ModeShape

Getting Started Guide

2.5.0.Beta2

by Randall M. Hauch, John Verhaeg, Stefano Maestri, and Serge Emmanuel Pagop

What this book covers

The goal of this book is to help you learn about ModeShape and how you can use it in your own applications to get the most out of your JCR repositories.

The first part of the book starts out with an introduction to content repositories and an overview of the JCR API, both of which are important aspects of ModeShape. This is followed by an overview of the ModeShape project, its architecture, and a basic roadmap for what's coming next.

The next part of the book covers how to download and build the examples, how to use ModeShape with existing repositories, and how to build and use custom sequencers.

If you have any questions or comments, please feel free to contact ModeShape's *user mailing list* [mailto:modeshape-users@lists.jboss.org], use the *user forums* [http://community.jboss.org/community/modeshape], or chat with the developers in the *IRC chat room* [http://www.jboss.org/modeshape/chat.html] . If you'd like to get involved on the project, join the *mailing lists* [http://www.jboss.org/modeshape/mailinglists.html] , *download the code* [http://www.jboss.org/modeshape/sourcecode.html] and get it building, and visit our *JIRA issue management system* [http://jira.jboss.org/browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:summary-panel] . If there's something in particular you're interested in, talk with the community - there may be others interested in the same thing.

Introduction

There are a lot of ways for applications to store information persistently so that it can be accessed at a later time and by other processes. The challenge developers face is how to use an approach that most closely matches the needs of their application. This choice becomes more important as developers choose to focus their efforts on application-specific logic, delegating much of the responsibilities for persistence to libraries and frameworks.

Perhaps one of the easiest techniques is to simply store information in *files* . The Java language makes working with files relatively easy, but Java really doesn't provide many bells and whistles. So using files is an easy choice when the information is either not complicated (for example property files), or when users may need to read or change the information outside of the application (for example log files or configuration files). But using files to persist information becomes more difficult as the information becomes more complex, as the volume of it increases, or if it needs to be accessed by multiple processes. For these situations, other techniques often have more benefits.

Another technique built into the Java language is *Java serialization* , which is capable of persisting the state of an object graph so that it can be read back in at a later time. However, Java serialization can quickly become tricky if the classes are changed, and so it's beneficial usually when the information is persisted for a very short period of time. For example, serialization is sometimes used to send an object graph from one process to another. Using serialization for longer-term storage of information is far less useful.

One of the more popular and widely-used persistence technologies is the *relational database*. Relational database management systems have been around for decades and are very capable. The Java Database Connectivity (JDBC) API provides a standard interface for connecting to and interacting with relational databases. However, it is a low-level API that requires a lot of code to use correctly, and it still doesn't abstract away the DBMS-specific SQL grammar. Also, working with relational data in an object-oriented language can feel somewhat unnatural, so many developers map this data to classes that fit much more cleanly into their application. The problem is that manually creating this mapping layer requires a lot of repetitive and non-trivial JDBC code.

Object-relational mapping libraries automate the creation of this mapping layer and result in far less code that is much more maintainable with performance that is often as good as (if not better than) handwritten JDBC code. The *Java Persistence API (JPA)* [http://java.sun.com/developer/technicalArticles/J2EE/jpa/] provide a standard mechanism for defining the mappings (through annotations) and working with these entity objects. Several commercial and open-source libraries implement JPA, and some even offer additional capabilities and features that go beyond JPA. For example, *Hibernate* [http://www.hibernate.org] is one of the most feature-rich JPA implementations and offers object caching, statement caching, extra association mappings, and other features that help to improve performance and usefulness. Plus, Hibernate is open-source (with support offered by *JBoss* [http://www.jboss.com]).

While relational databases and JPA are solutions that work well for many applications, they are more limited in cases when the information structure is highly flexible, the structure is not known *a*

priori, or that structure is subject to frequent change and customization. In these situations, *content repositories* may offer a better choice for persistence. Content repositories offer the storage capabilities of relational databases with the flexibility offered by other systems, such as using files. Content repositories also typically provide other capabilities as well, including hierarchical organization, versioning, indexing, search, access control, transactions, and observation. Content repositories are often used by content management systems (CMS), document management systems (DMS), and other applications that manage electronic files (e.g., documents, images, multi-media, web content, etc.) and metadata associated with them (e.g., author, date, status, security information, etc.). The *Content Repository for Java technology API* [http://www.jcp.org/en/jsr/detail?id=170] provides a standard Java API for working with content repositories. Abbreviated "JCR", this API was developed as part of the Java Community Process under *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] and has been revised under *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283].

The JCR API provides a number of information services that are needed by many applications, including: read and write access to information; the ability to structure information in a hierarchical and flexible manner that can adapt and evolve over time; ability to work with unstructured content; ability to (transparently) handle large strings; notifications of changes in the information; search and query; versioning of information; access control; integrity constraints; participation within distributed transactions; explicit locking of content; and of course persistence.

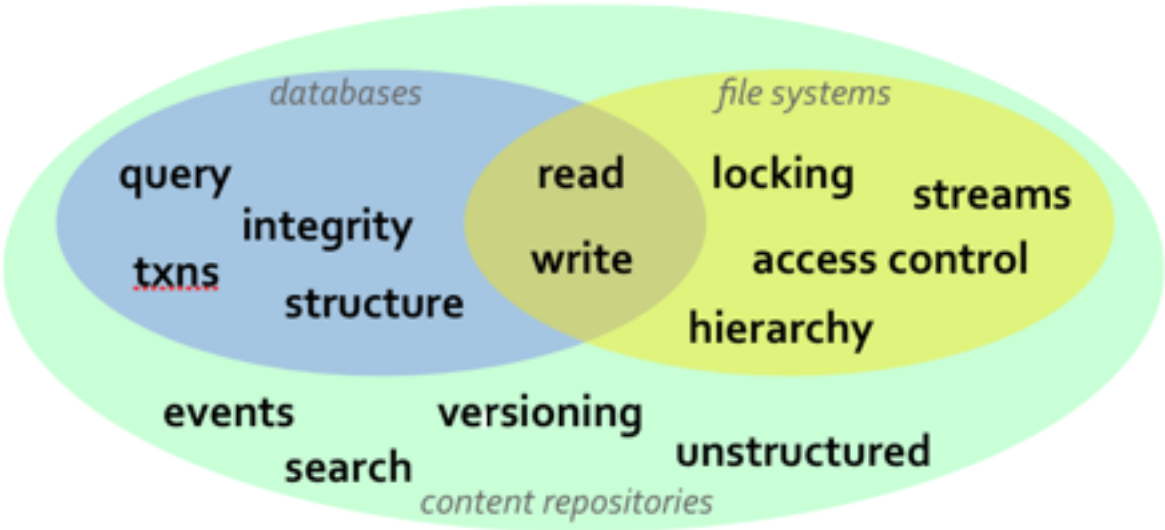

Figure 1.1. JCR API features

1.1. ModeShape

What makes JCR interesting, however, is that a JCR implementation provides all these features and capabilities without exposing where or how that information is stored. While other JCR implementations embed their own persistence technology, JCR becomes really interesting when it is used on top of existing information. *This is in fact the main purpose of ModeShape: provide a JCR implementation that provides access to content stored in many different kinds of systems,*

including the federation of multiple systems. A ModeShape repository isn't yet another silo of information, but rather it's a JCR view of the information you already have in your environment: files systems, databases, other repositories, services, applications, etc. **ModeShape can help you understand the systems and information you already have, through a standard Java API.**

Of course when you start providing a unified view of all this information, you start recognizing the need to store more information, including metadata about and relationships between the existing content. ModeShape lets you do this, too. And ModeShape even tries to help you discover more about the information you already have, especially the information wrapped up in the kinds of files often found in enterprise systems: service definitions, policy files, images, media, documents, presentations, application components, reusable libraries, configuration files, application installations, databases schemas, management scripts, and so on. As files are loaded into the repository, ModeShape can *sequence* these files to extract from their content meaningful information that can be stored in the repository, where your applications can find it by using the standard JCR API to search, access, and use the information.

So, ModeShape *is* a JCR 2.0 implementation that can be used as a traditional self-contained repository. But ModeShape can do so much more. It can automatically sequence files loaded into the repository, making it easier to reuse that information. It also lets your applications use the JCR API to access the content in other systems, and can unify the content from multiple external systems and multiple storage systems to provide a single, federated repository.

1.2. What's next

As we'll see in the *next chapter*, the ability of ModeShape to federate, integrate, and sequence information make ModeShape a powerful asset and tool. Then *Chapter 3* will show that once a ModeShape repository is set up, applications see ModeShape just as another JCR `javax.jcr.Repository` instance and use the standard JCR API to obtain a `javax.jcr.Session` and work with the content.

Chapter 4 walks you through downloading and building the ModeShape examples, while *Chapter 5* and *Chapter 6* will run these very simple examples and walk through their code. *Chapter 7* wraps things up with a discussion about the future of ModeShape and what you can do next to start using ModeShape in your own applications.

ModeShape Use Cases

There are lots of ways to use ModeShape in your own applications, but this chapter attempts to show some representative scenarios that take advantage of ModeShape's support for the JCR API as well as the federation, integration, and sequencing capabilities.

2.1. Service repository

In a SOA environment, one important component is a service registry that provides versioned storage of all the artifacts that describe the services, their capabilities/restrictions, and the policies that surround them. Service repositories contain information that define the services and their message models, ownership, availability, security requirements/abilities, auditing, funding, monitoring, provisioning, provenance, usage, discovery mechanism, configuration, documentation, relationships to other services, classification taxonomies, ontologies, and many other important aspects.

A JCR repository provides an excellent starting point for a service repository. The ability to store a wide range of content, ranging from structured information to documents, means that a JCR repository can offer the flexibility to manage and organize the information while maintaining the ability to adapt the structure and schema as needs evolve over time.

A service repository will contain lots of information represented in different forms, and it's important that the repository make it easy for users to quickly find what they need. Organization of the information (probably in multiple hierarchies and with tags) is important, but more important is the ability for users to use simple searching (or more advanced queries) to return ranked results that match the criteria. For search to be effective, it is important that the repository understand the different kinds of artifacts that are uploaded and the information they contain.

JCR repositories are naturally searchable and queryable, but also can be used to integrate a taxonomy (or folksonomic tags) with the content, allowing the same content to be presented in different hierarchical classifications. But ModeShape capabilities also offer a great advantage, since any file that is uploaded can be automatically sequenced and processed to extract information that's meaningful and useful but often locked up within the file. For example, when a WSDL file is uploaded, the appropriate sequencer(s) process the file and extract and store in the repository the structured information describing the types, message structures, operations, port types, bindings, and services found within the WSDL file. When an XML Schema Document is uploaded, ModeShape can do the same for the schema's complex and simple types, element and attribute declarations, model groups, namespaces, imports, includes, annotations, etc. And ModeShape can do the same for the various policy files, resource declarations, documentation, presentations, ontologies, etc.

Integration with a management system can be done in a similar manner. A ModeShape connector could access the management system to discover the servers and enable auto-discovery of the services, and "tag" the services' deployments with the lifecycle phase (dev, test, production, etc.). Plus, ModeShape sequencers can automatically process the uploaded artifacts to extract

a useful structured representation of their content, and can then store that additional information in the repository. So not only are the original artifacts stored in the repository, but a structured representation of their content is also stored. And all of it can be accessed, navigated, searched, and even updated.

By using ModeShape, a service repository could manage the wide range of artifacts required in a SOA or web-oriented architecture, yet be able to present a unified view of all service information.

2.2. Manage data sources and services

Many enterprise environments include numerous databases and data services, yet there is often no single place where all these different assets are described or related. A data source/service repository could provide information about the many databases running within the enterprise as well as their documentation, schema history, availability, usage policies, current users of the data (including applications, ETL processes, reporting), geographic deployments and synchronization, and the provenance of the data.

Some of this information may actually be defined or controlled within the data sources themselves or within other systems. For example, the DDL scripts used to migrate the database schemas are (hopefully) stored in a version control system, and the databases themselves have the ability to describe their current schemas.

Using ModeShape, the repository could use a connector to the version control system to expose the scripts, as well as connectors to the databases to expose (and cache) the current schema of the databases as structured content. ModeShape sequencers can automatically process the uploaded assets and artifacts to extract a useful structured representation of their content, and can then store that additional information in the repository. Applications can access, search, navigate, and update all of this metadata about the databases and data services, all through a single JCR repository using a standard JCR API and without having to touch the underlying systems.

However, the power of a data repository is really the ability to capture the relationships that otherwise were only captured in people's heads or trapped in documents spread throughout the network. A data repository can capture the policies that dictate how each data source should be used (which are for development purposes, or QA/testing purposes, or which are production, and how are they all related), and it can integrate with management systems to provide information about availability and deployment. As web services are created to provide service-based access to the data in databases, the repository can be used to maintain the relationships between these *data services* and the underlying sources. Similarly, the repository can track how the databases are used by applications, ETL processes, and reports. All of this information is just content that can be stored within the same JCR repository.

2.3. Configuration repository

Many applications and libraries have configuration files that allow the users (or developers) to dictate the setup and behavior. Often this involves multiple files in a specific structure on the file system. Invalid or inopportune changes to these files sometimes corrupt the environment, but creating a more robust configuration management system is often way beyond the desired effort.

An embedded ModeShape repository can provide a more formal and flexible configuration system with little effort. JCR's event system allows the system to be notified when the configuration changes, and versioning can help guarantee the ability to revert back to a previous (valid) configuration. ModeShape connectors can be used to integrate the files on the file system into the configuration system, keeping it natural for those wanting to view and change the configuration via the files. ModeShape sequencers can even process the configuration files to extract a more structured view of the system. And because ModeShape can used with a minimal footprint, it provides the ability to manage and version the configuration with little overhead.

ModeShape can even be used to centralize the configuration definition for a clustered or distributed system. In this mode, the configuration is managed in a central repository that is remotely accessible by the application. When a process is started, it examines the repository and reads the content containing its configuration. The application can monitor the configuration for changes so that it can modify itself and its components. For larger deployments, a central "enterprise configuration" repository can house the configuration of different kinds of systems, and can even be managed and manipulated through JCR.

As we'll see in the *next chapter*, this is actually the way in which ModeShape manages its own configuration. In the embedded case, the configuration repository is simply a local (in-memory) repository that is populated by the configuration file (or programmatic API). In a clustered mode, the repository can be centralized. But either way, to ModeShape the configuration is always defined in a repository.

2.4. What's next

The scenarios described in this chapter are representative of some of the ways in which ModeShape can be used, and hopefully give you ideas about how you can leverage ModeShape in your application or library.

In the *next chapter*, we'll show how you can set up ModeShape and use it via the standard JCR API.

Using ModeShape

Using ModeShape within your application is actually quite straightforward. Simply configure ModeShape with one or more repositories and the sources where the content for those repositories should be accessed and stored. Then, your application just uses the *JCR 2.0 API* [http://www.jcp.org/en/jsr/detail?id=283] to connect to and use those repositories.

Before we dive into how to configure ModeShape, let's start by looking at how your application will find and use the JCR repositories.

3.1. JCR's RepositoryFactory

The latest version of the JCR 2.0 API specification (*JSR-283* [http://www.jcp.org/en/jsr/detail?id=283]) defines a *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] interface that when coupled with the *Java Standard Edition Service Loader mechanism* [http://java.sun.com/javase/6/docs/api/java/util/ServiceLoader.html] lets your application find JCR *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instances using only the JCR API and Java interfaces, without using any implementation-specific interfaces.

ModeShape supports and recommends using this approach, which looks like this:

```
Properties parameters = new Properties();
parameters.load(...)   // typically loaded from property file or set programmatically
Repository repository;

for (RepositoryFactory factory : ServiceLoader.load(RepositoryFactory.class)) {
    repository = factory.getRepository(parameters);
    if (repository != null) break;
}
```

This code looks for all *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] implementations on the classpath (assuming those implementations properly defined the service provider within their JARs), and will ask each to create a repository given the supplied parameters. The first factory that understands these parameters will return a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance, while other factories will return null. The key, then, for defining which JCR *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] implementation your application uses are the parameters passed to the `getRepository(Map)` method. Simply load these from a properties file, and your application is set.

Note

This *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] approach is new to JCR 2.0. With JCR 1.0, your application likely used specific classes from the implementation to instantiate a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] implementation.

Once you've gotten hold of a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance, you can use it to create *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]s. JCR sessions are lightweight, so creating them is very fast. But they are *not* thread safe, so they shouldn't be used concurrently by multiple threads. Therefore, the JCR specification recommends applications create sessions to read, query or change repository content, and then quickly close the sessions:

```
Repository repository = // found earlier
Credentials credentials = ...; // JCR credentials
String workspaceName = ...; // The name of the workspace in the JCR repository
Session session = null;
try {
    // Obtain a JCR Session using simple authentication (or anonymous if configured)
    session = repo.login(credentials,workspaceName);

    // Use the JCR Session to read, query, or change repository content

    // Save any changes that were made ...
    session.save();
} catch (RepositoryException ex) {
    // Handle the error
} finally {
    if (session != null) session.logout();
}
```

JCR sessions are stateful, meaning they cache any information that is accessed to provide a single consistent view of the content, including any transient changes that haven't yet been saved. Thus, in applications with many concurrent sessions changing content, the cached data of a longer-lived session can become inconsistent with the stored content, and must be manually refreshed using the *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]'s `refresh()` method. It is for this reason that the JCR specification recommends using short-lived sessions.

Observing the repository for changes, however, will require registering listeners with a session, and will only receive events while that session is alive. Therefore, observation requires a longer-lived session. But the recommendation is that these longer-lived sessions are used only to register your application's listeners, and not used to read or update content.

These are the basics of writing an application that uses JCR. Next, we'll start looking at the specifics of ModeShape, starting with those *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] properties.

3.1.1. ModeShape's RepositoryFactory Properties

ModeShape's *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] implementation looks for a single property named "`org.modeshape.jcr.URL`". The value of this property is most often a URL pointing to a ModeShape configuration file, which is on the local file system at an absolute path:

```
file://path/to/configFile.xml?repositoryName=MyRepository
```

or a path relative to the running application:

```
file:configFile.xml?repositoryName=MyRepository
```

The configuration file can even be accessed from a web service (e.g., a web server, WebDAV, or version control system) using any resolvable URL, such as:

```
http://www.example.com/path/to/configFile.xml?repositoryName=MyRepository
```

This works great for self-contained applications, because ModeShape will create a new repository engine that runs embedded in the application. However, applications running in platforms (such as servlet containers or Java application servers) will likely prefer that ModeShape runs as a *central service in the platform* that can be shared by multiple applications. In these cases, the ModeShape engine will already be running and registered in JNDI, so the application will use a URL that points to this JNDI location:

```
jndi:name/in/jndi?repositoryName=MyRepository
```

Here's an example of a property file containing the single ModeShape property for *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html]:

```
# This URL use the repository named 'MyRepository' defined in the 'modeshape-configuration.xml'
 file
# located in the current directory. Use a different URL as needed.
#
org.modeshape.jcr.URL = file:modeshape-configuration.xml?repositoryName=MyRepository
```

In the *next section*, we'll take an introductory look at what these configuration files look.

3.2. ModeShape Configuration Files

The previous section showed how easy it was to obtain a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] and *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] using the standard JCR API. This section provides an introduction to ModeShape configuration files, although you will likely want to look at the *Reference Guide* [http://docs.jboss.org/modeshape/2.5.0.Beta2/manuals/reference/html/index.html] for more detail.

Each configuration file defines the components that are used to create the repository:

- **`Repository sources`** are the POJO objects that each describe a particular location where content is stored. Each repository source object is an instance of a ModeShape connector, and is configured with the properties that particular source. ModeShape's *RepositorySource* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html] classes are analogous to JDBC's `DataSource` [http://java.sun.com/javase/6/docs/api/javax/sql/DataSource.html] classes - they are implemented by specific connectors (aka, "drivers") for specific kinds of repository sources (aka, "databases"). Similarly, a *RepositorySource* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html] instance is analogous to a `DataSource` [http://java.sun.com/javase/6/docs/api/javax/sql/DataSource.html] instance, with bean properties for each configurable parameter. Therefore, each repository source definition must supply the name of the *RepositorySource* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html] class, any bean properties, and, optionally, the classpath that should be used to load the class.
- **`Repositories`** define the JCR repositories that are available. Each repository has a unique name that is used to obtain the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance, but each repository definition also can include the predefined namespaces (other than those automatically defined by ModeShape), various options, and the node types that are to be available in the repository without explicit registration through the JCR API.
- **`Sequencers`** define the particular sequencers that are available for use. Each sequencer definition provides the path expressions governing which nodes in the repository should be sequenced when those nodes change, and where the resulting output generated by the

sequencer should be placed. The definition also must state the name of the sequencer class, any bean properties and, optionally, the classpath that should be used to load the class.

- **`MIME type detectors`** define the particular MIME type detector(s) that should be made available. A MIME type detector does exactly what the name implies: it attempts to determine the MIME type given a "filename" and contents. ModeShape automatically uses a detector that uses the file extension to identify the MIME type, but also provides an implementation that uses an external library to identify the MIME type based upon the contents. The definition must state the name of the detector class, any bean properties and, optionally, the classpath that should be used to load the class.

3.2.1. Example configuration file

Here is the configuration file that is used in the repository example, though it has been simplified a bit and most comments have been removed for clarity):

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns:mode="http://www.modeshape.org/1.0" xmlns:jcr="http://www.jcp.org/jcr/1.0">
    <!--
    Define the JCR repositories
    -->
    <mode:repositories>
      <!--
      Define a JCR repository that accesses the 'Cars' source directly.
      -->
      <mode:repository jcr:name="car repository" mode:source="Cars">
        <mode:options jcr:primaryType="mode:options">
         <systemSourceName jcr:primaryType="mode:option" mode:value="system@Cars"/>
           <jaasLoginConfigName jcr:primaryType="mode:option" mode:value="modeshape-jcr"/>
           <!--
           As a convenience, ModeShape defaults to granting guest users full access.
         In a production system, you would want to limit this access by uncommenting one of the
           options below:

           for no access:
           <anonymousUserRoles jcr:PrimaryType="mode:option" mode:value="" />

           for read-only acces:
           <anonymousUserRoles jcr:PrimaryType="mode:option" mode:value="readonly" />
           -->
        </mode:options>
      </mode:repository>
    </mode:repositories>
```

```
    <!--
   Define the sources for the content.  These sources are directly accessible using the ModeShape-
specific
    Graph API.
    -->
    <mode:sources jcr:primaryType="nt:unstructured">
      <mode:source jcr:name="Cars"
                   mode:classname="org.modeshape.graph.connector.inmemory.InMemoryRepositorySource"
                   mode:retryLimit="3" mode:defaultWorkspaceName="workspace1">
        <predefinedWorkspaceNames>system</predefinedWorkspaceNames>
      </mode:source>
    </mode:sources>
    <!--
    Define the clustering configuration. This is an optional section; leave it out when
    running in a non-clustered (single-process) mode.
    -->
                <mode:clustering    clusterName="modeshape-cluster"    configuration="jgroups-
modeshape.xml" />
    <!--
    Define the sequencers. This is an optional section.
    -->
    <mode:sequencers>
      <mode:sequencer jcr:name="Image Sequencer"
                   mode:classname="org.modeshape.sequencer.image.ImageMetadataSequencer">
        <mode:description>Image metadata sequencer</mode:description>
        <mode:pathExpression>/foo/source => /foo/target</mode:pathExpression>
        <mode:pathExpression>/bar/source => /bar/target</mode:pathExpression>
      </mode:sequencer>
    </mode:sequencers>
    <!--
     Define how ModeShape will determine the MIME type of files. This is an optional section;
     if you do not specify a MIME type detector, ModeShape will use a built-in one that is based
     filename extensions for most commonly-used files.
    -->
    <mode:mimeTypeDetectors>
      <mode:mimeTypeDetector jcr:name="Detector"
            mode:description="Standard extension-based MIME type detector"/>
    </mode:mimeTypeDetectors>
</configuration>
```

3.3. Using ModeShape in Web Applications

Sometimes your applications can simply define a configuration file and use the *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] to access its repositories. This is very straightforward, and this is useful for many simple applications because the application will then own the ModeShape instance(s).

Web applications are a different story. Often, you would rather your web application not contain the code that initializes the JCR repository, but instead configure ModeShape as a central, shared service that all of your web applications can simply reference and use.

Unfortunately, there's no common way to deploy ModeShape into the various web or application servers, since they all have slightly different deployment and configuration techniques. The remainder of this section will talk about how to deploy ModeShape to two popular open source servers, the *JBoss Application Server* and *Apache Tomcat*.

3.3.1. Deploying ModeShape to JBoss AS

The *JBoss Application Server* [http://jboss.org/jbossas] (or JBoss AS) is a very popular open source Java application server, with an extremely healthy and active community. ModeShape offers a way to deploy ModeShape into JBoss AS as as a central, shared service that can be monitored and administered using the embedded console.

ModeShape provides a downloadable ZIP file that can be unzipped into any JBoss AS profile. When you do this, that profile will contain all the files necessary for ModeShape to run when the server is started. The default configuration is for a single, in-memory repository with two users. However, other than basic playing, you will want to edit the configuration files to define a more robust, persistent and secure configuration.

This JBoss AS distribution ZIP file contains several components:

- JAR files for the JCR 2.0 API and ModeShape's small extensions to the JCR API on the global classpath (that is, in the "`lib/`" directory). These APIs are available to all deployed applications, services and components. The JCR API contains the "`javax.jcr`" packages and has no other dependencies. ModeShape's extensions define interfaces in the "`org.modeshape.jcr.api`" packages; these extend a few of the standard JCR API interfaces and add several methods to make them more useful.
- The ModeShape Service, represented as an exploded JAR file in the "`deploy`" directory. This is where the *`JcrEngine`* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] is running, though any application (or other JBoss service) can access its JCR Repository instances using the standard *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] approach described *earlier* by using a URL such as:

```
jndi:jcr/local?repositoryName=repository
```

By default, there is a single in-memory repository named "repository", but this can be changed by simply editing the "`deploy/modeshape-services.jar/managedConfigRepository.xml`" configuration file. All of ModeShape's standard sequencers and connectors (and JARs for their dependencies) are included, meaning they can be configured for use without worrying about adding JARs to the classpath. Feel free to remove any of the JARs are not needed for your custom configuration.

- A pair of JAAS properties files, located in the "`conf/props/`" directory, that come out of the box with an "admin" user (with password "admin") that has full read, write, and administration privileges, and a "guest" user (with password "guest") that has only read and write privileges. Simply edit these files to change users, passwords, and roles, or to configure JAAS differently.
- The ModeShape RESTful API, represented as an exploded WAR file in the "`deploy`" directory. This allows remote applications to interact with ModeShape to access and manipulate repository content using a RESTful API that uses JSON in the requests and responses. All ModeShape repositories can be accessed, and authentication is done using the ModeShape JAAS configuration.
- The ModeShape WebDAV API, represented as an exploded WAR file in the "`deploy`" directory. This web application allows external clients to access and manipulate the content in the ModeShape repositories using the standard WebDAV protocol. For example, you can mount a repository (or parts of it) as a network drive on most operating systems, and then upload or download files and folders using standard OS operations and graphical tools. All ModeShape repositories can be accessed, and authentication is done using the ModeShape JAAS configuration.
- A plugin for the embedded JBoss AS console, represented as a WAR file in the "deploy" directory. This plugin also works with *RHQ* [http://support.rhq-project.org/display/RHQ/Home]/*JOPR* [http://jboss.org/jopr] administration, monitoring, alerting, operational control and configuration system. **This feature is currently incomplete, but is undergoing active development.**
- A JDBC driver for querying the repositories through JDBC. This driver is on the global classpath so it can be used in any deployed component. A single JDBC DataSource is also configured in the "`deploy/modeshape-services.jar/modeshape-jdbc-ds.xml`" file to use the single default in-memory repository available out of the box. Simply edit this file to add or change the DataSource definitions.

Here are the contents of this file:

```
conf/
conf/props/
conf/props/modeshape-roles.properties
conf/props/modeshape-users.properties
lib/
```

```
lib/jcr-2.0.jar
lib/modeshape-jcr-api-2.5.0.Beta2.jar
lib/modeshape-jdbc-2.5.0.Beta2.jar
deploy/
deploy/modeshape-jboss-beans.xml
deploy/modeshape-services.jar/
deploy/modeshape-services.jar/META-INF/
deploy/modeshape-services.jar/aperture-1.1.0.Beta1.jar
deploy/modeshape-services.jar/joda-time-1.6.jar
deploy/modeshape-services.jar/lucene-analyzers-3.0.2.jar
deploy/modeshape-services.jar/lucene-core-3.0.2.jar
deploy/modeshape-services.jar/lucene-regex-3.0.2.jar
deploy/modeshape-services.jar/lucene-snowball-3.0.2.jar
deploy/modeshape-services.jar/lucene-misc-3.0.2.jar
deploy/modeshape-services.jar/poi-3.6.jar
deploy/modeshape-services.jar/poi-scratchpad-3.6.jar
deploy/modeshape-services.jar/managedConfigRepository.xml
deploy/modeshape-services.jar/rdf2go.api-4.6.2.jar
deploy/modeshape-services.jar/META-INF/jboss-beans.xml
deploy/modeshape-services.jar/modeshape-cnd-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-common-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-connector-filesystem-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-connector-infinispan-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-connector-jbosscache-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-connector-jcr-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-connector-jdbc-metadata-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-connector-store-jpa-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-connector-svn-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-graph-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-jbossas-service-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-jcr-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-jdbc-ds.xml
deploy/modeshape-services.jar/modeshape-mimetype-detector-aperture-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-repository-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-search-lucene-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-sequencer-classfile-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-sequencer-cnd-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-sequencer-ddl-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-sequencer-java-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-sequencer-jbpm-jpdl-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-sequencer-msoffice-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-sequencer-teiid-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-sequencer-text-2.5.0.Beta2.jar
deploy/modeshape-services.jar/modeshape-sequencer-xml-2.5.0.Beta2.jar
```

```
deploy/modeshape-services.jar/modeshape-sequencer-zip-2.5.0.Beta2.jar
deploy/modeshape-rest.war/
deploy/modeshape-rest.war/META-INF/
deploy/modeshape-rest.war/WEB-INF/
deploy/modeshape-rest.war/WEB-INF/lib/
deploy/modeshape-rest.war/META-INF/MANIFEST.MF
deploy/modeshape-rest.war/WEB-INF/jboss-web.xml
deploy/modeshape-rest.war/WEB-INF/lib/jaxrs-api-1.2.1.GA.jar
deploy/modeshape-rest.war/WEB-INF/lib/jettison-1.1.jar
deploy/modeshape-rest.war/WEB-INF/lib/modeshape-jcr-2.5.0.Beta2.jar
deploy/modeshape-rest.war/WEB-INF/lib/modeshape-web-jcr-2.5.0.Beta2.jar
deploy/modeshape-rest.war/WEB-INF/lib/modeshape-web-jcr-rest-2.5.0.Beta2.jar
deploy/modeshape-rest.war/WEB-INF/lib/resteasy-jaxb-provider-1.2.1.GA.jar
deploy/modeshape-rest.war/WEB-INF/lib/resteasy-jaxrs-1.2.1.GA.jar
deploy/modeshape-rest.war/WEB-INF/lib/resteasy-jettison-provider-1.2.1.GA.jar
deploy/modeshape-rest.war/WEB-INF/lib/scannotation-1.0.2.jar
deploy/modeshape-rest.war/WEB-INF/web.xml
deploy/modeshape-webdav.war/
deploy/modeshape-webdav.war/WEB-INF/
deploy/modeshape-webdav.war/WEB-INF/lib/
deploy/modeshape-webdav.war/WEB-INF/jboss-web.xml
deploy/modeshape-webdav.war/WEB-INF/lib/aperture-1.1.0.Beta1.jar
deploy/modeshape-webdav.war/WEB-INF/lib/modeshape-jcr-2.5.0.Beta2.jar
deploy/modeshape-webdav.war/WEB-INF/lib/modeshape-mimetype-detector-
aperture-2.5.0.Beta2.jar
deploy/modeshape-webdav.war/WEB-INF/lib/modeshape-web-jcr-2.5.0.Beta2.jar
deploy/modeshape-webdav.war/WEB-INF/lib/modeshape-web-jcr-webdav-2.5.0.Beta2.jar
deploy/modeshape-webdav.war/WEB-INF/lib/webdav-servlet-2.0.jar
deploy/modeshape-webdav.war/WEB-INF/web.xml
deploy/admin-console.war/
deploy/admin-console.war/plugins/
deploy/admin-console.war/plugins/modeshape-jbossas-console-2.5.0.Beta2.jar
```

Your web application or JBoss service can use one of the JCR *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instances running inside the ModeShape service by simply using the *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] technique described *earlier*, with a URL such as:

```
jndi:jcr/local?repositoryName=repository
```

Be sure to use the correct repository name.

Since the JCR API JAR is on the global classpath, your web application can use the JCR API without having to include the JAR file in your application's WAR file. In fact, your application will likely get ClassCastExceptions if it does include the JCR API in its WAR file. Plus, if needed, your application can use ModeShape's "`org.modeshape.jcr.api`" extensions to the JCR API (again, on the global classpath), and should not need or use any of the classes or interfaces in the ModeShape implementation.

3.3.2. Deploying ModeShape to Tomcat

Each kind of web server or application server is different, but all servlet containers do provide a way of configuring objects and placing them into JNDI. ModeShape provides a *`JndiRepositoryFactory`* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/jcr/JcrRepository.html] class that implements and that can be used in the server's configuration. The *`JndiRepositoryFactory`* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/jcr/JcrRepository.html] requires two properties:

- **`configFile`** is the path to the *configuration file* resource, which must be available on the classpath
- **`repositoryName`** is the name of a JCR repository that exists in the JCR configuration and that will be made available by this JNDI entry

Here's an example of a fragment of the `conf/context.xml` for Tomcat:

```
<Resource name="jcr/local"
        auth="Container"
        type="javax.jcr.Repository"
        factory="org.modeshape.jcr.JndiRepositoryFactory"
        configFile="/resource/path/to/configuration.xml"
        repositoryName="Test Repository Source" />
```

Note that it is possible to have multiple `Resource` entries. The *`JndiRepositoryFactory`* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/jcr/JcrRepository.html] ensures that only one *`JcrEngine`* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] is instantiated, but that a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance is registered for each entry.

Before the server can start, however, all of the ModeShape jars need to be placed on the classpath for the server. JAAS also needs to be configured, and this can be done using the application server's configuration or in your web application if you're using a simple servlet container. For more details, see the *Reference Guide* [http://docs.jboss.org/modeshape/2.5.0.Beta2/manuals/reference/html/index.html].

Note

The ModeShape community has solicited input on how we can make it easier to consume and use ModeShape in applications that do not use Maven. Check out the *discussion thread* [http://community.jboss.org/thread/146589], and please add any suggestions or opinions!

Then, your web application needs to reference the `Resource` and state its requirements in its `web.xml`:

```
<resource-env-ref>
  <description>Repository</description>
  <resource-env-ref-name>jcr/local</resource-env-ref-name>
  <resource-env-ref-type>javax.jcr.Repository</resource-env-ref-type>
</resource-env-ref>
```

Note that the value of `resource-env-ref-name` matches the value of the name attribute on the `<Resource>` tag in the `context.xml` described above. This is a must.

At this point, your web application can perform the lookup of the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] object by using JNDI directly (or the more standard *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] technique shown earlier), create and use a *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html], and then close the *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]. Here's an example of a JSP page that does this:

```
<%@ page import="javax.naming.*, javax.jcr.*,
 org.jboss.security.config.IDTrustConfiguration" %>
<%!

static {
   // Initialize IDTrust
   IDTrustConfiguration idtrustConfig = new IDTrustConfiguration();
   try {
      idtrustConfig.config("security/jaas.conf.xml");
   } catch (Exception ex) {
      throw new IllegalStateException(ex);
   }
}
%>
```

```
<%
Session sess = null;
try {
    InitialContext initCtx = new InitialContext();
    Context envCtx = (Context) initCtx.lookup("java:comp/env");
    Repository repo = (Repository) envCtx.lookup("jcr/local");
    sess = repo.login(new SimpleCredentials("readwrite", "readwrite".toCharArray()));

    // Do something interesting with the Session ...
    out.println(sess.getRootNode().getPrimaryNodeType().getName());
} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    if (sess != null) sess.logout();
}
%>
```

Since this uses a servlet container, there is no JAAS implementation configured, so note the loading of IDTrust to create the JAAS realm. (To make this work in Tomcat, the security folder that contains the `jaas.conf.xml`, `users.properties`, and `roles.properties` needs to be moved into the `%CATALINA_HOME%` directory.)

Note

If you deploy your application to JBoss AS or EAP and deploy *ModeShape as a service*, your application doesn't have to do anything with JAAS, since that's provided by the platform.

3.4. Setting the Classpath

Deploying ModeShape as a service in *JBoss AS* is all set up with the correct classpaths and configurations. In other deployments, you'll have to ensure that all of the ModeShape JARs are available on the appropriate classpath. This section describes two different scenarios for doing this: Maven-based, and using JARs with the traditional classpath.

3.4.1. Building against ModeShape via Maven

By far the easiest way to use ModeShape is to use Maven, because with just a few lines of code, Maven will automatically pull all the JARs and source for all of the ModeShape libraries as well as everything those libraries need. All of ModeShape's artifacts for each release are published in the new *JBoss Maven repository* [https://repository.jboss.org/nexus/] under the "*org.modeshape* [https://repository.jboss.org/nexus/content/repositories/public/org/modeshape/]" group ID.

3.4.1.1. Using the JBoss Maven repository

The JBoss Maven repository not only contains all of the artifacts for ModeShape and other open source projects hosted at *JBoss.org* [http://www.jboss.org], but it also *proxies quite a few other repositories* [http://community.jboss.org/wiki/MavenRepository] that contain many other third-party libraries.

So if you're using Maven (or Ivy), first make sure your project knows about this new JBoss Maven repository. One way to do this is to add the following to your project POM (you'll still likely want to use other Maven repositories for third-party artifacts):

```
<repositories>
  <repository>
    <id>jboss</id>
    <url>http://repository.jboss.org/nexus/content/groups/public/</url>
  </repository>
</repositories>
```

Or, you can add this information to your `~/.m2/settings.xml` file. For more information, see the *JBoss wiki page* [http://community.jboss.org/wiki/MavenGettingStarted-Developers].

3.4.1.2. Add dependency to ModeShape

Then, simply modify your project's POM by adding dependencies on the ModeShape JCR library:

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-jcr</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
```

This adds only the minimal libraries required to use ModeShape. If your application is going to use clustering, you'll need to also depend upon the clustering module:

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-clustering</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
```

You also need to add dependencies for each of the connectors and sequencers you want to use. Here is the list of available sequencers:

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-cnd</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-ddl</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshapce</groupId>
  <artifactId>modeshape-sequencer-images</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-classfile</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-java</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-mp3</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-msoffice</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-xml</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-teiid</artifactId>
```

```
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-text</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-zip</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
```

Here is the list of available connectors:

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-filesystem</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-infinispan</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-jcr</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-jbosscache</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-jdbc-metadata</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-store-jpa</artifactId>
```

```
  <version>2.5.0.Beta2</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-svn</artifactId>
  <version>2.5.0.Beta2</version>
</dependency>
```

The sequencer and connector libraries you choose, plus every third-party library they need, will be pulled in automatically by Maven into your project.

3.4.2. Add dependencies for logging

ModeShape is designed to use the same logging framework as your application, and it uses SLF4J to accomplish this. In other words, ModeShape depends upon the SLF4J API library, but requires you to provide provide a logging implementation as well as the appropriate SLF4J binding JAR.

For example, if your application is using *Log4J* [http://logging.apache.org/log4j/], your application will already have a dependency for it, and so ModeShape log messages will be sent to the same logging system used in your application, you need to add a dependency to the SLF4J-to-Log4J binding JAR:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.5.11</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.16</version>
</dependency>
```

Of course, SLF4J works with other logging frameworks, too. Some logging implementations (such as *LogBack* [http://logback.qos.ch/]) implement the SLF4J API natively, meaning they require no binding JAR. For details on the options and how to configure them, see the *SLF4J manual* [http://www.slf4j.org/manual.html].

3.4.3. Building against ModeShape via JARs

If your application doesn't use Maven, you'll need to obtain the ModeShape JARs and place them onto your application's classpath. ModeShape provides a *single download* [http://downloads.jboss.org/modeshape/2.5.0.Beta2/modeshape-2.5.0.Beta2-all-with-

dependencies.jar] with all of the JARs for all ModeShape components and all dependencies. This file contains the following:

- **`modeshape-jcr-2.5.0.Beta2-jar-with-dependencies.jar`** contains all of the classes (except those under `javax.jcr`) necessary to run the core ModeShape JCR repository engine using the in-memory connector and the federating connector;
- one **`modeshape-connector-<type>-2.5.0.Beta2-jar-with-dependencies.jar`** for each type of connector, each containing all of the classes necessary for that connector, designed to be added to the classpath after the `modeshape-jcr-2.5.0.Beta2-jar-with-dependencies.jar` file;
- one **`modeshape-sequencer-<type>-2.5.0.Beta2-jar-with-dependencies.jar`** for each type of connector, each containing all of the classes necessary for that sequencer, designed to be added to the classpath after the `modeshape-jcr-2.5.0.Beta2-jar-with-dependencies.jar` file;
- **`modeshape-mimetype-detector-aperture-2.5.0.Beta2-jar-with-dependencies.jar`** containing all of the classes necessary for detecting the MIME type of files based upon their name and/or content, designed to be added to the classpath after the `modeshape-jcr-2.5.0.Beta2-jar-with-dependencies.jar` file;
- **`modeshape-jpa-ddl-gen-2.5.0.Beta2-jar-with-dependencies.jar`** contains all of the classes required to run the DDL generation utility as a standalone application.

Note that the core engine is required in all configurations. The `jcr-2.0.jar` file is not included and must be provided by you. And, as mentioned in the *previous section*, ModeShape uses SLF4J for logging and you must provide a logging implementation as well as the appropriate SLF4J binding JAR.

3.5. What's next

This chapter outline how you configure ModeShape, how you then access a `javax.jcr.Repository` instance, and use the standard JCR API to interact with the repository. The *next chapter* walks you through downloading and building the ModeShape examples, while *Chapter 5* and *Chapter 6* shows how to run the examples.

Building the example applications

This chapter provides instructions for downloading and compiling two sample applications that demonstrates how ModeShape works with a JCR repository to automatically sequence changing content to extract useful information. So read on to get the simple application running.

ModeShape uses Maven 2 for its build system, as do these examples. Using Maven 2 has several advantages, including the ability to manage dependencies. If a library is needed, Maven automatically finds and downloads that library, plus everything that library needs. This means that it's very easy to build the examples - or even create a maven project that depends on the ModeShape JARs.

To use Maven with ModeShape, you'll need to have *JDK 6* [http://java.sun.com/javase/downloads/index_jdk5.jsp] and Maven 2.0.9 (or higher).

Note

Maven can be downloaded from *http://maven.apache.org/*, and is installed by unzipping the `maven-2.0.9-bin.zip` file to a convenient location on your local disk. Simply add `$MAVEN_HOME/bin` to your path.

The examples are already configured to use the new *JBoss.org Maven repository* [https://repository.jboss.org/nexus], which provides a central location for the artifacts produced by the JBoss.org projects (well, at least those that use Maven) as well as proxying other repositories and caching artifacts for third party libraries. This simplifies the builds, helps ensure that developers have easy access to these artifacts (including sources) so that the project (and dependencies) can always be rebuilt when needed.

For more information about the JBoss Maven repository, see the *announcement* [http://community.jboss.org/en/build/blog/2010/04/20/announcement--new-maven-repository-infrastructure] and *documentation* [http://community.jboss.org/wiki/MavenRepository].

Previous versions of ModeShape made use of the older JBoss.org Maven repository, and required modifying your local `~/.m2/settings.xml` file. This is no longer required.

4.1. Downloading and compiling

The next step is to *download* [http://downloads.jboss.org/modeshape/2.5.0.Beta2/modeshape-2.5.0.Beta2-gettingstarted-examples.zip] the example for this Getting Started guide, and extract the contents to a convenient location on your local disk. You'll find the example contains the following files, which are organized according to the standard Maven directory structure:

```
examples/pom.xml
        sequencers/pom.xml
                /src/main/assembly
                        /config
                        /java
                        /resources
                    /test/java
                        /resources
        repository/pom.xml
                /src/main/assembly
                        /config
                        /java
                        /resources
                    /test/java
                        /resources
```

There are essentially three Maven projects: a `sequencers` project, a `repository` project, and a parent project. All of the source for the sequencing example is located in the `sequencers` subdirectory, while all of the source for the repository example is located in the `repository` subdirectory.

And you may have noticed that none of the ModeShape libraries are there. This is where Maven comes in. The two `pom.xml` files tell Maven everything it needs to know about what libraries are required and how to build the example.

In a terminal, go to the `examples` directory and run:

```
$ mvn install
```

This command downloads all of the JARs necessary to compile and build the example, including the ModeShape libraries, the libraries they depend on, and any missing Maven components. (These are downloaded from the JBoss repositories only once and saved on your machine. This means that the next time you run Maven, all the libraries will already be available locally, and the build will run much faster.) The command then continues by compiling the example's source code (and unit tests) and running the unit tests. The build is successful if you see the following:

```
$ mvn install
...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
```

```
[INFO] ------------------------------------------------------------------------
[INFO] Getting Started examples ..... SUCCESS [2.106s]
[INFO] Sequencer Examples ..... SUCCESS [9.768s]
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 12 seconds
[INFO] Finished at: Wed May 07 12:00:06 CDT 2008
[INFO] Final Memory: 14M/28M
[INFO] ------------------------------------------------------------------------
$
```

If there are errors, check whether you have the correct version of Maven installed and that you've correctly updated your Maven settings as described above.

If you've successfully built the examples, there will be a new `examples/sequencers/target/` directory that contains all of the generated output for the sequencers example, including a `modeshape-example-sequencers-basic.dir/` subdirectory that contains the following:

- **`run.sh`** is the *nix shell script that will run the sequencer example application.
- **`log4j.properties`** is the Log4J configuration file.
- **`sample1.mp3`** is a sample MP3 audio file you'll use later to upload into the repository.
- **`caution.gif`**, **`caution.png`**, **`caution.jpg`**, and **`caution.pict`** are images that you'll use later and upload into the repository.
- **`sequencing.cnd`** is a Compact Node Definition (CND) file that defines the node types used in the output from the sequencers.
- **`security`** subdirectory containing several files related to the JAAS implementation used for authentication.
- **`project1`** subdirectory contains some Java source that can be loaded into the repository.
- **`lib`** subdirectory contains the JARs for all of the ModeShape artifacts as well as those for other libraries required by ModeShape and the sequencer example.

Similarly, the `examples/repository/target/` directory contains all of the generated output for the repository example, including a `modeshape-example-repository-basic.dir/` subdirectory that contains the following:

- **`run.sh`** is the *nix shell script that will run the repository example application.
- **`run.cmd`** is the Windows command file that will run the repository example application.

- **`log4j.properties`** is the Log4J configuration file.
- **`configRepository.xml`** is an XML file containing the information that the example application loads as its configuration and which defines the sources, repositories, sequencers (if used), and other components that make up the ModeShape JCR engine.
- **`aircraft.xml`** is an XML file containing the information that the example application imports into its "Aircraft" repository.
- **`cars.xml`** is an XML file containing the information that the example application imports into its "Cars" repository.
- **`ufoSource`** subdirectory containing several folders and files used by the file system connector for the "UFOs" repository.
- **`aircraft.cnd`**, **`cars.cnd`**, and **`vehicles.cnd`** are the CND files used for the three different JCR Repositories set up in the example. The **`vehicles.cnd`** is just a combination of the other two (with duplicates removed). The UFO source doesn't need a CND file, since the file system connector uses the "nt:file" and "nt:folder" node types built into the JCR standard.
- **`security`** subdirectory containing several files related to the JAAS implementation used for authentication and authorization.
- **`lib`** subdirectory contains the JARs for all of the ModeShape artifacts as well as those for other libraries required by ModeShape and the repository example. There are a lot of libraries here, but almost all of them are from the JPA connector (which depends upon Hibernate), HSQLDB, Lucene, and the JAAS implementation.

4.2. What's next

In this chapter you downloaded, installed, and built the two example applications. In the next *two chapters* we'll run these examples and walk through the code.

The Sequencer Example

The previous chapter walked through the process of downloading and building the examples. This chapter will focus on the sequencer example, showing how to run the example and then walking through the code to describe what it's doing.

5.1. Running the sequencing example

The sequencing example consists of a client application that sets up an in-memory JCR repository and that allows a user to upload files into that repository. The client also sets up the ModeShape services with seven sequencers:

- an image metadata sequencer that processes PNG, JPEG, GIF, BMP or other image filetypes to extract the image's metadata (e.g., image format, physical size, pixel density, etc.)
- an MP3 sequencer that extracts the ID3 metadata (e.g., the author, title, album, year and comment)
- a Java source code sequencer that extracts the structure of Java classes by parsing the source code
- a Java class file sequencer that extracts the structure of Java classes by analyzing the class files
- a text file sequencer that extracts the comma-separated structured information contained in CSV files
- a text file sequencer that extracts the structured information contained in fixed-width files
- a ZIP archive sequencer that extracts the files and directory structure contained in ZIP and JAR files

These sequencers automatically extract content from the files and store that content in the repository.

Note

ModeShape includes several other sequencers, including sequencers for DDL files, Microsoft Office files, JCR Compact Node Definition (CND) files, and XML files. Feel free to experiment with the example and add these or even sequencers you write.

To run the client application, go to the `examples/sequencers/target/modeshape-example-sequencers-basic.dir/` directory and type `./run.sh`. You should see the command-line client and its menus in your terminal:

```
Starting repository and sequencing service ... done.
Starting sequencing service ... done.

---------------------------------
Menu:

u) Upload a file to the repository
s) Search the repository using extracted metadata

d) Display statistics

?) Show this menu
q) Quit
>
```

Figure 5.1. Example client

From this menu, you can upload a file into the repository, search for media in the repository, print sequencing statistics, or quit the application.

The first step is to upload one of the example images. If you type 'u' and press return, you'll be prompted to supply the path to the file you want to upload. Since the application is running from within the `examples/sequencers/target/modeshape-example-sequencers-basic.dir/` directory, you can specify any of the files in that directory without specifying the path:

```
---------------------------------
Menu:

u) Upload a file to the repository
s) Search the repository using extracted metadata

d) Display statistics

?) Show this menu
q) Quit
>u
Please enter the file to upload:
caution.png
Please enter the repository path where the file should be placed [/a/b/caution.png]:
/a/b/c/caution.png
>_
```

Figure 5.2. Uploading an image using the example client

You can specify any fully-qualified or relative path. The application will notify you if it cannot find the file you specified. The example client configures ModeShape to sequence MP3 audio files, Java source files, Java class files, ZIP files, text files (`.txt`), CSV files (`.csv`), or image files with one of the following extensions (technically, nodes that have names ending in the following): `jpg`, `jpeg`, `gif`, `bmp`, `pcx`, `png`, `iff`, `ras`, `pbm`, `pgm`, `ppm`, and `psd`. Files with other extensions in the repository path will be ignored. For your convenience, the example provides several files that will be sequenced (`caution.png`, `caution.jpg`, `caution.gif`, and `sample1.mp3`) and one image that will not be sequenced (`caution.pict`). Feel free to try other files.

After you have specified the file you want to upload, the example application asks you where in the repository you'd like to place the file. (If you want to use the suggested location, just press `return`.)

The client application uses the JCR API to upload the file to that location in the repository, creating any nodes (of type `nt:folder`) for any directories that don't exist, and creating a node (of type `nt:file`) for the file. And, per the JCR specification, the application creates a `jcr:content` node (of type `nt:resource`) under the file node. The file contents are placed on this `jcr:content` node in the `jcr:data` property. For example, if you specify `/a/b/caution.png`, the following structure will be created in the repository:

```
/a   (nt:folder)
  /b   (nt:folder)
    /caution.png   (nt:file)
              /jcr:content    (nt:resource)
                        @jcr:data = {contents of the file}
                        @jcr:mimeType = {mime type of the file}
                        @jcr:lastModified = {now}
```

Other kinds of files are treated in a similar way.

When the client uploads the file using the JCR API, ModeShape gets notified of the changes, consults the sequencers to see whether any of them are interested in the new or updated content, and if so runs those sequencers. The image sequencer processes image files for metadata, and any metadata found is stored under the `/images` branch of the repository. The MP3 sequencer processes MP3 audio files for metadata, and any metadata found is stored under the `/mp3s` branch of the repository. And metadata about Java classes are stored under the `/java` area of the repository. All of this happens asynchronously, so any ModeShape activity doesn't impede or slow down the client activities.

So, after the file is uploaded, you can search the repository for the image metadata using the "s" menu option:

```
---------------------------------
Menu:

u) Upload a file to the repository
s) Search the repository using extracted metadata

d) Display statistics

?) Show this menu
q) Quit
>s

1 image was found:
 Media 1
   Name: caution.png
   Path: /images/caution.png
   Type: image
   image:numberOfImages: 1
   image:physicalHeightInches: -1.0
   image:width: 48
   image:physicalWidthDpi: -1
   image:progressive: false
   image:physicalWidthInches: -1.0
   jcr:primaryType: image:metadata
   image:physicalHeightDpi: -1
   image:formatName: PNG
   image:bitsPerPixel: 24
   jcr:mimeType: image/png
   image:height: 48

>
```

Figure 5.3. Searching for media using the example client

Here are the search results after the `sample1.mp3` audio file has been uploaded (to the `/a/b/sample1.mp3` location):

```
---------------------------------
Menu:

u) Upload a file to the repository
s) Search the repository using extracted metadata

d) Display statistics

?) Show this menu
q) Quit
>s

2 images were found:
 Media 1
   Name: caution.png
   Path: /images/caution.png
   Type: image
   image:numberOfImages: 1
   image:physicalHeightInches: -1.0
   image:width: 48
   image:physicalWidthDpi: -1
   image:progressive: false
   image:physicalWidthInches: -1.0
   jcr:primaryType: image:metadata
   image:physicalHeightDpi: -1
   image:formatName: PNG
   image:bitsPerPixel: 24
   jcr:mimeType: image/png
   image:height: 48
 Media 2
   Name: sample1.mp3
   Path: /mp3s/sample1.mp3
   Type: mp3
   mp3:comment: This is a test audio file.
   mp3:title: Sample MP3
   mp3:album: Badwater Slim Performs Live
   mp3:year: 2008
   jcr:primaryType: mp3:metadata
   mp3:author: Badwater Slim

>
```

Figure 5.4. Searching for media using the example client

You can also display the sequencing statistics using the "d" menu option:

```
---------------------------------
Menu:

u) Upload a file to the repository
s) Search the repository using extracted metadata

d) Display statistics

?) Show this menu
q) Quit
>d

# nodes sequenced: 2
# nodes skipped: 10


>_
```

Figure 5.5. Sequencing statistics using the example client

These stats show how many nodes were sequenced, and how many nodes were skipped because they didn't apply to the sequencer's criteria.

Note

There will probably be more nodes skipped than sequenced, since there are more `nt:folder` and `nt:resource` nodes than there are `nt:file` nodes with acceptable names.

You can repeat this process with other files. Any file that isn't an image or MP3 files (as recognized by the sequencing configurations that we'll describe later) will not be sequenced.

5.2. Reviewing the example application

Recall that the example application consists of a client application that sets up an in-memory JCR repository and that allows a user to upload files into that repository. The client also sets up the ModeShape services with an image sequencer so that if any of the uploaded files are PNG, JPEG, GIF, BMP or other images, ModeShape will automatically extract the image's metadata (e.g., image format, physical size, pixel density, etc.) and store that in the repository. Or, if the client uploads MP3 audio files, the title, author, album, year, and comment are extracted from the audio file and stored in the repository.

The example is comprised of 5 classes and 1 interface, located in the `src/main/java` directory:

```
org/modeshape/example/sequencers/ConsoleInput.java
                                /ContentInfo.java
                                /JavaInfo.java
                                /MediaInfo.java
                                /SequencingClient.java
                                /UserInterface.java
```

`SequencingClient` is the class that contains the main application. `ContentInfo` is a simple class that encapsulate metadata generated by the sequencers and accessed by this example application, and there are two subclasses: `MediaInfo` encapsulates metadata about media (image and MP3) files, while `JavaInfo` is a subclass encapsulating information about a Java class. The client accesses the content from the repository and represents the information using instances of `ContentInfo` (and its subclasses) and then passes them to the `UserInterface`. `UserInterface` is an interface with methods that will be called at runtime to request data from the user. `ConsoleInput` is an implementation of this that creates a text user interface, allowing the user to operate the client from the command-line. We can easily create a graphical implementation of `UserInterface` at a later date. We can also create a mock implementation for testing purposes that simulates a user entering data. This allows us to check the behavior of the client automatically using conventional JUnit test cases, as demonstrated by the code in the `src/test/java` directory:

```
org/modeshape/example/sequencers/SequencingClientTest.java
                                /MockUserInterface.java
```

If we look at the `SequencingClient` code, there are a handful of methods that encapsulate the various activities.

Note

Some of the code samples included in this book have had some of the error handling and comments removed so that the code is more readable and concise.

The `main(String[] argv)` method is of course the method that is executed when the application is run. This code creates the ModeShape configuration using the programmatic style.

```
// Create the configuration.
String repositoryId = "content";
String workspaceName = "default";
JcrConfiguration config = new JcrConfiguration();
// Set up the in-memory source where we'll upload the content and where the sequenced output
 will
// be stored ...
config.repositorySource("store")
      .usingClass(InMemoryRepositorySource.class)
      .setDescription("The repository for our content")
      .setProperty("defaultWorkspaceName", workspaceName);
// Set up the JCR repository to use the source ...
config.repository(repositoryId)
      .addNodeTypes("sequencing.cnd")
      .setSource("store");
// Set up the image sequencer ...
config.sequencer("Image Sequencer")
      .usingClass("org.modeshape.sequencer.image.ImageMetadataSequencer")
      .loadedFromClasspath()
      .setDescription("Sequences image files to extract the characteristics of the image")
      .sequencingFrom(
         "//(*.(jpg|jpeg|gif|bmp|pcx|png|iff|ras|pbm|pgm|ppm|psd)[*])/jcr:content[@jcr:data]")
      .andOutputtingTo("/images/$1");
// Set up the MP3 sequencer ...
config.sequencer("MP3 Sequencer")
```

```
        .usingClass("org.modeshape.sequencer.mp3.Mp3MetadataSequencer")
        .loadedFromClasspath()
        .setDescription("Sequences mp3 files to extract the id3 tags of the audio file")
        .sequencingFrom("//(*.mp3[*])/jcr:content[@jcr:data]")
        .andOutputtingTo("/mp3s/$1");
// Set up the Java class file sequencer ...
config.sequencer("Java Class Sequencer")
        .usingClass(ClassFileSequencer.class)
        .setDescription("Sequences Java class files to extract the structure of the classes")
        .sequencingFrom("//*.class[*]/jcr:content[@jcr:data]")
        .andOutputtingTo("/classes");
// Set up the Java source file sequencer ...
config.sequencer("Java Sequencer")
        .usingClass("org.modeshape.sequencer.java.JavaMetadataSequencer")
        .loadedFromClasspath()
        .setDescription("Sequences Java files to extract the AST structure of the Java source code")
        .sequencingFrom("//(*.java[*])/jcr:content[@jcr:data]")
        .andOutputtingTo("/java/$1");
// Set up the CSV file sequencer ...
config.sequencer("CSV Sequencer")
        .usingClass("org.modeshape.sequencer.text.DelimitedTextSequencer")
        .loadedFromClasspath()
        .setDescription("Sequences CSV files to extract the contents")
        .sequencingFrom("//(*.csv[*])/jcr:content[@jcr:data]")
        .andOutputtingTo("/csv/$1");
// Set up the fixed width file sequencer ...
config.sequencer("Fixed Width Sequencer")
        .usingClass("org.modeshape.sequencer.text.FixedWidthTextSequencer")
        .loadedFromClasspath()
        .setDescription("Sequences fixed width files to extract the contents")
        .setProperty("commentMarker", "#")
        .setProperty("columnStartPositions", new int[] { 10, 20, 30, 40})
        .sequencingFrom("//(*.txt[*])/jcr:content[@jcr:data]")
        .andOutputtingTo("/txt/$1");

// Now start the client and tell it which repository and workspace to use ...
SequencingClient client = new SequencingClient(config, repositoryId, workspaceName);
client.setUserInterface(new ConsoleInput(client));
```

The first block of code configures the `JcrConfiguration` and sets up the "store" source, the "content" repository, and three sequencers. Again, this is done via the programmatic style. An alternative would be to load the entire configuration from a configuration file or from an existing

configuration repository. (The repository example shown in the *next chapter* shows how to load the configuration from a file.)

The second block simply instantiates the `SequencingClient` class, passing the configuration and the name of the repository and workspace, and finally sets the user interface (which then executes its behavior, which we'll see below).

The `startRepository()` method builds the `JcrEngine` component from the configuration, starts the engine, and obtains the JCR `javax.jcr.Repository` instance that the client will use. Note that the client has not yet obtained a `javax.jcr.Session` instance, since this will be done each time the client needs to access content from the repository. (This is actually a common practice according to the JCR specification, since Sessions are intended to be very lightweight.)

```
public void startRepository() throws Exception {
    if (this.repository == null) {
        try {
            // Start the ModeShape engine ...
            this.engine = this.configuration.build();
            this.engine.start();

            // Now get the JCR repository instance ...
            this.repository = this.engine.getRepository(repositoryName);
        } catch (Exception e) {
            this.repository = null;
            throw e;
        }
    }
}
```

The `shutdownRepository()` method requests the `JcrEngine` instance shuts down and, since that may take a few moments (if there are any ongoing operations or enqueued activities) awaits for it to complete the shutdown.

```
public void shutdownRepository() throws Exception {
    if (this.repository != null) {
        try {
            this.engine.shutdown();
            this.engine.awaitTermination(4, TimeUnit.SECONDS);
        } finally {
            this.repository = null;
```

```
        }
      }
}
```

None of the other methods really do anything with ModeShape *per se*. Instead, they merely work with the repository using the JCR API.

If we look at the `ConsoleInput` constructor, it starts the repository and a thread for the user interface. At this point, the constructor returns, but the main application continues under the user interface thread. When the user requests to quit, the user interface thread also shuts down the JCR repository.

```
public ConsoleInput( SequencerClient client ) {
 try {
    client.startRepository();

    System.out.println(getMenu());
    Thread eventThread = new Thread(new Runnable() {
       private boolean quit = false;
       public void run() {
          try {
             while (!quit) {
                // Display the prompt and process the requested operation ...
             }
          } finally {
             try {
                // Terminate ...
                client.shutdownRepository();
             } catch (Exception err) {
                System.out.println("Error shutting down sequencing service and repository: "
                                   + err.getLocalizedMessage());
                err.printStackTrace(System.err);
             }
          }
       }
    });
    eventThread.start();
 } catch (Exception err) {
    System.out.println("Error: " + err.getLocalizedMessage());
    err.printStackTrace(System.err);
 }
```

```
}
```

There is one more aspect of this example that is worth discussing. While the repository example in the *next chapter* does show how to use JAAS, this example intentionally shows how you might integrate a different security system into ModeShape. In the `createSession()` method, the `RepositoryClient` creates a `SecurityContextCredentials` wrapper around a custom `SecurityContext` implementation, then passes that credentials into the `login(Credentials,String)` method:

```
protected Session createSession() throws RepositoryException {
    SecurityContext securityContext = new MyCustomSecurityContext();
    SecurityContextCredentials credentials = new SecurityContextCredentials(securityContext);
    return this.repository.login(credentials, workspaceName);
}
```

where the custom `SecurityContext` implementation is as follows:

```
protected class MyCustomSecurityContext implements SecurityContext {
    /**
     * @see org.modeshape.graph.SecurityContext#getUserName()
     */
    public String getUserName() {
        return "Fred";
    }

    /**
     * @see org.modeshape.graph.SecurityContext#hasRole(java.lang.String)
     */
    public boolean hasRole( String roleName ) {
        return true;
    }

    /**
     * @see org.modeshape.graph.SecurityContext#logout()
     */
    public void logout() {
        // do something
    }
```

```
}
```

Obviously you would want to implement this correctly. If you're using ModeShape in a web application, your `SecurityContext` implementation would likely delegate to the `HttpServletRequest`.

But if you're using JAAS, then you could just pass in a `javax.jcr.SimpleCredentials` with the username and password, as long as your `JcrConfiguration`'s repository definitions are set up to use the correct JAAS login context name (see the repository example in the *next chapter*). Or, you could use the approach listed above and supply an instance of the `JaasSecurityContext` to the `SecurityContextCredentials`.

At this point, we've reviewed all of the interesting code in the example application related to ModeShape. However, feel free to play with the application, trying different things.

5.3. What's next

This chapter walked through running the sequencer example and looked at the example code. With the sequencer client, you could upload files into a JCR repository, while ModeShape automatically sequenced the source files you uploaded, extracted the metadata from the files, and stored that metadata inside the repository.

In the *next chapter* we'll do the same for the repository example.

The Repository Example

Chapter 4 walked through the process of downloading and building the examples, while the *previous chapter* showed how to run the sequencer example and walked through the code. In this chapter, we'll run the repository example and walk through that example code to see what it's doing.

6.1. Running the repository example

The repository example consists of a client application that sets up three ModeShape repositories (named "Cars", "Airplanes", and "UFOs") and a federated repository ("Vehicles") that dynamically federates the information from the other three repositories. The client application allows you to interactively navigate each of these repositories just as you would navigate the directory structure on a file system.

This collection of repositories is shown in the following figure:

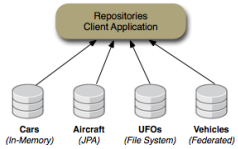

Figure 6.1. Repositories used in the example client

The "Cars" repository is an in-memory repository (using the In-Memory repository connector), the "Aircraft" repository is a JPA repository (using an in-memory HSQL database using the JPA repository connector), and the "UFOs" repository is a file system repository (using the File System repository connector). The federated "Vehicles" repository content is federated from the other repositories and cached into the "Cache" repository. This is shown in the following figure:

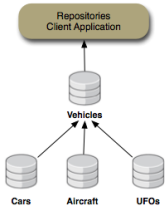

Figure 6.2. Vehicles repository content is federated from the Cars, Airplanes, UFOs, and Configuration repositories

To run the client application, go to the `examples/repository/target/modeshape-example-repositories-basic.dir/` directory and type `./run.sh`. You should see the command-line client and its menus in your terminal:

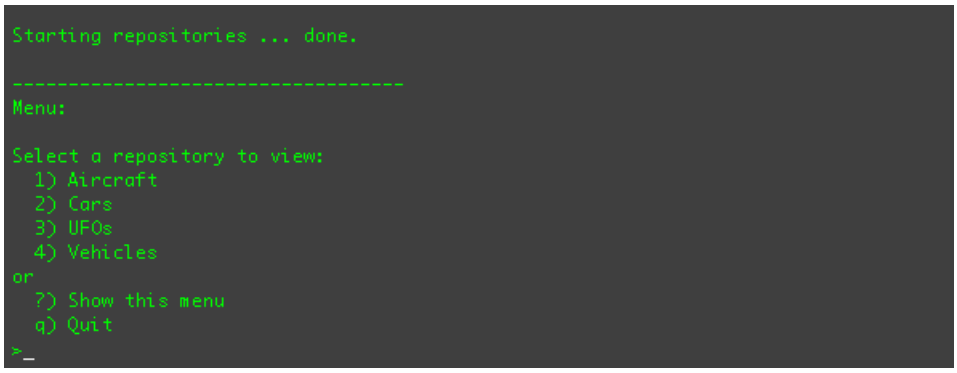

Figure 6.3. Example Client

From this menu, you can see the list of repositories, select one, and navigate through that repository in a manner similar to a *nix command-line shell (although the client itself uses the JCR API to interact with the repositories). Here are some of the commands you can use:

Table 6.1. Repository client commands to navigate a repository

Command	Description
pwd	Print the path of the current node (e.g., the "working directory")
ls [*path*]	List the children and properties of the node at the supplied path, where "*path*" can be any relative path or absolute path. If "*path*" is not supplied, the current working node's path is used.
cd *path*	Change to the specified node, where "*path*" can be any relative path or absolute path. For example, "`cd alpha`" changes the current node to be a child named "`alpha`"; "`cd ..`" changes the current node to the parent node; "`cd /a/b`" changes the current node to be the "`/a/b`" node.
exit	Exit this repository and return the list of repositories.

Note

The first time you access any repository, the application is automatically logging you in to ModeShape's JAAS-based security system. To make the application easier to use, it always logs in with the "jsmith" as the username and "secret" as the password. This matches what is configured in the "jaas.conf.xml" and

"users.properties" files. If you want to confirm that the security feature is working, change the password in `target/modeshape-example-repositories-basic.dir/users.properties` to something else and re-run the application. After you select a repository and try to view a directory with 'ls', you will get a LoginException!

If you were to select the "Cars" repository and use some of the commands, you should see something similar to:

```
-----------------------------------
Menu:

Select a repository to view:
  1) Aircraft
  2) Cars
  3) UFOs
  4) Vehicles
or
  ?) Show this menu
  q) Quit
>2

Entering the "Cars" repository.

Enter one of the following commands followed by RETURN:
      pwd          print the current node's path
      ls [path]    to list the details of the node at the specified absolute or relative path
                   (or the current path if none is supplied)
      cd path      to change to the node at the specified absolute or relative path
      exit         to exit this repository and return to the main menu

Cars> ls
 ./
 ../
 Cars/
 jcr:system/
 jcr:uuid        = 2f5363d5-9a43-45e4-8357-6962d59fe0c8
 jcr:primaryType = dna:root
Cars> cd Cars
/Cars
Cars> ls
 ./
 ../
 Hybrid/
 Sports/
 Luxury/
 Utility/
 jcr:primaryType = nt:unstructured
Cars> ls Hybrid
 ./
 ../
 Toyota Prius/
 Toyota Highlander/
 Nissan Altima/
 jcr:primaryType = nt:unstructured
Cars> _
```

Figure 6.4. Navigating the Cars repository

You can also choose to navigate the "Vehicles" repository, which projects the "Cars" repository content under the `/Vehicles/Cars` node, the "Airplanes" content under the `/Vehicles/Airplanes` branch, the "UFOs" content under the `/Vehicles/UFOs` branch, and the "Configuration" content under `/modeshape:system`.

Try using the client to walk the different repositories. And while this is a contrived application, it does demonstrate the use of ModeShape to federate repositories and provide access through JCR.

6.2. ModeShape connectors

As mentioned in the *Introduction*, one of the capabilities of ModeShape is to provide access through *JCR* [http://www.jcp.org/en/jsr/detail?id=170] to different kinds of repositories and storage systems. Your applications work with the JCR API, but through ModeShape you're able to accesses the content from where the information exists - not just a single purpose-built repository. This is fundamentally what makes ModeShape different.

How does ModeShape do this? At the heart of ModeShape and it's JCR implementation is a simple connector system that is designed around creating and accessing graphs. The ModeShape JCR implementation actually just sits on top of a single repository source, which it uses to access of the repositories content.

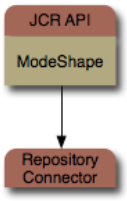

Figure 6.5. ModeShape's JCR implementation delegates to a repository connector

That single repository connector could access:

- a transient, in-memory repository
- an Infinispan data grid that acts as an extremely scalable, highly-available store for repository content
- a JBoss Cache instance that acts as a clustered and replicated store for repository content
- a JDBC database used as a store for repository content
- a repository that accesses existing JDBC databases to project the schema structure as read-only repository content
- a repository that accesses a file systems to present the files and directory structure as (updatable) repository content
- a repository that accesses an SVN repository to present the files and directory structure as (updatable) repository content
- a federated repository that presents a unified, updatable view of the content in multiple other systems (which are accessed via connectors)

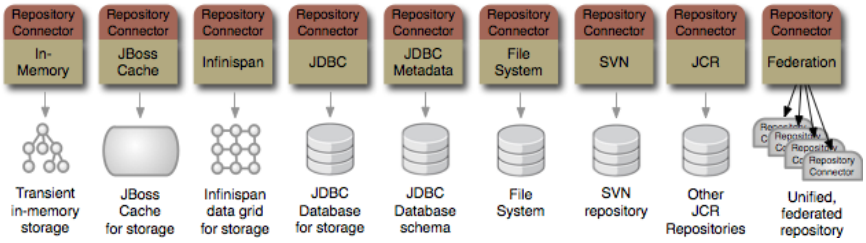

Figure 6.6. ModeShape can put JCR on top of multiple kinds of systems

And the ModeShape project has plans to create other connectors, too. For instance, we're going to build a connector to other JCR repositories. And another to access existing databases so that some or all of the existing data (in whatever structure) can be accessed through JCR. Of course, if we don't have a connector to suit your needs, you can write your own.

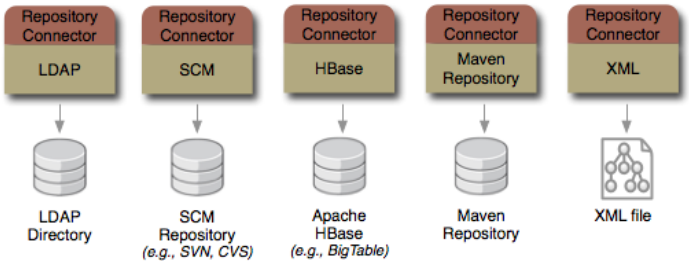

Figure 6.7. Future ModeShape connectors

Note

You might be thinking that these connectors are interesting, but what do they really provide? Is it really useful to use JCR to access a relational database rather than JDBC? Or, why access the files on a file system when there are already mechanisms to do that?

Maybe putting JCR on top of a single system (like a JDBC database) isn't that interesting. What *is* interesting, though, is accessing the information in multiple systems *as if all that information were in a single JCR repository*. That's what the federated repository source is all about. The ModeShape connector system just makes it possible to interact with all these systems in the same way.

Think of it this way: with ModeShape, you can use JCR to get to the schemas of multiple relational databases *and* the schemas defined by DDL files in your SVN repository *and* the schemas defined by logical models stored on your file system.

Before we go further, let's define some terminology regarding connectors.

- A **connector** is the runnable code packaged in one or more JAR files that contains implementations of several interfaces (described below). A Java developer *writes* a connector to a type of source, such as a particular database management system, LDAP directory, source

code management system, etc. It is then packaged into one or more JAR files (including dependent JARs) and deployed for use in applications that use ModeShape repositories.

- The description of a particular source system (e.g., the "Customer" database, or the company LDAP system) is called a **repository source**. ModeShape defines a *RepositorySource* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html] interface that defines methods describing the behavior and supported features and a method for establishing connections. A connector will have a class that implements this interface and that has JavaBean properties for all of the connector-specific properties required to fully describe an instance of the system. Use of JavaBean properties is not required, but it is highly recommended, as it enables reflective configuration and administration. Applications that use ModeShape create an instance of the connector's *RepositorySource* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html] implementation and set the properties for the external source that the application wants to access with that connector.

- A repository source instance is then used to establish **connections** to that source. A connector provides an implementation of the *RepositoryConnection* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/connector/RepositoryConnection.html] interface, which defines methods for interacting with the external system. In particular, the `execute(...)` method takes an *`ExecutionContext`* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] instance and a *`Request`* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/request/Request.html] object. The object defines the environment in which the processing is occurring, including information about the JAAS *`Subject`* [http://java.sun.com/javase/6/docs/api/javax/security/auth/Subject.html] and *`LoginContext`* [http://java.sun.com/javase/6/docs/api/javax/security/auth/login/LoginContext.html]. The *`Request`* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/request/Request.html] object describes the requested operations on the content, with different concrete subclasses representing each type of activity. Examples of commands include (but not limited to) getting a node, moving a node, creating a node, changing a node, and deleting a node. And, if the repository source is able to participate in JTA/JTS distributed transactions, then the *RepositoryConnection* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/connector/RepositoryConnection.html] must implement the `getXaResource()` method by returning a valid `javax.transaction.xa.XAResource` object that can be used by the transaction monitor.

As an example, consider that we want ModeShape to give us access through JCR to the schema information contained in a relational database. We first have to develop a connector that allows us to interact with relational databases using JDBC. That connector would contain a *`JdbcMetadataSource`* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/connector/meta/jdbc/JdbcMetadataSource.html] Java class that implements *RepositorySource* [http://docs.jboss.org/modeshape/2.5.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html], and that has all of the various JavaBean properties for setting the name of the driver class, URL, username, password, and other properties. If we add a

JavaBean property defining the JNDI name, our connector could look in JNDI to find a JDBC `DataSource` instance, perhaps already configured to use connection pools.

Note

Of course, before you develop a connector, you should probably check the *list of connectors* [http://docs.jboss.org/modeshape/latest/manuals/reference/html/provied-connectors-part.html] ModeShape already provides out of the box. With this latest release, ModeShape already includes this JDBC metadata connector! And we're always interested in new connectors and new contributors, so please consider developing your custom connector as part of ModeShape.

So with this very high-level summary, let's dive a little deeper and look at the repository example.

6.3. Reviewing the example repository application

Recall that the example repository application consists of a client application that sets up a repository service and the repositories defined in a configuration repository, allowing the user to pick a repository and interactively navigate the selected repository. Several repositories are set up, including several standalone repositories and one federated repository that dynamically federates the content from the other repositories.

The example is comprised of 2 classes and 1 interface, located in the `src/main/java` directory:

```
org/modeshape/example/repositories/ConsoleInput.java
                                  /RepositoryClient.java
                                  /UserInterface.java
```

`RepositoryClient` is the class that contains the main application. It uses an instance of the `UserInterface` interface to methods that will be called at runtime to obtain information about the files that are imported into the standalone repositories and the JAAS `CallbackHandler` implementation that will be used by JAAS to collect the authentication information. Finally, the `ConsoleInput` is an implementation of this that creates a text user interface, allowing the user to operate the client from the command-line. We can easily create a graphical implementation of `UserInterface` at a later date, or we can also create a mock implementation for testing purposes that simulates a user entering data. This allows us to check the behavior of the client automatically using conventional JUnit test cases, as demonstrated by the code in the `src/test/java` directory:

```
org/modeshape/example/sequencers/RepositoryClientTest.java
```

```
                    /RepositoryClientUsingJcrTest.java
```

If we look at the `RepositoryClient` code, there are a handful of methods that encapsulate the various activities.

Note

Some of the code samples included in this book have had some of the error handling and comments removed so that the code is more readable and concise.

The `main(String[] argv)` method is of course the method that is executed when the application is run. This code creates the ModeShape configuration by loading it from a file.

```
// Set up the JAAS provider (IDTrust) and a policy file (which defines the "modeshape-jcr" login
 config name)
IDTrustConfiguration idtrustConfig = new IDTrustConfiguration();
try {
    idtrustConfig.config("security/jaas.conf.xml");
} catch (Exception ex) {
    throw new IllegalStateException(ex);
}

// Now configure the repository client component ...
RepositoryClient client = new RepositoryClient();
for (String arg : args) {
    arg = arg.trim();
    if (arg.equals("--api=jcr")) client.setApi(Api.JCR);
    if (arg.equals("--api=modeshape")) client.setApi(Api.ModeShape);
    if (arg.equals("--jaas")) client.setJaasContextName(JAAS_LOGIN_CONTEXT_NAME);
                                                        if          (arg.startsWith("--
jaas=") && arg.length() > 7) client.setJaasContextName(arg.substring(7).trim());
}
// And have it use a ConsoleInput user interface ...
client.setUserInterface(new ConsoleInput(client, args));
```

The first block sets up the JAAS provider to be the IDTrust library and a policy file that defines the "modeshape-jcr" JAAS configuration.

The second block of code instantiates the `RepositoryClient` and passes in some options determined from the command-line. It then sets the user interface (which then executes its behavior, which we'll see below).

The `startRepositories()` method builds the `JcrEngine` component from the configuration, starts the engine, and obtains the JCR `javax.jcr.Repository` instance that the client will use. Note that the client has not yet obtained a `javax.jcr.Session` instance, since this will be done each time the client needs to access content from the repository. (This is actually a common practice according to the JCR specification, since Sessions are lightweight.)

```
public void startRepositories() throws IOException, SAXException {
    if (engine != null) return; // already started

    // Load the configuration from a file, as provided by the user interface ...
    JcrConfiguration configuration = new JcrConfiguration();
    configuration.loadFrom(userInterface.getRepositoryConfiguration());

    // Now create the JCR engine ...
    engine = configuration.build();
    engine.start();

    ...

    // For this example, we're using a couple of in-memory repositories (including one for the
   // configuration repository). Normally, these would exist already and would simply be accessed.
    // But in this example, we're going to populate these repositories here by importing from files.
    // First do the configuration repository ...
    String location = this.userInterface.getLocationOfRepositoryFiles();

    // Now import the content for the two in-memory repositories ...
    Graph cars = engine.getGraph("Cars");
    cars.importXmlFrom(location + "/cars.xml").into("/");
    Graph aircraft = engine.getGraph("Aircraft");
    aircraft.importXmlFrom(location + "/aircraft.xml").into("/");
}
```

This method does a number of different things. First, it checks to make sure the repositories are not already running; if so the method just returns. Then, it creates a ModeShape `JcrConfiguration` instance and loads the configuration from a file provided by the user interface. It then creates the `JcrEngine` from the configuration and starts it. Finally, it obtains the location of the content files

from the user interface, and imports them into the "Cars" and "Aircraft" repositories. Again, this is done to keep the example simple.

The `shutdown()` method of the example then logs out and requests that the `JcrEngine` instance shut down and, since that may take a few moments (if there are any ongoing operations or enqueued activities) awaits for it to complete the shutdown.

```
public void shutdown() throws InterruptedException, LoginException {
    logout();
    if (engine == null) return;
    try {
        // Tell the engine to shut down, and then wait up to 5 seconds for it to complete...
        engine.shutdown();
        engine.awaitTermination(5, TimeUnit.SECONDS);
    } finally {
        engine = null;
    }
}
```

A few of the other methods in the `RepositoryClient` class deal with the JAAS `LoginContext`. When needed, the client will authenticate the user (by asking the user interface for a callback handler that will be called when the authentication information is needed). The resulting authenticated `LoginContext` is wrapped by a custom `javax.jcr.Credentials` implementation. As long as the `Credentials` implementation has a `getLoginContext()` method that returns a `LoginContext` object, ModeShape's repository implementation will use that context to create the `javax.jcr.Session`. (Of course, the `javax.jcr.SimpleCredentials` can also be used to create a Session, and ModeShape will then attempt to use JAAS to authenticate the user given by the credentials.)

The `getNodeInfo(...)` method of the example is what is called when the properties and children of a particular node are requested by the user interface. (In the console user interface, this happens when the user navigates the graph structure.) There are really two different behaviors to this method, depending upon whether the JCR API is to be used or whether the ModeShape Graph API is to be used. The portion that uses JCR is shown below:

```
JcrRepository jcrRepository = engine.getRepository(sourceName);
Session session = null;
if (loginContext != null) {
    // Could also use SimpleCredentials(username,password) too
    Credentials credentials = new JaasCredentials(loginContext);
```

```
    session = jcrRepository.login(credentials);
} else {
    session = jcrRepository.login();
}
try {
    // Make the path relative to the root by removing the leading slash(es) ...
    pathToNode = pathToNode.replaceAll("^/+", "");
    // Get the node by path ...
    Node root = session.getRootNode();
    Node node = root;
    if (pathToNode.length() != 0) {
        if (!pathToNode.endsWith("]")) pathToNode = pathToNode + "[1]";
        node = pathToNode.equals("") ? root : root.getNode(pathToNode);
    }

    // Now populate the properties and children ...
    if (properties != null) {
        for (PropertyIterator iter = node.getProperties(); iter.hasNext();) {
            javax.jcr.Property property = iter.nextProperty();
            Object[] values = null;
            // Must call either 'getValue()' or 'getValues()' depending upon # of values
            if (property.getDefinition().isMultiple()) {
                Value[] jcrValues = property.getValues();
                values = new String[jcrValues.length];
                for (int i = 0; i < jcrValues.length; i++) {
                    values[i] = jcrValues[i].getString();
                }
            } else {
                values = new Object[] {property.getValue().getString()};
            }
            properties.put(property.getName(), values);
        }
    }
    if (children != null) {
        // Figure out which children need same-name sibling indexes ...
        Set<String> sameNameSiblings = new HashSet<String>();
        for (NodeIterator iter = node.getNodes(); iter.hasNext();) {
            javax.jcr.Node child = iter.nextNode();
            if (child.getIndex() > 1) sameNameSiblings.add(child.getName());
        }
        for (NodeIterator iter = node.getNodes(); iter.hasNext();) {
            javax.jcr.Node child = iter.nextNode();
            String name = child.getName();
            if (sameNameSiblings.contains(name)) name = name + "[" + child.getIndex() + "]";
```

```
            children.add(name);
        }
    }
} catch (javax.jcr.ItemNotFoundException e) {
    return false;
} catch (javax.jcr.PathNotFoundException e) {
    return false;
} finally {
    if (session != null) session.logout();
}
```

This code is literally just using the standard JCR API. First, it obtains a `javax.jcr.Session` instance (using the available `LoginContext`), finds the desired `javax.jcr.Node`, copies the properties and names of the children into collections supplied by the caller via method parameters, and finally logs out of the session.

The ModeShape Graph API is actually an internal API used within the different components of ModeShape (including the connector and sequencer frameworks), and provides low-level access to the exact same content. Though we do not recommend using this API in your client applications, if you need to write a connector or sequencer, you may need to know how to use the Graph API. Here is the portion of the `getNodeInfo(...)` method that does the exact same operation as the JCR code shown above:

```
// Use the ModeShape Graph API to read the properties and children of the node ...
ExecutionContext context = loginContext != null ? this.context.create(loginContext) : this.context;
Graph graph = engine.getGraph(context, sourceName);
graph.useWorkspace("default");
org.modeshape.graph.Node node = graph.getNodeAt(pathToNode);

if (properties != null) {
    // Now copy the properties into the map provided as a method parameter ...
    for (Property property : node.getProperties()) {
        String name = property.getName().getString(context.getNamespaceRegistry());
        properties.put(name, property.getValuesAsArray());
    }
}
if (children != null) {
    // And copy the names of the children into the list provided as a method parameter ...
    for (Location child : node.getChildren()) {
      String name = child.getPath().getLastSegment().getString(context.getNamespaceRegistry());
        children.add(name);
```

```
    }
}
```

Note that this code is significantly shorter than the equivalent code based upon the JCR API. This is in part because the Graph API doesn't have the notion of a stateful session. But some of it also is simply because the Graph API design requires less code to do the same kinds of operations.

None of the other methods in the `RepositoryClient` really do anything with ModeShape or JCR *per se*. Instead, they really facilitate interaction with the user interface.

If we look at the `ConsoleInput` constructor, it starts the repository and a thread for the user interface. At this point, the constructor returns, but the main application continues under the user interface thread. When the user requests to quit, the user interface thread also shuts down the JCR repository.

```
public ConsoleInput( SequencerClient client ) {
  try {
     client.startRepositories();

     System.out.println(getMenu());
     Thread eventThread = new Thread(new Runnable() {
        private boolean quit = false;
        public void run() {
           try {
              while (!quit) {
                 // Display the prompt and process the requested operation ...
              }
           } finally {
              try {
                 // Terminate ...
                 client.shutdown();
              } catch (Exception err) {
                 System.out.println("Error shutting down repository: "
                                    + err.getLocalizedMessage());
                 err.printStackTrace(System.err);
              }
           }
        }
     });
     eventThread.start();
  } catch (Exception err) {
     System.out.println("Error: " + err.getLocalizedMessage());
```

```
        err.printStackTrace(System.err);
    }
}
```

At this point, we've reviewed all of the interesting code in the example application related to ModeShape. However, feel free to play with the application, trying different things.

6.4. What's next

This chapter walked through running the repository example and looked at the example code. This example allowed you to walk through multiple repositories, including one whose content was federated from multiple other repositories. This was a very simplistic example that only took a few minutes to run.

In the *next chapter* we'll wrap up by summarizing what we've learned about ModeShape and provide information about where you can find out more about ModeShape.

Wrapping Up

This document provides a very high-level overview of ModeShape, introducing you to some basic concepts and showing what would be required to use ModeShape in your own application. We saw two simple examples that showed two different aspects of ModeShape: the sequencing system and the connector system. Each of these examples showed how to create a ModeShape configuration, how to start up the ModeShape engine, and how to then access a `javax.jcr.Repository` instance, and after that your application just uses the standard JCR API.

For a more in-depth description of ModeShape and the internal workings, see our *Reference Guide* [http://docs.jboss.org/modeshape/2.5.0.Beta2/manuals/reference/html/index.html]. This also describes how to write your own custom sequencers or connectors. If you have any questions or comments, please feel free to contact ModeShape's *user mailing list* [mailto:modeshape-users@lists.jboss.org], use our *discussion forums* [http://community.jboss.org/community/modeshape], or chat with the developers in the *IRC chat room* [http://www.jboss.org/modeshape/chat.html]. If you find a bug or have a suggestion, please let us know or (better yet) create a new issue in the project's *JIRA issue management system* [http://jira.jboss.org/browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:summary-panel] . If there's something in particular you're interested in, talk with the community - there may be others interested in the same thing.

7.1. Future directions

Although ModeShape 2.5.0.Beta2 includes several improvements and minor features, this release is primarily a bug-fix release, with numerous fixes for issues reported against 2.1 and 2.2. For details, see the *release notes* [http://docs.jboss.org/modeshape/2.5.0.Beta2/release.html].

ModeShape implements all of the required JCR 2.0 features: repository acquisition, authentication, reading/navigating, query, export, node type discovery, and permissions and capability checking. **ModeShape also implements most of the optional JCR 2.0 features**: writing, import, observation, workspace management, versioning, locking, node type management, same-name siblings, orderable child nodes, and shareable nodes. The remaining optional features (access control management, lifecycle management, retention and hold, and transactions) may be introduced in future versions.

Over the next few releases, we'll be adding more sequencers, connectors, and making a variety of improvements and bug fixes. Other items on our long-term roadmap include a web user interface, Seam integration, and integration with even more kinds of information systems and repositories.

7.2. Getting involved

If you're interested in getting involved with the ModeShape project, take a look at our *community pages* [http://www.jboss.org/modeshape/community.html], and consider picking up one of the sequencers or connectors on our *roadmap* [http://jira.jboss.org/browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:roadmap-panel]. Or, check out

JIRA [http://jira.jboss.org/browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:summary-panel] for the list of features we've thought of. If you think of one that's not there, please add it to JIRA!