEs requires a priority to be defined, as several ACEs can match for any given user.

Although negative ACEs are supported within the RF, they are not currently used. This is because they proved rather confusing for users.

A permission is usually granted for a user on a resource, if he or she is either the owner of the ACL or if at least one (positive) ACE exists that matches the given user and permission for the given resource.

If a resource has no ACL assigned to it, the resource’s parent ACL is used. This process of inheritance continues until either an ACL is found or the root resource is reached. According to this mechanism, a resource inherits its ACL from its parent if it does not have an ACL on its own.
If an ACL exists, all permissions are implicitly denied to users who are not owners of the ACL. Only those users explicitly stated by an ACE are granted permission.

If no ACL exists, all permissions are implicitly granted to all users.

→ see also: 5.2.4

### 2.2.5 Locking

If two users were to modify the same resource at the same time, this would lead to the “lost update problem”: User A edits resource $R$, while user B does the same and saves $R$, than A saves $R$ → the updates of user B for $R$ are lost.

Therefore a mechanism for serializing access is required:

The RF offers **locks** in order to achieve serialization: An application issues a lock request for a resource and user. If there is no other **blocking lock** (for another user) on this resource, the application can obtain the lock. If a blocking lock already exists for the resource (that is, for another user), the request is denied. Furthermore, access of a particular kind (write or read) is only allowed for users who obtain the relevant lock.

There are three important aspects that affect the behavior of locks:

#### Scope

- **Shared**: Shared locks allow several users to obtain a lock, but block exclusive locks and prevent any other non-locked access.
- **Exclusive**: An exclusive lock allows access for only one user and blocks any other user from obtaining a lock. It also prevents any other non-locked access.

#### Kind

- **Write**: A write lock blocks any other write lock and prevents any non-locked write operations whilst allowing further read locks. Write operations are any change of content, properties, or namespace.
- **Read**: A read lock allows further read locks to be obtained, but blocks any write locks and prevents any non-locked read operations. Read operations are the retrieval of resources, their properties, or content. *Read locks are not currently used within the RF!*

#### Depth

- **Shallow**: A shallow lock affects only the resource it was issued for.
- **Deep**: A deep lock affects all resources beneath the locked resource in the hierarchy. Linked resources (the link targets) are not affected by the lock as long as they are outside the locked resource’s hierarchy.

For example, a shallow, exclusive write lock prevents any other modifying access to the resource and blocks any other write lock from being obtained whilst allowing further read locks.

If a collection is locked, namespace operations are also considered as locked operations. In other words, in the given example, creating, deleting, moving or renaming would also be blocked.

The scope of a lock controls how many users can request locks for a resource, while the kind determines the intended type of operation and specifies, how many locks of this kind are permitted.
currently, only write locks are used in the RF.

locks can also have a **lock timeout**. when this timeout expires, the lock is automatically unlocked. this can be very useful in a distributed web scenario, if a remote application has locked a resource but cannot release it because the remote application’s connection to the server has become unavailable.

once an application has obtained a lock for a resource, it must not lock the resource again. therefore, methods are provided to check whether a lock already exists for a resource and the user to which it is assigned. if it is a lock with a timeout, the application should refresh the lock instead of relocking the resource. this extends the lock’s timeout.

→ see also: 5.2.5

**2.2.6 Events**

as shown in 2.2.3, some applications might need to keep track on changes to ‘their’ resources. to support this requirement, the RF offers several **events** for resources.

although the RF generates these events, which it defines (see below), clients cannot rely on events being supported for every resource. this is because events might be disabled (in the configuration) or as a result of some operations in the backend system that are neither routed through the RF nor reported by the repository.

in order to receive events, a client must first register itself as an **event receiver** with the **event broker** of the relevant resource’s repository. when a client registers, it can choose from the following two modes:

- **Synchronous**: the event receiver is called back while a operation is performed for a resource.
  
  the event, indicating the operation, is sent as a pre- and a post-event: the pre-event is sent before the operation is about to be performed and the post-event is sent when the operation has been performed successfully.
  
  synchronous event receivers must take performance into account, since the repository blocks any other operation on the resource while the operation (and the processing of the events) is in progress.
- **Asynchronous**: The event receiver is called back from a separate thread, which buffers events in an event queue.

  The event receiver might perform lengthy operations when receiving an event for an operation, because the repository is able to finish processing the operation without waiting for the event receiver to return from the call.

  On the other hand, the event receiver cannot rely on pre-events, because they might be received after the operation has been performed.
The following types of events are defined within the RF for listed operations:

- **Retrieval operations**
  - Get: Retrieving the content of a resource
  - Get children: Retrieving the list of children of a collection

- **Content and property operations**
  - Set: Changing the content of a resource.
  - Property get: Retrieving a resource property
  - Property set: Creating or updating a resource property
  - Property delete: Deleting a resource property

- **Namespace operations**
  - Create child: Creating a new child resource inside a collection (which is neither a link nor a collection).
  - Create collection: Creating a new sub collection inside a collection
  - Create link: Creating a new link inside a collection
  - Delete: Deleting a resource
  - Copy: Copying a resource
  - Move: Moving a resource
  - Rename: Renaming a resource

- **Locking operations**
  - Lock: Locking a resource
  - Unlock: Unlocking a resource

- **Version control operations** (see 2.3.2):
  - Enable versioning: Turning on version control for a collection
  - Checkout: Checking out a resource
  - Checkin: Checking in a resource
  - Undo checkout: Reverting the checkout of a resource

The events are sent as **pre-events**, before an operation starts and as **post-events**, after the operation completed. If the operation fails, no post-event is sent. To identify events that belong together (for example, pairs of pre- and post-events), events can provide a **correlation id**.

As well as its type, which defines the operation causing the event, the event contains the resource to which the operation was applied to.

A client that uses the resource as passed in the event must be aware that some operations invalidate the resource – namely a delete operation destroys the resource. If the resource has been invalidated, only its RID can be retrieved.

→ see also: 5.2.6
2.3 Expert Aspects – Semantic Objects and Versioning

2.3.1 Type Handling

When the RF matured from a document framework to a business object framework, a problem arose. Two aspects of business objects had to be covered:

- The unified aspects, such as the unified identifier (RID) and unified handling (through a unified namespace concept and unified operations).
- The semantic aspects, those specific to the object and that are necessary for application dealing with business objects.

Something like a type cast mechanism was therefore needed within the RF to “cast” a resource (which represents the unified aspects of an business object) to any type of semantic object (which represents the application specific aspects of an business object) and vice versa.

It might be too expensive for a repository, to expose its business objects as java objects, which implement all potentially relevant semantic objects they are able to represent. Because of this, no usual Java type cast could be used.

Therefore, a piece of code is needed to convert a resource into a specific semantic object (that is, to convert the internal handle of a repository to the real business object). This “converter” or object factory encapsulates the knowledge of how to convert a given resource into a specific semantic object. It simply de-serializes the specific semantic object from the resource (for example, using properties or the resource’s content). The object factories have to register themselves with the RF during start-up in a semantic object factory registry, so that the RF is aware of all available object factories (and therefore of all semantic objects into which a resource can be converted).

The reverse method of casting a semantic object back to a resource is not yet available in the RF, but will be made available in the future. In a very similar way to JNDI, where object factories are used to retrieve objects from the persistency and state factories are used to store objects into the persistency, a state factory will be used in the RF to serialize a semantic object into the properties or content of a resource. The state factory encapsulates the knowledge of how to convert a specific semantic object into a resource.

→ see also: 5.3.1
2.3.2 Versioning

This section describes the several versioning concepts within the RF. These are similar to those of WebDAV as described in [RFC3253].

2.3.2.1 Version Control

When a resource is updated, either by editing its content or changing a “dead” property value, the old content or property value is lost.

When resources are placed under version control, those old values are preserved when the resource is updated. As namespace operations (see 2.1.3 and 2.1.4) change the RID, changes of content or “dead” properties change the version controlled state of the resource. The version controlled state is therefore at least all content (structured and unstructured) of the resource – versioning allows the preservation of snapshots of these states over time.

A versionable resource is a resource that can be placed under version control. When the resource is placed under version control (when version control is enabled for this resource), it becomes a version controlled resource (or VCR for short) with the initial check-out state “checked-in”. In addition to this, an initial version resource (also called revision or version) is created with a new RID.

Only checked-out VCRs can be changed. When a VCR is checked-out, the VCR’s check-out state is changed to “checked-out” and the current version controlled state of the VCR is marked as the checked-out version.

After applying content or property changes, the checked-out resource has to be checked-in back again. This causes the VCRs check-out state to revert to “checked-in” and creates a new version from the version controlled state of the checked-in resource: Every time a checked-out resource is checked-in, version control creates a new version of the resource with its own RID – the behavior for how these RIDs are generated is repository-specific.
While each subsequent check-in generates a new version, a **version history** is generated over time. In the figure above, a linear version history is shown, where v1 has been edited and became v2, then v3 and the current version is v4. The first version, which started the version history (v1 in the example) and was created when version control was enabled for the resource, is also called the **root version**. As well as the version history, the VCR also supplies the content and the properties of the **current version**, which is the version controlled state of the latest checked-in version (unless not updated from another version).

When a VCR is checked-in, this checked-in VCR's version controlled state becomes the **successor** of the checked-out version and the formerly checked-out version becomes the **predecessor** of the version created on check-in of the VCR (that is, v3 is the successor of v2 and v2 is the predecessor of v3). An **ancestor** of one version (for example, v3) is any version that is connected to this one version by one or more successor relations (that is, v1 and v2 are ancestors of v3). Conversely, **descendants** of one version are versions that are connected to this one version by one or more predecessor relations (that is, v3 and v4 are descendants of v2).

A version history with each version having at most one successor (or no successor for the current version) is a linear version history, as shown in the example above.

When a second successor is added to a version, this creates a **fork** or **branch** in the version history – v3 and v4 are both successors to v2 in the example below.

![Diagram of version history](image)

If a version is created with more than one predecessor, this creates a **merge** in the version history, that is, v7 is merged from v5 and v6.

Both branching and merging are described in further detail below (see 2.3.2.3).

Versions can be tagged with **labels** in order to organize them. Versions in different version histories of different VCRs can be tagged with the same identifier. This can be used, to group sets of versions across different root versions (see 2.3.2.4).
2.3.2.2 Basic Versioning

The RF basic versioning supports linear version histories with in-place check-out for non-collection resources only.

When the client requests an in-place check-out on the VCR, the VCR’s check-out state simply changes to “checked-out”. The current checked-in version becomes the checked-out version of the VCR and might be updated by the client.

The repository might internally generate a RID for the version, which would be created on check-in in advance. Some repositories might support this expected RID being returned to the client, but this function is not supported by all repositories.

The client then changes the content and/or the properties of the checked-out VCR resource “in-place”. When completed, the client requests the check-in on the VCR. This causes a new version being to be created from the current version controlled state of the edited resource (its content and dead properties).

If no other version has been created in the meantime (see below), the expected RID – if generated on check-out – is used for the new version and the VCR’s check-out state is set to “checked-in” again.

If the user did not lock the VCR (see 2.2.5 on locking), it might be possible for a second user to edit the resource, check in the changes in and then check the resource out again. If the (first) user edits the resource then and checks it in (after the second user has already performed a check-in/check-out), the version is a successor of the version created on the second users check-in. To avoid such a situation, the version’s expected RID can be used, if it is supported by the repository.

→ see also: 5.3.2
### 2.3.2.3 Advanced Versioning: Working Resources

A **working resource** can be thought of as a temporary resource on the server’s side (in the backend), whereas with in-place check-out the client has to maintain the content and properties locally.

When the client requests the check-out of a working resource, an “intermediate” VCR for this version is created, which is marked as “checked-out”. The client then edits this “intermediate” VCR (with the “intermediate” VCR being the working resource – it does not edit the “original” VCR as with in-place check-out).

When the client requests the check-in on the checked-out “intermediate” VCR, the new successor of this version is created from the version controlled state of the “intermediate” VCR and then the “intermediate” VCR is released.

Working resources support a non-linear version history and enable users to work on different versions in parallel.

A branching scenario might then look like this:
While the first check-out/check-in (check-out 1 and check-in 1) creates a new version (v21) as described above, the second check-out (check-out 2) also refers to v1. When VCR2 is checked-in again, the new version (v22) is created as another successor of v1.

Branching can also be achieved explicitly, by checking out a specific version (not the current version as in the example above). This feature is useful if a given version has to be maintained while regular changes continue. Consider some development source code: When a code version is released to the customer, one branch might be used for bug fixes only, while another branch might be used for normal development and code enhancements.
2.3.2.4 Advanced Versioning: Workspaces

As explained above, the repository generates the RID of a checked-out resource internally – the client is not able to control the namespace. This might be irrelevant for a single resource, but if a hierarchy of resources is to be checked-out by the client, it is useful to preserve the namespace of such a hierarchy with the checked-out resources, too.

Therefore, the workspace concept was introduced: A workspace is a collection and each resource belonging to the workspace is a workspace controlled resource (WCR for short). A workspace is a bit like a temporary folder on the server, in which the client can store resources.

When the client requests that a workspace be created, it supplies a name for the workspace – the workspace is then created on the server side with a server generated RID. The client can then create non-versioned resources and collections as in a normal collection and can also create VCRs for specific versions in that workspace. The client is therefore able to control the RID of the working resources, when creating a version controlled resource (VCR) as a working resource inside the workspace.

There is one major restriction for working resources within a workspace: There are no VCRs that share their version history. In other words: a workspace does not contain VCRs for different versions of the same resource. Each version referenced by a VCR inside a workspace belongs to a different version history.

This feature is useful – especially when used with labels – if a set of versioned resources that are related to each other, has to be changed. A label can be used by the client to identify the versions that belong together, for example, by tagging a specific list of changes (change list).

The picture shows three resources (R1, R2 and R3) in the resource hierarchy with their versions. When checking out the versions of those resources, a client might use a workspace, to preserve the hierarchy (by creating parent collections for the three versions to check-out in the workspace). A label (CL4711) identifies the versions that belong together on the time line (R1v5, R2v4, R3v2).
### 2.3.2.5 Advanced Versioning: Auto-Versioning and Automatic Version Control

Collections can support **auto-versioning**: If enabled, any change to the children, which are VCRs, implies that a new version is created automatically for the VCR – without a check-out/check-in request issued by the client (so versioning-unaware clients will also create new versions for versioned resources).

Different from auto-versioning is **automatic version control**: A collection with this feature enabled automatically places each newly created plain child-resource under version control.

### 2.3.2.6 Advanced Versioning: Version Controlled Collections

While the state of a plain resource usually consists of its content and dead properties, the state of a collection also includes the local names of the collection’s children (and their ordering, if it is an ordered collection as described in 2.2.1).

Therefore, if version control is enabled on a collection, a version of a **version controlled collection** (VCC, or versioned collection) also preserves its children’s names (with ordering) – in addition to the dead properties and content. In other words: the version controlled state of a VCC also contains its children’s names and ordering information.

In the case of the list of children’s names of a VCC’s version, only the versioned children’s names (which are VCR’s) are saved. The non-versioned children are listed as children, but their names are not stored with each version of the VCC.

The graphic above shows a version controlled collection C, which initially contains R3 as a non-versioned child and R2 as a versioned child.

Then a new version controlled resource, R1, is created. C has to be checked out first in order to create R1. Thereafter, C is checked in again.

If a version for a VCC’s child VCR (R1 or R2) is created, no version is created for the VCC. Als, if a new non-versioned resource (R4) is created, no new version is created for the VCC.
This results in the following versions for C:

- C version 1 contains R2 and R3 as children
- C version 2 contains R1, R2 and R3 as children

C version 2 (Cv2') contains R1, R2, R3 and R4 as children, after R4 has been created, but it is not a new version.

Because the names of non-VCR children are not saved in versions of the VCC, the following situation can occur: If a versioned resource once existed for a given RID as a child of the collection, but was then deleted, and if finally a new resource with the same RID (without version control) is created, the once versioned resource is **eclipsed** by the newer, not-versioned resource.

In the graphic above, consider that the last version of R1 (R1v2) has been deleted and a new resource R1', with the same RID as R1, has been recreated; R1' now eclipses R1.

As depicted in the graphic above, only the references to the names of its child-VCRs are saved as the VCC’s version controlled state. It is therefore it’s not possible to preserve which versions of the VCC’s child-VCRs belong to a VCC’s version: Consider that Cv2 originally referenced R1 with R1v1 as the current version, R2 with R1v1 as current version, and R3, whereas Cv3 references R1 with R1v2 as current version instead, since a new version for R1 has been checked in. If C’s current version state were now be retrieved from Cv2, it would still reference R1 with R1v2 (as well as R2v1 and R3 of course), because Cv2 does not contain the information that R1 referenced R1v1 when Cv2 was created.

Such a **baseline** feature is not yet supported by the RF.
3 RF Extensions, API Layers and Concepts

This section focuses on the main RF building blocks and the general concepts used in the RF’s API. In addition, it provides a brief overview utility APIs that are offered by the RF but not directly related to the concepts that were introduced in section 2.

3.1 RF Extensions

From an application point of view, the RF offers access to the resources stored in the information sources by unified methods. Built-in or plugged-in extensions offer additional aspects on the unified RF resources. Some other plug-ins enable the casting of unified resources into objects that expose semantically specialized aspects of an information object to the application.

Internally, the RF’s core consists of various registries, where the different extensions are plugged-in.
There are the following types of RF extensions:

- **Repository Managers**: Handle the mapping between the RF objects and the backend system (information source). For example, the “File System Repository Manager” maps files and directories of the file system to resources and collections.

- **Repository Filters** and **Repository Filter Managers**: Repository filters allow the manipulation of RF objects as they are passed through the RF, whereas a repository filter manager is responsible for determining, when the filter has to be applied. For example, the “HTML Stylesheet Filter” adds a stylesheet-link for a given CSS to those resources with the content type `text/html`.

- **Repository Services**: Offer additional unified aspects for the resources of a specific repository. A repository service is related to a repository and can also communicate with the repository manager’s backend. For example, the “Application Properties Service” offers additional properties that can be stored along with a resource. An application can use these properties, to save additional data for that resource, such as a timestamp that indicates the application-specific lifetime of a resource.

- **Global Services**: Offer additional unified functionality for all RF resources. For example, the “Relation Service” enable the storage and retrieval of relations between resources, which an application might use to save interdependency information about resources, such as “document X is attached to document Y”.

- **Semantic Objects** and **Semantic Object Providers**: Semantic objects represent special aspects of the resource objects, whereas the semantic object providers handle the conversion of resource objects into semantic objects. For example, a semantic object provider is able to convert resources of a specific type into the semantic object “Team Room”.

There are other types of extensions that do not belong to the RF itself, but are offered by the built-in extensions of the RF:

- **Extensions for Extensions**: If an RF extension offer other applications to extend them, for example, the scheduler service supports `ISchedulerTasks` as its plug-in, which is run by the scheduler (see 5.4.3.4)
3.2 API Layers, Current and New API

The RF is built in several layers and relies on other software layers, as shown in the graphic below:

For historical reasons, two flavours of the RF API layer exist – the old and the new API:

- The current API, as introduced in EP 5.0, is not grouped into specific sections: com.sapportals.wcm.repository.* and com.sapportals.wcm.service.*

- The new API (currently only to be used for Repository Managers), as introduced with EP 6.0, is grouped into three major sections:
  - Common interfaces that are relevant to both, clients and repositories: com.sap.netweaver.bc.rf.common.*
  - Client interfaces that expose the interface for the applications that use the framework (this part of the API is not released yet, see below): com.sap.netweaver.bc.rf.ci.*
  - Repository manager interfaces that define the API for the repository managers: com.sap.netweaver.bc.rf.mi.*

The current API’s client interfaces (on the left-hand side) specify the current interfaces as used by RF applications. These interfaces are described in section 5 in further detail. The new API’s client interfaces (in the middle) will probably not be released before EP 6.0 SP3 (maybe even after SP3) and are not yet covered by this document.

➤ The current API is used for the development of applications and all RF extensions except repository managers!
The new API's repository manager interfaces (on the right-hand side), and the common interfaces used by them, are now released in EP 6.0 SP1 for restricted use. They are not yet finalized – small changes (such as adding methods for mass operations in order to increase performance) might still be applied to these APIs. The transition for the utility interfaces is also not yet completed. The new API will be released in SP2 for unrestricted availability.

The new API is currently (EP 6.0 SP1) used for the development of repository managers only!

The layer of the additional RF libraries includes some utility packages that are used by the RF. It also contains the component runtime. This layer does not deal with RF objects. However, since the RF uses this layer, and because it is developed by the RF team, it became an additional part of the RF API.

- **Utilities** (current API) and **utility interfaces** (new API) are split according to the RF API parts that are using them (see RF layer above). Their structure is described in section 4.
  - Current API: com.sapportals.wcm.util.*
  - New API: com.sap.netweaver.bc.rf.util.*

- The **component runtime (CRT)** manages the lifecycle of the several RF components (like repositories, services or repository filters, see section 6). Its APIs was already released in EP 5.0. Documentation is available in [CRT]: com.sapportals.wcm.crt.*

The layer with other frameworks and utilities contains:

- **Logging and Tracing**
  - com.sap.tc.logging.*

- **Configuration Framework**
  - com.sapportals.config.*

- **Exception Framework**
  - com.sap.exception.*

- **Usermanagement**
  - com.sapportals.portal.security.usermanagement.* for the old usermanagement
  - and com.sap.security.api.* for new usermanagement

Documentation for these components is available at [Logging], [CF], [Exceptions] and [Security].
3.3 (New) API Concepts

Most of the concepts described in this section apply to the new API only. Since the new API’s client interfaces are not released yet, this section might be most interesting for developers who want to develop a repository manager.

3.3.1 Aspects of RF objects in the API

Section 2 described the several aspects of the RF objects. These are

- Namespace
- Content
- Properties
- Property Search
- ID Mapping
- Locking
- Security
- Types
- Versioning

The new API reflects these aspects: The packages for common
coma.sap.netweaver.bc.rf.common, manager (com.sap.netweaver.bc.rf.mi) and client
coma.sap.netweaver.bc.rf.ci) interfaces are grouped accordingly:

- namespace (includes property search)
- content
- properties
- idmapper
- lock
- security
- type
- version

3.3.2 Supported Options, Read-Only and “Mutable” Interfaces

Not all the operations of an aspect are applicable for all RF objects. Some aspects or operations of an aspect might be only applicable for RF objects of a specific repository or only for specific RF objects. For example, files on a CD-ROM drive might only be exposed as read-only by a repository manager. Or it might be that a file cannot be placed under version control because the repository manager does not support it. It is also possible that a file system repository might not support content operations for collections and links.

To reflect these restrictions, supported options are used: Repository managers report the options that they do support to the RF.
In the current API, the possible options are defined in a single SupportedOption enumeration in the package com.sapportals.wcm.framework.enum. With the new API, the SupportedOption classes are separated for each aspect and are therefore located in each aspect com.sap.netweaver.bc.rf.common package (for example, com.sap.netweaver.bc.rf.common.namespace.SupportedOption for supported namespace operations).

If an operation that is not supported by the repository manager is requested on a resource, the repository manager throws aNotSupportedException (in the current API) or an OperationNotSupportedException (in the new API).

Although clients can retrieve the list of a resource’s supported options, this list does not necessarily reflect the list of unsupported operations for the client, because the RF might map certain operations that are not supported by a repository itself to its internal default implementation of the operation (for example, if a repository does not support the copy operation, the RF maps it to an internal implementation that uses the create operation instead).

Within the new API, all operations are additionally grouped as read-only or read-write in separate interfaces. If the read-only operations are located in an interface, their corresponding read-write operations are located in a mutable interface (prefixed by Mutable).

For example, operations for reading the content of a resource from a repository manager are located in the interface IContentManager, whereas the operations for writing content are located in IMutableContentManager.

3.3.3 Exceptions

The new API uses the SAP exception framework as the foundation for its exception handling. Further documentation on the SAP exception framework can be found at [Exceptions].

The RF’s new API offers one central exception:

com.sap.netweaver.bc.rf.common.exception.RepositoryException.

Two exceptions are derived from this exception:

- OperationNotCompletedException, which is used for multi status results (see 3.3.6)
- ResourceException, which is the base class of all RF exceptions in the current API.

All other exceptions used throughout the new API are derived from this ResourceException.

Exceptions that are not specialized for one aspect belong to the common exception package (com.sap.netweaver.bc.rf.common.exception) and can be found there. Exceptions specialized for one aspect (for example, a WrongVersionException is only used by the version package), are placed in the aspect’s package.
The most frequently used exceptions in the RF are:

- **Current: NotSupportedException / new: OperationNotSupportedException:** Method is not supported at all, for example `enableVersioning()` is called, but the repository does not support versioning.

- **MethodNotAllowedException**
  Although the method is supported in general, it is not supported for the given resource, for example, `setContent()` is not supported for a collection, but it is supported for a plain resource.

- **Current: IOException / new: IOOperationFailedException:** A problem with the connection to the backend system, for example, a broken connection.

- **ResourceNotFoundException:**
  The resource specified does not exist in the backend.
  Methods for retrieval, that is, `getResource()` do not throw this exception but return `null` instead (if the resource is not found) or throw an IOException to indicate a problem with the backend system.

- **AccessDeniedException:**
  The user in the context is not authorized to perform the requested operation, for example, if a user only has read permission but requests to update the content, the permission is denied and the method `updateContent()` throws an AccessDeniedException.

### 3.3.4 Descriptors

When the RF’s new API was designed, implementing the operations for all those aspects of RF objects would have led to multiple overloads of the RF object’s operations. For example, the create operation must be made available for plain resources, collections, links, versions, and so on.

In order to avoid a large number of overloaded methods that would differ only in certain details and in order to keep a lean API, descriptors were used instead of overloading. Descriptors group coherent parameter blocks together in one reusable unit. In the example above for the create operation, there are appropriate descriptors for creating a resource, collection, link or version – but only one create method.

The “overloading” is now carried out in the various constructors for the descriptors. This allows the existing API to remain unchanged when new aspects are introduced. Only the new aspect and its new descriptors have to be introduced in the API.

In the current API, descriptors are not used consistently; they are used for certain operations (for example, as ICopyParameter for the copy operation).

Despite its advantages, this solution has its drawbacks: A descriptor is an additional object that needs to be allocated and removed from the memory, if no longer referenced by the garbage collection. To minimize housekeeping efforts, descriptor objects should be reused where possible.

Therefore, two simple rules should be adhered to when using descriptors:

1. Create as few descriptors as possible. Do this by pooling and modifying the ones you create.
2. Do not do this for those descriptors you didn’t create. In particular, do not modify them.
3.3.5 Typed vs. Un-Typed Collections and Iterators

Within the current API, typed collections and iterators have been used where possible. For example, the returned list of a collection’s getChildren() method call is an IResourceList. The corresponding Iterator for such an IResourceList is an IResourceListIterator.

With the new API, un-typed collections and iterators from the standard JDK API are used instead in order to reduce the number of classes in the RF API. For example, a collection’s getChildren() method returns a java.util.List.

3.3.6 Mass Calls and Multi Status

For performance reasons, mass calls are also included in the RF. Mass calls enable clients to apply an operation on several resources at once, instead of applying the operation in a loop, for example, retrieving a specific property for a list of resources.

Applying the same operation with one mass call might be more efficient, than executing several calls one after the other, especially when sending a request to a remote backend system.

Despite the improved performance, the only special feature of mass calls is the error handling: When an error occurs (for example, if retrieving the requested property causes an error for one specific resource), the mass call continues trying to apply the operation on the remaining resources (that is, tries to retrieve the requested property for the remaining resources).

When the mass call completes without errors, it returns a list of results (usually a list of items or a map, using resource handles as keys and the corresponding return value of the (single call) operation for the key resource as values).

In the current API, mass calls were only used for setting properties (or for calls from the RF to the repository managers – see 6). If an error occurred (when several properties were set), a special exception with a list of results is returned (SetPropertiesException in package com.sapportals.wcm.repository.).

If an error occurred in the new API, an OperationNotCompletedException is thrown (package com.sap.netweaver.bc.rf.common.exception). This exception includes a multi status error result that consists of:

- A list of ResourceExceptions for the failed resources, to which the operation could not be applied (getThrowables()). The ResourceExceptions contains the RID of the resource (getRID()) and the Throwable (getCause()) that caused the error.

- The result for the processed resources, to which the operation was successfully applied (getPartiallyComputedResult()).
3.3.7 Atomicity and Mass Calls

The RF defines all API calls as atomic, as long as it is not explicitly stated otherwise. The mass calls are not atomic, which is also explicitly stated in the Java documentation.

All API calls must be atomic on their work units, which are – as per definition – atomic. A work unit is, for example, an operation that is applied only to one resource handle or property only.

Therefore, all API calls that process several work units non-atomically (for example, mass calls), have to report both, the committed work units and the rolled back work units (as, for example, mass calls do with the multi status exception).

All independently executable work units (where a previous work unit is not required for a following work unit), are processed. For example, when a mass call applies an operation to several resources, this results in independent work units (one for each resource), all of which are processed.

3.3.8 Interfaces vs. Classes

As is usual in Java, most of the RF’s definitions are interfaces. For most of the RF interfaces, a default implementation class is provided for the corresponding interface, for example, Content for IContent in the com.sap.netweaver.bc.rf.common.content package.

Only Exceptions, enumerations and some parameter structures (for example, Position in package com.sap.netweaver.bc.rf.common.namespace for the definition of a resource’s position in an ordered collection) have no corresponding interfaces.

Neither RF applications nor RF extensions should work on a special implementation for an RF interface only. For example a repository’s content submanager (see 6.2.6) should not check for its own Content implementation on setContent(), because the client might have used the default implementation or its own.

Exceptions to this rule are:

- The repository manager’s implementation of IResourceHandle (see 6.2.3)
- The repository manager’s implementation of IQueryExpressions (see 6.2.8.3)
4 Utility Packages

4.1 Current API: com.sapportals.wcm.util

The following three lists provide a brief overview of the packages beneath com.sapportals.wcm.util and the interfaces and classes they provide:

The first group consists of packages that are related to the RF:

- **com.sapportals.wcm.util.content**
  The IContent interface represents the (unstructured) content of a resource as a java.io.InputStream.
  ➔ New API: com.sap.netweaver.bc.rf.common.content

- **com.sapportals.wcm.util.enum**
  AbstractEnum is the base class for all enumerations of RF constants in the current API.
  ➔ New API: com.sap.netweaver.bc.rf.util.enum

- **com.sapportals.wcm.util.events**
  Foundation for the eventing mechanism (see 2.2.6).
  ➔ New API: com.sap.netweaver.bc.rf.util.event

- **com.sapportals.wcm.util.logging**
  Provides LoggingFormatter.extractCallstack() for extracting an exception’s call stack for tracing.
  ➔ New API: com.sap.netweaver.bc.rf.util.logging

- **com.sapportals.wcm.util.name**
  Interface IName and class Name for handling an XML (namespace, name) naming scheme (see 2.1.7 and [RFC2518]).
  ➔ New API: com.sap.netweaver.bc.rf.util.namespace

- **com.sapportals.wcm.util.resource**
  Offers ResourceBundles that extend java.util.ResourceBundle for multi language support and Descriptions that support the handling of descriptions in several languages.
  ➔ New API: com.sap.netweaver.bc.rf.util.resource

- **com.sapportals.wcm.util.uri**
  Classes for handling RIDs (see 2.1.2), URIs, and URLs (see [RFC2396]).
  ➔ New API: com.sap.netweaver.bc.rf.common

- **com.sapportals.wcm.util.uuid**
  Offers UUID for generating Universally Unique Identifiers (UUIDs) as specified by the Internet-Draft “A UUID URN Namespace” (see IETF-UUID).
  ➔ New API: com.sap.netweaver.bc.rf.util.uuid
The next group of packages are those that are not related to the RF. These packages will not be incorporated into the new API. Most of these packages deal with content and string handling:

- **com.sapportals.wcm.util.base64**
  Base64Encoder for encoding and Base64Decoder for decoding a stream or string into Base64 format, used for MIME data.

- **com.sapportals.wcm.util.html**
  Several classes for parsing HTML content.

- **com.sapportals.wcm.util.http**
  Definitions for client side HTTP access and a tiny implementation in com.sapportals.wcm.http.slim.

- **com.sapportals.wcm.util.mmparser**
  Classes for handling multipart MIME messages.

- **com.sapportals.wcm.util.regex**
  Provides two pattern matchers:
  - PathPatternMatcher matches for expressions that use an “Ant” like syntax (see [ANT]).
  - Matcher matches for regular expressions that are represented as Pattern.

- **com.sapportals.wcm.util.threads**
  Interfaces and classes for handling semaphores and reader-writer locks.

- **com.sapportals.wcm.util.xml**
  A wrapper for XML parsers and an XML serializer.
The third group of packages are implementations of features that should only be used internally. They should not be used by application developers for the following reasons: Either they provide basic functions, making it likely that either the portal runtime or the J2EE Engine will provide the same functionality in the future, or the functionality is relevant for RF extensions only, or the package is already deprecated.

- **com.sapportals.wcm.util.acl**
  Will be provided by the portal’s user management.

- **com.sapportals.wcm.util.cache**
  Will be provided by the J2EE Engine.

- **com.sapportals.wcm.util.channels**
  Will be provided by the Portal Runtime.

- **com.sapportals.wcm.util.config**
  Deprecated. The CRT encapsulates handling of configuration data or the extensions. Applications should use the Configuration Framework directly (see [CF]).

- **com.sapportals.wcm.util.controlstatus**
  Only relevant for the Knowledge Management’s Next Generation UI.

- **com.sapportals.wcm.util.factories**
  Has been superseded by the CRT concept. The `ThreadUtils` will be replaced by J2EE Engine’s functionality.

- **com.sapportals.wcm.util.log**
  Deprecated. The Logging Framework in `com.sap.tc.logging` is used now.

- **com.sapportals.wcm.util.systemconfig**
  Used internally by RF extensions to allow special rights for some special users.

- **com.sapportals.wcm.util.usermanagement**
  Only used by the Knowledge Management’s UI. The main functionality is now provided through the portal’s user management.
4.2 New API: `com.sap.netweaver.bc.rf.util`

The following list is just a brief summary of the packages beneath `com.sap.netweaver.bc.rf.util`.

- `com.sap.netweaver.bc.rf.util.context`
  Utilities for storing thread-bound context objects.

- `com.sap.netweaver.bc.rf.util.enum`
  Enum is the base class for all enumerations with RF constants in the new API.
  ➔ Current API: `com.sapportals.wcm.util.enum`

- `com.sap.netweaver.bc.rf.util.event`
  Copy of the current API, not yet used.
  ➔ Current API: `com.sapportals.wcm.util.event`

- `com.sap.netweaver.bc.rf.util.exception`
  The base classes for the RF exceptions (see 3.3.3).

- `com.sap.netweaver.bc.rf.util.flyweight`
  A base class for implementing the flyweight pattern.

- `com.sap.netweaver.bc.rf.util.logging`
  Provides `LoggingFormatter.extractCallstack()` for extracting an exception’s call stack for tracing (copied from the current API), and some helper classes to initialize, configure, and format the logging.
  ➔ Current API: `com.sapportals.wcm.util.logging`

- `com.sap.netweaver.bc.rf.util.namespace`
  Interface `IName` and class `Name` for handling an XML (namespace, name) naming scheme (see 2.1.7 and [RFC2518], copied from the current API).
  ➔ Current API: `com.sapportals.wcm.util.name`

- `com.sap.netweaver.bc.rf.util.resource`
  Offers `ResourceBundles` that extend `java.util.ResourceBundle` for multi-language support (copied from the current API).
  ➔ Current API: `com.sapportals.wcm.util.resource`

- `com.sap.netweaver.bc.rf.util.uuid`
  Offers `UUID` for generating Universally Unique Identifiers (UUIDs) as specified by the Internet draft “A UUID URN Namespace” (see IETF-UUID, copied from the current API).
  ➔ Current API: `com.sapportals.wcm.util.uuid`
5 Using the RF

This section focuses on the RF API from an application developer’s point of view. As mentioned in 3.1, applications are built using the current API, that consists of some portions of the com.sapportals.wcm.repository.* packages and of some portions of the com.sapportals.wcm.util.* packages.

For simplified reading, these packages have been abbreviated as follows:

- com.sapportals.wcm.repository is omitted
- com.sapportals.wcm.repository.<subpackage> as <subpackage>
- com.sapportals.wcm.util as …util

5.1 Basic Aspects

5.1.1 RF Objects, RID, Resource Context and the Resource Factory

IResource represents an RF object (resource).
ICollection represents a collection and extends IResource.
IResource in turn extends ITypeCast, which represent a semantic object (see 5.3.1).

The class …util.uri.RID represents a RID …util.uri.RID and offers the static factory method getRID(), for creating a RID from a String.
IResourceContext represents a resource context, where ResourceContext is the RF’s according implementation. To create a ResourceContext, at least the user on whose behalf the resource context is created must be supplied (see [Security] on how to manage user information and the portal’s user management).

The main entry point to the RF is the resource factory, represented by IResourceFactory. An instance of the IResourceFactory is retrieved by calling the ResourceFactory’s static method getInstance().

The following sample code illustrates, how to retrieve the IResourceFactory, create a resource context, and retrieve the resource for the RID /etc:

```java
com.sapportals.portal.security.usermanagement.IUser user = …
IResourceContext resourceContext = new ResourceContext(user);
RID rid = RID.getRID("/etc");
try {
    IResource resource = ResourceFactory.getInstance()
        .getResource(rid, resourceContext);
    if( resource != null ) {
        // resource found
        System.out.println(“resource ” + resource.getRID() + “ found”);
    } else { // resource not found
        System.out.println(“resource ” + resource.getRID() + “ does not exist”);
    }
} catch( ResourceException e ) { // problem while retrieving the resource
    System.out.println(“exception while trying to get resource ” + e.getRID()
        + “: ” + e.getMessage());
}
```

A resource’s RID and its resource context can be retrieved with getRID() and getContext().

A collection is retrieved in the same way as a resource and is then simply casted to ICollection:

```java
if( resource.isCollection() ) {
    ICollection collection = (ICollection)resource;
    IResourceList children = collection.getChildren();
}
```

The children of a collection are returned by the ICollection’s getChildren() method.
5.1.1.1 Name Info

The paths and names, as used for repositories resources, might be restricted in length or by the set of available characters that can be used.

A client can retrieve this information by calling the `manager.IRepositoryManager.getNameInfo()` method:

```
ICollection root = ...  
NameInfo nameInfo = root.getRepository().getNameInfo()  
```

A `NameInfo` offers the following methods:

- `getMaxPathLength()` returns the maximum length allowed for a RID’s path part.
- `getMaxNameLength()` returns the maximum length allowed for a RID’s name part (and thus for the path segments, too).
- `getReservedResourceNameChars()` returns the array of forbidden characters. These characters are not allowed for resource names (where resources are resources that are not collections).
- `getReservedCollectionNameChars()` returns an array of forbidden characters that are not allowed for collection names (and thus not for the path segments).
- `checkName()` throws an `InvalidNameException` if the given resource name contains characters from `getReservedResourceNameChars()`.

5.1.1.2 Links

Links are not represented by a dedicated class – the relevant operations are assigned to the `IResource` interface. Internal and external link types are identified by appropriate `enum.LinkType` values, as returned by the `IResource`’s `getLinkType()` method:

```
if( LinkType.INTERNAL.equals(resource.getLinkType()) ) {  
    // resource is an internal link to another resource  
    IResource target = resource.getTargetResource();  
    if( target == null ) {  
        // link is broken, because target does not exist anymore  
        URL targetURL = resource.getTargetURL();  
    }  
} else if( LinkType.EXTERNAL.equals(resource.getLinkType()) ) {  
    // resource is an external link to an URL  
    URL target = resource.getTargetURL();  
} else { // if( LinkType.NONE.equals(resource.getLinkType()) )  
    // resource is not a link  
}
```

The method `getTargetResource()` returns the target resource of an internal link – if it is not broken and if the target resource exists. The method `getTargetURL()` returns the URL for an external link’s target or the RID of an internal link’s target (which might be relative, as it is returned as passed when `setTargetResource()` is called or the link was created, see below).

As with EP 6.0 SP 1, flexible links or dynamic links are now supported within the RF. In any case, the RF offers no methods that allow a client to control whether an internal link is to be created as a static or dynamic link – this decision is up to the repository.
A client is only able to determine whether an internal link is a flexible (dynamic) link or just an internal (static) link: To do this, the IResource has to be "casted" to an IInternalLinkResource semantic object.

```java
if( LinkType.INTERNAL.equals(resource.getLinkType()) ) {
    if( resource.isA(IInternalLinkResource.class) ) {
        IInternalLinkResource internalLink = (IInternalLinkResource)resource .as(IInternalLinkResource.class);
        if( internalLink.isDynamic() ) {
            // it’s a flexible link (dynamic internal link), which never becomes broken
        } else {
            // it’s a static internal link, which might be broken
        }
    }
}
```

### 5.1.2 Namespace Operations

Since only collections may contain resources, only ICollection offers methods for creating resources.

To create a resource, the client must provide a name for resource in the collection. The collection where the create...()-operation was requested, becomes the parent collection of the newly created resource.

ICollection offers the following create operations:

- createResource() for a plain resource with content and properties
- createCollection() for a collection with properties
- createLink() for a link with properties, link type, and target

```java
IResourceContext context = new ResourceContext(user);
ICollection parent = (ICollection) ResourceFactory.getInstance() .getResource("/documents", context);
ICollection collection = parent.createCollection("folder", null);
ILink link = parent.createLink("link", "/documents/file", LinkType.INTERNAL, null);
```

The example above retrieves the /documents folder and creates the empty resource (without properties or content) /documents/file, the collection /documents/folder without properties and the internal link /documents/link (which points to /documents/file) in it.
The other namespace operations for copying, deleting, moving and renaming are available from the IResource interface:

- `copy()` copies the resource and expects the destination RID and an optional ICopyParameter. This parameter is a descriptor (see 3.3.4) that might supply additional parameters (for example, if children should be copied recursively, too).
- `delete()` deletes the resource, collection or link itself. When a collection is deleted, all its children are deleted, too.
- `move()` moves the resource to another parent collection and expects the same parameters as the `copy()` method.
- `rename()` changes the names of a resource and expects the resource’s new name as a parameter.

Using the `resource` from the example above:

```java
IResource copied = resource.copy("/documents/copy", null);
IResource moved = copied.move("/documents/folder", null);
IResource renamed = moved.rename("renamed");
renamed.delete();
```

This example would create a copy of `/documents/file` as `/documents/copy`, then move it to the folder `/documents/folder` as `/documents/folder/copy` and then rename it as `/documents/folder/renamed`.

When `delete()` is called, all the IResources are invalidated, since all of them reference the same resource. In other words, calling `copied.delete()` or `moved.delete()` after `renamed` has been deleted, would cause a ResourceException to be thrown.
5.1.3 Content

Content is represented by `util.content.IContent`. The RF provides the class `Content` as a default implementation.

For retrieving the content of a resource, `getContent()` is used.

```java
IContent oldContent = resource.getContent();
```

Setting the content is either done during the creation of a resource (see create) or by using `updateContent()`:

```java
String dataString = new String("content");
ByteArrayInputStream dataStream = new ByteArrayInputStream(dataString.getBytes());
IContent newContent = new Content(
    dataStream,
    "text/plain",
    dataStream.available() )
;
resource.updateContent(newContent);
```

The example above would set the resources content to the text "content" as plain text.

Changing the content of a resource also updates the values of the corresponding live properties for content length, content type, and encoding, as given by the `IContent` object.

As another side effect, the values of the other content-related live properties (entity tag, last modified, last modified by) are also updated accordingly.
5.1.4 Properties

A property’s read-only operations are assigned to IProperty. The corresponding implementation is Property. The write operations are assigned to IMutableProperty, with MutableProperty as the corresponding implementation.

The name of a property is modeled as IPropertyName, with PropertyName as the implementing class. IPropertyName is derived from ...util.name.IName, which holds the tuple of (namespace, name).

The property’s attributes are modeled as java.util.Properties.

Each property has an enum.PropertyType, which defines the type of the property’s value:

- PropertyType.BOOLEAN for boolean values
- PropertyType.DATE for Date values
- PropertyType.INT for int values
- PropertyType.LONG for long values
- PropertyType.STRING for String values
- PropertyType.XML for XML values

Please note, that a property with a specific name is either set to an existing value – or it does not exist. It is therefore not possible to store a null value as a property’s value. An undefined property value should be represented by a non-existing property.
The following example shows, how a property is retrieved:

```java
String namespace = "http://com.sap.netweaver.bc.rf.sample/xmlns/sample";
String name = "property";
IPropertyName propertyName = new PropertyName(namespace, name);
IProperty property = resource.getProperty(propertyName);
if( property != null ) {
    // property exists
    String value = property.getValueAsString();
} else {
    // property is not set for this resource
}
```

The following example shows how to use the `enum.PropertyType` for checking the type of the property's value:

```java
if( property.isMultivalued() ) {
    // property is multi valued
    List values = property.getValues();
    if( values.size() > 0 ) {
        // property is multi valued and a value element exists
        if( PropertyType.BOOLEAN.equals(property.getType()) ) {
            boolean firstValue = ((Boolean)list.iterator().next()).booleanValue();
        } else if ( PropertyType.DATE.equals(property.getType()) ) {
            Date firstValue = (Date)list.iterator().next();
        } else if ( PropertyType.INT.equals(property.getType()) ) {
            int firstValue = ((Integer)list.iterator().next()).intValue();
        } else if ( PropertyType.LONG.equals(property.getType()) ) {
            long firstValue = (Long)list.iterator().next().longValue();
        } else { // PropertyType.STRING and PropertyType.XML are mapped to String
            String firstValue = (String)list.iterator().next();
        }
    } else {
        // property is multi valued but has no value element
    }
} else {
    if( PropertyType.BOOLEAN.equals(property.getType()) ) {
        boolean value = property.getBooleanValue();
    } else if ( PropertyType.DATE.equals(property.getType()) ) {
        Date value = property.getDateValue();
    } else if ( PropertyType.INT.equals(property.getType()) ) {
        int value = property.getIntValue();
    } else if ( PropertyType.LONG.equals(property.getType()) ) {
        long value = property.getLongValue();
    } else { // PropertyType.STRING and PropertyType.XML are mapped to String
        String value = property.getValueAsString();
    }
}
```

Setting a property is achieved as follows:

```java
boolean value = true;
IProperty property = new Property(propertyName, value);
resource.setProperty(property);
```

This creates a property with a `boolean` type and value.

When changing the value of an already created or retrieved property, `IMutableProperty` is used:

```java
IMutableProperty mutableProperty = property.getMutable();
mutableProperty.setBooleanValue(true);
resource.setProperty(property);
```

This would set the value of the `boolean` property to `true`. If the new value were to cause a type mismatch with the previous value (for example, assigning a string as a value for a property with integer type), a `ResourceException` would be thrown.
Since creating and disposing of a large number of objects has an impact on the VM’s garbage collector, Property and PropertyName should be reused where possible.

Therefore, the class PropertyName offers several methods for retrieving PropertyName for the RF’s live properties (see below for examples of display names and resource types).

The IPropertySearchManager’s isUnderstood() method can be used to check whether a given property name identifies a live property (see 5.2.2 on IPropertySearchManager).

The display name property is named as returned by PropertyName.createDisplayname(). The IResource offers the method getDisplayName() as a shortcut for getProperty(PropertyName.createDisplayname()) with two different signatures:

The first one, without any parameter, just returns the display name, or null if it is not set.

```java
// retrieving the display name or null if not set
String displayname = resource.getDisplayName(); // or .getDisplayName(false);
if( displayname == null ) {
    // retrieving the RID’s name if display name not set
    displayname = resource.getRID().name(); // use RID’s name as fallback
}
```

The second signature takes a boolean parameter: If false, it returns the same result as without any parameter. If true, it returns the display name as shown in the example above. Therefore the example above could be reduced to:

```java
// retrieving the display name or RID’s name if not set
String displayname = resource.getDisplayName(true);
```

If an application wants to update the display name, the Property class offers the createDisplaynameProp() method for creating a display name property from a given string.

```java
// update the display name
resource.setProperty(Property.createDisplaynameProp(displayname));
```

The resource type property is handled in the same manner: It is represented by the property name as returned by PropertyName.createResourceType() and the IResource offers getResourceType() as a shortcut for

```java
// retrieving the resource’s type
String resourcetype = resource.getResourceType();
if( resourcetype == null ) {
    // not set
}
```

To build a resource type property for updating a resource’s type, Property.createPropertyTypeProp() creates a resource type property from the given String.

Note: In order to avoid the ambiguous naming of resource types (the values of the resource type property), this resource type’s property value has to adhere to the following namespace scheme: All resource types of the RF are prefixed by the value defined with com.sapportals.wcm.IWcmConst.RESOURCE_TYPE_NAMESPACE.FRAMEWORK, while the resource types of applications are prefixed by the value defined with com.sapportals.wcm.IWcmConst.RESOURCE_TYPE_NAMESPACE.APPLICATIONS and separated by an IWcmConst.NAMESPACE_SEPARATOR.
If the application "myApp" wants to invent the new resource type value "myResourceType", the proper coding for updating the resource type might then look like the following:

```java
// myApp’s namespace
public final String myTypeNS = IWcmConst.RESOURCE_TYPE_NAMESPACE.APPLICATIONS
    + IWcmConst.NAMESPACE_SEPARATOR
    + "myApp";

// myApp’s full resource type value
public final String myTypeName = myTypeNS
    + IWcmConst.NAMESPACE_SEPARATOR
    + "myResourceType";

// setting a resource’s resource type property to myTypeName
resource.setProperty(Property.createResourceTypeProp(myTypeName));
```
5.2 Advanced Aspects

5.2.1 Ordered Collections

Ordered collections are ICollection that return enum.OrderType.MANUAL for getOrderType().

An application might enable ordering for an ICollection by using setOrderType(), if it is supported (otherwise, a NotSupportedException is thrown).

If an application wants to reorder the order of an ordered collection’s children, the definition for reordering is modeled as IReorderList, with ReorderList as the implementation.

IReorderList defines several “positioning commands” as IPositioning (positioning for implementation). Such a “positioning command” defines the resource to which it is to be supplied and the new position to choose for the resource in question.

The positions are defined as IPosition, where Position is the implementation. There are four kinds of positions, where the “position kind” is defined by enum.OrderPosition:

- OrderPosition.FIRST: The first element
- OrderPosition.LAST: The last element
- OrderPosition.BEFORE: The predecessor of another resource
- OrderPosition.AFTER: The successor of another resource

The third and fourth OrderPosition are relative position kinds, and therefore require a resource to which they can refer in relation.
Consider the collection \(c\), with \(\text{getOrderType()}\), returning \(\text{OrderType.MANUAL}\) and that contains the following children in the specified order:

\[1st, 2nd, 3rd, 4th\]

The following examples demonstrate the several \(\text{OrderPosition}\)s:

```java
IReorderList reordering = new ReorderList();
reordering.add(new Positioning("4th", new Position(null, OrderPosition.FIRST)));
reordering.add(new Positioning("1st", new Position(null, OrderPosition.LAST)));
reordering.add(new Positioning("2nd", new Position("1st", OrderPosition.BEFORE)));
reordering.add(new Positioning("3rd", new Position("4th", OrderPosition.AFTER)));
orderedCollection.reorder(reordering);
```

In this example's first step, \(4th\) is positioned as the first element.

\[4th, 1st, 2nd, 3rd\]

Then \(1st\) is positioned as the last element.

\[4th, 2nd, 3rd, 1st\]

Next, \(2nd\) is positioned relatively before \(1st\) (which is now the last element), making it the third element.

\[4th, 3rd, 1st, 2nd\]

The last step positions \(3rd\) relatively after \(4th\) (which is now the first element), making it the second element (which it is already).

\[4th, 3rd, 1st, 2nd\]

### 5.2.2 Property Queries

Property queries are built using an \(IQueryBuilder\). A property query consists of one or more \(IQueryExpression\)s that can be combined using boolean operations (\(\text{and()}\), \(\text{or()}\) and \(\text{not()}\)). An \(IQueryExpression\) represents one \(\text{WHERE}\) clause, defining the matching values for a given property as specified by its \(IPropertyName\).
There are two options for retrieving an IQueryBuilder:

- The search.IGenericQueryFactory offers a generic IQueryBuilder that is applicable to any repository but is not optimized for a specific repository. This is especially useful if a query spans several repositories.

- The manager.IPropertySearchManager, which can be retrieved from a resource's manager.IRepositoryManager, offers an IQueryBuilder that is specialized for the given repository. Unlike a query expression built with an IGenericQueryFactory, the query expression built by a specific repository's query builder cannot be used with another repository.

To select the resources for the given query, execute() has to be called, depending on where to apply the SELECT:

- If a generic query is to be performed, the IQueryExpression must be converted into a search.IGenericQuery. Then the search.IGenericQuery's execute() has to be called.

- If a specialized query is to be performed, the manager.IPropertySearchManager's (the one that returned the IQueryBuilder) execute() has to be called.

For example, the following WHERE clause is to be used:

( '{http://sapportals.com/xmlns/cm}contenttype' like '%.jpg' or '{http://sapportals.com/xmlns/cm}contenttype' like '%.gif' ) and '{http://sapportals.com/xmlns/cm}createdby' = 'admin'

This WHERE clause's corresponding IQueryExpression could be constructed with the following code:

```csharp
ICollection start = (ICollection)factory.getResource("/documents", context);
IGenericQueryFactory queryFactory = GenericQueryFactory.getInstance();
IQueryBuilder queryBldr = queryFactory.getQueryBuilder();
IQueryExpression queryExpr = queryBldr like(PropertyName.createContentType(), "%.jpg").or(
    queryBldr like(PropertyName.createContentType(), "%.gif").and(
    queryBldr.eq(PropertyName.createCreatedBy(), "admin")
);)
IGenericQuery query = queryFactory.toGenericQuery(queryExpr);
IResourceList result = query.execute(
    start, // the resource to start at
    Integer.MAX_VALUE, // maximum depth
    Integer.MAX_VALUE, // maximum result size
    false // don’t include versions
);
```

The example above shows how the generic query is used.

Using the IResource's search() method is equivalent to query.execute():

```csharp
_ = start.search(query, Integer.MAX_VALUE, Integer.MAX_VALUE, false)
```
The following example shows the same thing for a repository specific query, which is usually faster (since the query expression is already set up for the repository) but can only be applied to this type of repository:

```csharp
ICollection start = (ICollection)factory.getResource("/documents", context);
IRepositoryManager repositoryMgr = start.getRepositoryManager();
IPropertySearchManager searchMgr = repositoryMgr.getPropertySearchManager(start);
IQueryBuilder queryBldr = searchMgr.getQueryBuilder();
IQueryExpression queryExpr =
    queryBldr.like(PropertyName.createContentType(), "%.jpg").or(
        queryBldr.like(PropertyName.createContentType(), "%.gif").and( 
            queryBldr.eq(PropertyName.createCreatedBy(), "admin") 
        )
    );
IResourceList result = searchMgr.execute( 
    queryExpr, // the WHERE clause 
    start, // the resource to start at 
    Integer.MAX_VALUE, // maximum depth 
    Integer.MAX_VALUE, // maximum result size 
    false // don’t include versions 
);
```

Note: If a query builder is not able to construct a query expression for a given property and operation, the returned query expression might be `null`. The examples above do not check for `null` being returned, but "real" code should do so.

### 5.2.3 Unique ID

In the current API, unique IDs are provided by a global service. See 5.4.2.6 for details.

### 5.2.4 Security

Security constraints that are defined for the operations on the RF’s resource aspects are handled by the repository manager (see 6.2.8.5). This means that all a client application usually has to concern itself with is the proper handling of `AccessDeniedException`S. Such exceptions are thrown if an operation has been requested by a user for a resource and the user does not have the necessary permissions.

For example:

```java
// retrieval requires no permission
IResource resource = ResourceFactory.getInstance().
    getResource(RID.getRID("/documents/myResource"), context);
if( resource != null ) { 
    // resource exists 
    try { 
        resource.delete(); // request an operation, e.g. delete()
    } 
    catch( AccessDeniedException e ) { 
        // show some message to the user 
        System.out.println( 
            e.getPermissionName() + " not granted for " + 
            e.getUserID() + " on " + 
            e.getRID().name()
        );
    }
}
```
Please note that the resource can be retrieved even if the user has no permission at all for the resource specified by the given RID. Permissions are not checked, unless an operation that requires a specific permission is requested. The only method that does not require a permission for an IResource is getRID().

To prevent users from “seeing” the children of a resource, the ICollection's getChildren() method checks for the “list children” permission (see 6.2.8.5 for details on checked permissions).

An extension or application can use the manager.ISecurityManager’s isAllowed() method to check whether a specific permission is granted for a user:

```java
ISecurityManager securityManager = resource.getRepositoryManager().getSecurityManager(resource);
if( securityManager.isAllowed(resource, user, IPermission.PERMISSION_DELETE) {
    // delete permission granted
} else {
    // delete permission denied
}
```

If an extension or application wants to establish its own security constraints, it can use the service ACLs (see 5.4.1.2) to define its own permissions and to define application specific security semantics.
Both, the repository manager’s `manager.IAclSecurityManager` and the
`service.serviceacl.IAclService` offer the `getAclManager()` method for retrieving the
corresponding `security.IResourceAclManager`.

The `security.IResourceAclManager` handles the retrieval, creation, and removal of the
`security.IResourceAcl` – either those for the repository manager or those of the service’s ACL.
The `manager.IAclSecurityManager` and `service.serviceacl.IAclService` use it to
retrieve their ACLs and perform the `isAllowed()` check.

An `IResourceAclManager` needs to know about available permissions, represented by
`...util.acl.IAclPermission`. Permissions are identified by their IDs. Permissions used by the RF
are defined by the constants in `manager.IPermission` (see 6.2.8.5 for more details).

After a permission has been created by `createPermission()`, it has to be assigned to a
`...util.acl.IObjectType` as a supported permission before it can be used. To do this, an
application or extension uses `addSupportedPermission()`, passing the relevant object type and
permission as the methods arguments. The `IObjectType.OBJECT_TYPE_NODE` is used for node
resources (collections) and `IObjectType.OBJECT_TYPE_LEAF` is used for leaf resources
(resources that are not collections).

An ACL, represented as `security.IResourceAcl`, is bound to an `IResource` and has at least
one owner. It can contain zero, one, or more `security.IResourceAclEntry`s that specify the
permissions for a specific principal. ACLs are created with the `createAcl()` method, which takes the
resource to which the ACL is to be assigned as a parameter. Therefore, the created ACL initially only
gives full control (all supported permissions are granted) to the owner, taken from the resource’s
class.

An `IResourceAclEntry` contains the principal and the permission that is granted (or revoked, see
below) for a user. An instance of a `IResourceAclEntry` is created by the
`IResourceAclManager``s `createAclEntry()` method, which takes the principal and the
permission at parameters. A flag for positive or negative entries is also supported. This flag indicates
whether the entry grants or revokes a specific permission, as well as a sort-index that controls when to
apply a “revoke” or “grant” (for example, if an ACL has two entries, `E1` for revoking permission `P` for
user `U`, and `E2` for granting permission `P` to group `G`. User `U` is member of group `G`. If `E1`
owns has a lower sort index than `E2`, Permission `P` is revoke for user `U`, but it is granted by `E2` if `E1`
would have a higher sort index than `E2`).

Note: Users became very confused when negative ACLs where used. The KM’s ACL editor therefore
does not support negative ACL entries or the sortindex.
5.2.5 Locking

A resource can be locked by using the IResource's lock() method. If no parameters are supplied, the resource is locked for the user specified in the resource's context. The lock created is an exclusive, shallow write lock with an infinite timeout.

To specify another scope, depth, or timeout, the application has to supply the values bundled in ILockProperties, where LockProperties is the according implementation.

Scope, type, and depth are modeled as enum.LockScope, enum.LockType and enum.LockDepth. enum.LockType.READ is not used in the RF at the moment.

The methods isLocked() and isLockedByMe() are used to check whether a resource is locked.

The methods refreshLock() and unlock() require the ILockInfo that was returned by lock() to refresh or release a lock.

The ILockInfo supplies a lock token (a String that uniquely identifies the lock among the locks of the resource). This can be used to retrieve the ILockInfo required for unlock() by calling getLockByToken().

For example:

```java
ILockProperties lockProperties = new LockProperties(
    LockScope.SHARED, // shared
    LockType.WRITE, // write
    LockType.SHALLOW, // shallow
    (60 * 60 * 1000) // timeout 1h
);

ILockInfo lockInfo = resource.lock(lockProperties);
String lockToken = lockInfo.getLockToken();

ILockInfo restoredLockInfo = resource.getLockByToken(lockToken);
resource.unlock(restoredLockInfo);
```

This would lock the resource with a shared, shallow write lock, with a timeout of one hour (60 minutes * 60 seconds * 1000 milliseconds).

The lock token can then be stored and later restored. For unlocking, the lock info is retrieved using the restored lock token.
5.2.6 Events

Each repository manager owns a `manager.IResourceEventBroker` that can be retrieved by a call to the `manager.IRepositoryManager`’s `getEventBroker()` method.

The event broker is the central object, where both senders and receivers are registered. Senders are represented by `manager.IResourceEventSender` and receivers by `manager.IResourceEventReceiver`.

All these interfaces are derived from the corresponding interfaces in `...util.events.*`, simply by adding the resource-related methods.

A generic event is modeled by `...util.events.IEvent`. The type is just an integer constant (defined in `manager.ResourceEvent`) that specifies the event’s type. The event’s `isLike()` method checks whether two events are similar with respect to their type and Java class (disregarding the parameter). The `isLike()` method treats the `ResourceEvent.ALL` type a little bit differently than the other types: `ResourceEvent.ALL` matches any other type, so `isLike()` would return `true` when compared to any other type of the same event Java class.

The parameter object returned by `getParameter()` is just optional information that is associated to this event (for example, the old RID of a resource when it was moved).

A resource event `manager.IResourceEvent` is derived from `...util.events.IEvent`. It adds the resource to which the event refers to the event.
An application must implement the `manager.IResourceEventReceiver` interface in order to receive events:

```java
public class MyEventReceiver
    implements IResourceEventReceiver {
    private IResourceEvent DELETE_EVENT
        = new ResourceEvent(ResourceEvent.DELETE, null);
    private static IResourceEvent MOVE_EVENT
        = new ResourceEvent(ResourceEvent.MOVE, null);
    private static IResourceEvent RENAME_EVENT
        = new ResourceEvent(ResourceEvent.RENAME, null);

    public MyEventReceiver() {
    }

    public void register(IResource resource)
        throws WcmException {
        IResourceEventBroker broker = resource.getRepositoryManager().getEventBroker();
        if( broker == null ) return;
        broker.register(this, DELETE_EVENT, IEventBroker.PRIO_MIN, true);
        broker.register(this, MOVE_EVENT, IEventBroker.PRIO_MIN, true);
        broker.register(this, RENAME_EVENT, IEventBroker.PRIO_MIN, true);
    }

    public void unregister(IResource resource)
        throws WcmException {
        IResourceEventBroker broker = resource.getRepositoryManager().getEventBroker();
        if( broker == null ) return;
        broker.unregister(this, DELETE_EVENT, IEventBroker.PRIO_MIN);
        broker.unregister(this, MOVE_EVENT, IEventBroker.PRIO_MIN);
        broker.unregister(this, RENAME_EVENT, IEventBroker.PRIO_MIN);
    }

    public void received(IEvent event) {
        try {
            if( event instanceof IResourceEvent ) {
                IResourceEvent resourceEvent = (IResourceEvent)event;
                switch( resourceEvent.getType() ) {
                    case ResourceEvent.DELETE:
                        // resourceEvent.getResource().getRID() was deleted
                        break;
                    case ResourceEvent.RENAME:
                        // RID.getRID((String)resourceEvent.getParameter()) was renamed
                        // to resourceEvent.getResource()
                        break;
                    case ResourceEvent.MOVE:
                        // resourceEvent.getResource() was moved to
                        // (IResource)resourceEvent.getParameter() was renamed
                        break;
                    default:
                        // ignore other IResourceEvents
                        break;
                }
            }
        } catch( Exception e ) {
        }
    }
}
```

When the `MyEventReceiver` class is registered to a repository's event broker, it receives the events it registered for:

```java
IResource resource = factory.getResource("/documents", context);
MyEventReceiver aReceiver = new MyEventReceiver();
aReceiver.register(resource);
```
This causes the `MyEventReceiver` to register itself with the event brokers of the given resource’s `/documents` repository manager for DELETE, MOVE and RENAME events. The receiver registers itself with minimum priority as an asynchronous receiver (indicated by the additional `boolean true` parameter on registration).

The priority determines the event receiver’s position in the list of the event broker’s receivers.

If an event receiver wants to use the correlation ID, it must use the `ResourceEvent` class that provides the methods `getCorrelationId()` and `getUniqueId()`:

```java
public void received(IEvent event) {
   ...
   if( event instanceof ResourceEvent ) {
      ResourceEvent resourceEvent event = (ResourceEvent)event;
      String correlationId = resourceEvent.getCorrelationId();
      switch( resourceEvent.getType() ) {
         case ResourceEvent.PRE_DELETE:
            // resourceEvent.getResource().getRID() is about to be deleted
            break;
         case ResourceEvent.DELETE:
            // resourceEvent.getResource().getRID() was deleted
            break;
      }
   }
   ...
}
```

The correlation ID identifies a sequence of events that belong together – for now, only corresponding pre-and post-events, as shown in the example above, are associated with the same correlation ID.

Once an event receiver registered with an event broker, it receives events until it deregisters.

If a receiver performs an operation on a resource when handling a received event, this might trigger other events. In the worst case scenario, this can lead to unintended recursions. To prevent this, an event receiver can change its state to “suspended” by calling the event broker’s `suspend()` method. This temporarily disables events to be sent to the suspended receiver until it changes its state back to “receiving” by calling `resume()`. Events that are triggered while the receiver is suspended are simply dropped. If the dropping of events is not desired, another mode, similar to suspend, is available: “hold” is invoked by calling the event broker’s `hold()` method. While on “hold”, triggered events are collected by the event broker, allowing the receiver to handle them later. Depending on the number of events being triggered, this might cause a memory overflow if the receiver is to slow in resetting the list of collected events.
The following picture summarizes the states of an event receiver:

In order to send events, an application has to:
- Introduce its new event types
- Implement the IResourceEventSender interface
- Register an instance of its IResourceEventSender implementation with the appropriate event broker

Defining a new event type:

```java
public class MyEvent implements IEvent {
    public static final int MYEVENT = 4711;
    private IResource parameter;

    public MyEvent(IResource resource) {
        this.parameter = resource;
    }

    public IResource getResource() {
        return this.parameter;
    }

    public String getDescription() {
        return getDescription(java.util.Locale.getDefault());
    }

    public String getDescription(java.util.Locale locale) {
        return "MyEvent";
    }

    public Object getType() {
        return null;
    }

    public boolean isLike(IEvent template) {
        // since only one type-code is used, it’s enough to check the Java class
        return (template instanceof MyEvent);
    }
}
```
Please note that it might not be a good idea, to derive new resource events from `IResourceEvent`, since most event receivers only check for `instanceof IResourceEvent` and then use the value returned from `getType()` to select the proper event type.

If a new event implements `IResourceEvent` and uses a type code that is already used by another `IResourceEvent` implementation, the event receivers also try to handle the new event.

Implementing a separate event sender for the `IResourceEventSender` interface:

```java
public class MyEventSender implements IResourceEventSender {
    private static final MyEvent myEventTemplate = new MyEvent(null);
    private IEventList myEventList = new EventList();
    private IResourceEventBroker broker = null;

    public MyEventSender() {
        this.myEventList.add(myEventTemplate);
    }

    public void register(IResource resource) throws WcmException {
        if( this.broker != null ) broker.unregister(this);
        this.broker = resource.getRepositoryManager().getEventBroker();
        if( this.broker == null ) return;
        broker.register(this);
    }

    public void unregister(IResource resource) throws WcmException {
        if( this.broker == null ) return;
        this.broker.unregister(this);
        this.broker = null;
    }

    public IEventList getEvents() {
        return this.myEventList;
    }

    public IEventList getEvents(IResource resource) {
        return this.myEventList;
    }

    public void send(IResource resource) {
        try {
            if( this.broker == null ) return;
            this.broker.send(new MyEvent(resource), this);
        } catch( Exception e ) {
            // handle exception
            
        }
    }
}
```

An instance of the `MyEventSender` class is registered with a repository’s event broker as follows:

```java
IResource resource = ResourceFactory.getInstance()
    .getResource("/documents", context);
MyEventSender mySender = new MyEventSender();
resource.getRepositoryManager().getEventBroker().register(mySender);
```

To send a `MyEvent`:

```java
mySender.send(resource);
```
If a receiver wants to handle the new type of event, its `received()` method might look like the following:

```java
public void received(IEvent event) {
    try {
        if (event instanceof MyEvent) {
            MyEvent myEvent = (MyEvent)event;
            // myEvent.getResource() returns the resource of MyEvent
        } else if (event instanceof IResourceEvent) {
            IResourceEvent resourceEvent = (IResourceEvent)event;
            // handle IResourceEvents here
        }
        // other event implementations are ignored
    } catch (Exception e) {
    }
}
```
5.3 Expert Aspects

5.3.1 Type Handling

Type handling is a mechanism that allows the RF’s unified resources to be “casted” into semantic objects (see 2.3.1).

The IObjectFactoryRegistry holds a list of all known IObjectFactories: Each IObjectFactory has to register itself with the IObjectFactoryRegistry; an instance of the IObjectFactoryRegistry is provided by ObjectFactoryRegistry.getInstance():

```java
IObjectFactoryRegistry objectFactoryRegistry = ObjectFactoryRegistry.getInstance();
objectFactoryRegistry.registerObjectFactory(new MyObjectFactory());
```

The object factories are responsible for “casting” IResources into other objects:

A IObjectFactory offers isA() to check whether a given resource can be “casted” into an object of a specific class or interface. To “cast” a resource into an object of a specific class or interface, the method as() is used. The method listTypes() lists all the available classes and interfaces to which the given resource can be “casted”.

A RF application might either implement its own object factory, which has to implement the IObjectFactory interface, or use the RF’s DefaultObjectFactory instead:

The DefaultObjectFactory implements the IMutableObjectFactory, which allows new classes to be registered as possible objects to which a resource can be “casted”. The DefaultObjectFactory uses the resource’s type as the selection criteria. The resource type is specified by the value of the resource type property (Property.createResourceTypeProp(), as
returned by the IResource’s getResourceType() method, see 5.1.4). When a cast is requested, the DefaultObjectFactory uses the resource type and the class to which the resource should be “casted” to select the appropriate object to be created.

For example:

```
IObjectFactoryRegistry objectFactoryRegistry = ObjectFactoryRegistry.getInstance();

// create the list of interfaces
Vector interfaces = new Vector();
interfaces.add(IMyInterface.class);

// register the class and its interfaces with the default object factory
objectFactoryRegistry.getDefaultObjectFactory().registerClass(
    MyObject.TypeName,
    MyObject.class,
    interfaces
);
```

Where MyInterface and MyObject might look like the following:

```java
public interface MyInterface {
}

public class MyObject
extends AbstractTypeCast
implements IMyInterface {

    static public String TypeName = IWcmConst.RESOURCE_TYPE_NAMESPACE.APPLICATIONS
        + IWcmConst.NAMESPACE_SEPARATOR + "myApp"
        + IWcmConst.NAMESPACE_SEPARATOR + "myType";

    public MyObject(IResource resource) {
        super(resource);
    }
}
```

The example above would register MyObject as “castable” object for resources with the resource type "http://sap.com/xmlns/cm/app/myApp/myType”. All resources of this type could then be converted into MyObjects by using resource.as(IMyInterface.class) or resource.isA(MyObject.class).

The DefaultObjectFactory requires the registered class to be derived from AbstractTypeCast, because it relies on the specified constructor, which takes an IResource as parameter. While the DefaultObjectFactory uses the resource type, it is not possible to register several classes for the same resource type and interface at the same time. To use a selection criteria other than resource type and class/interface, you could implement your own IObjectFactory implementation.
For example:

```java
public class MyObjectFactory
    implements IObjectFactory {

    private boolean isMyObject(resource) {
        return resource.getRID().extension().equalsIgnoreCase("MYOBJ");
    }

    public boolean isA(Class objectClass, IResource resource) throws ResourceException {
        if ( isMyObject(resource) ) // check source of cast
            && ( objectClass.isAssignableFrom(MyObject.class) ) // check destination
        ) {
            return true;
        }
        return false;
    }

    public Object as(Class objectClass, IResource resource) throws ResourceException {
        if( this.isA(objectClass, resource) ) {
            return new MyObject(resource);
        } else {
            return null;
        }
    }

    public Collection listTypes(IResource resource) {
        Collection allClasses = new ArrayList();
        if( isMyObject(resource) ) {
            Class[] objectClasses = MyObject.class.getClasses();
            for( int i = 0; i < objectClasses.length; i++ ) {
                allClasses.add(objectClasses[i]);
            }
        }
        return allClasses;
    }
}
```

The example would use the extension of a resource’s RID to check whether it is a “castable” object (those resources with extension .MYOBJ in this case).
5.3.2 Versioning

5.3.2.1 Basic Versioning

Before a resource can be checked out, version control has to be enabled for the resource. This is achieved using the `enableVersioning()` method. The method `isVersioned()` returns `true`, if versioning is enabled (that is, if the resource is a VCR).

```java
IResource resource = ...;
// turn on versioning, if it’s not already enabled
try {
    if( !resource.isVersioned() ) {
        resource.enableVersioning(true);
    }
} catch( OperationNotSupported e ) {
    System.out.println("versioning not supported for " + resource.getRID());
}
```

The `checkOut()` method is used to check-out a resource, while `undoCheckOut()` cancels the check-out state and `checkIn()` performs a check-in, thereby creating a new version (also called revision) of the resource.

For example:

```java
IResource resource = ...;
// check-out the VCR
resource.checkOut();
IContent oldContent = resource.getContent();
// usually the user or some client application modifies the content or properties
IContent newContent = ...;
// update content and properties and check-in the VCR
resource.checkIn(newContent, null);
```

Note: the current API’s `checkIn()` method is a convenience method that does two things: Firstly, it updates the VCR’s version controlled state from the content and properties supplied and secondly, it changes the check-out state to “checked-in”. Although update and check-in are atomic operations, the `IResource`’s `checkIn()` method is not atomic.

It is also possible, to perform only the check-in by specifying `null` as content or properties.

For example:

```java
// first, update the content
String dataString = new String("new version");
ByteArrayInputStream data = new ByteArrayInputStream(dataString.getBytes());
IContent newContent = new Content(data, "text/plain", data.available());
resource.updateContent(newContent);
// then, only the check-in
ICheckInInfo checkInInfo = resource.checkIn(null, null);
```

The check-in returns an `ICheckInInfo` object that provides the version’s ID (version number) and the RID of the version’s resource.

A version returns `true` on `isRevision()`.

As versioning is only related to the version controlled state of a resource, there are no concepts of “ownership”. There is therefore no way of determining who checked out a VCR. If an application wants to prevent concurrent modification, it should use locking. When combined with versioning, the resource is then locked before check-out and unlocked after check-in.
The expected RID, as returned by `checkOut()`, specifies the RID assigned to the revision that is created when the VCR is checked in. A client can use that RID in advance, for example to create a reference to that revision in the content, before it is checked in.

If a repository returned an expected RID, it checks for the expected RID on `checkIn()` as follows: If the client specifies the expected RID on check in, and it does not match the RID of the revision to be created on check-in (for example, because another application checked in a new version and then checked the resource out again), an `ExpectedCheckInRIDException` is thrown.

This is an example of how to use the expected RID if the repository supports checking:

```java
CheckOutInfo checkOutInfo = resource.checkOut();
// edit the resource
if( checkOutInfo != null ) {
    // application might use checkOutInfo.getExpectedRevisionRID() here
    try {
        ICheckInInfo checkInInfo = resource.checkIn(
            newContent, null, // content+props
            true, // ignore property failures
            checkOutInfo
                .getExpectedRevisionRID()
        );
    }
    catch( ExpectedCheckInRIDException e ) {
        // expected RID would not match the new version’s RID
    }
} else { // repository returned null checkout-info -> doesn’t support expected RID
    ICheckInInfo checkInInfo = resource.checkIn(
        newContent, null, // content+props
        true, // ignore property failures
    );
}
```

Please note, that this feature is optional.

The method `getVersionHistory()` is used to retrieve the (linear) version history of a resource under basic version control. It returns the list of versions, sorted by age and starting with the oldest version. The last entry is the current version.

This method throws a `NotSupportedException` either if the revision is not versioned or if it is versioned but does not have a linear version history.

### 5.3.2.2 Advanced Versioning

While the basic versioning features are located in the `IResource` interface, the advanced aspects are defined in other interfaces. In order to for these features to be used, an `IResource` has to be “casted” to these interfaces first, using the type handling mechanism (see 5.3.1).

Even for basic versioning, the interfaces `IVersionControlledResource` (representing a VCR) and `IVersionResource` (representing a version or revision) are available, but they are rarely used except for the advanced versioning features.

An `IResource` can be “casted” to a `IVersionControlledResource` semantic object, if `isVersioned()` returns true. It can be “casted” to an `IVersionResource` semantic object, if `isRevision()` returns true. For example:

```java
if( resource.isA(IVersionControlledResource.class) ) {
    // resource is a VCR
    IVersionControlledResource vcr = (IVersionControlledResource)resource
        .as(IVersionControlledResource.class);
}
```
5.3.2.2.1 Working Resources

From a client’s point of view, working resources just differ from in-place checkout in so far as the resource they are applied to: An in-place checkout refers to the (checked-in) version controlled resource (VCR), whereas the checkout of a working resource refers to a version (revision).

A working resource can also serve as a temporary resource on the server side that can hold the content and properties of the version to be checked-in (with basic versioning’s in-place check-out, the client has to store the content and properties on its own).

Working resources are represented by the IWorkingResource interface and created by a checkOut() call for an IVersionResource (while basic versioning works on IVersionControlledResources).

For example:

```java
IResource resource = ...;
IVersionHistory versions = resource.getVersionHistory();
IResource currentVersion = versions.get(versions.size()-1);
// now currentVersion is the current version of resource
currentVersion.checkOut();
```

5.3.2.2 Workspaces

The methods for creating a workspace are located in the IExtendedCollection interface. To retrieve such an “extended collection”, a normal collection (for example, the repository’s root collection) has to be “casted” to an IExtendedCollection semantic object:

```java
ICollection collection = ...;
if( collection.isA(IExtendedCollection.class) ) {
    IExtendedCollection extendedCollection = (IExtendedCollection)collection.as(IExtendedCollection.class);
}
```

To create a workspace, the method createWorkspace() is called, which returns an ICollection representing an IWorkspaceResource:

```java
ICollection collection = ...;
if( collection.isA(IExtendedCollection.class) ) {
    IExtendedCollection extendedCollection = (IExtendedCollection)collection.as(IExtendedCollection.class);
    ICollection workspace = extendedCollection.createWorkspace("myWorkspace", null);
}
```

This collection can then be treated as a normal collection, using the createCollection() and createResource() methods for non-versioned collections and resources as usual.

To create a VCR within the namespace of the workspace, the method createVersionControlledResource() is used.
Consider the example in 2.3.2.4, where R1v5 is a revision of R1, and assume that /repository/Ra is the parent of R1. To place R1v5 as a checked-out workspace resource in the workspace, the code might look like the following:

```java
IResource R1 = ... // create the parent in the workspace
IExtendedCollection Wa = (IExtendedCollection)workspace.createCollection("Ra", null).as(IExtendedCollection.class);
// get the version the workspace controlled resource should refer to:
IVersionHistory R1versions = R1.getVersionHistory(); // first get the history
IResource R1v5 = R1versions.getRevision("v5"); // then the "v5" version of R1
// now create the VCR for R1v5
IResource R1v5VCR = Wa.createVersionControlledResource(R1v5.getRID(), null);
// and finally check it out
R1v5VCR.checkOut();
```

Since R1v5VCR in the example above is a workspace controlled resource (WCR), it could be “casted” to an IWorkspaceControlledResource semantic object. This interface provides getWorkspaceRID() for retrieving the RID of the workspace to which the WCR belongs.

To check-in the checked-out resources of a workspace, the workspace has to be “casted” to an IWorkspaceResource semantic object. This interface offers getCheckOutSet() for retrieving the RIDs of all checked-out resources in this workspace.

For example:

```java
IWorkspaceResource myWorkspace = (IWorkspaceResource)workspace.as(IWorkspaceResource.class);
// retrieve the RID's of all checked-out resources
IRidSet rids = myWorkspace.getCheckOutSet();
// check the resources in again
IRidIterator iterator = rids.iterator();
while(iterator.hasNext()) {
    IResource resource = ResourceFactory.getInstance().getResource(iterator.next(), context);
    resource.checkIn(null, null);
}
```

5.3.2.2.3 Auto-Versioning and Automatic Version Control

Automatic versioning (see 2.3.2.5) is not currently be set by the client. The IVersionControlledResource interface offers getAutoVersioningMode() for retrieving the auto-versioning mode as provided by the repository for the given resource.

If automatic version control (see 2.3.2.5) is available for a collection, it can be “casted” to an IVersionController semantic object that offers the necessary methods enable(), disable() and isEnabled() for enabling, disabling, or retrieving the automatic version control flag for the given collection.

5.3.2.2.4 Version Controlled Collections

Version controlled collections (see 2.3.2.6) are created by simply enabling version control on an ICollection instead of an IResource.
5.4 Built-In Extensions and Additional Aspects

The following sections give a brief overview of the RF's built-in extensions, which can be used by RF clients. The built-in repository managers SFSRepositoryManager and CmRepositoryManager are not covered here, since a client should not rely on the specific implementations. However, a client could use the /etc repository to retrieve application data (for example, configuration data) and might as well configure a repository instance, where it could store its data.

5.4.1 Repository Services

Repository services supply additional aspects that are not known to the RF, but are related to a repository (see 6.4).

5.4.1.1 Application Properties

Application properties behave like resource properties. In addition, application properties can be dependent on resource and user, whereas resource properties are assigned to the resource only. The default implementation of an IApplicationProperties repository service, as provided by the RF, stores the application properties in a database table – it does not rely on the repository's ability to handle resource properties. It is therefore independent of the repository implementation used and can be attached to any repository. However it relies on events to delete application properties when a resource is deleted.

5.4.1.2 Service ACLs

Service ACLs can be used by an application or extension to store additional ACLs that define the specific permissions for that application. For example, the subscription service introduces a new permission "subscription allowed" and uses this service to store the ACEs with these permissions.

The registration of new permissions has to be carried out on start-up by the application, using the IResourceAclManager’s addSupportedPermission() method. The service ACL’s IResourceAclManager is provided by the IAclService’s getAclManager() method.

ACEs are created or modified by an administrator, who uses the KM’s ACL editor to define the ACLs of resources (see 5.2.4).

Whenever an application wants to perform an operation that requires a specific permission (one of those registered at start-up) on a resource, it uses the IAclManagers isAllowed() method to check for the required permission, the given resource, and the current user from context.

The RF’s default IAclService repository service implementation uses a database to store its ACLs and ACEs. As it is independent of the repository implementation, it can be attached to any repository, but it relies on events to delete ACLs of resources that are deleted.
5.4.2 Global Services

Global services offer additional operations on resources (see 6.5).

5.4.2.1 MIME Type Handler

MIME types are used by applications to determine the “type” of a resource’s content and information about that type.

Since not all repositories support the correct MIME type for their resource’s content, and because an application might need to determine the content type to be created for a resource that is still empty, the IMimeHandlerService is used: It returns the MIME type information in an IMime object for a given resource’s RID, based on the RID’s extension. It retrieves the information from its configuration: An administrator specifies which extension should map to which piece of MIME type information.

The IMime object contains information on which icon to display in the navigation and what description should be displayed.

For example, the extension .GIF might be mapped to the MIME type image/gif using image.gif stored in /etc/public/mimes/images as an icon, and .HTML to text/html using html.gif in the same folder.

5.4.2.2 Notifications

Notifications are predefined forms that an application can send to users. For example, the subscription service uses notifications to inform users that a document has changed.

In order to use the INotificatorService, an application must implement its own notifications. This is achieved either by implementing the INotification interface anew or by extending the AbstractNotification, which already implements a property-based XML-/XSLT-handling: The application provides the properties that represent the data to be sent. The XML files represent the language-specific texts to be used for the notification. The XSLT specifies how the properties and the content of the XML are merged to build the content for a certain output channel (for example, for an email message).

To send a notification, an application then uses the INotificatorService’s send() method.

5.4.2.3 Object Type Handler

The object type handler allows IActions to be defined for resources. The actions for a resource are selected by using a combination of RIDs (with wildcards) and/or resource type as criteria.

An application developer specifies the IActions in an .oth XML-File in the /etc/oth collection, together with the selection criteria. Each XML file specifies an IObjectTypeHandler.

The application then retrieves the IActions by getting the appropriate IObjectTypeHandlers, which match the specified criteria, using the IObjectTypeHandlerService’s getObjectTypeHandler() methods.

For example, the action inbox uses these IActions, to determine the available actions for the several types of action inbox items.
5.4.2.4 Publishing Pipeline and XSLT Pipeline

The publishing pipeline service `IPipelineService` enables the general transformation of resource content from one format to another format or from one layout to another:

- An `IProducer` is responsible for providing the content of a resource, stream, or string to the publishing pipeline as a `RequestData` object.
- An `IProcessor` applies the transformation to the `RequestData`, for example, adds a new XML element to an XML document.
- The `IFormatter` finally converts the `RequestData` back into a specific output format, for example, into a stream again.

As well as the publishing pipeline, the `IXsltPipelineService` offers certain convenient methods for creating a pipeline for processing XML with XSLT.

5.4.2.5 Relations

The relation service `IRelationService` stores relations between RF objects in a separate database. Relations belong to a relation type that defines what type of objects are related to each other, and the roles of the source and target of the relations. For example, the relation type “attachments” links two resources, where the source is the document to which the target is attached.

The operands are represented as `IRelationOperand`s, and their type as `IRelationOperandType`. Currently supported types are `IResourceOperandType`, which represents resources as operands and `IStringOperandType`, which can be used for non-resource objects, identified by a unique key.

Each tuple for such a relationship (for example, “document A is attached to document B”) is represented as an `IRelation` object.

The relation service offers methods for the fast retrieval of relations for a resource or an object ID and the ability to manage the relations for specific resource operations (for example, a relation might be automatically deleted if its source operand is deleted).

5.4.2.6 URI-Mapping

As discussed in 2.2.3, the RID of a resource can change over time. If an application requires a unique ID, which does not change but also identifies a resource, the `IUriMapperService` can be used.

It offers just three methods:

- `getCreateConstantID(RID rid)`: tries to retrieve the mapped unique ID for a given RID and creates a new mapping between a new unique ID and the RID if it does not already exist.
- `getConstantID(RID rid)`: retrieves an existing mapped unique ID for the given RID.
- `getRIDFromConstantID(String constID)`: retrieves the RID for the given unique ID if such a mapping between the given unique ID and RID exists.

The service stores the mappings in a database table and uses the event mechanism, to update the mappings if when a resource is moved or renamed. It also deletes the mapping if the resource is deleted. An application will therefore only be able to retrieve a mapped unique ID for an undeleted resource – it should register itself for the `PRE_DELETE` and `DELETE` event if it also needs to remove references to the unique ID of the deleted resource.
5.4.2.7 URL-Generator

If an application needs an HTTP-URL to address a resource in the portal, for example, in order to send an e-mail that includes a link to a specific resource, the IURLGeneratorService provides methods for resolving the URLs of the relevant KM applications.

5.4.3 Basic Services

The basic services are additional services that are not related to resources but are provided as global services for now as well.

5.4.3.1 Application Log

If an application or extension wants to log logging information, it can use the IAppLogService to write its logging information into an IAppLog. The IAppLog is similar to the logging and tracing API, described in 6.1.3, and might be replaced by the APIs described there. For now, its implementation allows applications to write the lock and it offers a log viewer that can be used by an administrator.

5.4.3.2 Caches

The ICacheService offers getCache() for retrieving an configured implementation of an ICache (package com.sapportals.wcm.util.cache). This cache can be used by an application or extension to cache frequently requested objects. It is configured using the KMs configuration UI. The caches implement local caches and are not cluster-aware. The application is therefore responsible for handling the proper cache invalidation in a clustered environment.

5.4.3.3 System Landscape

Since the RF’s main task is to integrate various backend systems, the applications and extensions might require information about the systems they should connect to. This is handled by the ILandscapeService.

It offers the method isClusterInstallation() for determining whether the entire system is running in a clustered installation. The method getSystemFactory() retrieves the ISystemFactory, which is used in turn to retrieve information about a specific system. A system is identified by its ID, whereas the information about it is represented as ISystem.

A system’s type determines its ICredentials, which specify the logon credentials required to logon to a system of that type and to carry out the usermapping between portal and backend system.
5.4.3.4 Scheduler

The scheduler service ISchedulerService supports ISchedulerTasks that represent tasks that have to be executed periodically. An application or extension has to define its own scheduler tasks in the configuration in order to have them configured to run at a certain time.

The ISchedulerService’s method createSchedulerEntry() creates a task that has to be re-registered again on the next start-up.

An implementation of a ISchedulerTask is instantiated by the scheduler on the start-up of the RF. Its run() method is then called periodically, as specified by its ISchedulerTimeTable.

5.4.3.5 Task Queue

In order to support “large” time-consuming computations being sourced out, an application can use the scheduler to perform batch tasks at times when only few users are logged in. The task queue offers another method for separating those batch tasks from the application that is useful especially in a load-balanced installation:

The application retrieves an ITaskQueueWriter for a specific namespace and ID from the ITaskQueueService and uses it to place ITasks into an ITaskQueue. The ITaskQueue stores the tasks and distributes them to the ITaskQueueReaders.

On the other side, the application batch job retrieves the appropriate ITaskQueueReader (for the applications namespace and ID) from the ITaskQueueService and uses it to get the ITasks sent by the application from the ITaskQueue.
6 Extending the RF

As illustrated in 3.1, the following types of RF extensions can be plugged into the RF:

- **Repository Managers** are used to expose objects from a backend system with the RF’s unified aspects.
- **Repository Filters** and **Filter Managers** are used to manipulate the RF resources.
- **Repository Services** are used to add additional unified aspects to objects of a backend system.
- **Global Services** are used to add additional (unified) functionality to the RF resources.
- **Semantic Objects** and **Semantic Object Providers** are used to convert RF resources into application-specific objects.

As all these extensions dwell inside the RF, their runtime environment will be described first. The different extension types will then be described in more detail.

6.1 Runtime of RF Components

All extensions (regardless of type) and the built-in services are referred to as RF components.

6.1.1 Configuration Framework

The Configuration Framework (CF) stores the configuration data for all RF components. It offers a GUI that allows administrators to change the configuration for the RF’s components (see [Admin Guide] on how to use the Knowledge Management's configuration UI).

Usually, developers only have to concern themselves with the configuration classes that define the set of parameters that an extension needs. For example, a CmRepositoryManager.cc.xml specifies the configuration class of the RF’s default repository manager implementation. Configuration classes are built using the PDK Eclipse Plugin “Configuration Wizard”.

The configurables are instances of configuration classes, for example the configuration data for the /documents repository (stored in documents.co.xml) is an instance of CmRepositoryMananager.cc.xml. New configurables are created with the help of the configuration UI.

For further details on how to create configuration classes and using configurables, see [CF].

The retrieval of a component’s configuration data is handled by the component runtime, which is described in the following section.
6.1.2 Component Runtime

The Component Runtime (CRT) manages the start-up and shutdown of RF components. It also provides the RF components with their configuration data from the Configuration Framework. The components’ state can be inspected using the CRT component monitor, so an administrator can find out which components were started and which could not be started and why.

Usually, the RF extensions do not need to concern themselves with the CRT, because extensions are derived from the relevant RF classes (AbstractManager, AbstractFilterManager, AbstractRepositoryService and AbstractService; see 6.2, 6.3, 6.4 and 6.5), which already implement the required CRT interfaces for starting, stopping and configuring a component with its configuration data.

However, for implementing additional features, it might be helpful to know the CRT’s interfaces:

Most of these interfaces are tagging interfaces, like java.lang.Cloneable, for indicating a specific feature of the component.
6.1.2.1 Components, the Component Hierarchy, and Component Information

All components within the CRT implement the IComponent interface. Each component

- Is implemented by the Java class specified in the class attribute of the component's configuration in the Configuration Framework
- Has a public default constructor (without any arguments)
- Has a key that identifies the component within its container (see below)
- Has a unique URI (for example, crt://cm/repository_manager/documents) with the URI’s protocol set to crt. The path of the URI consists of the component keys along the component hierarchy. This URI can be used to lookup a component.

Components that manage child components implement the IContainer interface. For example, the AbstractManager, which is the base class for a repository manager, contains its event broker as a child component (see 6.2).

The IComponentInfo and ILifecycleInfo interfaces support the retrieval of some status information that is used by the CRT component monitor to show the current status of a component.

6.1.2.2 Lifestyle and Singletons

Components can have different lifestyles. The lifestyle defines how a component is instantiated:

- The IThreadSafe interface indicates that a component is created only once by the CRT. Each lookup for the component then returns the same instance of the component, handling the component like a singleton.
- IPooleable indicates a component whose instances are being pooled by the CRT.
- Usually, IPooleable is combined with IReusable, which indicates that a component’s instance can be reset (the CRT calls reset() and then reused). IPooleable or IReusable must not be combined with IThreadSafe.

If a component does not implement one of these three interfaces, a new instance is created each time the component is looked up or retrieved, for example when a global service is retrieved using IServiceFactory.getService().

6.1.2.3 Lifecycle, State and Configuration

The IStartable interface indicates that the component performs some initialization on start-up and carries out a certain amount of clean up during shutdown.

The initialization during start() should be used by all components (instead of doing this in the components constructor). In particular, lookup of other components must not be carried out in the components constructor, since this can lead to indissoluble lookup sequences during start-up, causing a CyclicDependencyException to be thrown.

IAutoStartable indicates components that are instantiated automatically during the start-up of the CRT (whereas components that are not started automatically are not be instantiated until the first lookup). Because automatic start-up only makes sense for a singleton, IAutoStartable is also derived from IThreadSafe.
IConfigurable indicates a component that requires configuration data. The CRT retrieves the components configuration and calls configure().

A component implements the IReconfigurable interface in order to enable hot reload. Hot reload is a component’s ability to handle changes to its configuration without requiring a system restart. When a component implements IReconfigurable, the CRT calls reconfigure() with the changed configuration. The component must then change its internal state according to the new configuration.

IReconfigurable can be used in conjunction with ISuspendable, which indicates that a component should be stopped during reconfiguration and resumed afterwards. This is especially useful if a component is not able to process any other requests during reconfiguration. The CRT calls suspend() before and resume() after the reconfiguration for an ISuspendable, and the component will be marked as stopped during reconfiguration, which inhibits any further lookups for that component during the suspension.

A component that implements all available lifecycle interfaces (state and configuration) has the following methods called by the CRT:

1. Constructor: The component is instantiated.
2. IContextualizable.contextualize(): The component receives a CRT context object that might contain parameters from the component’s container (its parent component).
3. IConfigurable.configure(): the component receives its configuration data.
4. IStartable.start(): The component initializes and acquires the resources necessary to provide its functionality.
5. ISuspendable.suspend(): The component is suspended for reconfiguration.
6. IReconfigurable.reconfigure(): The component receives new configuration data.
7. ISuspendable.resume(): The component is accessible again after reconfiguration.
8. IStartable.stop(): The component releases the acquired resources.
Each time a component changes its state (when a lifecycle interface method is called by the CRT), the component should call the appropriate `pre...()` - and `post...()` - methods of the `ComponentStateHandler` when entering or respectively leaving the method in order to have its state automatically updated in its `ILifecycleInfo`.

For example, a component’s `reconfigure()` method might look like this:

```java
public void reconfigure(IConfiguration config) throws ConfigurationException {
    this.stateHandler.preReconfigure();
    try {
        ... // do something for reconfiguration here
    } catch (Exception e) {
        // handle exception and throw ConfigurationException
    } finally {
        this.stateHandler.postReconfigure();
    }
}
```

### 6.1.2.4 Component Events

If the component depends on other components (for example, a repository service usually depends on the repository manager it is assigned to), it can implement the `IComponentEventListener` interface in order to be notified whenever a component is added, removed, or changed within the CRT. The CRT calls `notify()` with a `ComponentEvent`, which holds the ID of the added, changed, or removed component.

### 6.1.2.5 Usage of CRT Interfaces for RF Extensions

The following table shows the CRT interfaces that are implemented by the different RF base classes (semantic object factories do not currently use the CRT):

<table>
<thead>
<tr>
<th></th>
<th>Abstract Manager</th>
<th>Abstract Repository Service</th>
<th>Abstract Service</th>
<th>Abstract Filter Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>IComponent</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IComponentManager</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IComponentInfo</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ILifecycleInfo</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>IConfigurable</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IStartable</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IAutoStartable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IThreadSafe</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IComponentEventListener</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For further information on the CRT, see [CRT].
6.1.3 Logging and Tracing

Logs are used for reporting problems, that is, for example, security auditing or database logging. The class \texttt{com.sap.tc.logging.Category} can be used for logging to such a (distinct) log. While several processes or objects might write to the same log, a \texttt{com.sap.tc.logging.Location} represents the “source” location of the information logged.

Traces are used for tracing code execution – they are special logs used for support and development. A trace is the default category, and the class should be used as the tracing location within the RF.

For example:

```java
import com.sap.tc.logging.Location;
...
private static Location trace = Location.getLocation(myClass.class);
...
    trace.errorT("myMethod", "myMessage");
```

The example creates a location instance \texttt{trace}. If \texttt{trace.errorT()} is called, an error trace is written to the RF’s trace file.

The different methods for the several \texttt{severities} are:

- \texttt{fatalT()}: Should be used for errors that cause the system to stop or prevent further action (for example, “out of memory”).
- \texttt{errorT()}: Should be used for errors that cannot be resolved automatically and when data might be lost (for example, when a backend connection becomes unavailable).
- \texttt{warningT()}: Should be used for error that can be recovered by the system (for example, database connection lost but recovered after a reconnect).
- \texttt{infoT()}: Should be used for tracing the various paths through the program’s execution (usually at least one trace per relevant method and potentially one for the \texttt{if-else}-branches of relevant \texttt{if} statements).
- \texttt{debugT()}: Should be used for tracing the detailed developer information required to trace a problem, if the result of the execution is not as expected.

For performance reasons, tracing should only be performed for warning, info, or debug traces if the appropriate severity is set. Moreover, since the line number is not included in the trace output automatically, it might be a good idea to include it in the first parameter.

For example (using 123 as line number):

```java
if( trace.beInfo() ) {
    trace.infoT("myMethod(123)", "my info message");
}
```
To trace the backlog of an `Exception` or `Throwable`, the utility class `LoggingFormatter` is provided:

```java
import com.sap.wcm.util.logging.LoggingFormatter;
...
try {
...
}
catch( Exception e ) {
    trace.errorT(
            "myMethod",
            "caught exception " + e.getMessage()
            + ":" + LoggingFormatter.extractCallstack(e)
    );
}
```

SAP Logging and Tracing offers a lot of additional functionalities. For further details see [Logging].

### 6.1.4 Clustered Installation

Since the RF has to support a large number of users working concurrently, performance is one of the major issues in the RF.

In order to support more users than one server might be able to handle, the RF can be distributed across several servers – this is called a **clustered installation**. Within a cluster, several instances of the RF run on different machines (in different VMs), but these instances use the same database (configuration, backend connections, and so on).

Each RF extension must be aware of supporting a clustered installation by implementing the relevant synchronization mechanisms.

An extension can check for the clustered installation using the following code fragment:

```java
boolean isClustered = true;
try {
    ILandcapeService landscapeService =
    (ILandscapeService)ResourceFactory
        .getInstance()
        .getServiceFactory()
        .getService(IServiceTypesConst.LANDSCAPE_SERVICE);
    if( landscapeService != null ) {
        isClustered = landscapeService.isClusterInstallation();
    }
} catch( ResourceException e ) {
    // trace exception
}
```
6.2 Repository Manager

Since the new API is used to build a repository manager, all packages described in this section are located beneath com.sap.netweaver.bc.rf. To shorten references to the classes and interfaces, com.sap.netweaver.bc.rf is omitted from the package names.

6.2.1 Implementation Considerations

As explained in 3.1, a repository manager is responsible for the mapping of backend objects and their operations to the RF’s unified resources and their aspects.

One task, when implementing a repository manager, is to design this mapping. This involves the following issues:

- **Namespace:**
  How should the resources’ RIDs be built, what business objects can be represented as collections (containing other resources but do not have unstructured content), and what objects require unstructured content or should be treated as leaf nodes of the hierarchy (becoming resources)?

- **Content and properties:**
  Which parts of the business objects data should be represented as structured properties and which data should be treated as unstructured content? How should objects that handle several blocks of unstructured data be represented?
  (This is closely related to the namespace design.)

- **Security and locking:**
  How to map users and the permissions of the backend system to the RF’s permissions and what resource context data is required to handle it?

Another task is the design of the backend implementation. This depends very much on the specific requirements of the backend system. Things to consider are:

- **Establishing a backend connection:**
  When to use a dedicated connection for each client request and when to use connection pooling instead?
  Which parameters are required to establish a backend connection?
  (This might be related to the security design, when a backend uses the connection data to apply security checks)

- **Achieving performance:**
  Which data can be cached and how are changes propagated to the caches, especially when running in a clustered environment?

The following sections give some hints on how to design the mapping. How to design the backend connection is not covered here, since this is probably too specific to each backend system.
6.2.1.1 Some Rules of Thumb

These “rules of thumb” should only be considered as guidance on how to design a repository manager. These are no “hard rules” that have to be adhered to in every repository manager design:

- All relations between objects that are to be used for navigation in the KM UI have to be modeled as parent-child relations.
  (see the employee – manager relation in the following example).

- Links should be used for modeling relations between resources, where the relation's target is already somewhere else in the hierarchy with a “primary” RID.
  If not modelled at all, navigation will not be possible; if modeled as a “virtual” resource instead (see below), applications will see two “different” resources. This can lead to duplicate entries in the search result list.

- A meaningful display name should be provided for the resources.
  It is also a good idea for names of collections and resources to be meaningful to human readers. *)
  (For example /example/all_users/developers/smith instead of /example/0x072393/2422 for the following example).

- A collection should not have too many children.
  If an object has many other objects related to it, it might be better to group the different types of relations in different sub-collections (see the following example).
  Otherwise, navigation will be cumbersome. *)

- If the KM UI is to be used for editing an object's attributes, these attributes should be modeled as properties.
  (In the following example, if users should be able to edit their e-mail, the e-mail address should be exposed as a property.)

- If an object or a part of it can be represented as a stream, the stream should be exposed as a resource’s content.
  (See the picture of a user in the following example.)

- If an object has several parts, all of which can be represented as streams, the object should be modeled as a collection containing several “virtual” resources. These “virtual” resources represent intermediate objects that are not mapped to an object in the backend. Each “virtual” resource serves as a container for contents of the different parts.
  (see the following example)

- Collections should not have content since the KM UI does not support it.

- Implementing versioning might cause some headaches (especially advanced versioning)... It is therefore recommended that you check the availability of some good pain relievers first. *)

*) Might not be an issue if no end user navigation is required and only search and query-based taxonomies have to be supported.
6.2.1.2 Example Scenario

To illustrate some of the design issues, consider the following scenario:

A company’s phone book is to be made available in the SAP Enterprise Portal. For navigation, the KM’s flexible UI is to be used; the KM’s search and subscription features are to be used as well.

In order to achieve this, a repository manager for the phone book’s objects has to be implemented. The phone book contains users and groups. The user data contains name, office, phone number, mobile phone number, fax, e-mail, the user’s manager and a picture of the user. The group data contains the list of users and groups that belong to the group (groups can contain other groups), an e-mail distribution list, and a responsible administrator for the group.

When designing the hierarchy, it seems obvious that groups – since they contain other groups and users – should be modeled as collections. Users are a little bit trickier: Modeling them as resources implies that all attributes of a user have to be modeled either as properties or content, preventing navigation for them. This might not be a problem if no further navigation is required, but if, for example, navigation to the user’s manager is required, the user has to be modeled as a collection that contains a link to its manager (see “rule of thumb” for modelling relations between objects).

These issues show how the namespace design affects the data representation: If the user is modeled as a collection, its picture could then be modeled as a resource in a “user collection”, while its attributes might be modeled as the collection’s properties. This also adheres to the “rules of thumb” on stream data (for the picture) and on editable attributes (for the other attributes, for example, e-mail). If the user contains other data streams, for example, picture and curriculum vitae, the “user collection” would then contain two “virtual” resources, one for the “picture” data and one “vitae” resource, providing the curriculum vitae as its content.

It might also be possible to expose the attributes as XML elements in an XML file as another “virtual” resource inside the “user collection”.

Following the “rule of thumb” for modelling relations, the user’s manager is an example of how a link should be used: The manager is already represented in the hierarchy using its “primary” path (as a “normal” user); the link references it from somewhere else via a “second” path (as another user’s manager).
6.2.1.3 Mass Calls

For the sake of a better overview, in the following descriptions of the repository manager and its components the mass calls have been ignored.

Nevertheless, implementing an interface requires all interface methods to be implemented. A repository manager should therefore do one of the following:

- Implement the mass calls by throwing an `OperationNotSupportedException`
- Implement them by calling the “single” calls in a loop, collecting their result in a list (this is automatically done for `getResource()` by the RF, if not supported by the repository manager).
- Implement them in the most efficient way possible (which is also the best)
6.2.2 Components of a Repository Manager

As shown in 3.1, repository managers are registered with the RF’s manager registry, which forwards the RF’s requests to the manager.

A repository manager has to be derived from mi.AbstractManager. This base class handles the various interfaces that are required for interaction with the component runtime and handles registration with the manager registry.

The AbstractManager’s methods startUpImpl() and shutDownImpl() can be overwritten by the repository manager if something has to be initialized on start-up (for example, if a connection to the backend system has to be established) or has to be de-initialized on shutdown (for example, when the connection has to be closed again).

In order to implement the functions for the various aspects of resources, the repository manager might contain several submanagers, one for each aspect of a resource. As explained in section 3, all these aspects – and therefore the submanagers that implement the aspect’s functionality – are optional.

The repository manager’s submanagers have to be derived from mi.AbstractSubManager. As for the mi.AbstractManager, the mi.AbstractSubManager’s methods startUpImpl() and shutDownImpl() can be overwritten by the submanager implementation, if initialization or de-initialization are required during start-up or shutdown.
The repository manager

- Must be derived from `mi.AbstractManager`
- Must implement the `getSupportedOptions()` method (from `mi.IManager`, see 6.2.3)
- Must implement the various `lookup()` methods (from `mi.IManager`, see 6.2.4)
  (However only the signature `lookup(RID)` has to be implemented while the others might throw an `OperationNotSupportedException`)
- Should implement the `getNameInfo()` method (see 6.2.4)
- Can implement `startUpImpl()` and `shutDownImpl()`
- Might implement a default constructor (without parameters)

Each submanager

- Must be derived from `mi.AbstractSubManager`
- Must implement a constructor that takes the `mi.IManager` as parameter
- Must implement one of the read-only aspects (for example, `mi.namespace.INamespaceManager`)
- Can implement `startUpImpl()` and `shutDownImpl()`
- Can implement the mutable interface for the aspect it implements (for example, `mi.namespace.IMutableNamespaceManager`)

6.2.3 Mapping RF Objects to Backend Objects

As explained in 5.1.1, applications deal with IResource and ICollection, which are the representation for RF objects from an application’s point of view. They represent the unified aspects of the RF’s objects.

A repository manager can implement an internal reference for the backend objects, which are passed to the RF as IResourceHandle implementations. The RF encapsulates these objects from the repository manager in its internal IResource (new API) implementations. The RF’s IResource implementation acts as a proxy for the repository manager’s IResourceHandle implementation (see below), preventing applications from using the repository manager specific implementations but enhancing the IResourceHandle (from which IResource is derived), with additional unified operations.

The common.IResourceHandle serves as an abstract pointer for linking the repository manager’s internal representation of a backend object to the RF’s unified representation. It provides just one method, getRid(), which returns the RID assigned to the RF object, this handle refers to. In the new API, the RID is represented as a common.IRid.

If a client requests a lookup for a given resource’s RID, the RF determines the responsible repository manager and forwards the lookup request to this manager. The repository manager retrieves its internal representation of the resource from the backend and returns its implementation of IResourceHandle to the RF, which then references the manager’s backend object.

The RF in turn wraps this handle into its implementation of an IResource and exposes its unified representation of the resource to the client.

When the client requests an operation to be performed on the IResource object, the RF determines the responsible repository’s submanager for the relevant aspect, checks whether the submanager is available, and whether it supports the requested operation. If the submanager exists and supports the operation, the RF forwards the request to the submanager, passing the repository manager’s implementation of IResourceHandle back again to the submanager. The submanager uses the reference from IResourceHandle to retrieve its internal backend object representation of the resource.
When designing the internal representation of an `IResourceHandle`, the following UML diagram might give a hint on how to manage the several RF objects internally:

The repository

- Should provide `MyHandle`, which is the implementation of `IResourceHandle`. `MyHandle` implements `getRid()`, and should contain a reference to `MyNode` and to the resource’s access context data, which is relevant for accessing the object in the backend system (for example, the user).

- Should provide `MyNode`, which is the base class for all types of backend objects to expose. It should contain all basic functionality common to all types of backend objects (`MyNode` represents a general resource).

- Can provide `MyResource` and any other class for implementing specific operations for the RF object types (notably the collection `MyCollection` and link `MyLink`) that extend the common functionality provided by `MyNode` (`MyResource` represents a leaf resource and `MyCollection` a node resource).

If the backend system exposes various types of objects, another class hierarchy can be designed to reflect the backend’s object type hierarchy. The classes derived from `MyNode` can then perform the mapping to the classes of the backend object hierarchy.
A sample implementation of a handle might look like:

```java
public class MyHandle implements IResourceHandle {
    private final MyNode node;
    private final IUser user;

    MyHandle(MyNode node, IUser user) {
        this.node = node;
        this.user = user;
    }

    public IRid getRid() {
        return this.node.getRid();
    }

    protected MyNode getNode() {
        return this.node;
    }

    protected IUser getUser() {
        return this.user;
    }
}
```

To indicate which operations are supported for a given object, the repository manager has to implement the `getSupportedOptions()` method, which receives an `IResourceHandle` as a parameter and returns a `Set` of supported options (see 3.3.2) for the type of object, the handle refers to. If the handle is null, `getSupportedOptions()` should return all options it supports.

```java
private final static HashSet collectionOptions;
private final static HashSet resourceOptions;
private final static HashSet allOptions;
static {
    collectionOptions = new HashSet(5);
    collectionOptions.add(….common.property.SupportedOption.GET_PROPERTIES);
    collectionOptions.add(….common.property.SupportedOption.SET_PROPERTIES);
    collectionOptions.add(….common.namespace.SupportedOption.CREATE_RESOURCE);
    collectionOptions.add(….common.namespace.SupportedOption.CREATE_COLLECTION);
    collectionOptions.add(….namespace.SupportedOption.DELETE);
    resourceOptions = new HashSet(5);
    resourceOptions.add(….common.property.SupportedOption.GET_PROPERTIES);
    resourceOptions.add(….common.property.SupportedOption.SET_PROPERTIES);
    resourceOptions.add(….content.SupportedOption.GET_CONTENT);
    resourceOptions.add(….common.content.SupportedOption.UPDATE_CONTENT);
    resourceOptions.add(….common.lock.SupportedOption.LOCK);
    allOptions = new HashSet(collectionOptions);
    allOptions.addAll(resourceOptions);
}
```

```java
public Set getSupportedOptions(IResourceHandle handle) {
    if (handle == null) {
        return allOptions;
    }
    if (!((MyHandle)handle).getNode() instanceof MyCollection) {
        return resourceOptions;
    }
    return collectionOptions;
}
```

The example above would indicate that all types of resources allow properties to be retrieved or to be set. Collections allow new resources and sub-collections to be created or existing resources to be deleted. Only resources that are no collections support locks and the retrieval of content as well as the updating of content.

The example assumes that `MyCollection` is derived from `MyNode`. 
6.2.4 RIDs, Lookup and Resource Context

6.2.4.1 RIDs and Lookup
As explained in 2.1.2, an RID consists of a path, name and query. The repository should expose any restrictions that apply to the path and name of an RID using the `IManager.getNameInfo()` method. It returns a `NameInfo` object, which provides information on:

- The maximum path length
- The maximum name length
- The characters not to be used in the name of a collection
- The characters not to be used in the name of a resource

As the path's segments refer to the names of collections, the easiest way to implement the `lookup()` method is a recursive lookup of the parent collections for the requested resource up to the root resource. If the parent collection is found, the requested resource is retrieved as a child (with the name given by the RID) from the parent collection or `null` if the resource does not exist:

```java
public IResourceHandle lookup(IRid rid) throws ResourceException {
    try {
        IUser user = getUserFromContext(rid); // see Resource Access Context below
        if( rid == null ) {
            return null; // null is an invalid RID
        }
        if( rid.equals(this.root.getRid()) ) {
            // the given RID is the RID of the repository’s root resource
            return new MyHandle(this.root, user);
        }
        // recursive lookup of the parent resource for the given RID
        IResourceHandle parentHandle = this.lookup(rid.parent());
        if( parentHandle == null ) {
            return null; // parent handle not found -> wasn’t a valid RID
        }
        MyNode parentNode = ((MyHandle)parentHandle).getNode();
        if( parentNode == null ) {
            return null; // parent node not found -> wasn’t a valid RID
        }
        if( !parentNode.isCollection() ) {
            return null; // parent is not a collection -> wasn’t a valid RID
        }
        // parent found, requested the resource from the parent for the given name
        MyNode node = ((MyCollection)parentNode).getChild(rid.name().getPath());
        if( node == null ) {
            return null; // no resource exists for the given RID in parent collection
        }
        // the parent returned the resource with the given name
        return new MyHandle(node, user);
    }
    catch( Exception e ) {
        // ignore exception in this sample and treat them as if backend unavailable
        throw new ServiceNotAvailableException(rid, "backend unavailable");
    }
    return null;
}
```
If the repository could not access the backend system, it should indicate this with an appropriate exception, not by returning null. Returning null should only be used to indicate that the resource definitively does not exist.

The example above assumes that the repository manager knows its root resource, which is assigned to the repository’s prefix. This prefix is returned as a String by calling AbstractManager.getRidPrefix().

Another assumption is that MyCollection provides a getChild() method, which returns the child resource specified by a given name from this collection.

The lookup() method is the only required method for a repository manager (all other operations for the various aspects are optional). It is the method used to implement the IResourceFactory's getResource() method by the RF.

However, this is not sufficient for navigation: As mentioned, navigation uses the parent-child relations that are provided through the INamespaceManager’s getChildren() method (see 2.1).

### 6.2.4.2 Resource Access Context

The resource’s access context (see 2.1.2) is handled different in the new API than in the current API: Before an operation is requested from the repository manager, the resource’s access context is pushed to the current thread’s context by the RF using a common.context.AccessContextFactory.

The repository manager retrieves the access context through AccessContextFactory. This technique eliminates the resource’s access context as an additional parameter on each call to the manager, making the signatures of the several methods a little less complicated.

This AccessContextFactory is derived from the util.context.AbstractIndependentThreadContextFactory, which handles the stacking of java.lang.Objects within the thread context. The AccessContextFactory extends it with type-safe methods for stacking common.context.IAccessContexts. The AccessContext class provides a default implementation for the IAccessContext.

The repository manager can use this technique to push another IAccessContext to the stack, before calling other methods.
The following example shows how an implementation of `getUserFromContext()` (as used in the `lookup()` example above) for retrieving the information from the access context. It could look like:

```java
protected IUser getUserFromContext(IRid rid)
throws ResourceException {
    try {
        IAccessContext context = AccessContextFactory.getInstance().getContext();
        return context.getUser();
    }
    catch( ContextUnavailableException e ) {
        trace.errorT("getUserFromContext()", "no context found");
        throw new ResourceException(rid, e);
    }
}
```

### 6.2.5 Namespace Manager

The `mi.namespace.INamespaceManager` interface defines the methods required for a read-only namespace submanager.

The first group of functions are those for determining the type of a resource:

- `boolean isCollection(IResourceHandle handle)
  throws ResourceException;

  Returns true if the given handle references a collection, false if not.

- `boolean isLink(IResourceHandle handle)
  throws ResourceException;

  Returns true, if the given handle references a link, false if not.

- `ILinkDescriptor getLinkDescriptor(IResourceHandle handle)
  throws ResourceDescriptor;

  Returns an `ILinkDescriptor` that holds the `LinkType`:
  - `NONE` if it's not a link
  - `INTERNAL_STATIC` if it is an internal (static) link
  - `EXTERNAL_STATIC` if it is an external link
  - `REPOSITORY_DYNAMIC` if it's a flexible (internal dynamic) link

  It also holds the target's RID (for internal and flexible links) or URI (for external links).

The second group of functions are those used for retrieving the children of a resource:

- `List findResources(IResourceHandle rootHandle, IFindResourcesDescriptor findDescriptor, int offset, int length, Object resultStateDescriptor)
  throws ResourceException, OperationNotSupportedException;

  Returns a list of handles for the children of the given `rootHandle` that match the given `findDescriptor`.
This variation of `findResources()` is used for navigation, that is to retrieve all the direct children of a collection.

- The `findDescriptor` specifies the selection criteria that the resources to be returned must match (see the following section for a detailed description of `IFindResourcesDescriptor`).
- An offset defines how many matching resources should be skipped and will are not included in the result list. If the offset is 0, the result list starts with the first matching resource. An offset less than 0 is not allowed.
- `length` defines how many matching resources are to be included in the result list. If the length is less than 0, the list contains all matching resources, starting at the given offset. Using offset and length, a client can retrieve a list of resources in blocks (see example below).
- The `resultStateDescriptor` is an optional parameter that can be used by the namespace submanager to retrieve a “session state” of a previous query. In order to receive the session state of a previous query, the namespace submanager has to return a special `List` implementation, which implements the `common.namespace.IResultStateList` interface: Such a list provides a `getResultSetStateDescriptor()` method that is used by the RF to retrieve the namespace manager’s session state for the result. The Object returned by this method is then passed again to the next call to `findResources()` (see example below).

```java
Iterator findResources(IResourceHandle rootHandle,
                       IFindResourcesDescriptor findDescriptor,
                       int startIndex,
                       int fetchSize)
throws ResourceException,
         OperationNotSupportedException;
```

Returns an `Iterator` instead of a result list. This method is similar to `findResources()` returning a `List` (see above). This variation of `findResources()` is used by the crawlers, when a repository is indexed for search. The namespace manager might simply return the `List.iterator()` or implement its own `Iterator` that handles the `Iterator`’s `next()` method more efficiently for the given result.

- `startIndex` works like the offset parameter for `findResources()` returning a list.
- `fetchSize` is a hint for the namespace manager as to how many consecutive calls to `next()` are usually called by the client.

```java
long countResources(IResourceHandle rootHandle,
                    IFindResourcesDescriptor findDescriptor)
throws ResourceException,
         OperationNotSupportedException;
```

Returns the number of children for the given `rootHandle` that will be found when calling `findResources()` with the given `findDescriptor`. 
The following example illustrates how the RF uses blocked retrieval:

```java
IResourceHandle start = ...;
IFindDescriptor descriptor = new BasicChildrenFindResourcesDescriptor();
Object state = null;
int blockSize = 10;
int blockIndex = 0;
int i = 0;
while( true ) {
    // retrieve block
    List result = findResources(start, descriptor, blockIndex, blockSize, state);
    // handle the retrieved block of resources
    if( result.size() == 0 ) break; // no more resources found
    Iterator iterator = result.iterator();
    while( iterator.hasNext() ) {
        System.out.println( "position " + ((blockIndex * blockSize) + i) + " is " + ((IResourceHandle)iterator.next()).getRid() );
    }
    // goto next block, passing the result state for this block to the next blockIndex += blockSize;
    if( List instanceof IResultStateList ) { state = ((IResultStateList)list).getResultStateDescriptor(); }
}
```

The following UML diagram shows the hierarchy of the various IFindResourcesDescriptors:

```
+getQueryExpression() +getCollator()
+getDepth()
```

© 2003 SAP AG
• The first two descriptors return the children of the given collection. These are used for mapping the current API's `IResource`'s `getChildren()` method:
  o The `IBasicChildrenFindResourcesDescriptor` indicates that all the children of a collection should be retrieved.
  o An `IAdvancedChildrenFindResourceDescriptor` supports additional filter criteria that are represented by a query expression (see 2.2.2). A collator, represented by an `ICollator`, might also specify sorting rules that have to be applied to the resulting list of children (see 2.2.1).

• The remaining descriptors return all the resources in the hierarchy where the given collection is the starting point (they are now used, to return the result for a property search – see 5.2.2):
  o The `IAdvancedDeepFindResourcesDescriptor` also supports collators, like the `IAdvancedChildrenFindResourceDescriptor`. However, unlike the later, the former indicates that the entire hierarchy (and not only the plain child-resources on the first level) should be retrieved.
  o The `IAdvancedDeepVersionedFindResourcesDescriptor` also selects all the versions of versioned resources, whereas the other descriptors only select the current versions.

The other groups of functions are related to advanced topics and will be discussed later (see 6.2.8.1 and 6.2.8.3).

A sample implementation of `findResources()`, returning a `List`, might look like the following:
```java
public List findResources(IResourceHandle startHandle, IFindResourcesDescriptor findDescriptor, int offset, int length, Object resultStateDescriptor) throws ResourceException, OperationNotSupportedException {
    if(   ( findDescriptor instanceof IAdvancedChildrenFindResourcesDescriptor )
        || ( !(findDescriptor instanceof IBasicChildrenFindResourcesDescriptor) )
        || ( offset != 0 )
        || ( length != -1 )
        || ( resultStateDescriptor != null )
     )
        throw new OperationNotSupportedException(startHandle.getRid(), "only basic find descriptor is supported");
    return ((MyCollection)((MyHandle)startHandle).getNode())
        .getChildren(getUserFromContext(startHandle.getRid()));
}
```

The example above assumes, that `MyCollection` provides a `getChildren()` method that retrieves all direct child nodes of a collection.
The remaining namespace operations for creating, copying, renaming/moving, and deleting resources are defined by the `mi.namespace.IMutableNamespaceManager` interface. These are:

**IResourceHandle** `createResource`(`ICreateDescriptor createDescriptor`) throws ResourceException;

Creates a new resource, as specified by the `createDescriptor`. Returns the handle of the newly created resource. If the descriptor contained an `ILockDescriptor`, when the method was called (to create a new resource and lock it in one atomic transaction), the descriptor contains an `ILockInfo` as the result of the atomic lock operation performed during creation afterwards.

**List** `deleteResource`(`IResourceHandle resourceHandle`) throws ResourceException, OperationNotCompletedException;

Deletes a resource.

Returns a list of handles for the deleted resources, where the first element of the list is the handle of the deleted resource specified by `resourceHandle`. Other entries will exist if the `resourceHandle` specified a collection and the children of this collection were also deleted.
List `copyResource` (IResourceHandle resourceHandle, 
ICopyDescriptor copyDescriptor)
throws ResourceException, OperationNotCompletedException;

Copies a resource, link, or collection to somewhere else, as specified by the `copyDescriptor`

Returns a list of handles for the copied resources, where the first element of the list is the handle of the copy for the resource that was given by `resourceHandle`. Other entries will exist if the `resourceHandle` specified a collection and the collection’s children also had to be copied.

List `moveResource` (IResourceHandle resourceHandle, 
IMoveDescriptor moveDescriptor)
throws ResourceException, OperationNotCompletedException;

Moves or renames a resource as given by the `moveDescriptor` (see above).

Returns a list of handles for the moved resources, where the first element of the list is the handle of the moved resource that was specified by `resourceHandle`. Other entries will exist, if the `resourceHandle` specified a collection and the collection’s children where also moved.

void `setLinkDescriptor` (IResourceHandle resourceHandle, 
ILinkDescriptor linkDescriptor)
throws ResourceException

Changes the type or target of a link.
6.2.6 Content Manager

The methods for reading content are defined by `mi.content.IContentManager`:

```java
IContent getContent(IResourceHandle handle)
throws ResourceException;
```

Returns the content for a given resource, specified by `handle`, throws a `common.content.ContentUnavailableException` if the resource has no content assigned.

In the new API, content is represented by `common.content.IContent` with `common.content.Content` as the default implementation. A content submanager can either use the default implementation or implement its own implementation (which might also derive from `Content`).

`IContent` provides the resource’s content as an `java.io.InputStream` using `getInputStream()`; the content’s metadata is provided using `getContentMetadata()`.

The content’s metadata bundles the information about the content (see 2.1.6) and is represented by `common.content.IContentMetadata`.

```java
+getInputStream()
+getContentMetadata()
«interface» IContent
+setMetadata()
«interface» IMutableContent
+getContentLength()
+getContentType()
+getContentEncoding()
+getETag()
+getExpires()
+getLastModified()
«interface» IContentMetadata
```

`common.content.IMutableContent` is used to update the content of a resource. The `mi.content.IMutableContentManager` handles the update of a resource’s content:

```java
void setContent(IResourceHandle handle,
               IMutableContent content,
               Boolean compareETags)
throws ResourceException,
       ContentMetadataMismatchException;
```

Updates the resource’s content with the mutable content given. The content’s metadata is updated as well.

If `compareETags` is `true`, the content is only be updated if the resource’s content has not been modified since the `getContent()` call that originally retrieved the content. In other words: if `compareETag` is `true`, `content.getETag()` must be equal to the manager’s internal ETag for the resource’s content. Otherwise, a `common.content.ContentMetadataMismatchException` is thrown.
Note: A repository manager that supports content and would want to support queries on content metadata as well has to expose the content metadata as properties. The repository manager must therefore implement a property submanager that handles the exposed content metadata properties.

### 6.2.7 Property Manager

Although the properties in the new API are built on the current API’s properties (see 5.1.4), the implementation for property objects has changed a little:

While the current API offers only one implementation for all type of properties, the new API offers specific implementation for the several combinations of types with single-/multivalued properties. These classes are located in `common.property` for the client applications. `common.property.PropertyName` defines the property names that are known to the RF (for example, `CREATEDBY` – see table at the end of this section).

A property submanager uses property classes derived from those in `common.property`, located in `mi.property`. The `mi.property` property implementations provide additional constructors that allow for the attributes of the property to be set.
The mi.property.IPropertyManager handles the read-only property requests:

```java
IProperty getProperty(IResourceHandle handle,
                     IPropertyName propertyName)
    throws ResourceException;
```

Returns the property for the given property’s name and the specified resource or null if it does not exist.

```java
Map getAllProperties(IResourceHandle handle)
    throws ResourceException;
```

Returns all the existing properties of the specified resource in a Map, indexed by the property names as key entries.

```java
Map getListedProperties(IResourceHandle handle,
                         List propertyNameList)
    throws ResourceException;
```

Returns existing properties specified in the propertyNameList for the given resource. The result is a Map, where the properties are the entries and the keys are the property names.

The RF offers the mi.property.SystemPropertyFactory for helping system properties to be created more efficiently by the property submanager (see the table at the end of this section).

The corresponding write requests are defined by the mi.property.IMutablePropertyManager:

```java
void updateProperty(IResourceHandle handle,
                    IPropertyUpdateDescriptor propertyUpdateDescriptor)
    throws ResourceException;
```

Updates a resource property as specified by the propertyUpdateDescriptor.

```java
void updateProperties(IResourceHandle handle,
                      List propertyUpdateDescriptorList)
    throws ResourceException;
```

Updates several properties as specified by the propertyUpdateDescriptors contained in the propertyUpdateDescriptorList.
A `common.property.IPropertyUpdateDescriptor` is either of type `PropertyUpdateType.SET` or `PropertyUpdateType.REMOVE`, implemented by one of the following classes:

- `common.property.PropertyUpdateDescriptor` provides a property with name and value. It indicates that an existing property should be updated with the provided one – or that it has to be created, if it does not already exist.
- `common.property.PropertyRemoveDescriptor`, which provides a property name. It indicates that a property should be removed, if it already exists.

When a client updates a property, it retrieves it, creates a copy by calling `getMutable()`, changes the property’s value, and then passes the changed mutable property as a parameter to the `updateProperty()` method (see 5.1.4).
Because properties are also used for property queries (see 6.2.8.3), some **system properties**, which describe a resource and content, are predefined by the RF. The following table lists those system properties’ names from `common.property.PropertyName` that a repository’s property submanager should be able to provide:

<table>
<thead>
<tr>
<th>property name</th>
<th>description</th>
<th>resource</th>
<th>content</th>
<th>properties</th>
<th>application</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLECTION</td>
<td>true, if the resource is a collection</td>
<td>resource creation</td>
<td>*</td>
<td>single</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>CONTENTENCODING</td>
<td>encoding of the content</td>
<td>resource creation or content update</td>
<td>*</td>
<td>single</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>CONTENTLANGUAGE</td>
<td>the language the content is for</td>
<td>resource creation or content update</td>
<td>*</td>
<td>single</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>CONTENTLENGTH</td>
<td>content’s length in bytes</td>
<td>resource creation or content update</td>
<td>*</td>
<td>single</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>CONTENTTYPE</td>
<td>MIME type of the content</td>
<td>resource creation or content update</td>
<td>*</td>
<td>single</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>CREATED</td>
<td>resource’s creation timestamp</td>
<td>resource creation</td>
<td>*</td>
<td>single</td>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>CREATEDBY</td>
<td>user ID with the resource’s creator</td>
<td>resource creation</td>
<td>*</td>
<td>single</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>a description of the resource</td>
<td>application</td>
<td>*</td>
<td>single</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>DISPLAYNAME</td>
<td>display name of the resource</td>
<td>application</td>
<td>*</td>
<td>single</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>ETAG</td>
<td>entity tag for the content</td>
<td>resource creation or content update</td>
<td>*</td>
<td>single</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>HIDDEN</td>
<td>true, if the resource is hidden</td>
<td>application</td>
<td>*</td>
<td>single</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>LINKTYPE</td>
<td>the code of the resource’s link type</td>
<td>resource creation or link update</td>
<td>*</td>
<td>single</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>MODIFIED</td>
<td>timestamp of the resource’s last modification</td>
<td>resource creation or link update</td>
<td>+*</td>
<td>*</td>
<td>single</td>
<td>Date</td>
</tr>
<tr>
<td>MODIFIEDBY</td>
<td>user ID with the resource’s last modifier</td>
<td>resource creation or content/property update</td>
<td>+*</td>
<td>*</td>
<td>single</td>
<td>String</td>
</tr>
<tr>
<td>READONLY</td>
<td>true, if the resource is read-only</td>
<td>application</td>
<td>*</td>
<td>single</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>RESOURCENAME</td>
<td>the name-part of the resource’s RID</td>
<td>resource’s RID</td>
<td>*</td>
<td>single</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>RESOURCETYPE</td>
<td>the resource’s type</td>
<td>application</td>
<td>*</td>
<td>single</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>

As mentioned, the `SystemPropertyFactory` can be used to construct these properties.
The rows 3 through 6 in the table above indicate the “area” in which the property is used:

- The 3rd row (resource) contains a * for those properties that depend on the resource itself, while a + indicates properties that relate to collections only.
- The 4th row (content) indicates those properties related to the content with a *.
- The 5th row (properties) indicates properties related to property changes with a *.
- The 6th row (application) contains a * for those properties that are usually specified by an application.

These “areas” only describe intended usage – any property submanager can provide these properties as immutable, using the backend’s values for them instead.
6.2.8 Advanced Aspects

6.2.8.1 Ordered Collections

In order to support ordered collections, the IMutableNamespace submanager must support common.namespaceICollectionCreateDescriptors and regard their common.namespace.OrderMechanism as well as handling the setCollectionOrderMechanism() method:

```java
void setCollectionOrderMechanism(IResourceHandle handle,
                                 OrderMechanism orderMechanism)
throws ResourceException;
```

A submanager that does not support ordered collections only allows OrderMechanismType.NONE. A submanager, which does support ordered collections has to support OrderMechanismType.MANUAL or OrderMechanismType.SERVER, or both (see 2.2.1).

If a submanager supports manual ordering, it must be able to handle the reorder method:

```java
void reorderCollection(IResourceHandle handle,
                      List orderPositions)
throws ResourceException;
```

Reorders the children of the collection, specified by the given handle, according to the common.namespace.OrderPositions in the List. (See 5.2.1 for an example of how the feature is used with the current API.)

Please note that the sequence of OrderPositions in the list is significant to the result of the reorder operation.
The new API’s class OrderPosition contains all information of the current API’s interfaces IPositioning and IPosition (see 5.2.1): The type specifies whether the position is either absolutely the first or the last element, or if it is a relative position before or after another resource. The method getResourceHandleName() gets the name part of the resource’s RID that this position is for. For the relative positions BEFORE and AFTER, the method getReferencePointName() returns the name part of the resource’s RID that the position relates to.

### 6.2.8.2 Collators

Since collators are not yet used by the client applications, they will be described here only briefly:

![Collator and CollatorEntry Diagram]

An ICollator extends the List interface and contains several ICollatorEntries. Each ICollatorEntry specifies one ordering criteria: It is the name of the property to sort by and the flag to show whether sorting is ascending or descending.

In order to support collators, the INamespace submanager must support those common.namespace.IAdvancedChildrenFindResourceDescriptor\_s that return a common.namespace.ICollator on getCollator().
6.2.8.3 Property-Queries

In order to support property queries, the INamespace submanager has to return an implementation of common.namespace.IQueryBuilder.

The submanager must also observe IAdvancedChildrenFindResourcesDescriptors, which supply a query expression on getQueryExpression(). The submanager can choose to accept only its own IQueryExpression implementations, built from its specific IQueryBuilder implementation.

An example of how a client uses the query builder to construct a query is given in 5.2.2. The following example sketches what a query builder’s relevant methods might look like:

```java
private IQueryExpression buildExpression(IPropertyName name, String sqlOperand, String value) {
    if (name == null) {
        return null; // name must be specified
    }
    if (PropertyName.CONTENTTYPE.equals(name)) {
        return new MyQueryExpression("contenttype " + sqlOperand + " ?", value);
    }
    if (PropertyName.CREATEDBY.equals(name)) {
        return new MyQueryExpression("author " + sqlOperand + " ?", value);
    }
    return null; // query builder doesn’t handle the given property
}

public IQueryExpression like(IPropertyName name, String value) {
    return buildExpression(name, "like", value);
}

public IQueryExpression eq(IPropertyName name, String value) {
    return buildExpression(name, "=", value);
}
```
The example above uses the following `MyQueryExpression`, which sketches how a query expression might be implemented:

```java
class MyQueryExpression {
    private String sql;
    private List values;

    MyQueryExpression (MyQueryExpression part1,
            MyQueryExpression part2,
            String sql) {
        this.sql = " (" + part1.sql + ") " + sql + " (" + part2.sql + ")";
        this.values = part1.values.clone();
        this.values.addAll(part2.values);
    }

    MyQueryExpression(String sql;
            String value) {
        this.sql = sql;
        this.values = new ArrayList();
        if( value == null ) {
            this.values.add("");
        } else {
            this.values.add(value);
        }
    }

    PreparedStatement getSqlStatement(Connection connection) {
        String sql = "SELECT rid FROM myTable WHERE " + this.sql;
        PreparedStatement statement = connection.prepareStatement(sql);
        for( int i = 0; i < this.values.size(); i++ ) {
            statement.setString(i, (String)this.values.get(i));
        }
        return statement;
    }

    IQueryExpression and(IQueryExpression expression) {
        if( expression instanceof MyQueryExpression ) {
            return new MyQueryExpression(this, (MyQueryExpression)expression, " and ");
        }
        throw new OperationNotSupportedException();
    }

    IQueryExpression or(IQueryExpression expression) {
        if( expression instanceof MyQueryExpression ) {
            return new MyQueryExpression(this, (MyQueryExpression)expression, " or ");
        }
        throw new OperationNotSupportedException();
    }
}
```

This query expression sample collects the `WHERE` clauses and the relevant parameter values. The namespace submanager uses `getSqlStatement()` to get the appropriate `java.sql.Statement` for a fictitious database table `myTable`, in order to retrieve the RID’s of the relevant resources.

This table might have been created as follows:

```sql
create table myTable (  
    rid varchar(512) not null,
    contenttype varchar(128) not null,
    author varchar(128) not null,
    ...
)
```

To support property queries on content metadata, the repository manager should be able to handle the appropriate property names in its query expressions (see 6.2.7).
6.2.8.4 Unique ID

Although unique IDs are provided by a global service, a repository’s backend system can support unique IDs on its own.

Note: In the current API, unique IDs are only provided by a global service. See 5.4.2.6 for details. As a result, repository managers do not currently have to support this.

In order to expose a backend’s unique IDs, a submanager that implements the mi.idmapper.IIdMapperManager interface has to be implemented. The submanager’s interface specifies the following methods:

```java
String getUniqueId(IResourceHandle handle)
        throws ResourceException;
```

Retrieves the mapped unique ID for a resource handle.

```java
IResourceHandle lookup(String uniqueId)
        throws ResourceException;
```

Retrieves the resource handle for a mapped unique ID.

An ID mapping submanager, which also implements the IMutableIdMapperManager interface, has to offer the following methods:

```java
void assignToResource(IResourceHandle handle,
                      String uniqueId)
        throws ResourceException;
```

This method assigns a mapping between the given resource handle and the specified unique ID to an existing resource.

```java
void assignToRid(IRid rid,
                  String uniqueId)
        throws ResourceException;
```

Assigns a mapping between the given RID and the specified unique ID. The RID does not have to refer to an existing resource.

These methods should be implemented if a backend system is capable of assigning (foreign) unique IDs to a resource or an RID.
6.2.8.5 Security

A specific security manager implementation is not necessary if all security constraints are defined in their entirety in the backend system. The repository manager can then rely on the backend’s security checks and only forward access violations as `common.security.AccessDeniedExceptions` to the RF.

However, an implementation might be required if:

- The backend does not support security checks on its own
- or
- The repository manager wants to expose the backend’s security constraints in the RF’s unified fashion in order to support:
  - Locking
  - The setting of the backend’s ACLs from within the portal with the help of the KM’s UI.

These two facets are described in the following sections:

- The first section lists which operations have to be subjected to a security check by the repository manager and which RF permissions are used (for example, `getContent()` must be subjected to a security check for `Permission.READ_CONTENT`). This is needed to enable security for a backend system that doesn’t support it on its own and has the RF’s default ACL security manager attached.

- The second section describes how the security check itself can be implemented by the repository manager’s own `ISecurityManager` implementation, and what has to be done to make it available for the RF’s default ACL edit dialog.
6.2.8.5.1 Using the ISecurityManager

When performing a security check, the repository’s submanagers have to check whether the user requesting the operation has the appropriate permissions.

The RF’s permissions are specified by the constants in `common.security.Permission`. They correspond to those mentioned in 2.2.4.

The following list specifies which checks have to be performed for which operation:

- `checkFind(resource)` on `findResources`, `countResources()`:
  - `LIST` (only relevant if the resource is a collection)
- `checkCreate(resource)` on `createResource()`:
  - `CREATE` for resource’s parent
- `checkDelete(resource)` on `deleteResource()`:
  - If resource is a collection `DELETE` else `DELETE_NODE` and `WRITE_NODE_PROPERTIES` for resource's parent
- `checkGetContent(resource)` on `getContent()`:
  - `READ_CONTENT` for the resource
- `checkSetContent(resource)` on `setContent()`:
  - `WRITE_CONTENT` for the resource
- `checkGetProperty(resource)` on `getProperty()`:
  - If the resource is a collection `READ_NODE_PROPERTIES` else `READ_PROPERTIES`
- `checkSetProperty(resource)` on `updateProperty()`:
  - If the resource is a collection `WRITE_NODE_PROPERTIES` else `WRITE_PROPERTIES`
- `checkMove(source, target)` on `moveResource()`:
  - If it is a rename (source’s and target’s parent are the same):
    - `checkSetProperty(parent)` to check if a folder allows write
  - else (if it’s a real move):
    - `checkDelete(source)`, to check if the source can be deleted
    - and if the target already exists:
      - `checkDelete(target)` to check if the existing target can be overwritten
    - else (if target will be newly created):
      - `checkCreate(target)` to check if the non-existing target can be created
- `checkCopy(source, destination)` on `copyResource()`:
  - If the source is a collection `READ_NODE_PROPERTIES and LIST` else `READ_PROPERTIES and READ_CONTENT` and if destination does not exist:
    - `checkCreate(destination)`, to check if destination can be created
  - else (destination already exists)
    - `checkSetProperty(destination’s parent)`, to check if folder allows write
    - and if destination is not versioned
      - `checkDelete(destination)` to check if the existing, non-versioned target can be overwritten
    - else (if destination is versioned)
      - `checkSetProperties(destination)`
      - and `checkSetContent(destination)`
The remaining operations use the checks specified above:

- `getLocks()` : `checkGetProperty()`
- `lock()` : `checkSetProperty()`
- `refreshLock()` : `checkSetProperty()`
- `unlock()` : `checkSetProperty()`
- `getCollectionOrderMechanism()` : `checkGetProperty()`
- `setCollectionOrderMechanism()` : `checkSetProperty()`
- `reorderCollection()` : `checkSetProperty()`
- `getLinkDescriptor()` : `checkGetProperty()`
- `setLinkDescriptor()` : `checkSetProperty()`
- `setVersionControlEnabled()` : `checkSetProperty()`
- `checkIn()` : `checkSetProperty()` and `checksetContent()`
- `updateFromVersion()` : `checkSetProperty()` and `checksetContent()`

Methods for the RF’s aspects other than those listed above (especially `lookup()` ) should not check for permissions.

A sample implementation of such a `check...()` method might look like the following:

```java
private MyRepositoryManager manager = (MyManager) this.repositoryManager;
private ISecurityManager securityManager = this.manager.getSecurityManager();

public void checkGetContent(IResourceHandle handle) throws AccessDeniedException, ResourceException {
    IUser user = getUserFromContext(handle.getRid());
    if( !this.securityManager.isAllowed(handle, user, Permission.GET_CONTENT) ) {
        throw new AccessDeniedException(handle.getRid(),
                                         Permission.GET_CONTENT.getName(),
                                         user.getId());
    }
}
```

The example above uses the `getUserFromContext()` method as shown in 6.2.4.2.
### 6.2.8.5.2 Implementing a security submanager

Because the KM commands that are used by the KM UI to perform locking, require a security submanager, a repository manager might implement the `mi.security.ISecurityManager` submanager. It offers:

```java
boolean isAllowed(IResourceHandle handle,
                  IPrincipal principal,
                  IAclPermission permission)
  throws ResourceException;
```

Checks whether the specified permission is granted to the given principal for the resource to which the handle refers.

To do this, the submanager has to map the backend’s permissions to those of the RF (see above) and forward this check to the backend’s security checking mechanism.

```java
List getSupportedPermissions(IResourceHandle handle)
  throws ResourceException;
```

Returns a `List` of all the `IPermission`s for the given resource (see 2.2.4. and above for an overview of the RF permissions and the resource’s object types used for permissions).

The RF offers `mi.security.Permission` as an implementation of `IPermission`.

### 6.2.8.5.3 Implementing a ACL submanager

If a repository manager wants to expose the backend’s ACLs to allow them to be edited by the KM’s ACL editor, note that the RF’s permission are mapped again to only three permissions displayed in the KM’s ACL editor:

- **Read**: Contains
  ```
  leaf_read_content,
  leaf_read_properties,
  node_list_children,
  node_read_properties
  ```

- **Write**: Contains
  ```
  leaf_write_content,
  leaf_write_properties,
  node_create_child,
  node_write_properties
  ```

- **Delete**: Contains
  ```
  leaf_delete,
  node_delete
  ```

In order to expose a backend’s ACLs, a `mi.security.acl.IAclSecurityManager` has to be implemented, which offers the methods for retrieving, creating, and removing IAclS. The mechanism is very similar to the `IResourceAclManager`, as described in 5.2.4. The only differences are the names of the interfaces and their packages:
The ACL submanager has to implement its own IAcI and IAcIEntry implementations (for example, MyAcI and MyAcIEntry), which are passed through the RF to the client and back again.

An IAcISecurityManager might therefore only accept its own ACL and ACE implementations, except for the assignAcl() method, which has to accept a (foreign) ACL.

An IAcISecurityManager should apply security constraints on ACLs itself: Usually, only owners of an ACL are usually allowed to change it.
6.2.8.6 Locking

The **ILockManager** interface only contains this method:

```java
List getLocks(IResourceHandle handle) throws ResourceException;
```

Returns the list of **ILockInfo**s for the given resource.

In order to support locking as described in 5.2.5, the submanager must also implement the **IMutableLockManager** interface, which offers:

```java
ILockInfo lock(IResourceHandle handle,
    ILockDescriptor lockDescriptor) throws ResourceException;
```

Locks the resource specified by the handle with a lock, as specified by the **ILockDescriptor**, for the current user.

```java
ILockInfo refreshInfo(IResourceHandle handle,
    String lockToken) throws ResourceException;
```

Refreshes a lock and resets the remaining time for a lock's timeout if such a timeout is set for the current user.

```java
void unlock(IResourceHandle handle,
    String lockToken) throws ResourceException;
```

Unlocks the specified lock for the current user.

The RF provides a default implementation for the **ILockInfo** interface in **mi.lock**, whereas the default implementation for **ILockDescriptor** is located in **common.lock**.

Note: Due to the current implementation of the KM's locking command, which is used within the KM UI, locking requires a security submanager attached to the repository manager.
6.2.8.7 Events

Usually, a repository manager does not need to send events because the RF handles the appropriate events being generated. However, the RF is only able to handle this if resources are accessed through the RF.

If resources are manipulated in the backend without the RF being involved, the RF is not able to generate the corresponding events.

To avoid this, a repository manager can send events on its own to indicate changes made in the backend, using the mechanism as described in 5.2.6: The mi.AbstractManager implements an IResourceEventSender that is already registered at its own event broker. The event broker is returned by the AbstractManager’s method getEventBroker().

To send an event, the repository manager would implement the following:

```java
protected void sendEvent(IResourceEvent event) {
    this.getEventBroker().send(event, this);
}
```

In addition, the event sender’s methods getEvents() and getEvents(IResource) have to be overloaded, to return the appropriate list of events supported by the repository manager.

Repository manager or submanagers should not implement an IResourceEventReceiver, because using a resource from a resource event might cause other events, which might in the end trigger an avalanche of events.
6.2.9 Expert Aspects

6.2.9.1 Type Handling

The ISemanticObjectFactoryRegistry (see 5.3.1) offers a generic way for registering ISemanticObjectFactorys, which are able to “cast” an IResource into a semantic object. Since a repository usually already “knows”, which semantic objects it provides, an easier and faster way of exposing these “casting” abilities would be nice – that is what the ITypeManager is for:

The mi.type.ITypeManager offers only two of the ISemanticObjectFactory's methods:

```java
public boolean isA(IResourceHandle handle, Class objectClass) throws ResourceException;
Checks whether the object identified by the given handle could be “casted” to an object of the requested class.

public Object as(IResourceHandle handle, Class objectClass) throws ResourceException;
Returns the object identified by the given handle as an instance of the requested class.
```

If a client calls an IResource's isA() or as() methods, the RF forwards this call to the resource’s repository ITypeManager first. If the type handling submanager is not able to “cast” the object (or if doesn’t exist), the call is then forwarded to the ISemanticObjectFactoryRegistry.

The mi.type.IMutableTypeManager is not yet used.
6.2.9.2 Versioning

As described in 2.3.2, the versioning in the RF comes in different flavors: basic versioning and advanced versioning, which in turn offers options for workspaces, branches, labels and versioned collections.

The interfaces for the versioning submanager are divided accordingly and will briefly be described here. For further details, see the RF's JavaDoc.

Note: The KM’s versioning command as used by the KM’s UI issues a lock request before it performs a checkout on a resource. If the KM’s UI is to be used to access the repositories versioning features, therefore the IMutableLockManager must also be implemented (see 6.2.8.6).

6.2.9.2.1 Basic Versioning

The mi.version.IBasicVersioningManager interface specifies the methods required for a read-only basic versioning submanager. It provides methods to retrieve the (linear) version history of a versioned resource as well as the checked-in and checked-out version of a VCR:

```java
boolean isVersionControlEnabled(IResourceHandle resourceHandle)
    throws ResourceException

Returns true if the given resource handle refers to a VCR.

boolean isCheckedOut(IResourceHandle vcrHandle)
    throws ResourceException

Returns true if the given resource handle refers to a VCR in check-out state “checked-out”.

IResourceHandle getCheckedOutVersion(IResourceHandle vcrHandle)
    throws ResourceException

Returns the handle for the resource from which the given checked-out VCR has been checked out.

boolean isCheckedIn(IResourceHandle vcrHandle)
    throws ResourceException

Returns true if the given resource handle refers to a VCR in check-out state “checked-in”.

IResourceHandle isCheckedInVersion(IResourceHandle vcrHandle)
    throws ResourceException

Returns the handle for the resource the given checked-in VCR is based on.

List getVersionHistory(IResourceHandle resourceHandle)
    throws ResourceException

Returns the version history of the given resource if the resource handle refers to a VCR and if the version history is a linear version history. The list of versions in the version history is in ascending order, where the first element is the oldest version and the last element the newest.

Set getPredecessorVersions(IResourceHandle resourceHandle)
    throws ResourceException

Retrieves the predecessor for the version given by the resource handle. With basic versioning, the set will contain only the single predecessor in the linear version history for this version.

© 2003 SAP AG
Set `getSuccessorVersions(IResourceHandle resourceHandle)`
throws ResourceException

Retrieves the successor for the version given by the resource handle. With basic versioning, the `Set` will contain only the single successor in the linear version history for this version.

The `mi.version.IMutableBasicVersioningManager` specifies the according methods for in-place check-out and check-in:

```java
void setVersionControlEnabled(IResourceHandle resourceHandle,
                                boolean enabled)
throws ResourceException
```

Enables or disables version control on the resource referred to by the resource handle, according to the `enabled` parameter.

```java
ICheckOutInfo checkOutInPlace(IResourceHandle vcrHandle,
                                boolean forkOk)
throws ResourceException
```

Performs an in-place check-out for the given VCR that changes the VCR's check-out state to "checked-out". It optionally returns an object that implements the `ICheckOutInfo`, if the server supports the `expectedCheckInRid` feature (see `checkIn()` below).

The `forkOk` parameter is only used with advanced versioning and should be ignored.

```java
ICheckInInfo checkIn(IResourceHandle resourceHandle,
                      IRid expectedCheckInRid,
                      boolean forkOk,
                      boolean keepCheckedOut)
throws ResourceException
```

Performs a check-in for a checked-out VCR, creating a new version from the current version controlled state of the VCR and returns an instance of `ICheckInInfo`. This `ICheckInInfo` object contains the RID and the revision ID of the newly created version. If the `keepCheckOut` flag was set it also contains the `ICheckOutInfo` from the implicit check-out operation (see below).

If the `expectedCheckInRid` is set (is not `null`), the method must fail if the server cannot assign the given `expectedCheckInRid` to the new version, created from the VCR.

As with `checkOutInPlace` the `forkOk` parameter should be ignored with basic versioning.

The `keepCheckOut` parameter specifies if the given VCR should be checked-out again, immediately after the check-in succeeded.

```java
Set undoCheckOut(IResourceHandle vcrHandle)
throws ResourceException
```

Rolls back a previous checkout operation. The specified VCR’s version controlled state (its content and dead properties) will be updated from its checked-out version first (the version which has the same version controlled state as the VCR had when the check-out was requested). Then, the VCR will be set to check-out state “checked-in”.

```java
Set updateFromVersion(IResourceHandle vcrHandle,
                      IResourceHandle versionHandle)
throws ResourceException
```
Updates the given VCR's checked-in version's version controlled state from the version controlled state of the given version (the VCR's content and properties will be updated from the version specified). With basic versioning it returns the updated VCR in the Set.

6.2.9.2.2 Advanced Versioning: Working Resources and Branches

The mi.version.IMutableAdvancedVersioningManager contains the method for creating a working resource:

ICheckOutInfo checkOutWorkingResource(IResourceHandle resourceHandle,  
boolean applyToVersion,  
boolean forkOk)  
throws ResourceException

Creates a working resource for the given version.

If the resource handle refers to a VCR and applyToVersion is true, the version referred by the VCR's checked-in version (see getCheckedInVersion() and updateFromVersion() above) is used. If it refers to a VCR but applyToVersion is false, the VCR will be automatically updated from the newly created version on check-in (in addition to the behaviour when applyToVersion is true).

If the forkOk parameter is true subsequent forks in the version history are allowed on check-in (depending on getCheckOutForkBehaviour(), see below).

Using forkOk true will enable forks or branches in the version history (see below).

Within the mi.version.IMutableBasicVersioningManager, the following methods also respect the forkOk parameter:

ICheckOutInfo checkOutInPlace(IResourceHandle vcrHandle,  
boolean forkOk)  
throws ResourceException

As with checkOutWorkingResource, if the forkOk parameter is true, subsequent forks in the version history are allowed on check-in (depending on getCheckOutForkBehaviour(), see below).

ICheckInInfo checkIn(IResourceHandle resourceHandle,  
IRid expectedCheckInRid,  
boolean forkOk,  
boolean keepCheckedOut)  
throws ResourceException

If the forkOk parameter is true, a forks in the version history is allowed (depending on getCheckInForkBehaviour())

The methods for checking the server's fork-behaviour on check-out and check-in are specified in the mi.version.IAdvancedVersioningManager interface:

Set getCheckOutForkBehaviour(IResourceHandle versionHandle)  
throws ResourceException

Returns as Set (for compatibility with WebDAV, see [RFC3253]), which contains an IName object describing the fork-behaviour on check-out for the given version.
Set `getCheckOutForkBehaviour(IResourceHandle versionHandle)` throws ResourceException

Returns as Set (for compatibility with WebDAV – see [RFC3253]) that contains an IName object, which describes the fork-behaviour on check-in for the given version.

The IName is either one of the constants defined in the current API’s `com.sapportals.wcm.repository.IVersionResource`:

- **FORKBEHAVIOUR_FORBIDDEN** (= {DAV:}forbidden) if forks are explicitly not allowed (the forkOk flag is ignored and treated as false)
- **FORKBEHAVIOUR_DISCOURAGED** (= {DAV:}discouraged) if forks are not explicitly allowed; only then the forkOk flag will define the fork behaviour
- **FORKBEHAVIOUR_ALLOWED** (= {DAV:}allowed), if forks are explicitly allowed (the forkOk flag is ignored and treated as true)

As advanced versioning allows forks, the semantics for the version history became more complex. The relevant methods in `mi.version.IBasicVersioningManager` might then return a Set, containing several handles instead of just one:

Set `getPredecessorVersions(IResourceHandle resourceHandle)` throws ResourceException

With advanced versioning, this method retrieves the predecessors for the version given by the resource handle, even if it’s not a linear version history. The Set will contain all predecessors in the version history for this version.

Set `getSuccessorVersions(IResourceHandle resourceHandle)` throws ResourceException

With advanced versioning, this method retrieves the successors for the version given by the resource handle, even if it’s not a linear version history. The Set will contain all successors in the version history for this version.

In addition, the `mi.version.IAdvancedVersioningManager` offers the following methods:

IResourceHandle `getVersionHistoryResource(IResourceHandle versionHandle)` throws ResourceException

Returns the version history of a version as a resource handle, instead of returning the version history as a list of versions, as the `IBasicVersioningManger`’s `getVersionHistory()` does.

To traverse a version history resource then, `getPredecessorVersion()` and `getSuccessorVersion()` are used. To retrieve the starting node, the following method is provided:

IResourceHandle `getRootVersion(IResourceHandle versionHistoryHandle)` throws ResourceException

It returns the root version of the version history specified by the given version history resource handle. The version returned is the first version in the version history (without any predecessors).
6.2.9.2.3 Advanced Versioning: Labels

Another advanced versioning feature is label support. The interface `mi.version.IAdvancedVersioningManager` specifies the following methods:

```
Set getLabelSet(IResourceHandle versionHandle) throws ResourceException
```

Returns the `Set` of labels for the given version.

```
IResourceHandle getVersionResourceByLabel(IResourceHandle versionHistoryHandle, String label) throws ResourceException
```

Retrieves the version for the given label from the specified version history resource.

The corresponding methods for assigning and removing labels are specified in the `mi.version.IMutableAdvancedVersionManager` interface:

```
void addLabel(IResourceHandle versionHandle, String label) throws ResourceException
```

Tags the given version with the given label. If the label is already used as tag for another version in the same version history, the method fails.

```
void setLabel(IResourceHandle versionHandle, String label) throws ResourceException
```

Tags the given version with the given label. This method does not fail, if the label is already used to tag another version in the same version history – it removes the label from that other version then.

```
void removeLabel(IResourceHandle versionHandle, String label) throws ResourceException
```

Removes the given label from the specified version, if the label exists as tag for the given version.

6.2.9.2.4 Advanced Versioning: Workspaces

The `mi.version.IWorkspaceManager` interface contains the methods for retrieving workspaces and checked-out versions in a workspace:

```
Set getWorkspaceCollectionSet(IResourceHandle resourceHandle) throws ResourceException
```

Returns a `Set` of collections, where workspaces can be created in (see `createWorkspace()` below) for the given resource.

```
IResourceHandle getWorkspaceResource(IResourceHandle resourceHandle) throws ResourceException
```

Returns the handle of the workspace, the specified resource belongs to if it’s a workspace controlled resource (`null`, if not).
Set `getCheckedOutResources(IResourceHandle workspaceHandle)`

`throws` `ResourceException`

Returns the `Set` of all checked-out VCRs belonging to the workspace as specified by the given handle.

The `mi.version.IMutableWorkspaceManager` then contains the methods for creating workspaces and version controlled resources inside a workspace:

```
IResourceHandle `createWorkspace(IResourceHandle collectionHandle, String name)`
`throws` `ResourceException`
```

Creates a workspace in a collection (returned from `getWorkspaceCollectionSet()`) with the given name.

```
IResourceHandle `createVersionControlledResource(IResourceHandle collectionHandle, IResourceHandle baseVersionHandle, String name)`
`throws` `ResourceException`
```

Creates a new VCR, based on the specified base version in the collection referenced by `collectionHandle`, for the given name. The `collectionHandle` has to refer to a workspace controlled collection (a collection in a workspace). If the workspace contains already another VCR for the same version history, the method fails.

### 6.2.9.2.5 Auto-Versioning and Automatic Version Control

To check auto-versioning and automatic version control for a collection, the `mi.version.IBasicVersioningManager` specifies the following two methods:

```
boolean `isAutoChildVersionControlEnabled(IResourceHandle collectionHandle)`
`throws` `ResourceException`
```

Returns `true` if auto-versioning is enabled for the children of the specified collection.

```
IName `getAutoVersioningMode(IResourceHandle resourceHandle)`
`throws` `ResourceException`
```

Returns the `IName` for the server's auto versioning mode of the given resource or `null`, if no auto versioning mode is defined.

The following modes are possible (constants defined in the current API's `com.sapportals.wcm.repository.IVersionControlledResource`):

- `AUTOVERSIONING_MODE_CHECKOUT_CHECKIN` (`{DAV:}checkout-checkin`)
- `AUTOVERSIONING_MODE_CHECKOUT_UNLOCKED_CHECKIN` (`{DAV:}checkout-unlocked-checkin`)
- `AUTOVERSIONING_MODE_CHECKOUT` (`{DAV:}checkout`)
- `AUTOVERSIONING_MODE_LOCKED_CHECKOUT` (`{DAV:}locked-checkout`)

These reflect the modes as specified in [RFC3253], Section 3.2.2.
Auto-versioning can be switched using the method specified in
`mi.version.IMutableBasicVersioningManager`:

```java
Set setAutoChildVersionControlEnabled(IResourceHandle collectionHandle, boolean modifyPlainChildren, boolean returnModified, boolean enabled) throws ResourceException
```

Enables or disables auto-versioning for the specified collection, according to the `enabled` flag.

6.2.9.2.6 Advanced Versioning: Versioned Collections

Since version controlled collections (VCCs) are treated as other version controlled resources (VCRs),
the only “special” methods are specified by `mi.version.VersionedCollectionManager`:

```java
Set getEclipsedSet(IResourceHandle vcrHandle) throws ResourceException
```

Returns the `Set` of names for those children of the specified version controlled collection, which are
eclipsed by other children of the VCC.
6.3 Repository Filter and Filter Manager

Repository filters modify the resources as they are passed through the RF. Several filters can be cascaded in a chain:

When a client requests a read operation the RF retrieves the specified resource data from the repository manager. It then retrieves the last filter to apply from the read filter managers by requesting their filters. Then the RF requests the last filter to apply itself to the data. In order to retrieve the data, the filter in turn requests the filtered data from its predecessor, resulting in all filters in the chain being called by their successors. Finally the filtered data is returned to the client.

Same for write operations, where the RF first applies the write filters and then sends the filtered data to the resource's repository manager.
The **repository filter manager** decides when to apply a filter. When the RF requests the filters to be applied from the repository filter manager it uses `getFilterForRead()` when a read request’s data has to be filtered, and `getFilterForWrite()` when a write request’s data is to be filtered. The filter’s predecessor in the filter chain is passed as parameter to the `getFilterFor...()` method.

The filter manager should then decide (based on the resource returned by the predecessor’s filter `getResource()` method) whether to apply its filter or not. If the filter is to be applied the filter manager should return its own filter (see below). If the filter should not be applied, it must return the predecessor as passed to the `getFilterFor...()` method. The filter manager must not return `null`. If the filter manager returned its own filter, that filter has to contain a reference to its predecessor.
The following types of filters are depending on the type of a resource's data to be filtered:

- **Namespace filter** hide existing resources from the hierarchy
- **Property filter** hide or modify existing properties of resources, or add virtual properties to resources
- **Content filter** modify a resource's content

Filters are located in the `com.sapportals.wcm.repository.filter` package in the current API.

### 6.3.1 Namespace Filter

Namespace filters affect the visibility of resources and are applied to collections.

A namespace filter implements `INamespaceFilter`.

The filter's `getCollection()` method returns the collection the filter is to be applied to. This is used by the filter manager for determining whether to apply the filter or not.

The filter's `filter()` method returns the filtered list of child resources for the collection.

### 6.3.2 Property Filter

Property filters affect the properties of resources.

A property filter implementation should be derived from `AbstractPropertyFilter`, which handles all the `IPropertyFilter`'s methods except `filter()`.

The filter's `getResource()` returns the resource the filter is to be applied to – it might return `null`, when the filter is called, while the resource is being created. In this case, the method `getRepositoryManager()` returns the repository manager where the resource will be created in.

The `getFilterMode()` method returns the current filter mode, that is one of the following values:

- `PropertyFilterMode.SINGLE_PROPERTY` only one property is requested by the client.
- `PropertyFilterMode.PROPERTY_LIST` a list of properties for the given names is requested by the client.
- `PropertyFilterMode.ALL_PROPERTIES` all properties of the resource are requested by the client.

If `getFilterMode()` is `SINGLE_PROPERTY` or `PROPERTY_LIST` the method `getPropertyNameList()` returns a list with the name(s) of the properties requested.

The `getFilterContext()` method returns some `java.util.Properties`, which can be used to pass some context information from filter to filter.
6.3.3 Content Filter

Content filter affect the content of resources.

A content filter implements IContentFilter and IContent.

The filter's getResource() method returns the resource, the filter is to be applied to – it might return null, when the filter is called, while the resource is being created. In this case, the method getRepositoryManager() returns the repository manager, where the resource will be created in.

The methods for IContent behave as described in 5.1.3.
### 6.4 Repository Service

Implementing a repository service might be done for one of the following two reasons:

- **To implement a unified aspect, already specified by the RF as repository service (e.g. application properties or the KM’s subscription service), but for a specific repository.**
  
  This is usually done to gain better performance for that repository service, since a repository specific implementation can be built, using integration with the backend system more efficiently than the RF’s or KM’s generic implementation does.

- **To add new unified aspects to the RF.**

  Specifying a new aspect through an repository service will introduce that aspect for the RF resources, but the implementation might be specific for a repository. When specifying a new aspect (e.g. `IMyAspect`) one should keep in mind, that other repositories might have to implement this aspect as well (or to provide a generic repository service, suitable for any repository, as well).

The following UML diagram shows the relevant interfaces and classes:

![UML Diagram]

A repository service implementation (e.g. `MyRepositoryService`) must derive from `AbstractResourceService` (package `com.sapportals.wcm.repository.manager`), which implements `IRepositoryService`. This base class already handles all interfaces required for the interaction with the component runtime (see 6.1.2).

The `MyRepositoryService` must then overwrite the `startUpImpl()` and `shutDownImpl()` methods, to perform initialisation on start-up and de-initialisation on shutdown.

A repository service is assigned to one (or more, if it’s a singleton, see 6.1.2.2) repositories, therefore the list of repository managers, the repository service is initially assigned to, is passed as parameter to `startUpImpl()`.
The methods `addRepositoryAssignment()` and `removeRepositoryAssignment()` are only required to be overwritten by hot-reload enabled repository services: These methods are called by the RF when a new repository was added to the RF or if a repository has been removed, which this repository service assigned to.

Last but not least, the `MyRepositoryService` should implement the interface(s), which specify the appropriate aspect (e.g. `IAppProperty` when implementing a specific version of the RF’s predefined application properties or `IMyAspect` when implementing a new aspect).
6.5 Global Service

Global services add additional “unified” functionality to the RF’s resource objects and use the unified aspects.

While repository services might be related to specific repositories only, global services provide functionalities, which are not closely related to the backend system (as aspects do) but are more general. For example, the KM’s indexing service (which handles indexing of resources in order to enable fast searching) works on all resources, regardless which repository – by using the unified namespace, content and property aspects of the resource.

The following UML diagram shows the relevant interfaces and classes:

A global service implementation (e.g. MyService) must be derived from AbstractService (package com.sapportals.wcm.service), which in turn implements IService. This base class AbstractService already handles the interaction with the component runtime (see 6.1.2).

The MyService should then overwrite the startUpImpl() and shutDownImpl() methods, to perform initialisation on start-up (e.g. connecting to a database) and de-initialisation on shutdown (e.g. committing all open transactions and closing the connection).

And, of course, it has to implement the interface, which specifies the functionalities provided by that global service (e.g. IMyFunctionality).
7 References


[PDK-Start] “PDK - Getting Started”; iView Studio

[PDK-Services] “PDK - Services”; iViewStudio


[RFC2291] Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web; IETF RFC 2291

[RFC2396] Uniform Resource Identifiers (URI): Generic Syntax; IETF RFC 2396

[RFC2518] HTTP Extensions for Distributed Authoring – WEBDAV; IETF RFC 2518

[RFC2616] Hypertext Transfer Protocol – HTTP/1.1; IETF RFC 2616

[RFC3253] Versioning Extensions to WebDAV; IETF RFC 3253

[IETF-UUID] UUID URN Namespace Internet Draft; IETF Network Working Group, draft-mealling-uuid-urn-00.txt

[CF] Configuration Management in Portal Application; (included in the PDK)

[CRT] RF Component Runtime API;

[Logging] SAP Logging and Tracing API;

[Exceptions] SAP Exception Framework API;

[Security] SAP User Management API;

[ANT] Apache Ant; http://ant.apache.org/
8 Index

A
access context, 11, 99
ACE, 20
ACL, 20
administrator, 7
application developer, 7
attributes, 8
automatic version control, 32
auto-versioning, 32

B
backend system, 6, 95
baseline, 33
branch, 27
broken link, 13

C
check-in, 26, 73
check-out, 26, 73
clustered installation, 88
collators, 17
collection, 8, 11, 47, 100
collection order mechanism, 17
component runtime (CRT), 83
configuration framework, 82
content, 8, 14, 52
  encoding, 14
  entity tag, 14
  language, 14
  length, 14
  type, 14
  current version, 27

dead properties, 15
display name, 16, 55

E
eclipsed resource, 33
end user, 7
entity tag, e-tag, 14
event, 22
  broker, 22
  correlation id, 66
  receiver, 22
expected RID, 28
expires, 14
extension developer, 7
external link, 13

F
flexible links, 13
fork, 27

G
global service, 82, 138

H
hierarchical name, 8
hierarchical namespace, 10

I
in-place check-out, 28
internal link, 13

L
label, 27
last modified, 14
leaf resource, 9, 62, 96
link, 13, 49
  target, 13
  type, 13
live properties, 15
lock, 21, 63
  depth, 21
  kind, 21
  scope, 21
  timeout, 22
  token, 63
  logs, 87
  lookup, 98

M
mass calls, 92
merge, 27
meta-data, 14
Repository Framework – Concepts

N
name, 10
namespace, 10
node resource, 9, 62, 96

O
object factory, 25, 70
ordered collection, 17, 57
  position, 17, 57

P
parent-child relation, 10
path, 10
permission, 20
plain resource, 9
property, 8, 14, 53, 107
  description, 16
  local name, 14
  name, 14
  namespace, 14
property attributes, 15
  hidden, 15
  multivalued, 15
  readonly, 15
  required, 15
property query expression, 18, 58
property search, 18
property type, 15
  boolean, 15
  dateonly, 15
  datetime, 15
  integer, 15
  long, 15
  string, 15
  timeonly, 15
  XML, 15

Q
query builder, 18

R
repository, 6, 9
  filter, 82
  manager (RM), 6, 82, 89
  prefix, 10
  service, 82, 136
Repository Framework (RF), 6
resource, 8, 47
resource context, 11, 48
resource factory, 48
resource handle, 95
resource type, 55
revision, 26
RF component, 82
RF extension, 35
RF object, 8, 47, 95
RID, 10, 47, 98
root resource, 10
root version, 27

S
semantic object, 8, 25
store, 6
structured data, 14
system properties, 110

T
top-level resource, 10
traces, 87
type cast, 25

U
unique ID, 19
unstructured data, 14

V
version, 26
version control, 26, 73
version controlled collection (VCC), 32
version controlled resource (VCR), 26, 73
version controlled state, 26
version history, 27
versionable resource, 26

W
workspace, 31
workspace controlled resource (WCR), 31

X
XML local name, 14
XML namespace, 14