Chromattic Reference Guide

Chromattic

Julien Viet (eXo Platform), Alain Defrance (eXo Platform)

Preface

Chromattic development started during July 2009 when I had to develop a rich model called MOP (Model Object for Portals) that was persisted in a JCR repository. The development started with the prototyping of the model as a set of Java interfaces, just bare interfaces plus a set of value objects. Once I was satisfied with the initial model, I decided it was time to write the JCR persistence implementation and quickly I realized that I would not be able to achieve it without the help of a tool.

Obviously the idea of using a mapping framework stroke me and I fell in the Not Invented Here syndrom for some reason.

If you are reading this chapter it's probably because you are not yet convinced that Chromattic can do something useful for you (if you are already convinced, read this chapter anyway so you can convince other persons) and *some reason* is probably not enough to convince you.

JCR defines a set of base node types for modelling a file system and that's a perfect example to use:

- `nt:hierarchyNode`: a super type for file and folder, its purpose is mainly to define a common node type for children of a folder
- `nt:resource`: a node type for modelling a resource, basically it's data
- `nt:file`: a node type for a file, it contains data via a `jcr:content` child node of type `nt:resource`
- `nt:folder`: a node type for a folder with children of type `nt:hierarchynode`

The following examples list the content of a directory structure and we have two versions, one using the native JCR API and one using Chromattic objects mapped onto the same node types.

Example 1. Directory listing with the JCR native API

```
private void list(Node node) throws RepositoryException {
  if (!node.isNodeType("nt:hierarchyNode")) {
    throw new IllegalArgumentException("The provided node is not a hierarchy node");
  }
  if (node.isNodeType("nt:file")) {
    if (node.hasNode("jcr:content")) {
      Node content = node.getNode("jcr:content");
      String encoding = null;
      if (content.hasProperty("jcr:encoding")) {
        encoding = content.getProperty("jcr:encoding").getString();
      }
      String mimeType = content.getProperty("jcr:mimeType").getString();
      System.out.println("File[name=" + node.getName() + ",mime-type=" + mimeType +
        ",encoding=" + encoding +  "]");
    }
  } else if (node.isNodeType("nt:folder")) {
    System.out.println("Folder[" + node.getName() + "]");
    for (NodeIterator i = node.getNodes();i.hasNext();) {
      list(i.nextNode());
    }
  }
}
```

Example 2. Directory listing with Chromattic objects

```
private void list(NTHierarchyNode hierarchy) {
  if (hierarchy instanceof NTFile) {
    NTFile file = (NTFile)hierarchy;
    Resource content = file.getContentResource();
    if (content != null) {
      System.out.println("File[name=" + file.getName() + ",mime-type=" + content.getMimeType() +
        ",encoding=" + content.getEncoding() +  "]");
    }
  } else {
    NTFolder folder = (NTFolder)hierarchy;
    System.out.println("Folder[" + folder.getName() + "]");
    for (NTHierarchyNode child : folder.getChildren().values()) {
      list(child);
    }
  }
}
```

There are several difference between the two versions, but the most important one is **type safety**. The JCR version use `javax.jcr.Node` objects and the main drawback is that the effective type of a node is never known until runtime. Chromattic main purpose is to provide type safety to Java programs that use JCR:

- The `list` method argument is typed with `NTHierarchyNode` and that guarantees that the method will never be invoked with an appropriate node type, this guarantee is enforced during the compilation of any program that wants to invoke the `list` method.
- The `instanceof` operator is what a Java developer uses when he wants to determine the type of an object. The JCR version performs the same operation but there is more work to do.

The second benefit is object oriented programming: each node turns into a Chromattic object, and on that object you can add any method you need to. This is just what we use in this example with the `getContentResource()` method on the `NTFile` object.

The third benefit is productivity: modern IDEs provide an impressive set of tools that gives a lot of power to the developer, Chromattic type safe and object oriented nature is a perfect fit:

- A Chromattic object is a Java object and the IDE is able to perform code completion.
- Refactoring is a commodity offered by any IDE that can be leveraged on a Chromattic model.

There are many other reasons left to use Chromattic, let's discover them in this guide.

Chapter 1. Getting started with Chromattic

This chapter introduces you to the basic of Chromattic and the Java Content Repository to object mapping. We will show the most basic Chromattic application and focus on the various steps to build this application.

1.1. The website example

The project example models a web site persisted in a JCR server. The site contains web pages that are organized according to a tree structure making easy to display the pages on the web. The natural JCR hierarchy tree shape will model the hierarchy of the pages.

1.1.1. The Page object

The `org.chromattic.docs.reference.gettingstarted.Page` class is the object representation of a web page. The `Page` object is mapped to the JCR *page* node type. The `Page` class contains the properties we want for the representation of a web page:

- The property `name` is the web page name and is mapped to the JCR node name.
- The property `title` is the page title and is mapped to the JCR *title* node property of type `STRING`.
- The property `content` is the page content and is mapped to the JCR *content* node property of type `STRING`.

Important

The Javabean properties needs to be modelled as abstract methods because it allows Chromattic to implement them to make the mapping between objects and node possible.

Example 1.1. The Page class

```
/**
 * The page of a site.
 */
@PrimaryType(name = "gs:page") ❶
public abstract class Page {

  /**
   * Returns the page name.
   * @return the page name
   */
  @Name
  public abstract String getName(); ❷

  /**
   * Returns the page title.
   * @return the page title
   */
  @Property(name = "title")
  public abstract String getTitle(); ❸

  /**
   * Updates the page title.
   * @param title the new page title
   */
  public abstract void setTitle(String title);

  /**
```

```
    * Returns the page content.
    * @return the page content
    */
  @Property(name = "content")
  public abstract String getContent(); ❹

  /**
    * Updates the page content.
    * @param content the new page content
    */
  public abstract void setContent(String content);

}
```

❶ The Page class is mapped to the page node type
❷ The name property is mapped to the node name
❸ The title property is mapped to the title node property
❹ The content property is mapped to the content node property

Chromattic uses code annotations to declare which and how classes are mapped to node types. The most important annotation is the `@org.chromattic.api.annotations.PrimaryType` that declares the mapping of a class to a node type. Our class is annotated with the `@PrimaryType` annotation, the `name` parameter specifies the name of the node type mapped to the class.

Note

JCR defines two kinds of node types which are primary node type and mixin node type. By default we denote by node type a primary node type. Mixin node type can also be mapped by Chromattic that is explained in the chapter XYZ.

The `@org.chromattic.api.annotations.Name` annotation targets Javabean property getters or setters and indicates that the property is mapped to the name of the node. Indeed each JCR node has a mandatory name and this is the way to expose it on a class. As a result the `Page` *name* property is mapped to the node name.

Like the `@Name` annotation the `@org.chromattic.api.annotations.Property` annotation targets Javabean properties. It specifies how a property is mapped to a node property. It has a mandatory `name` parameter that specifies the node property name. The node property type does not need to be specified as it is deduced from the class property. In our example, we map the *content* Javabean property to a *content* node property.

1.1.2. The JCR node types

Node types are important for JCR, they define the schema of the node data. In our application we have a `page` node type that is modelled after the `Page` class. Chromattic can generate for you the node type definition when the classes are compiled. It results in a `nodetype.xml` file resources located in the class output of the Java™ compiler.

The annotation `org.chromattic.api.annotations.NodeTypeDefs` instructs the compiler to generate the the node type definitions in the XML format that can be used by the JCR server to create the node type. The annotation targets a package and it generate the node type for any Chromattic class inside this package and in the sub packages.

Example 1.2. The org.chromattic.docs.reference.gettingstarted.package-info.java file package

```
@NodeTypeDefs package org.chromattic.docs.reference.gettingstarted;
```

```
import org.chromattic.api.annotations.NodeTypeDefs;
```

Warning

The node type generation is still a work in progress and should be considered as an experimental feature

1.1.3. The client

We have designed and mapped our `Page` object and now we will examine how to interact with a JCR server via Chromattic. The goal of the client is very simple and focus on demonstrating the bootstrap of Chromattic and the persistence of a simple `Page` in the Java Content Repository.

1.1.3.1. Chromattic bootstrap

The boostrap is the creation and the configuration of the Chromattic runtime. Usually the bootstrap occurs during the initialization of the application, for instance in a web application, it is most often performed in a `ServletContextListener` initialization.

Chromattic bootstrap relies mainly on the `ChromatticBuilder` object. The builder is configured with the Chromattic application classes to obtain an instance of the `Chromattic` object. The `Chromattic` object can be used to create `ChromatticSession` objects. The `ChromatticSession` is the main runtime API used to interract with Chromattic.

```
    ChromatticBuilder builder = ChromatticBuilder.create(); ❶
    builder.add(Page.class); ❷
    Chromattic chromattic = builder.build(); ❸
```

❶ Creates the builder object
❷ We add the Page class to the builder object
❸ Now the Chromattic object can be created

1.1.3.2. Interacting with Chromattic objects

We have just explained how to obtain a `Chromattic` object thanks to the builder. Now it is time to show how to obtain and use the `ChromatticSession` with the goal to insert a new page node. Let's examine the code:

```
    ChromatticSession session = chromattic.openSession(); ❶
    try
    {
      Page page = session.insert(Page.class, "index"); ❷
      page.setTitle("Hello Page"); ❸
      page.setContent("Hello World"); ❹
      session.save(); ❺
    }
    finally
    {
      session.close(); ❻
    }
```

❶ Any Chromattic interaction requires to open a session
❷ A new page is inserted under the /index path
❸ Set the title property
❹ Set the content property

❺ Saves the session to persist changes in the repository
❻ We must close the session to properly release the session

1.1.4. Project build

The project build is an important piece of the software infrastructure and Chromattic has been developped to integrate seamlessly with the build.

Chromattic leverages the Java™ 6 Annotation Processor Tool (abbreviated as APT) that works at the Java™ compiler level and therefore there is nothing much to do to integrate Chromattic in the build itself.

As many Object Relational Mapping tool, Chromattic needs a bit of instrumentation to make the magic work. Chromattic does not modify existing classes, it takes the existing classes and adds new classes and that is achieved thanks to the APT plugin. It means that instrumentation is performed at the compilation time by generating Java™ source file that are compiled by the compiler instead of generating those classes at the load time in the virtual machine.

The only condition to enable Chromattic instrumentation is to have the Chromattic APT jar on the compilation classpath. Nothing more, nothing less.

1.1.4.1. Building with Maven

Building with Maven is very easy and only requires a dependency on the Chromattic API and APT module in your pom file.

- The API dependency provides the Chromattic API classes prefixed with the `org.chromattic.api` package.

```
<dependency>
  <groupId>org.chromattic</groupId>
  <artifactId>chromattic.api</artifactId>
</dependency>
```

- The APT dependency triggers the Chromattic instrumentation.

```
<dependency>
  <groupId>org.chromattic</groupId>
  <artifactId>chromattic.apt</artifactId>
</dependency>
```

And that's it, we have just configured our project.

1.1.5. Running the client

The client requires different jars for running

todo : provide an uber client jar

Chapter 2. Type mapping

Chromattic establishes a correspondance between a Java class and a JCR node type. In most case there is a trivial mapping between a Java class and a JCR node type, however both models are not the same. Chromattic offers solutions for mapping JCR concepts like mixin and multiple type inheritance which are not native to the Java language.

2.1. Primary type mapping

The `org.chromattic.api.annotations.PrimaryType` annotation creates a unique correspondance between a Java class and JCR primary node type. The mapping between an annotated class and the primary type must be unique for the JCR node type, therefore it is not possible to have the same node type mapped to more than one class inside the same Chromattic application.

2.2. Property mapping

2.2.1. Property type mapping

JCR defines the following set of property types:

- The `STRING` type
- The `BOOLEAN` type
- The `LONG` type
- The `DOUBLE` type
- The `DATE` type
- The `NAME` type
- The `BINARY` type
- The `PATH` type
- The `REFERENCE` type

Any of those types except the `REFERENCE` type can be mapped to an object property.

`REFERENCE` types can be used, however this type is not mapped to a specific Java type, instead Chromattic supports it thanks to the concept of relationship that will be explained in the Chapter 4, *Reference mapping* .

2.2.1.1. Generic data types

JCR provides two types to map generic data types:

- The JCR `STRING` type is mapped to the Java `java.lang.String` type.
- The JCR `BINARY` type is mapped to the Java `byte[]` type or the `java.io.InputStream` type.

The string type is pretty straightforward to use, you simplet get or set the string that is mapped to the JCR property.

The binary type can be used in two different manners, the first one maps the `BINARY` type to a byte array. This mapping style is similar to the string mapping except that a byte array is not immutable. The client has the opportunity to alter the array as Chromattic cannot prevent it to be modified. This mapping style is very straightforward too but has the inconvenient to load the whole stream into memory which is not always desirable for very large streams.

The other manner maps the `BINARY` type to an `java.io.InputStream`. This behavior is actually the JCR native behavior and Chromattic provides it as well, as it has the benefit to use an input stream to read and write binary data which is efficient for large binary content. This approach does not force to hold all the data in memory, unlike the byte array approach. However it requires a little extra work from the developer to use the input stream carefully.

To read the data, the property getter returns an input stream that provides access to the binary data. The stream should be used as any other kind of input stream: consume data until the stream is empty and then close the stream in a finally block. The stream must be used corrected, otherwise the entire content could be loaded in memory and that would defeat the purpose of the stream based approach.

To write data, the property setter must be called with an input stream that is used to consume all the data available. It means that on the return of the setter, the input stream shouldn't be used anymore for reading data as Chromattic will close the stream. Again here, the stream must be used carefully.

2.2.1.2. Primitive types

The types `BOOLEAN`, `LONG` and `DOUBLE` are mapped to Java primitive types:

- The JCR `BOOLEAN` type is mapped to the Java `boolean` type
- The JCR `LONG` type is mapped to either the Java `int` or `long` type
- The JCR `DOUBLE` type is mapped to either the Java `double` or `float` type

For each of those types, there is the choice between either the Java primitive type or the Java wrapper type.

2.2.1.3. Temporal type

JCR defines a `DATE` type that represents a date. Chromattic provides three different mappings for this type:

1. Java date objects
   a. `java.util.Calendar` mapping, the same type exposed by the native JCR API.
   b. `java.util.Date` mapping
2. `java.lang.Long` or `long` mapping exposing the value returned by `Calendar#getTimeMillis()`

Date objects objects are mutable by nature and Chromattic clones them when it is necessary to preserve the data. A date object returned by Chromattic can be modified without changing mapped JCR value, likewise a property update will read the value once and copy it.

2.2.2. Simple property mapping

The `org.chromattic.api.annotations.Property` annotation binds an object to a node property. Our `Page` shows several examples of property mapping using the `@Property` annotation. This annotation has a mandatory `name` parameter to provide the name of the corresponding JCR property.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Property {

  /**
   * The jcr property name.
   *
   * @return the jcr property name
   */
  String name();

  /**
   * Specify the property type of the mapped property, the value must be a legal value referenced by
   * {@code javax.jcr.PropertyType}. The default value returned is -1 which means that the value is guessed
   * by Chromattic according to the type of the annotated property.
   *
   * @return the property type value.
   * @since 1.1
   */
  int type() default -1;

}
```

The `Property` annotation can either annotate the getter or annotate the setter but it should be used only once with read/write accessible properties.

2.2.2.1. Single valued property mapping

The most common mapping style between a single valued class and a node property. The object property must provide at least a setter method or a getter method, probably both in most use cases, those methods must use the same java property type.

A property getter method returns the JCR property value. If the property does not exist, the null value is returned when the java property type is not a primitive type. Sometimes it can happen that the JCR property does not exist but this property is mapped to a primitive type. When the situation occurs Chromattic throws a `NullPointerException`, that behavior is similar to what happens when a null value is unboxed to its corresponding primitive type.

A property setter method updates the JCR property value when it is invoked. For non primitive type it is possible to delete the property by providing a null argument.

```
  /**
   * Returns the page title.
   *
   * @return the page title
   */
  @Property(name = "title")
  public abstract String getTitle(); ❶

  /**
   * Updates the page title.
   *
   * @param title the new page title
   */
  public abstract void setTitle(String title);

  /**
   * Returns the date of the page last modification.
   *
   * @return the date of the last modification
   */
  @Property(name = "lastmodifieddate")
  public abstract Date getLastModifiedDate(); ❷
```

```
  /**
   * Updates the date of the page last modification.
   *
   * @param date the date of the last modification
   */
  public abstract void setLastModifiedDate(Date date);
```

❶ The title property is mapped the `STRING` type
❷ the last modified date property is mapped to the `DATE` type

The corresponding JCR node defines a title property and lastModifiedDate property:

```
<propertyDefinition autoCreated="false" mandatory="false" multiple="false" name="title" onParentVersion="COPY" p
  <valueConstraints/>
</propertyDefinition>
<propertyDefinition autoCreated="false" mandatory="false" multiple="false" name="lastmodifieddate" onParentVersi
  <valueConstraints/>
</propertyDefinition>
```

2.2.2.2. Multi valued property mapping

JCR naturally provide support for multi valued properties, so does Chromattic. Chromattic gives you the choice to use either an array or a `java.util.List` to access the data. A primitive array can be used when the type is a primitive type.

```
  /**
   * Returns the list of the page tags.
   *
   * @return the list of tags
   */
  @Property(name = "tags")
  public abstract List<String> getTags(); ❶
```

❶ the tags property is mapped to a multi valued `STRING` type

The corresponding JCR node defines a tags properties:

```
<propertyDefinition autoCreated="false" mandatory="false" multiple="true" name="tags" onParentVersion="COPY" pro
  <valueConstraints/>
</propertyDefinition>
```

When a list of values is returned by a getter method, any modification to this list is only visible to this list and does not affect the JCR property values. When the JCR property does not exist, a null value is returned to the caller.

To update the values of a JCR property, the property setter has to be invoked. The list of values is read once and copied to the corresponding JCR property. If the list is null, it simply delete the property.

2.2.2.3. Mixing multi value and single value styles

It can be convenient to map a single valued property to a multi valued property. For instance a multi valued JCR property exposed as a single valued property provides access to the first value of the values.

	JCR single valued	**JCR multi valued**
Java single valued	trivial mapping	access the first element
Java multi valued	a list of size 1	trivial mapping

The same multi valued JCR property can be exposed both as a single and multi valued property. The multi valued property gives access to the complete list of values and the single valued property is useful when the first value needs to be accessed.

2.3. Overview of other mapping styles

Chapter 3. Hierarchical mapping

Chromattic makes the usage of the JCR node hierarchy very natural thanks to relationship mapping. Chromattic defines two mapping styles one-to-many/many-to-one and one-to-one mapping. The one-to-one mapping is useful for accessing the particular child of a node, the one-to-many-many-to-one mapping is useful for accessing residual node definitions defined by a wildcard (*) name.

3.1. One-to-many/many-to-one hierarchical relationship mapping

The usage of Java generics combined with different types of collection provides a flexible mapping. Java generics allows collection filtering based on the type of the collection, it becomes handy when you need to access the a subset of the child nodes filtered with a specific node type (make a chapter on genericity).

Chromattic provides access to the children of node with a Java collection. A bean annotates a collection valued getter with the `@OneToMany` annotation.

```
/**
 * Returns the collection of page children.
 *
 * @return the children
 */
@OneToMany
public abstract Collection<Page> getChildren();
```

The getter method never returns a null value as a node always provides a set of children even if this set is empty. Unlike for multi valued property collection, any modification to this collection will be reflected directly by the underlying JCR node children and vice versa:

- The `add(Page page)` adds a page
- The `remove(Object o)` removes a page
- The `clear()` removes all the page children
- The `iterator()` returns an `iterator` that can be used to remove any child

The other collection methods of the collection class are read methods that won't modify the children and provides various ways to deal with the children.

The `Page` object also provides to its parent with a property annotated with the `ManyToOne` annotation. The getter method returns the object associated to the parent node.

```
/**
 * Returns the page parent.
 *
 * @return the parent
 */
@ManyToOne
public abstract Page getParent();

/**
 * Update the page parent.
 *
 * @param page the parent
 */
public abstract void setParent(Page page);
```

A null value can be obtained in two particular situations:

1. When an object is associated to the root node, indeed the root node is the only node without a parent
2. When an object has a parent of a JCR node type that is not mapped to the Chromattic object returned the getter

It is legal for an object to have several parent accessors when the corresponding JCR node type can have different parent node types. When the various parent types share a common parent class, this class can be used to have a single accessor instead. Ultimately it is possible to use the `java.lang.Object` type that is implicitely mapped to the `nt:base` node typen, the `nt:base` node type is the super type of all JCR node types. (todo: make a section about that somewhere else to clarify)

3.1.1. Adding a child node

There are several ways for adding a child and we are going to examine two of them in this section.

The first way to add a child is to use the collection returned by the parent object. As said earlier, any modification to the collection is directly reflected into the corresponding JCR node.

```
Page child = session.create(Page.class, "bar"); ❶
Collection<Page> children = page.getChildren(); ❷
children.add(child); ❸
assertSame(page, child.getParent()); ❹
```

❶ Create the transient page object
❷ Obtain the children collection from the parent
❸ The child becomes persistent and the bar node is created under the foo node
❹ The parent is set to foo

The second way is to use to add a child is to use the parent setter.

```
Page child = session.create(Page.class, "bar"); ❶
child.setParent(page); ❷
assertTrue(page.getChildren().contains(child)); ❸
```

❶ Create the transient page object
❷ The child becomes persistent and the bar node is created under the foo node
❸ The children collection contains the child

Setting the parent to the child has the same effect than adding the child to the collection. Indeed we can notice in both examples that the when one style is used, we get the same result: the parent getter returns the parent object and the children collection contains the child.

In both case, Chromattic will use the name set on the child before it is inserted in its parent. The session `create` method call takes as second argument the name of the future child. This name is stored temporarily on the create child and is used when the node is effectively inserted.

3.1.2. Destroying a node

We have explained two ways for adding a child to a parent, we will now see that we can use the same methods to destroy a node and its relationship to its parent (indeed in JCR, the only node with no parent is the root node).

When a child is removed from its parent collection, it is removed.

```
    children.remove(child); ❶
    assertFalse(page.getChildren().contains(child)); ❷
```

❶ Removing the child from the collection destroys the child
❷ And the parent does not contain the child anymore

Setting the parent of a Chromattic object to null forces Chromattic to remove the object and the associated node.

```
    child.setParent(null); ❶
    assertFalse(page.getChildren().contains(child)); ❷
```

❶ Setting the parent to null destroys the child
❷ And the parent does not contain the child anymore

3.1.3. Collection types

In our example we have examined the `ManyToOne` side of the relationship based on a `java.util.Collection` interface. Two other type of mapping are available `java.util.List` and `java.util.Map`, let's study what would become our example with such mappings.

3.1.3.1. `java.util.List` mapping

The list mapping must be only used when the corresponding node type has defined its children to be ordered. The list interface adds the notion of order to the collection interface, and using the order oriented method on the list will affect the order of the children.

Example 3.1. Moving a child from the first position to the last position

```
children.add(children.get(0));
```

3.1.3.2. `java.util.Map` mapping

The map interface adds the notion of key which is very useful when the children needs to be accessed by their key. Previously we have seen that when the child is created from the session, its name has to be specifed. When the map interface is used, this is not necessary anymore, as the child name is specified when it is inserted with the `put(String key, Page value)` operation.

Example 3.2. Child insertion

```
Page page = session.create(Page.class);
children.put("foo", page);
```

Example 3.3. Obtaining a particular child

```
Page foo = children.get("foo");
```

Example 3.4. Child removal

```
children.remove("foo");
```

3.2. One-to-one hierarchical relationship mapping

In the Section 3.1, “One-to-many/many-to-one hierarchical relationship mapping ” we explained how to map a node and a its children. One to one hierarchical mapping is about mapping a node and one of its named children thanks to a one-to-one relationship. The most important difference between the two mapping styles is that a one-to-one relationship acts on a precise child defined by its name.

In our example, this type of relationship is used to model the relationship between a website and the root of the page hierarchy of this website. The `WebSite` object is mapped to the *website* node and this node has a child named *rootpage*. The one-to-one relationship between `WebSite` objects and `Page` objects is precisely defined for the *rootpage* child of the *website* node.

Mapping one-to-many/many-to-one hierarchical relationship was only requiring the `@OneToMany` and `@ManyToOne` annotations. One-to-one relationship mapping requires two additional annotations:

- The `@Owner` annotation makes the distinction between the parent and the child of the relationship. The parent object must be annotated with the `@Owner` annotation and the child not.
- The `@MappedBy` annotation provides the name of the node by which the relationship is maintained. It contains a single parameter the is the name of the child.

Example 3.5. The rootPage property

```
/**
 * Returns the root page of the website.
 *
 * @return the root page
 */
@Owner
@OneToOne
@MappedBy("root")
public abstract Page getRootPage();

/**
 * Sets the root page of the website.
 *
 * @param root the root page
 */
public abstract void setRootPage(Page root);
```

Example 3.6. The site property

```
/**
```

```
    * Returns the parent site.
    *
    * @return the parent site
    */
   @OneToOne
   @MappedBy("root")
   public abstract WebSite getSite();
```

/todo explain the dynamic of relationship life cycle

```
    Page root = session.create(Page.class); ❶
    site.setRootPage(root); ❷
    assertEquals(site, root.getSite()); ❸
    session.save();
```

❶ Create the transient page object
❷ The page becomes persistent and the *root* node is inserted under the *site* node
❸ The parent of the root page is the site object

```
    site.setRootPage(null); ❶
```

❶ Setting the root page to null destroys the relationship

Chapter 4. Reference mapping

The hierarchical tree structure supported by JCR is the default way to organize data. JCR provides a reference mechanism for relationship between nodes, a node has a pointer to target node via a property. The relationship is based on two specific property types:

- The `REFERENCE` property type references a target node using its UUID.
- The `PATH` property type references a target node using its path.

The single kind of relationship supported by reference is one-to-many/many-to-one: a node references a target node and a node can be the target of multiple nodes.

Note

Technically it should be possible to support many-to-many relationship using a multivalued reference property. This feature could be implemented in the future.

4.1. One-to-many/many-to-one reference relationship mapping

Mapping single valued reference properties to Chromattic relationship relationship relies on Java collections, in a similar manner hierarchical one-to-many/many-to-one relationship does.

The `@OneToMany` and `@ManyToOne` annotations declares the relationship, however as it is not a hierarchical parent child relationship, the type of the annotation must be set to `RelatonshipType.REFERENCE`.

The `Page` object has a reference to a `Content` object. The `@ManyToOne(type = RelationshipType.REFERENCE)` annotation on the *content* property of `Page` object declares the relationship from the `Page` side.

```
/**
 * Returns the content associated to this page.
 *
 * @return the content
 */
@ManyToOne(type = RelationshipType.REFERENCE)
@MappedBy("content")
public abstract Content getContent();

/**
 * Set thet content on this page
 *
 * @param content the content
 */
public abstract void setContent(Content content);
```

Conversely the `Content` object owns a collection of `Page` objects, each of those having a reference pointing to this object. The `#OneToMany(type = RelationshipType.REFERENCE)` annotation on the *pages* property declares the relationship from the `Content` side. Unlike the one-to-many relationship, the only possible type of collection is the `java.util.Collection` interface because there isn't any notion of order, not name in such relationship.

```
/**
 * Returns all the pages associated with this content.
 *
 * @return the associated pages
 */
@OneToMany(type = RelationshipType.REFERENCE)
@MappedBy("content")
public abstract Collection<Page> getPages();
```

Again here, the relationship between two objects is established when a `Page` object is added to the pages collection of a `Content` object or when a `Content` object is set by invoking the `setContent(Content content)` method on the `Page` object.

4.2. One-to-many/many-to-one path relationship mapping

/todo /todo

Chapter 5. Groovy integration

5.1. Differences with the java version

In the Groovy version of Chromattic, the types are not abstract and annotations can be used directly on properties. The Chromattic engine in the Groovy version is exactly the same than the Java version. Actually Groovy and Java are interroperable

- Groovy and Java Chromattic objects can be used in the same Chromattic application
- A Chromattic application can be used by both Java or Groovy code

Before reading this part, you should already be familliar with Chromattic described in this guide. A simple example of code with the Groovy version of Chromattic : `Page.groovy` (the equivalent of Page.java) in Groovy is

```
package org.chromattic.docs.reference.groovy

import org.chromattic.api.annotations.Name
import org.chromattic.api.annotations.Property
import org.chromattic.api.annotations.PrimaryType

/**
 * @author <a href="mailto:alain.defrance@exoplatform.com">Alain Defrance</a>
 * @version $Revision$
 */
@PrimaryType(name = "gs:page")
class Page {
  /**
   * The page name.
   */
  @Name def String name ❶

  /**
   * The page title.
   */
  @Property(name = "title") def String title ❷

  /**
   * The page content.
   */
  @Property(name = "content") def String content ❸
}
```

❶ The name property is mapped to the node name
❷ The title property is mapped to the title node property
❸ The content property is mapped to the content node property

5.2. Building a Groovy project with Chromattic

Chromattic is plugged to Groovy classes at compile time (this operation is based on AST transformation). So the only thing to do is to have the *chromattic.groovy* jar in the compilation classpath.

5.2.1. Building with Maven

Just add the Maven dependencies in the pom.xml.

```
...
<dependency>
  <groupId>org.chromattic</groupId>
  <artifactId>chromattic.groovy</artifactId>
  <score>compile</score>
</dependency>
...
```

5.2.2. Building with ANT

Add the *chromattic.groovy* jar in the classpath in the *build.xml*.

```
<classpath>
  <pathelement path="${classpath}"/>
  <pathelement location="lib/chromattic.groovy-1.1.0-SNAPSHOT-jar-with-dependencies.jar"/>
</classpath>
```

5.2.3. Compiling with *groovyc*

Just add the *chromattic.groovy* jar in the classpath with the `-classpath` argument

```
groovyc Page.groovy -classpath chromattic.groovy-1.1.0-SNAPSHOT-jar-with-dependencies.jar
```

5.3. Runtime dependencies

To use Chromattic, you should have a JCR implementation in the runtime classpath. For example *chromattic.exo* Maven dependency:

```
<dependency>
  <groupId>org.chromattic</groupId>
  <artifactId>chromattic.exo</artifactId>
  <scope>runtime</scope>
</dependency>
```

5.4. How to access to JCR data through Chromattic objects in Groovy

Simply access to the property content thanks to getter, setter or property :

```
package org.chromattic.docs.reference.groovy

import junit.framework.TestCase
import org.chromattic.api.ChromatticBuilder
import org.chromattic.api.Chromattic
import org.chromattic.api.ChromatticSession
import org.chromattic.docs.reference.groovy.Page
/**
 * @author <a href="mailto:alain.defrance@exoplatform.com">Alain Defrance</a>
 * @version $Revision$
 */
class GroovyTestCase extends TestCase {
  void testGroovy() {
    ChromatticBuilder builder = ChromatticBuilder.create(); ❶
    builder.add(org.chromattic.docs.reference.groovy.Page.class); ❷
    Chromattic chromattic = builder.build(); ❹

    ChromatticSession session = chromattic.openSession(); ❺
```

```
    try
    {
      Page page = session.insert(Page.class, "index"); ❻
      page.setTitle("Hello Page"); ❼
      page.content = "Hello World"; ❽
      session.save(); ❾

      String title = page.title; ❿
      String content = page.getContent(); (11)
    }
    finally
    {
      session.close(); ❸
    }
  }
}
```

❶ Creates the builder object
❷ We add the Page class to the builder object
❸ We must close the session to properly release the session
❹ Now the Chromattic object can be created
❺ Any Chromattic interaction requires to open a session
❻ A new page is inserted under the /index path
❼ Set the title property with setter
❽ Set the content property without setter
❾ Saves the session to persist changes in the repository
❿ Get the title property without getter
Get the title property with getter