BEA WebLogic Portal®

Content Management SPI Development Guide

Version 10.2
Revised: April 2008

Contents

Introduction to Content Management SPI

This chapter introduces the BEA WebLogic Portal Content Management (CM) Service Provider Interface (SPI) by way of an FAQ. It contains the following sections:

- Content Management SPI FAQ
- Architecture Overview of an SPI Implementation

Content Management SPI FAQ

The following questions and answers provide an overview of the Content Management SPI.

What is the CM SPI?

SPI is an open extension point in the WebLogic Portal Content Management Virtual Content Repository (VCR). It supports integrating data from an external system into the WebLogic Portal VCR.

SPI implementations can be "plugged into" the VCR to enable data access and modification in external stores of given type. This gives you the ability to use WebLogic Portal Interaction Management, display templates, CM APIs, content search, WebLogic Portal Administration Tools, and other WebLogic Portal VCR features with the external data. For example, you could create SPI implementations for accessing SharePoint, file system, RSS, or database data.

The SPI is a group of CM interfaces supplied with WebLogic Portal. For example, there is a `Repository` interface, a `Ticket` interface, and various optional operations interfaces, such as

`NodeOps`, `ObjectClassOps`, and `SearchOps`. Some of these interfaces are required, others are optional.

An SPI implementation is a group of classes that implements the SPI to support persisting and/or storing data externally. The SPI is not tied to any particular architecture, protocol, or data source. This means that it is applicable to a wide range of implementations, including database-centric, file-system based, and network protocol-based systems. Some SPI implementations may delegate to a remote server, while other implementations run in same JVM as WebLogic Server.

For example, you could create an SPI to access external syndicated feeds (RSS) and present the feed data as VCR nodes and properties. The CM tags and display template features of WebLogic Portal could be used to access and present the data.

The SPI is designed to support incremental SPI implementations. For example, you could start with a simple read-only SPI implementation, and then later add search and read-write capabilities.

Why Write an SPI?

The main reason to write an SPI is to expose data via the VCR.

Creating an SPI implementation enables the WebLogic Portal features built on the CM VCR to be used with external system data. For example, content selectors, placeholders, WebDAV, display templates, CM tags, syndicated feeds, and the WebLogic Portal Administration tools may all run against external system data exposed by an SPI implementation. The exact capabilities available from the VCR depend on which features the SPI implements.

The optional interface, optional method approach allows an SPI Implementation to be quickly created and gracefully upgraded over time to support additional capabilities.

What are the Structures and Data Types that can be Exposed?

The CM SPI supports exposing hierarchical data via a structured (typed) mechanism. If data can be represented in a folder structure (including just a single folder level), it can be exposed via the CM SPI.

Each node in the hierarchy has an implicit name (the name of the hierarchy position) and unique ID. A node without a type has only a name and an ID. You can refer to a node either by its unique hierarchical path from the repository root, such as `/foo/bar/bas`, or by direct access with its unique (opaque) ID.

If the node has an associated type (`ObjectClass`), the node can have additional metadata. The allowable metadata depends on the type (`ObjectClass`) definition. For example, suppose there is an `ObjectClass` named City, with a numerical property “population” and a binary property “image”. A node “Denver” of type City can then have a “population” value as well as an “image” value, in addition to its name and ID.

What is the Difference Between an SPI Implementation and a Repository?

An SPI implementation is a way to interact with an external system of a given type. For example, you could create one SPI implementation to access Documentum data, another SPI to access file system data, and a third to access data stored in a database.

The VCR may have multiple named instances (repositories) of the same SPI implementation running against different external data sources. For example, one “FileSystem” Repository may represent data rooted in one directory tree and another “FileSystem” Repository may represent data rooted in another directory tree, but both may use the file system SPI implementation classes, and be deployed in the same enterprise application.

A repository has an associated `RepositoryConfig` object that contains a name, an SPI implementation class name, and associated property configuration data.

Who Uses the SPI?

The CM VCR uses an SPI implementation on behalf of clients, such as WebLogic Portal Administration tools.

Client code does not directly use the SPI implementation; all access occurs indirectly via the VCR. WebLogic Portal administrators register SPI implementations with the VCR by creating or updating a repository configuration, which lets the VCR know about an SPI implementation and its associated configuration. In order to service VCR clients, the VCR uses the repository configuration information to connect to the SPI implementation, which then connects to the external system for accessing and modifying data.

How is an SPI Implementation Packaged and Made Available to the VCR?

The SPI implementation classes are placed in one or more JARs. You can then place the JARs in the system or application classpath. A WebLogic Portal administrator creates a repository configuration to register and configure a Repository instance of a particular SPI implementation using (for example) the WebLogic Portal Administration tools. The administrator specifies various configuration settings, including the repository name, the SPI Repository implementation class, a username and password (optional), and other properties.

Where Does the SPI Implementation Run?

The SPI implementation classes are loaded into the WebLogic Server's enterprise application classloader and run inside the WebLogic Portal server. The same SPI implementation may be loaded multiple times into different applications in the same server.

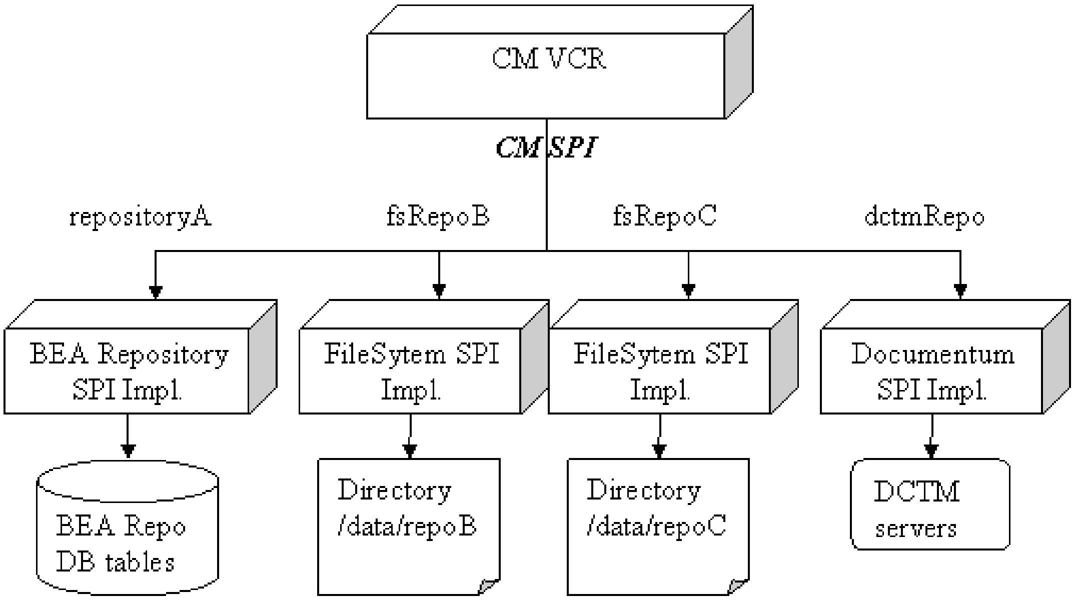
The same SPI implementation may be loaded multiple times, into the same application with different repository names and configurations.

Note: Because they are loaded into the WebLogic Server, they can also be clustered.

Architecture Overview of an SPI Implementation

Figure 1-1shows an architectural diagram of Content Management VCR-SPI implementation.

Figure 1-1 Content Management VCR-SPI Architecture



What components are involved in the SPI?

Several interfaces in the `com.bea.content.spi.flexpi` package must be implemented by the SPI, such as `Repository` and `Ticket`. These interfaces must have implementations for the SPI to be loaded and provide a means for calling the SPI and for authenticating a user.

Additionally, some optional interfaces can be fully or partially implemented to support various types of functionality. For example, to support `Node` operations, you can implement the `com.bea.content.spi.flexspi.ticket.NodeOpsV1` interface. Within this interface, you could implement all the methods except those related to link properties.

Other components involved in SPI include SPI data, such as `Id`, `Node`, `ObjectClass`, `PropertyDefinition`, `QueryCriteria`, and `QueryResult`, that are passed through the SPI interfaces. They are located in either in `com.bea.content` or `com.bea.content.spi.flexspi.common`.

What is the Relationship of the VCR Repository Construct and the WebLogic Portal Application?

The VCR Repository construct is enterprise-application scoped. When a repository is registered in an enterprise application with a name, an SPI Implementation class, and a set of configuration properties, it is available (via the repository name) to all users in the same enterprise application. In other words, all users in the enterprise application can access the same set of configured repositories. The particular capabilities each WebLogic Portal user can have for a given repository in the current enterprise application depend on the user's entitlements. User capability may range from none to full.

It is possible for multiple enterprise applications to duplicate a repository configuration, using the same repository name and other configuration settings, to refer to the same external system data.

What Authentication Models are Available?

Three supported authentication models are provided. The exact behavior of the authentication depends on which model is being use, as described in the following list:

- No authentication – Some SPI implementations may not have a concept of authentication. If this is the case, the repository configuration contains no user or password credentials, and all VCR users in the same enterprise application have access to the repository. WebLogic Portal CM entitlements can define the data each user is permitted to access. From an SPI implementation perspective, the `Repository.connect()` methods should always create and return a `Ticket` implementation object.

- Enterprise application authentication – Some SPI implementations have a concept of authentication, which means the enterprise application is essentially authenticated to the repository. In this case, the repository configuration is configured with a username and a password. All VCR users in the same enterprise application use the same shared SPI credentials. WebLogic Portal CM entitlements can define which data each user can access. From an SPI implementation perspective, the `Repository.connect()` methods should check the application credentials before creating and returning a Ticket implementation object. If authentication fails, the method should throw an `AuthenticationException`.
- Per-user authentication – Some SPI implementations may additionally support the optional feature of per-user authentication on top of either of the previously listed approaches. In this model, individual WebLogic Portal users may "upgrade" their username and password credentials from the global per-application credentials to credentials associated with their username. When these users access the repository, the implementation uses their personal credentials rather the global credentials. For users who do not have personal credentials, the global credentials are used. From an SPI implementation perspective, the `Repository.connect()` methods should check the passed credentials (which may be application or user credentials) before creating and return a `Ticket` implementation object, and throw an `AuthenticationException` if authentication fails.

How Does the VCR Interact With the SPI Implementation?

After you complete configuration, the VCR needs to connect to the SPI implementation before any SPI implementation methods can be invoked.

The first time the VCR contacts the SPI implementation, the VCR usually:

1. Loads and instantiates the SPI class, which implements the `Repository` interface via `Class.forName()` and `Class.newInstance()`.
2. Queries the `Repository` object for its capabilities. For more information, see Chapter 3, "SPI Capabilities and Versions."
3. Calls `Repository.connect(…)` to authenticate credentials to a repository and receive a `Ticket` object.
4. Queries the `Ticket` object for its capabilities.

After the initial setup is complete, future VCR access directly invokes the SPI methods.

For example, if the (optional) `operation` interface is supported and the capabilities also indicate the method is supported, the VCR invokes the appropriate SPI operation method. To illustrate, if the client calls `INodeManger.getNodes( ContentContext, ID )`, the VCR may try to call the SPI method `NodeOpsV1.getNodeChildrenWithQueryCriteria( ID, int, QueryCriteria)`, (if the SPI implements this method).

Note: The VCR may cache data returned by the SPI or the SPI may use its own caches.

SPI Data Model

This chapter describes the SPI data model. It includes the following sections:

About the Content Management Data Model

External system data is represented by two primary types:

- Type data—information about what data a node of a given type can contain.
- Node data— information about the node itself.

To address and access data, both `Nodes` and `ObjectClasses` have a unique ID. The ID contains both the repository name and an opaque repository-specific (`String`) UUID. Generally, from an SPI implementor's perspective, the UUID is the only field in the ID that should be examined and all data should be tied to the repository-specific UUID rather than to the ID. However, you need to construct the ID as the SPI needs to return data. The repository name is assigned by the VCR, as it receives the object from the SPI.

Note: The UUID field must never be null, as this value is reserved for the repository root ID.

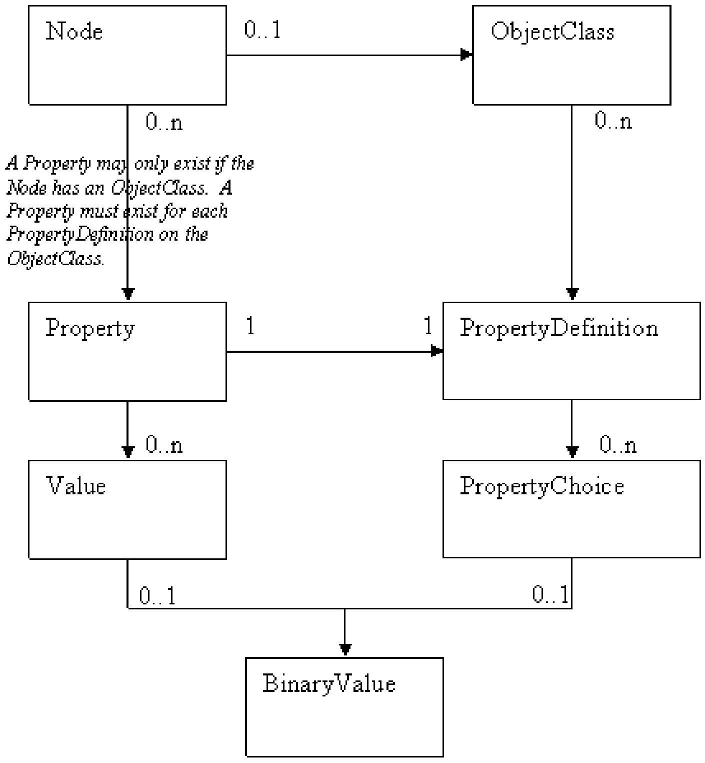
Nodes have a unique implicit hierarchical path from the repository root that ends with the node name. You can retrieve nodes based on this path. For example, a node with name "foo" might be available at repository path `/someNode/anotherNode/foo`.

From an SPI implementor's perspective, all paths start with the repository root node, that is `/foo`. In contrast from a VCR client perspective, all paths start with the repository name, that is `/SomeRepo/foo`. The VCR manages this path modification.

The special path "`/`" is reserved for representing the repository root and the special ID `UUID=null` is reserved for repository root ID. The repository root is an artificial address construct for retrieving the top-most nodes in a repository. You cannot directly fetch the root node, but you can retrieve its children.

Figure 2-1shows an overview of the SPI data model.

Figure 2-1 Content Management Data Model Diagram


Node
0..1
ObjectClass
0..n
0..n
A Property may only exist if the Node has an ObjectClass. A Property must exist for each PropertyDefinition on the ObjectClass.
Property
1
1
PropertyDefinition
0..n
0..n
Value
PropertyChoice
0..1
0..1
BinaryValue

Type Data Representation

An `ObjectClass` defines the schema for a `Node` (that is, what kind of data a node of that type can contain). Besides the schema, the `ObjectClass` contains a name and ID that uniquely identifies the `Node` within the repository, plus an array of property definitions that describe the data (`Properties`) that a `Node` of that type can hold. For example, you could create a type "City" with property definitions for "population" (String) and "image" (binary).

`ObjectClasses` support, but do not require, type-inheritance. A type `City` may inherit from type `Location`, thus receiving property definitions from `Location`. Single inheritance is supported. Additionally, `ObjectClasses` support, but do not require, type nesting, which is a containment structure. This allows you to construct a hierarchy of `Node` data rather than a flat list of properties.

An `ObjectClass` contains a `PropertyDefinition` that defines the schema for a single `Property` that can exist on a `Node` of that `objectClass`. This is actually the schema for a named piece of data. The primary fields in it include:

- Property name
- Type (String, binary, boolean, and so on)
- Single-valued or multi-valued
- Read only
- Mandatory (if it is required to have a value)

A `PropertyDefinition` may define a set of `PropertyChoices` that you can use to provide some predefined values. For example, a `String` property may have a set of property choices, one for each state.

A `PropertyChoice` can be demarcated as default. When a node is created, if the associated property has no value, it is created with the default `PropertyChoice` value.

Node Data Representation

A `Node` represents a set of external system data. A `Node` has a name, and you can address a node using a unique hierarchical path (ending with its name), or a unique ID.

If a node has a type, then you can include a set of `Property` values that hold external system data. The set of `Properties` available depends on the node's type (`ObjectClass`). For example, if `Node` myNode is of type City, and City has `PropertyDefinitions` for size and image, the myNode can have a property called `size` and another called `image`, with values for each of these properties.

Nodes also have implicit system metadata including:

- name

- ObjectClass
- Creation date
- Created by
- Modified date
- Modified by
- Parent node ID (the ID of the node’s parent)

A `Property` represents a named piece of external system data. It exists as part of a `Node` object. Each `Node` can have multiple properties providing its `ObjectClass` defines multiple properties. A `Property` has a name that maps up to a `PropertyDefinition` on the node’s `ObjectClass` with the same name.

Properties may have zero or more `Values`. The schema for the property values depends on the associated `PropertyDefinition`. For example, if `Node` myNode is of type City, and City has property definitions for size (String) and image (binary), then myNode can have a property called “size” with a string value, and another called “image”, with a binary value.

A `Value` represents an unnamed piece of external system data. It exists as part of a `Property` object. A `Property` may have multiple values if its `PropertyDefinition` is multi-valued. The `Value` contains the real value of the data type specified on the property’s `PropertyDefinition`. For example, if the “Person” `ObjectClass` has a “favoriteColors” `PropertyDefinition` (a multi-valued String), a node of type “Person” can have a `Property` named “favoriteColors” that contains zero or more string `Values`. Valid value types are:

- binary (a `binaryvalue` object holds the data.)
- boolean
- calendar (date and time)
- double
- ID (a node address, also called a link)
- Long
- String
- `Property[]` (used for nested `ObjectClass` values)

A `BinaryValue` represents an unnamed piece of external system binary data. It exists as part of a `Value` object. There can be only one `BinaryValue` object per `Value` object. In addition to an `InputStream`, the `BinaryValue` also includes metadata:

- Binary Name
- Mime Type (Content Type)
- Size
- Checksum

`BinaryValues` are generally loaded on-demand for better performance.

CHAPTER 3

SPI Capabilities and Versions

This chapter describes how an SPI implementation exposes the capabilities it supports to the VCR. This chapter includes the following topics:

- About SPI Capabilities
- VCR Detection of the SPI Implementation Capabilities
- SPI Interface Versions

About SPI Capabilities

The SPI implementation exposes the capabilities it supports to the VCR. For example, an SPI implementation may report which methods it supports, and whether it supports Node creation. This allows the VCR and VCR client code to customize behavior based on the capabilities of the underlying SPI implementation. There are three types of capabilities:

- Method capabilities
- Feature capabilities
- Repository defined capabilities

Method capabilities indicate which methods on an interface. These are exposed via:

- Repository – `Repository.getCapabilitySupport()`
- Ticket – `Ticket.getCapabilitySupport()`

Feature capabilities are exposed with `Repository.getCapabilitySupport()`. CM features may span multiple methods. For example, `NodeFeatureCapability.NodeUpdate` affects several methods and indicates the general ability to update a node.

Repository-defined capabilities are optional SPI implementation-specific functions. These capabilities are exposed with `Repository.getRepositoryDefinedCapabilities() and` report various (usually dynamic) abilities of the underlying repository to the VCR client code for a specific SPI implementation. For example, you could use an SPI implementation to report to client code that the SPI implementation is performing case-insensitive sorting. A basic SPI implementation likely would not use this feature.

VCR Detection of the SPI Implementation Capabilities

For method and feature capabilities, the VCR passes the set of capabilities it is aware of to the SPI. The SPI implementation must then report its capability support (supported or not supported) for each capability passed. The SPI implementation must not report any additional capabilities beyond those it was passed. If additional capabilities exist, the implementation should use repository-defined capabilities for those capabilities.

For repository-defined capabilities, the VCR simply asks the repository for its repository defined capabilities and associated support levels.

During the connection process, the VCR calls the methods:

- `Repository.getCapabilitySupport(…)` – Defines how the SPI implementation reports its feature capability support. If the Repository implements any optional repository operation interfaces, such as `RepositoryConfigOpsV1`, it must report a `MethodCapability` for each method in these interfaces.
- `Ticket.getCapabilitySupport(…)` – If the `Ticket` implements any optional ticket operation interfaces, such as `NodeOpsV1`, it must report a `MethodCapability` for each method in these interfaces.
- `Repository.getRepositoryDefinedCapabilities()` – Defines how the SPI implementation can report any repository-defined capabilities.

SPI Interface Versions

The SPI model included in WebLogic Portal allows the VCR and SPI implementation to have different versions of the SPI interfaces. For example, the VCR may use `NodeOpsV3` while the SPI implementation may be written to `NodeOpsV2`. This flexibility allows the SPI to expand over time without breaking any existing SPI implementations and allow additional interfaces to be introduced that expose additional functionality. The SPI version numbers are unrelated to the WebLogic Portal product version; they are incremented anytime an interface change is made.

Note: Once an SPI interface version is released with WebLogic Portal, it will not change because changing it may break any SPI implementations that implement its version.

Versioned interfaces exist for optional repository operation interfaces, such as RepositoryConfigOpsV1 and optional ticket operation interfaces, such as NodeOpsV1. When you create an SPI implementation, you write the implementation for specific versions of the SPI interfaces.

For example, suppose that WebLogic Portal v11 includes NodeOpsV1, ObjectClassOpsV1, and ObjectClassOpsV2 (among others) and you write an SPI implementation against WebLogic Portal v11 to implement the latest versions: NodeOpsV1 and ObjectClassOpsV2. After a period of time, when you use WebLogic Portal v12, which includes NodeOpsV1, NodeOpsV2, ObjectClassOpsV1, ObjectClassOpsV2, and ObjectClassOpsV3, the SPI implementation, built to earlier versions, works properly.

Tip: You should use the latest available versions of the interfaces for your SPI implementations. Over time, SPI interface versions may be deprecated and eventually removed, so it's best to start with the latest available.

You do not need to implement all methods in the interfaces. For example, you could implement only a single method in NodeOpsV1.

Some SPI Implementation methods report the interface version they implement. For example, `Repository.getAllInterfaces()` and `Ticket.getAllInterfaces()` return a `Map<String, ISPIMarker>`, which has a key with the interface version: `SPIRepositoryInterfaces.REPOSITORY_CONFIG_OPS_V1` or `SPITicketInterfaces.NODE_OPS_V1`.

Note: You should not bundle the versioned SPI interface classes in the SPI JARs with your SPI implementations; they are included with WebLogic Portal.

CHAPTER 4

SPI Implementation Creation

This chapter provides information about creating an SPI implementation. It contains the following sections:

- VCR SPI Implementation Interaction
- Primary Classes for a Basic SPI Implementation
- Repository Guidelines when Creating an SPI Implementation
- Basic SPI Implementation

VCR SPI Implementation Interaction

When the VCR needs to access a specific repository from the set of application repository configurations, the VCR loads and creates an instance of the configured SPI implementation class, which implements the VCR SPI Repository interface. The VCR invokes methods on the repository implementation to obtain objects, such as a `Ticket`, which implement other VCR SPI interfaces.

The VCR invokes methods on the Repository and Ticket implementations to query the SPI implementation for its capabilities. (Capabilities are what operations the implementation supports.) The VCR then invokes methods to retrieve the operation interfaces, such as NodeOpsV1, that the SPI exposes. Finally, the VCR invokes methods on the operation interfaces.

Primary Classes for a Basic SPI Implementation

The two primary classes for an SPI implementation are:

- `Repository`
- `Ticket`

The `Repository` class is instantiated directly by the VCR. This class provides anonymous repository access. For example, an SPI implementation can report its version and other basic information. Most importantly, this is the entry point for authenticating to an SPI implementation to obtain a Ticket. Only authenticated users can obtain a Ticket; the Ticket provides access to the important repository operations.

The `Ticket` class provides authorized repository access, and provides access to the authorized operations interfaces such as `NodeOpsV1`. Only authenticated users can obtain a `Ticket`; the `Ticket` provides access to the important repository operations. This class is instantiated by `Repository.connect(…)`.

The `Repository` can optionally expose additional authorized ticket operations interfaces, such as `NodeOpsV1`, `SearchOpsV1`, via its implementation of `getAllInterfaces()` and `getInterface()`.

The general flow at runtime is:

- Connection process:
  - VCR instantiates `Repository` class.
  - VCR configures `Repository` object.
  - VCR queries `Repository` object.
  - VCR invokes `Repository.getAllInterfaces()` to retrieve all repository interfaces.
  - VCR invokes `Repository.connect()` to obtain a `Ticket`.
  - VCR invokes `Ticket.getAllInterfaces()` to retrieve all ticket interfaces.
- Call process:
  - VCR accesses the desired interface, such as `NodeOps`.
  - VCR invokes method on the desired interface, such as `NodeOpsV1.getNodeWithId( ID )`.

Repository Guidelines when Creating an SPI Implementation

Use the following design guidelines when creating an SPI implementation:

- Standalone – The SPI implementation should be standalone. The VCR calls into the SPI implementation. The SPI implementation should not call into the VCR federated interfaces, although it can use the VCR data objects.
- Stateless – Generally, the SPI implementation should be stateless. All changes should be propagated to the external system immediately.

  There are several reasons for this. Primarily, because in a cluster scenario or multiple enterprise applications on the same server, changes to external system data should be persisted immediately. This allows the data to appear "live" regardless from which server and enterprise application the data is accessed.
- Object caching – The SPI implementation should not cache the data objects instances it returns (`Nodes`, `Properties`, `Values`, `ObjectClasses`, `PropertyDefinitions`, and `PropertyChoices`). The VCR and client code own these objects, which they can modify. For example, the VCR may change the node path before returning the node to the client.

  A safe approach is to cache the objects, then return a clone of these objects, which the SPI implementation will not have references to.

  Because the data object types are shared between the SPI implementation, VCR, and client code, not all methods on these data objects are appropriate for each type of caller. (See the WebLogic Portal Javadoc for clarification.) For example, new data objects, such as `Properties` passed when creating a `Node` from the VCR to the SPI implementation do not have UUID on the `PropertyID`. This happens because the UUID is assigned by the SPI implementation and the new object has not yet been persisted by the SPI implementation. The UUID is present only for data objects retrieved or returned by the SPI.
- VCR Repository interface – SPI implementations must implement the VCR Repository interface. Repositories are loaded dynamically by the VCR and require a public default constructor. When the VCR loads the SPI implementation, it calls the `Repository.setName()` and `Repository.setProperties()` methods before it calls the `connect()` method.

  Each Repository instance corresponds to a single repository configuration for a single enterprise application and for use by a single user. At runtime, the VCR can create many instances of Repositories for use by numerous current WebLogic Portal users. For this reason they should be relatively lightweight objects, as hundreds of these objects may exist.

In an enterprise application for a given SPI implementation, there may be multiple active repository configurations. For example, three repository configurations may exist for an application that the VCR is managing, and each repository will have its own configuration data and its own Repository instance that is created at runtime.

- Exceptions – Anytime the SPI implementation is unable to perform an operation, it should throw a `RepositoryException`. Several subclasses of `RepositoryException` are defined, and these should be used when a good match exists. For example, if the SPI implementation does not support an operation, it should throw a `UnsupportedRepositoryOperationException`.
- Ticket re-use – The `Ticket` object is cached and re-used by the VCR. Generally, it should be stateless for concurrency reasons. The VCR associates the `Ticket` with a user on a `HTTPSession`. If multiple requests on the same `HTTPSession` arrive for the same repository, multiple operations on the same `Ticket` object can be performed simultaneously. A stateless design will avoid issues in this situation.
- Support for multiple repository instances of the SPI implementation – To work properly when multiple repositories of your SPI implementation are used simultaneously, be careful with static caches by ensuring that the data is tied to a specific repository name or instance. One option is to incorporate the repository name in the data key when caching data. Another option is to use named singletons to access cached data. By keeping the data segregated by repository name, the SPI implementation should work properly in this situation.
- Performance – SPI caching is important; the SPI should generally cache data so it does not need to consult the external system frequently. The SPI can lazy-load several settings to boost performance, such as:
  - Node properties – If the Node is created with a null `Property[]`, the properties are lazy-loaded.
  - Node `objectClass` – If the Node is created with a null `ObjectClass`, but the `ObjectClassId` assigned via `setObjectClassId`, the `ObjectClass` is lazily-loaded.
- Binary property values – When the node's properties (and values) are loaded, the binary property `inputstream` can be returned as null to boost performance.

Basic SPI Implementation

To create a basic SPI implementation:

1. Create a Java class that implements the interface `com.bea.content.spi.flexspi.Repository`.

   This class is a required interface that must be implemented. It is included with WebLogic Portal.

2. Create a public default constructor.

3. Implement `setProperties()` to store the repository configuration properties passed by the VCR. Store in a local variable.

4. Implement `setName()` to store the repository configuration name passed by the VCR. Store in a local variable.

5. Implement `getProperties()`and `getName()` to read the local variables.

6. Implement `getDescription()` to report the repository description data.

   This is essentially a `Properties` bucket, with some well-defined keys in `SPIDescriptionKeys`. It includes the vendor, version, and so on.

7. Implement `getRepositoryDefinedCapabilities()` to report which custom capabilities the SPI implementation supports.

   For a basic SPI implementation, just return `Collections.emptySet()`.

8. Implement `getAllInterfaces()` to return all repository (not `ticket`) operations interfaces.

   These are optional interfaces, such as `RepositoryConfigOpsV1`, that a Repository can implement.

   For a bare-bones SPI implementation, just return an empty `HashMap`.

9. Implement `getInterface( String interfaceName )` to be consistent with `getAllInterfaces()`.

   For a bare-bones SPI implementation, just return null.

10. Implement `getCapabilitySupport( Set<ICapabilityDefinition> )` to report which feature capabilities this SPI implementation supports.

    It also report which method capabilities the repository supports across the optional repository operation interfaces.

11. Implement `connect( Credentials )` and `Connect( String username, String password )` to allow a caller to obtain a ticket.

`connect( username, password )` is called if a username and password are available (generally from the repository configuration data.)

`connect( credentials )` is called if no username/password are available. The credentials includes the caller's implicit identity.

For a basic SPI implementation, create and return a new `Ticket` instance. More advanced implementations may authenticate the credentials, and only return a `Ticket` instance if successful.

12. Implement any optional `Repository` operation interfaces, such as `RepositoryConfigOpsV1`.

    For any repository operation interfaces returned by `Repository.getAllInterfaces()`:

    a. Implement the interface.

       Any methods that are not implemented should throw an `UnsupportedRepositoryOperationException` and the `getCapabilitySupport()` method should not report this method as supported.

    b. Modify `Repository.getAllInterfaces()` and `getInterface()` to return the interface implementation.

    c. Modify `Repository.getCapabilitySupport()` to report the status (supported or unsupported) of each method on the repository operation interface.

13. Create a Java class that implements the interface `com.bea.content.spi.flexspi.Ticket` (supplied with WebLogic Portal)

    a. Implement `getAllInterfaces()` to return all ticket operations interfaces. These are optional interfaces, such as `NodeOpsV1`, which a `Ticket` can implement. For a basic SPI implementation, return an empty `HashMap`.

    b. Implement `getInterface( String interfaceName )` to be consistent with `getAllInterfaces()`. For a basic SPI implementation, just return null.

    c. `Implement getCapabilitySupport( Set<ICapabilityDefinition> )` to report which method capabilities this ticket supports across the optional ticket operation interfaces.

14. Implement any optional `Ticket` operation interfaces, such as `NodeOpsV1`.

    For any ticket optional interfaces returned by `Ticket.getAllInterfaces()`:

    a. Implement the interface.

Any methods that are not implemented should throw an `UnsupportedRepositoryOperationException` and the `getCapabilitySupport()` method should not report this method as supported.

b. Modify `Ticket.getAllInterfaces()` and `getInterface()` to return the interface implementation.

c. Modify `Ticket.getCapabilitySupport()` to report the status (supported or unsupported) of each method on the ticket operation interface.

Basic SPI Repository Implementation Code Example

Listing 4-1shows a simple SPI Repository code example of a Repository with the following limitations:

- It does not authenticate.
- It does not support any feature capabilities.
- It does not support any optional repository operation interfaces, such as `RepositoryConfigOpsV1`, and therefore does not support any method capabilities.
- It does not provide any repository defined capabilities.

Listing 4-1 Basic SPI Implementation

```
import com.bea.content.spi.flexspi.Repository;
import com.bea.content.spi.flexspi.Ticket;
import com.bea.content.spi.flexspi.common.capability.ICapabilityDefinition;
import com.bea.content.spi.flexspi.common.capability.CapabilityLevel;
import
com.bea.content.spi.flexspi.common.capability.FeatureCapabilityDefinition;
import com.bea.content.spi.flexspi.common.ISPIMarker;
import com.bea.content.spi.flexspi.common.SPIRepositoryInterfaces;
import com.bea.content.spi.flexspi.common.SPIDescriptionKeys;
import com.bea.content.capability.NodeFeatureCapability;
import com.bea.content.*;
import java.util.*;
public class FlexRepositoryImpl implements Repository
{
   // VCR configures these before connect() is called
   private String repositoryName;
   private Properties props;

   // the description key/values
   private Map<String, String> descMap = new HashMap<String, String>();
   public FlexRepositoryImpl()
   {
      // SPI implementation MUST have a public default ctor
      //standard keys
      descMap.put(SPIDescriptionKeys.VENDOR_KEY,
         "Some third party vendor");
      descMap.put(SPIDescriptionKeys. VERSION_KEY, "0.1.1");
      descMap.put(SPIDescriptionKeys. DESCRIPTION_KEY, "Simple SPI");

      //custom keys can also be added
      descMap.put("InternalVersion", "Build 31141");
   }
   public Ticket connect(Credentials credentials)
      throws AuthenticationException, RepositoryException
   {
       // this SPI does not perform authentication
       return new FlexTicketImpl(this);
   }
   public Ticket connect(String username, String password)
      throws AuthenticationException, RepositoryException
   {
      // this SPI does not perform authentication
       return new FlexTicketImpl(this);
   }
   public String getName() {
      return repositoryName;
   }
   public void setName(String name) {
      repositoryName = name;
   }
   public Properties getProperties() {
      return props;
   }
   public void setProperties(Properties properties)  {
      props = properties;
   }
   public Set<ICapabilityDefinition> getRepositoryDefinedCapabilities() {
      return Collections.emptySet();
   }
   public Map<String, String> getDescription() {
       return descMap;
   }
   public Map<String, ISPIMarker> getAllInterfaces() {
       // no repository operation interfaces
      return new HashMap<String,ISPIMarker>();
   }
   public ISPIMarker getInterface(String interfaceName) {
       // no repository operation interfaces
```

```
        return null;
    }
    public Map<ICapabilityDefinition, CapabilityLevel>
        getCapabilitySupport(Set<ICapabilityDefinition> capabilities)
    {
        HashMap<ICapabilityDefinition, CapabilityLevel> capMap
            =new HashMap<ICapabilityDefinition, CapabilityLevel>();
        // start out with everything not supported; we will mark
         // individual feature capabilities as supported.
        for (ICapabilityDefinition capDef : capabilities) {
            capMap.put(capDef, CapabilityLevel.NotSupported);
    }
        // here we would override the unsupported value if we supported anything
        // but we don't…

        // everything unsupported
        return capMap;
    }
}
```

Basic SPI Ticket Implementation Code Example

Listing 4-2 shows a simple SPI Ticket code example of a Ticket that does not support any optional ticket operation interfaces, such as `NodeOpsV1` and `SearchOpsV1`, and therefore does not support any method capabilities

Listing 4-2 Ticket Implementation Code Example

```
import com.bea.content.spi.flexspi.Ticket;
import com.bea.content.spi.flexspi.Repository;
import com.bea.content.spi.flexspi.common.ISPIMarker;
import com.bea.content.spi.flexspi.common.SPITicketInterfaces;
import com.bea.content.spi.flexspi.common.capability.ICapabilityDefinition;
import com.bea.content.spi.flexspi.common.capability.CapabilityLevel;
import com.bea.content.spi.flexspi.common.capability.MethodCapabilityDefinition;

import java.util.*;

public class FlexTicketImpl  implements Ticket
{
  Repository repository;  // the repository this ticket was created from

  //1=flex interface name, 2=flex interface object
  private Map<String, ISPIMarker> advertisedInterfaces;
  public FlexTicketImpl(Repository repository)  {
    this.repository = repository;

    advertisedInterfaces= new HashMap<String,ISPIMarker>();
    // no interfaces yet
  }

  public Map<String, ISPIMarker> getAllInterfaces() {
    return advertisedInterfaces;
  }

  public ISPIMarker getInterface(String interfaceName) {
    return advertisedInterfaces.get(interfaceName);
  }

  public Map<ICapabilityDefinition, CapabilityLevel>
    getCapabilitySupport(Set<ICapabilityDefinition> capabilities)
  {
    // no interfaces, no methods, no capabilities; everything is unsupported

    HashMap<ICapabilityDefinition, CapabilityLevel> capMap
      =new HashMap<ICapabilityDefinition, CapabilityLevel>();

      // start out with everything not supported; we will mark
      // individual feature capabilities as supported.
      for (ICapabilityDefinition capDef : capabilities) {
        capMap.put(capDef, CapabilityLevel.NotSupported);
      }

      // here we would override the unsupported value if we supported anything
      // but we don’t…

      // everything unsupported
      return capMap;
  }
}
```

Optional SPI Interfaces Implementation

You use optional SPI interfaces to expose nodes and types to the VCR. Generally, client code has certain assumptions about the capabilities of the repositories the code runs against, such as read-only access to nodes. As the VCR delegates to the repository, the client code presumes that certain VCR methods work properly.

The optional PSPI interfaces are:

- Repository operation interfaces – The `RepositoryConfigOpsV1` provides repository callbacks as repository configurations that are modified by VCR clients (generally, administrators). For example, `createRepository(…)`, `updateRepository(…)`, and `removeRepository(…)`.
- Ticket operation interfaces – The following interfaces provide CRUD operations:
  - Nodes – `NodeOpsV1`.
  - Types – `ObjectClassOpsV1`.
  - Operations for searching for nodes – `SearchOpsV1`.
  - Workflows – `WorkflowOpsV1`.

-Exposing an Optional SPI Interface

To expose an optional SPI interface such as `NodeOpsV1`.

1. Create a class to implement the SPI interface.

   For example `MyNodeOps` implements `NodeOpsV1`. Generally, you should make this a light-weight class, as many objects may be created.

2. Write implementations of the SPI interface methods.

   Note: Any methods that are not implemented should throw an `UnsupportedRepositoryOperationException`. Additionally, the `Ticket.getCapabilitySupport()` method should report this method as not supported.

3. `Modify Ticket.getAllInterfaces()` and `getInterface()` to return the interface implementation.

To reduce object creation, create the interfaces when the ticket is created, hold onto them, and then return them in `getAllInterfaces()` and `getInterface()`. For example:

```
private Map<String,ISPIMarker> ifaces;
public FlexTicketImpl( Repository repository ) {
   this.repository= repository;
//init interfaces
   ifaces= new HashMap<String,ISPIMarker>();
   ifaces.put(SPITicketInterfaces.NODE_OPS_V1,
      new MyNodeOps(…);
   . . .
}

public Map<String, ISPIMarker> getAllInterfaces() {
   return ifaces;
}

public ISPIMarker getInterface( String ifaceName ) {
   return ifaces.get( ifaceName );
}
```

4. Modify `Ticket.getCapabilitySupport()` to report the status (supported or unsupported) of each method on all the ticket operation interfaces. For example:

```
public Map<ICapabilityDefinition, CapabilityLevel>
   getCapabilitySupport(Set<ICapabilityDefinition> capabilities)
{
   Map<ICapabilityDefinition, CapabilityLevel> capMap
      =new HashMap<ICapabilityDefinition, CapabilityLevel>();

   // start out with everything not supported; we will mark
   // individual feature capabilities as supported.
   for (ICapabilityDefinition capDef : capabilities) {
      capMap.put(capDef, CapabilityLevel.NotSupported);
   }

   // now override the unsupported values where it makes sense
   final String[] supportedNodeOpsMethodNames = new String[] {
      NodeOpsV1.MethodName.getNodeChildren.toString(),
      NodeOpsV1.MethodName.getNodeChildrenAsNodeIds.toString(),
      NodeOpsV1.MethodName.getNodesWithIds.toString(),
      NodeOpsV1.MethodName.getNodeWithId.toString(),
      NodeOpsV1.MethodName.getNodeWithPath.toString(),
   };

   for (String methodName : supportedNodeOpsMethodNames) {
      ICapabilityDefinition capDef
         = new MethodCapabilityDefinition(
            SPITicketInterfaces.NODE_OPS, methodName
            );
         if (capabilities.contains(capDef)) {
                capMap.put(capDef, CapabilityLevel.FullySupported);
            }
        }

        return capMap;
    }
```

SPI Interface Result Collections, Sorting, and Filtering

Optionally, the SPI implementation can include the ability to sort and/or filter results. The collections of items returned by the SPI are returned in a `QueryResult` object (including an ordered list of results) and a `QueryCriteria` object that describes how the returned collection is sorted and filtered (if at all). For instance, a `QueryResult<Node>` may contains a collection of nodes that are both sorted and filtered.

Filtering and Sorting Results with the SPI

Many of the methods that return results collections take a `QueryCriteria` parameter. This parameter allows the caller to indicate how results should be sorted or filtered or both (if possible). For example, the caller may request that the results be sorted by name. At present, the caller can specify only one sort criteria and one filter criteria.

The `SortCriteria` contains a property (criteria) and an ascending or descending flag, such as `name ascending`.

The `FilterCriteria` contains a property (criteria), a `FilterMethod` operand, such as unfiltered, equals, not equals, contains, not contains, begins with, ends with, greater than, and less than, and a value. For example, `name contains 'foo'`.

The sort and filter criteria set of properties are related to the JavaBean properties on the data objects being used. For example, `Node` contains a JavaBean property for `name` because it has a `getName()` method; for `createdDate` because it has a `getCreatedDate()` method; and so on.

Native sorting and filtering performs best. The VCR also supports mechanisms for sorting and filtering in-memory if the SPI implementation is unable to service a sort or filter request. Client code (and the VCR) can ask the SPI which properties it can natively sort and filter.

The SPI implementation reports its sorting and filtering capabilities (the properties it can sort and filter on) to the VCR for objects on an interface by implementing methods such as `NodeOpsV1.getNativeSortableProperties()` and `NodeOpsV1.getNativeFilterableProperties()`. For example, the SPI may report that it can sort nodes by `name` and `createDate` properties.

The sorting and filtering capability reporting is currently done at the interface granularity. In other words, a given interface such as `NodeOpsV1` has a primary data object. The primary data object, a `Node` for example, reports the capabilities for sorting and filtering across the results collection methods in the interface.

Tip: If the SPI does not support native sorting or filtering, it should return an empty set, such as `Collections.emptySet()` rather than null.

The SPI implementation may receive a `QueryCriteria` parameter that requests sorting or filtering capabilities beyond what it can support. If this happens, the SPI implementation should not throw an exception. Instead the SPI implementation should do its best to report how the results are currently sorted and filtered. For example, the SPI implementation should create an unsorted and/or unfiltered `QueryResult` to express what it was unable to sort and/or filter. If the SPI implementation can perform one of the requests, it should do so and report the results appropriately. For instance, if it can sort, the SPI implementation should report the results as sorted, but unfiltered.

If necessary the VCR may sort/filter the query results in memory to ensure the query results are sorted/filtered as the client requests.

Common SPI Interface Objects for Sorting and Filtering

Use the following objects for sorting and filtering:

- `QueryCriteria` – Some methods pass a `QueryCriteria` object as a means for a caller to request sorting and filtering of results. These methods include `SortCriteria` and `FilterCriteria` objects. Currently, only a single `SortCriteria` or `FilterCriteria` can be specified; multi-criteria sorting or filtering is not supported.
- `SortCriteria` – Provides the ability for ascending or descending sorting on a specific property (criteria). A `sortResults` flag indicates whether the results are sorted or unsorted. `SortCriteria` also includes a `property` that is one of the JavaBean properties on the data object.
- `FilterCriteria` – Provides the ability to filter on a specific property (criteria). A `filterResults` flag indicates whether the results are filtered or unfiltered. `FilterCriteria` includes a `property` that is one of the JavaBean properties on the data object, and a `filterMethod` that indicates the filter operation, such as begins with, contains, equals, greater than, unfiltered, and so on.

- `QueryResult` – Represents a set of results returned by the SPI implementation. `QueryResult` also includes a `QueryCriteria` object that describes how the results are sorted and filtered (or neither), plus an ordered list of the data objects.

CHAPTER 5

Interface Topics

This chapter provides useful information for writing an SPI implementation. It contains the following sections:

- NodeOpsV1 SPI Interface Topics
- ObjectClassOpsV1 SPI Interface Topics
- SearchOpsV1 SPI Interface Topics
- Indexing Content
- SPI Testing Topics

NodeOpsV1 SPI Interface Topics

The following FAQs provides useful information when implementing the NodeOpsV1 interface. It contains the following questions, answers, and examples:

- What Types of Operations are Supported by the NodeOpsV1 SPI Interface?
- What Type of Hierarchical Paths are Passed To and From the SPI layer?
- How Should the SPI Implementation Create Node Data Objects to be Returned?
- How Should the SPI Implementation Create Property Data Objects to be Returned
- What Should the SPI Implementation Do when Node Metadata is not Available?
- How are the Node ID and Property ID related?

- What Node Names are Valid?
- Example – Creating a Node with no ObjectClass
- Example – Creating a Node with an ObjectClass and Property Values

What Types of Operations are Supported by the NodeOpsV1 SPI Interface?

The `NodeOpsV1` interface supports all the operations on nodes, properties, and values:

- Creating nodes.
- Retrieving nodes by ID or Path.
- Reading groups of nodes by hierarchy positions and by link property location.
- Updating nodes.
- Deleting nodes.
- Copying nodes.
- Moving nodes.
- Retrieving node properties.
- Retrieving node binary values.

What Type of Hierarchical Paths are Passed To and From the SPI layer?

The paths used in the SPI layer must start with a path delimiter (`/`) and extend based on the repository path. For example, `/foo/bar/bas`. The repository name is not part of the path unlike a VCR federated path. For example, `/MyRepository/foo/bar/bas` is not a valid SPI path, but it is a valid VCR path.

How Should the SPI Implementation Create Node Data Objects to be Returned?

The SPI implementation can directly instantiate the `Node` via a constructor such as:

```
new Node( Calendar createdDate, String createdBy, boolean hasChildren, ID
id, String modifiedBy, Calendar modifiedDate, ObjectClass objectClass, ID
parentId, String path, Property[] properties )
```

The `Node ID` needs a non-null UUID set; however, the `Node ID` does not need the `repositoryName` set, as the VCR takes care of this function.

Previous versions of WebLogic Portal had a concept of "node type." Starting with WebLogic Portal 10.2, SPI implementations should use the node type of `Node.CONTENT` when a node type is necessary.

If the node has an `ObjectClass`, you can create a `Property` on the `Node` for any `PropertyDefinition` in the `ObjectClass`.

To optimize performance, the SPI implementation can optionally pass `null` for the `Property[]` properties. This allows the properties to be lazy-loaded by the VCR when needed. Additionally, the SPI implementation can optionally use a null `ObjectClass` and call `setObjectClassId()` to set the `ObjectClass` identifier. This allows the `objectClass` to be lazy-loaded by the VCR when needed.

You can use these approaches on a per-method basis. In general, use these approaches for methods that may return a large collection of nodes. For methods returning a single node, it is generally not advisable to lazy-load the properties.

How Should the SPI Implementation Create Property Data Objects to be Returned

The SPI implementation can directly instantiate each `Property` via a constructor such as:

```
new Property( ID id, String name, int type, Value[] values )
```

The `Property ID` needs a non-null UUID set; it does not need the `repositoryName` set, as the VCR takes care of the repository name.

What Should the SPI Implementation Do when Node Metadata is not Available?

The SPI implementation requirements, when metadata (created by, modified date, and so on) is not available, depends on what data the VCR clients use. For maximum flexibility with clients, it is best for the SPI implementation to return constant values rather than null. For example, the SPI could return `system` as the "created by" or "modified by" `String`, or return a constant `Date` as the created date or modified date.

How are the Node ID and Property ID related?

The `Property ID` is a finer granularity than the `Node ID`; it represents data within a node. The exact relationship depends on the back-end system and how it represents and can access node and property data.

The `Node UUID` must uniquely identify the node within an implicit repository. You use the `Node UUID` retrieve all the properties (and values) for a given node.

The `Property UUID` must uniquely identify a single property on a single node within an implicit repository. You use the `Property UUID` to retrieve a single property on a node.

The `Node UUID` is not always supplied when the properties are retrieved. For example, with `NodeOpsV1.getPropertyBytes( ID propertyId )` the `Property UUID` needs to be able to "stand alone" and work on its own. One option for linking the IDs together is to have the `Property ID` contain the `Node ID`. For example, if the `Node UUID` is "41431", the `Property UUID` could be "41431/stringProperty".

What Node Names are Valid?

In general, the node name must be a non-empty string, must not contain forward or backslashes, and the last token in a node's hierarchical path must be the node name. For detailed information, see the WebLogic Portal Javadoc.

Example – Creating a Node with no ObjectClass

Listing 5-1 shows an example of how to create a node without an `ObjectClass`.

Listing 5-1 Creating a Node with no ObjectClass

```
ID id = new ID(uid);
ID parentId = new ID(parentUid);
Node node =  new Node(createDate, createdBy, false, id, createdBy,
   createDate, null /* no ObjectClass */, parentId, path, null);
```

Example – Creating a Node with an ObjectClass and Property Values

Listing 5-2 shows an example of how to create a node with an `ObjectClass` and property values.

Listing 5-2 Creating a Node with an ObjectClass and Property Values

```
ID id = new ID(uid);
ID parentId = new ID(parentUid);
Property [] props = getPropertiesToUse();
Node node =  new Node(createDate, createdBy, false, id, createdBy,
createDate, null /* lazy-load ObjectClass */, parentId, path, props);
ID ocId = new ID(objectClassUid);
node.setObjectClassId(ocId);   //lazy-load ObjectClass
```

ObjectClassOpsV1 SPI Interface Topics

This section contains information about implementing `ObjectClassOpsV1`. It contains the following sections:

- What are the Supported Operation Types?
- How Should the SPI Implementation Create ObjectClass Objects?

What are the Supported Operation Types?

The `ObjectClassOpsV1` interface supports all the operations on `ObjectClasses`, `PropertyDefinitions`, and `PropertyChoices`, which are:

- Creating `objectClasses` and `propertyDefinitions`.
- Retrieving an `objectCass` by ID or name.
- Retrieving a `propertyDefinition` by ID.
- Retrieving all `propertyDefinitions` by `objectClass` ID.
- Updating an `objectClass` or property definition.

- Deleting `objectClasses` and `propertyDefinitions`.
- Renaming an `objectClass`.
- Retrieving some or all `objectClasses`.
- Retrieving property choice binary values.

To support type inheritance, `objectClasses` support a hierarchical structure. `ObjectClasses` have a `Name` and an `ID` that are used for identification. They also have a path. When type inheritance is used, the `ObjectClass` path indicates its relationship to other `ObjectClasses`.

How Should the SPI Implementation Create ObjectClass Objects?

The SPI implementation can directly instantiate each `ObjectClass` using a constructor such as:

```
new ObjectClass( ID id, String name, PropertyDefinition
   primaryPropertyDefinition, PropertyDefinition[] propertyDefinitions,
   boolean hasPropertyDefinitions );
```

The `hasPropertyDefinitions` flag indicates whether `PropertyDefinitions` exist for this type in the external system and supports lazy-loading of `propertyDefinitions`. For example, `propertyDefinitions` may exist in the external system, but not be returned in the `ObjectClass`.

To enhance performance, the SPI implementation can optionally pass `null` for the `PropertyDefinition[] propertyDefinitions`. This allows the `propertyDefinitions` to be lazy-loaded by the VCR when needed.

SearchOpsV1 SPI Interface Topics

The `SearchOpsV1` interface supports node search and indexing operations. The following operations are supported:

- Searching for nodes matching criteria, either metadata search or full-text search.
- Manually indexing or re-indexing a node tree by path.
- Manually indexing or re-indexing all nodes with a specified `ObjectClass`.

The `SearchOpsV1` interface supports the following search criteria:

- System metadata – The JavaBean properties on the `Node`, such as name, `ObjectClass` name, created by, and so on.
- User metadata – The `ObjectClass` properties.

Indexing Content

Content indexing allows for fast lookups, especially for full-text searches. For example, when nodes are created, data can be indexed into a full-text search engine for fast retrieval.

This section contains the following topics:

- How Is Content Indexed?
- How Can an Event Listener Perform Content Indexing?

How Is Content Indexed?

You can index content in two ways:

- Synchronously, during node creation. To do this, use the SPI implementation of `NodeOpsV1.create` methods or use a synchronous event listener that is registered to listen to node create events.
- Asynchronously, after node creation. To do this, use an asynchronous event listener that is registered to listen to node create events controlled via an external timer or mechanism.

  Use the index_cm_data script to re-index by path or content type.

The synchronous approach makes node creation slower, but supports immediate searches for the data. The asynchronous approaches make node creation faster, but the data is not available immediately.

Note: If you use event listeners, you can temporarily disable content indexing, then create a group of nodes, re-enable content indexing, and then manually re-index the relevant data tree with the index_cm_data script.

How Can an Event Listener Perform Content Indexing?

The event listener must implement the interface `com.bea.p13n.events.EventListener`. You can register the event listener via a `META-INF/p13n-config.xml` file with an entry such as:

```
<?xml version="1.0" encoding="UTF-8"?>
<p13n-config xmlns="http://www.bea.com/ns/p13n/90/p13n-config">
```

```
   <event-service>
      <listener>com.xxx.ContentExporterListener</listener>
   </event-service>
</p13n-config>
```

The event listener should listen for the event type `ContentEventHelper.CONTENT_EVENT_BATCH_TYPE`. The event listener's `handleEvent( Event)` method is called as content items are created, updated, or deleted. This method receives events of type `ContentEventBatch`, which contains a group of events that have fired.

SPI Testing Topics

This section contains informations to aid in testing an SPI implementation. It contains the following topics:

- How to Configure a Repository for SPI Parameter and Response Data Checking
- How to Monitor Repository and Ticket Method Invocations and Performance
- How to Monitor SPI Operation Interface Method Invocations and Performance

How to Configure a Repository for SPI Parameter and Response Data Checking

During testing, you can configure a repository to perform SPI parameter and response data checking. The `vcrValidation` and `repositoryValidation` repository configuration settings enable additional runtime checks on the data passed to and from the SPI. These settings catch some of the common errors. By default, both settings are false (disabled) to maximize performance.

To enable VCR validation, add these settings to your repository configuration:

```
Property name: 'vcrValidation'
Property value: 'true' (default is false)

Property name: 'repositoryValidation'
Property value: 'true' (default is false)
```

Alternatively, you can place this code snippet in your `MTA-INF/content-config.xml` file in the appropriate section for your repository configuration:

```
<repository-property>
   <name>vcrValidation</name>
   <value>true</value>
</repository-property>

<repository-property>
   <name>repositoryValidation</name>
   <value>true</value>
</repository-property>
```

How to Monitor Repository and Ticket Method Invocations and Performance

To debug repository and ticket operations, add debug lines to your `debug.properties` file:

```
spi.com.bea.content.manager.internal.RepositoryHelper: ON
spi.com.bea.content.manager.internal.RepositoryManagerImpl: ON
```

To collect timing information, add:

```
timing.com.bea.content.manager.internal.RepositoryHelper: ON
timing.com.bea.content.manager.internal.RepositoryManagerImpl: ON
```

How to Monitor SPI Operation Interface Method Invocations and Performance

To enable SPI invocation, add a debug line to your `debug.properties` file:

```
spi.com.bea.content.manager.internal.delegate: ON
```

To enable SPI method timing, add a debug line to your debug.properties file:

```
timing.com.bea.content.manager.internal.delegate: ON
```