ModeShape

Reference Guide

1.0.0.Final

by Randall M. Hauch and Brian Carothers

Target audience

This reference guide is for the developers of ModeShape and those users that want to have a better understanding of how ModeShape works or how to extend the functionality. For a higher-level introduction to ModeShape, see the *Getting Started* [http://docs.jboss.org/modeshape/1.0.0.Final/manuals/gettingstarted/html/index.html] document.

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

Introduction to ModeShape

ModeShape is a JCR implementation that provides access to content stored in many different kinds of systems. A ModeShape repository isn't yet another silo of isolated 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.

To your applications, ModeShape looks and behaves like a regular JCR repository. Using the standard JCR API, applications can search, navigate, version, and listen for changes in the content. But under the covers, ModeShape gets its content by federating multiple back-end systems (like databases, services, other repositories, etc.), allowing those systems to continue "owning" the information while ensuring the unified repository stays up-to-date and in sync.

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, you can make ModeShape automatically sequence these files to extract from their content meaningful information that can be stored in the repository, where it can then be searched, accessed, and analyzed using the JCR API.

This document goes into detail about how ModeShape works to provide these capabilities. It also talks in detail about many of the parts within ModeShape - what they do, how they work, and how you can extend or customize the behavior. In particular, you'll learn about ModeShape *connectors* and *sequencers*, how you can use the implementations included in ModeShape, and how you can write your own to tailor ModeShape for your needs.

So whether you are a developer on the project, or you're trying to learn the intricate details of how ModeShape works, this document hopefully serves a good reference for developers on the project.

1.1. Use cases for ModeShape

ModeShape repositories can be used in a variety of applications. One of the more obvious use cases for a metadata repository is in provisioning and management, where it's critical to understand and keep track of the metadata for models, database, services, components, applications, clusters, machines, and other systems used in an enterprise. Governance takes that a step farther, by also tracking the policies and expectations against which performance of the systems described by the repository can be verified. In these cases, a repository is an excellent mechanism for managing this complex and highly-varied information.

But these large and complex use cases aren't the only way to use a ModeShape repository. You could use an embedded ModeShape repository to manage configuration information for an application, or you could use ModeShape just to provide a JCR interface on top of a few non-JCR systems.

The point is that ModeShape can be used in many different ways, ranging from the very tiny embedded repository to a large and distributed enterprise-grade repository. The choice is yours.

1.2. What is metadata?

Before we dive into more detail about ModeShape and metadata repositories, it's probably useful to explain what we mean by the term "metadata." Simply put, *metadata* is the information you need to manage something. For example, it's the information needed to configure an operating system, or the description of the information in an LDAP tree, or the topology of your network. It's the configuration of an application server or enterprise service bus. It's the steps involved in validating an application before it can go into production. It's the description of your database schemas, or of your services, or of the messages going in and coming out of a service. ModeShape is designed to be a repository for all this (and more).

There are a couple of important things to understand about metadata. First, many systems manage (and frequently change) their own metadata and information. Databases, applications, file systems, source code management systems, services, content management systems, and even other repositories are just a few types of systems that do this. We can't pull the information out and duplicate it, because then we risk having multiple copies that are out-of-sync. Ideally, we could access all of this information through a homogenous API that also provides navigation, caching, versioning, search, and notification of changes. That would make our lives significantly easier.

What we want is *federation*. We can connect to these back-end systems to dynamically access the content and project it into a single, unified repository. We can cache it for faster access, as long as the cache can be invalidated based upon time or event. But we also need to maintain a clear picture of where all the bits come from, so users can be sure they're looking at the right information. And we need to make it as easy as possible to write new connectors, since there are a lot of systems out there that have information we want to federate.

The second important characteristic of the metadata is that a lot of it is represented as files, and there are a lot of different file formats. These include source code, configuration files, web pages, database schemas, XML schemas, service definitions, policies, documents, spreadsheets, presentations, images, audio files, workflow definitions, business rules, and on and on. And logically if files contain metadata, we want to add those files to our metadata repository. The problem is, all that metadata is tied up as blobs in the repository. Ideally, our repository would automatically extract from those files the content that's most useful to us, and place that content inside the repository where it can be much more easily used, searched, related, and analyzed. ModeShape does exactly this via a process we call *sequencing*, and it's an important part of a metadata repository.

The third important characteristic of metadata is that it rarely stays the same. Different consumers of the information need to see different views of it. Metadata about two similar systems is not always the same. The metadata often needs to be tagged or annotated with additional information. And the things being described often change over time, meaning the metadata has to change, too. As a result, the way in which we store and manage the metadata has to be flexible and able to adapt to our ever-changing needs, and the object model we use to interact with the repository must accommodate these needs. The graph-based nature of the JCR API provides this flexibility while also giving us the ability to constrain information when it needs to be constrained.

1.3. What is JCR?

There are a lot of choices for how applications can 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 more risky.

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 new *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 are almost a hybrid with the storage capabilities of relational databases and the flexibility offered by other systems, such as using files. Content repositories also typically provide other capabilities as well, including versioning, indexing, search, access control, transactions, and observation. Because of this, content repositories are 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.

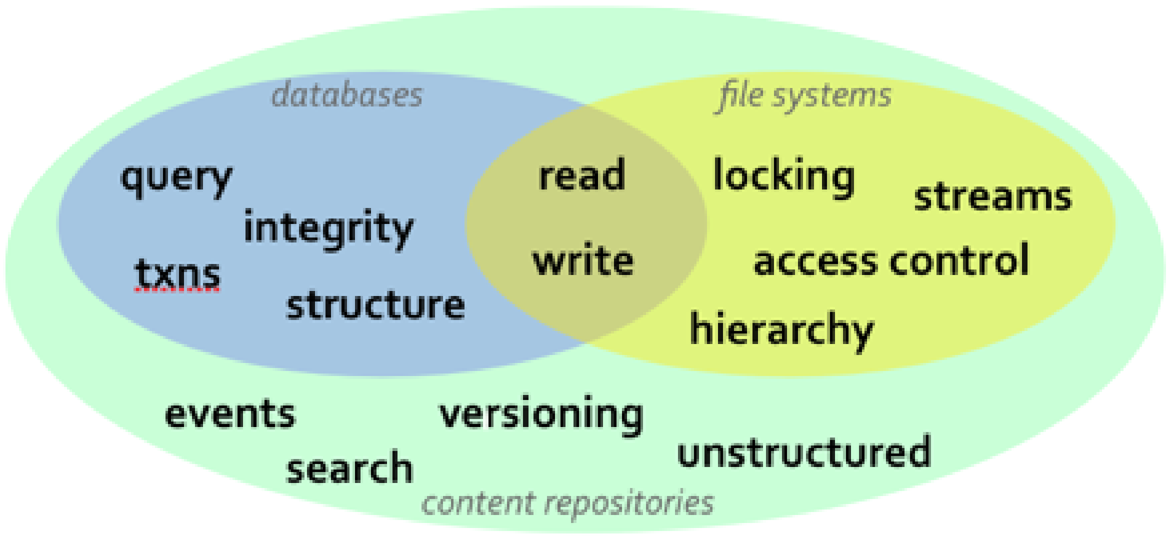

Figure 1.1. JCR API features

1.4. Project roadmap

The roadmap for ModeShape is managed in the project's *JIRA instance* [http://jira.jboss.org/jira/browse/MODE] . The roadmap shows the different tasks, requirements, issues and other activities that have been targeted to each of the upcoming releases. (The *roadmap report* [http://jira.jboss.org/jira/browse/MODE?report=com.atlassian.jira.plugin.system.project:roadmap-panel] always shows the next three releases.)

By convention, the ModeShape project team periodically review JIRA issues that aren't targeted to a release, and then schedule them based upon current workload, severity, and the roadmap. And if we review an issue and don't know how to target it, we target it to the *Future Releases* [http://jira.jboss.org/jira/browse/MODE?report=com.atlassian.jira.plugin.system.project:roadmap-panel] bucket.

At the start of a release, the project team reviews the roadmap, identifies the goals for the release, and targets (or retargets) the issues appropriately.

1.5. ModeShape modules

ModeShape consists of the following modules:

- **modeshape-jcr** contains ModeShape's implementation of the JCR API. If you're using ModeShape as a JCR repository, this is the top-level dependency that you'll want to use. The module defines all required dependencies, except for the repository connector(s) and any sequencer implementations needed by your configuration. As we'll see later on, using ModeShape as a JCR repository is easy: simply create a *configuration*, start ModeShape's *JCR engine*, get the JCR Repository object for your repository, and then use the JCR API. This module also uses the JCR unit tests from the reference implementation to verify the behavior of the ModeShape implementation.
- **modeshape-repository** provides the core ModeShape graph engine and services for managing repository connections, sequencers, MIME type detectors, and observation. If you're using ModeShape repositories via our graph API rather than JCR, then this is where you'd start.
- **modeshape-cnd** provides a self-contained utility for parsing CND (Compact Node Definition) files and transforming the node definitions into a graph notation compatible with ModeShape's JCR implementation.
- **modeshape-graph** defines the Application Programming Interface (API) for ModeShape's low-level graph model, including a fluent-style API for working with graph content. This module also defines the APIs necessary to implement custom connectors, sequencers, and MIME type detectors.
- **modeshape-common** is a small low-level library of common utilities and frameworks, including logging, progress monitoring, internationalization/localization, text translators, component management, and class loader factories.

There are several modules that provide system- and integration-level tests:

- **modeshape-integration-tests** provides a home for all of the integration tests that involve more components that just unit tests. Integration tests are often more complicated, take longer, and involve testing the integration and functionality of multiple components (whereas unit tests focus on testing a single class or component and may use stubs or mock objects to isolate the code being tested from other related components).

The following modules are optional extensions that may be used selectively and as needed (and are located in the source under the `extensions/` directory):

- **modeshape-connector-infinispan** is the preferred ModeShape repository connector for persistently storing content. *Infinispan* [http://infinispan.org] is an extremely scalable, highly available data grid platform that distributes the data across the nodes in the grid. This connector makes it possible for repository content to be stored in a very efficient, fast, highly-concurrent (essentially lock- and synchronization-free), and reliable manner, even when the content size grows to massive sizes. This connector is capable of storing any kind of content, and dictates how the content is stored on the data grid. Therefore, this connector cannot be used to access the content of existing data grids created by/for other applications.
- **modeshape-connector-jbosscache** is a ModeShape repository connector that stores content within a *JBoss Cache* [http://www.jboss.org/jbosscache/] instance. JBoss Cache is a powerful cache implementation that can serve as a distributed cache and that can persist information. The cache instance can be found via JNDI or created and managed by the connector. This connector is capable of storing any kind of content, and dictates how the content is stored in the cache. Therefore, this connector cannot be used to access the content of existing cache instances created by/for other applications.
- **modeshape-connector-jdbc-metadata** is a ModeShape repository connector that provides read-only access to metadata and schema information from relational databases through a JDBC connection. This connector provides an optional and configurable caching facility to prevent frequent requests to the database.
- **modeshape-connector-store-jpa** is a ModeShape repository connector that stores content in a JDBC database, using the Java Persistence API (JPA) and the very highly-regarded and widely-used *Hibernate* [http://www.hibernate.org] implementation. This connector is capable of storing any kind of content, and dictates the schema in which it stores the content. Therefore, this connector cannot be used to access the data in existing created by/for other applications.
- **modeshape-connector-filesystem** is a ModeShape repository connector that accesses the files and folders on (a part of) the local file system, providing that content in the form of `nt:file` and `nt:folder` nodes. This connector *does* support updating the file system when changes are made to the `nt:file` and `nt:folder` nodes. However, this connector does not support storing other kinds of nodes.
- **modeshape-connector-svn** is a ModeShape repository connector that accesses the content of an existing Subversion repository, providing that content in the form of `nt:file` and `nt:folder`

nodes. This connector *does* support updating the SVN repository when changes are made to the `nt:file` and `nt:folder` nodes. However, this connector does not support storing other kinds of nodes.

- **modeshape-sequencer-cnd** is a ModeShape sequencer that extracts JCR node definitions from JCR Compact Node Definition (CND) files.
- **modeshape-sequencer-ddl** is a ModeShape sequencer that extracts the structure and content from DDL files. *This is still under development and includes support for the basic DDL statements in in the Oracle, PostgreSQL, Derby, and standard DDL dialects.*
- **modeshape-sequencer-zip** is a ModeShape sequencer that extracts the files (with content) and directories from ZIP archives.
- **modeshape-sequencer-xml** is a ModeShape sequencer that extracts the structure and content from XML files.
- **modeshape-sequencer-classfile** is a ModeShape sequencer that extracts the package, class/type, member, documentation, annotations, and other information from Java class files.
- **modeshape-sequencer-java** is a ModeShape sequencer that extracts the package, class/type, member, documentation, annotations, and other information from Java source files.
- **modeshape-sequencer-jbpm-jpdl** is a prototype ModeShape sequencer that extracts process definition metadata from jBPM process definition language (jPDL) files. *This is still under development.*
- **modeshape-sequencer-msoffice** is a ModeShape sequencer that extracts metadata and summary information from *Microsoft Office* [http://office.microsoft.com/en-us/] documents. For example, the sequencer extracts from a PowerPoint presentation the outline as well as thumbnails of each slide. Microsoft Word and Excel files are also supported.
- **modeshape-sequencer-images** is a ModeShape sequencer that extracts the image metadata (e.g., size, date, etc.) from PNG, JPEG, GIF, BMP, PCS, IFF, RAS, PBM, PGM, and PPM image files.
- **modeshape-sequencer-mp3** is a ModeShape sequencer that extracts metadata (e.g., author, album name, etc.) from MP3 audio files.
- **modeshape-sequencer-text** is a ModeShape sequencer that extracts data from text streams. There are separate sequencers for character-delimited sequencing and fixed width sequencing, but both treat the incoming text stream as a series of rows separated by line-terminators with each row consisting of one or more columns.
- **modeshape-search-lucene** is an implementation of the *SearchEngine* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/search/SearchEngine.html] interface that uses the *Lucene* [http://lucene.apache.org/java/] library. This module is one of the few extensions that is used directly by the `modeshape-jcr` module.

- **modeshape-mimetype-detector-aperture** is a *MimeTypeDetector* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] implementation that uses the *Aperture* [http://aperture.sourceforge.net/] library to determine the best MIME type given the name and contents of a file.

- **modeshape-classloader-maven** is a small library that provides a *ClassLoaderFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/component/ClassLoaderFactory.html] implementation that can create *ClassLoader* [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html] instances capable of loading classes given a Maven Repository and a list of Maven coordinates. The Maven Repository can be managed within a JCR repository.

The following modules make up the various web application projects (and are located in the source under the `web/` directory):

- **modeshape-web-jcr-rest** provides a set of JSR-311 (JAX-RS) objects that form the basis of a RESTful server for Java Content Repositories. This project provides integration with ModeShape's JCR implementation (of course) but also contains a service provider interface (SPI) that can be used to integrate other JCR implementations with these RESTful services in the future. For ease of packaging, these classes are provided as a JAR that can be placed in the WEB-INF/lib of a deployed RESTful server WAR.

- **modeshape-web-jcr-rest-war** wraps the RESTful services from the modeshape-web-jcr-rest JAR into a WAR and provides in-container integration tests. This project can be consulted as a template for how to deploy the RESTful services in a custom implementation.

- **modeshape-web-jcr-rest-client** is a library that uses POJOs to access the REST web service. This module eliminates the need for applications to know how to create HTTP request URLs and payloads, and how to parse the JSON responses. It can be used to publish (upload) and unpublish (delete) files from ModeShape repositories.

There are also documentation modules (located in the source under the `docs/` directory):

- **docs-getting-started** is the project with the *DocBook* [http://www.docbook.org/] source for the ModeShape Getting Started document.

- **docs-getting-started-examples** is the project with the Java source for the example application used in the ModeShape Getting Started document.

- **docs-reference-guide** is the project with the *DocBook* [http://www.docbook.org/] source for this document, the ModeShape Reference Guide document.

Another module provides some utility functionality:

- **modeshape-jpa-ddl-gen** provides a standalone utility that can generate the DDL for the database schema used by the JPA connector. Because it uses Hibernate, it can generate DDL for any of the databases that the connector can use.

There is another module that runs the full suite of JCR TCK tests, and which at the moment still contains some failures.

- **modeshape-jcr-tck** provides a separate testing project that executes all reference implementation's JCR TCK tests on a nightly basis to track implementation progress against the JCR 1.0 specification. This module will likely be retired when the ModeShape JCR implementation is complete, since `modeshape-jcr` and `modeshape-integration-tests` will be running the full suite of JCR TCK unit tests.

Finally, there is a Maven parent `pom.xml` file that aggregates all of the other projects, provides common defaults for Maven plugins and dependency versions used throughout the modules, and definition of various asset files to help build the necessary Maven artifacts during a build.

Each of these modules is a Maven project with a group ID of `org.modeshape` . All of these projects correspond to artifacts in the *JBoss Maven 2 Repository* [http://repository.jboss.com/maven2/] .

1.6. What's new?

With version 1.0.0.Final, ModeShape introduces support for JCR *query and search* with a number of query languages, including the *JCR XPath language* (required by the 1.0 specification), the *JCR-SQL2 dialect* defined by the JCR 2.0 specification, and a *full-text search language*. This release also adds support for JCR locking and observation.

This means that **ModeShape now implements all of the JCR Level 1 and Level 2 features, along with the optional locking and observation features**. The only optional feature not implemented is versioning, and that will be coming soon. This version passes more than 95% of the JCR TCK tests, and all of the failures are because of a handful of known issues. Fortunately, most of these are either less-frequently-used features of JCR or issues that can be worked around.

This release also introduces a number of new and improved connectors. Both the *file system connector* and *SVN connector* were reworked to support reads and updates, and they both offer a preview of an optional caching system. The *JPA storage connector* was dramatically improved and is significantly faster, more capable, and more efficient. The new *JDBC metadata connector* provides read-only access to the schema information of relational databases through JDBC.

ModeShape 1.0.0.Final includes a number of new and improved sequencers. The new *text sequencer* is able to extract structured data from comma-separated or fixed-width text files. The new *DDL sequencer* is capable of parsing a number of DDL dialects to extract the more important DDL statements. The *CND sequencer* was rewritten and dramatically simplified to perform better, fix a number of known issues, and eliminate a dependency on a third-party library. There is also a new *Java class file sequencer* that operates on Java class files and produces output that is comparable to the *Java source file sequencer*, and that can be used in conjunction with the *ZIP file sequencer* to extract the Java metadata from JARs, WARs, and EAR files.

This release also brings numerous bug fixes and improvements, and upgrades all third-party dependencies to the latest versions available at the time of release. The build system now supports *running all of the tests against a variety of databases* [http://jbossmodeshape.blogspot.com/2009/

09/running-tests-against-different-dbmses.html], making it very easy to test against DBMSes that ModeShape doesn't directly test against. A new DDL generation utility was also introduced that produces the DDL for the database used by the JPA connector. And JCR repositories now support the use of *anonymous users* - this is enabled by default but can easily be changed for production purposes.

Note

If you're migrating from JBoss DNA 0.7, please see the *migration* section for a straightforward recipe.

Part I. Developers and Contributors

The ModeShape project uses a number of process, tools, and procedures to assist in the development of the software. This portion of the document focuses on these aspects and will help developers and contributors obtain the source code, build locally, and contribute to the project.

If you're not contributing to the project but are still developing custom *connectors* or *sequencers*. this information may be helpful in establishing your own environment.

Developer tools

The ModeShape project uses *Maven* as its primary build tool, *Subversion* for its source code repository, *JIRA* for the issue management and bug tracking system, and *Hudson* for the continuous integration system. We do not stipulate a specific integrated development environment (IDE), although most of us use *Eclipse* and rely upon the code formatting and compile preferences to ensure no warnings or errors.

The rest of this chapter talks in more detail about these different tools and how to set them up. But first, we briefly describe our approach to development.

2.1. Development methodology

Rather than use a single formal development methodology, the ModeShape project incorporates those techniques, activities, and processes that are practical and work for the project. In fact, the committers are given a lot of freedom for how they develop the components and features they work on.

Nevertheless, we do encourage familiarity with several major techniques, including:

- ***Agile software development*** **[http://en.wikipedia.org/wiki/Agile_software_development]** includes those software methodologies (e.g., Scrum) that promote development iterations and open collaboration. While the ModeShape project doesn't follow these closely, we do emphasize the importance of always having running software and using running software as a measure of progress. The ModeShape project also wants to move to more frequent releases (on the order of 4-6 weeks)
- ***Test-driven development (TDD)*** **[http://en.wikipedia.org/wiki/Test-driven_development]** techniques encourage first writing test cases for new features and functionality, then changing the code to add the new features and functionality, and finally the code is refactored to clean-up and address any duplication or inconsistencies.
- ***Behavior-driven development (BDD)*** **[http://behaviour-driven.org/]** is an evolution of TDD, where developers specify the desired behaviors first (rather than writing "tests"). In reality, this BDD adopts the language of the user so that tests are written using words that are meaningful to users. With recent test frameworks (like JUnit 4.4), we're able to write our unit tests to express the desired behavior. For example, a test class for sequencer implementation might have a test method `shouldNotThrowAnErrorWhenStreamIsNull()`, which is very easy to understand the intent. The result appears to be a larger number of finer-grained test methods, but which are more easily understood and easier to write. In fact, many advocates of BDD argue that one of the biggest challenges of TDD is knowing what tests to write in the beginning, whereas with BDD the shift in focus and terminology make it easier for more developers to enumerate the tests they need.
- ***Lean software development*** **[http://en.wikipedia.org/wiki/Lean_software_development]** is an adaptation of *lean manufacturing techniques* [http://en.wikipedia.org/wiki/

Lean_manufacturing], where emphasis is placed on eliminating waste (e.g., defects, unnecessary complexity, unnecessary code/functionality/features), delivering as fast as possible, deferring irrevocable decisions as much as possible, continuous learning (continuously adapting and improving the process), empowering the team (or community, in our case), and several other guidelines. Lean software development can be thought of as an evolution of agile techniques in the same way that behavior-driven development is an evolution of test-driven development. Lean techniques help the developer to recognize and understand how and why features, bugs, and even their processes impact the development of software.

2.2. JDK

Currently, ModeShape is developed and built using *JDK 6* [http://java.sun.com/javase/downloads/widget/jdk6.jsp]. So if you're trying to get ModeShape to compile locally, you should make sure you have the JDK 6 installed and are using it. If you're a contributor, you should make sure that you're using JDK 6 before committing any changes.

When installing a JDK, simply follow the procedure for your particular platform. On most platforms, this should set the `JAVA_HOME` environment variable. But if you run into any problems, first check that this environment variable was set to the correct location, and then check that you're running the version you expect by running the following command:

```
$ java -version
```

If you don't see the correct version, double-check your JDK installation.

2.3. JIRA

ModeShape uses *JIRA* [http://jira.jboss.org/jira/browse/MODE] as its bug tracking, issue tracking, and project management tool. This is a browser-based tool, with very good functionality for managing the different tasks. It also serves as the community's roadmap, since we can define new features and manage them alongside of the bugs and other issues. Although most of the issues have been created by community members, we encourage any users to suggest new features, log defects, or identify shortcomings in ModeShape.

The ModeShape community also encourages its members to work only issues that are managed in JIRA, and preferably those that are targeted to the current release effort. If something isn't in JIRA but needs to get done, then create an issue before you start working on the code changes. Once you have code changes, you can upload a patch to the JIRA issue if the change is complex, if you want someone to review it, or if you don't have commit privileges and have fixed a bug.

2.4. Subversion

ModeShape uses Subversion as its source code management system, and specifically the instance at *JBoss.org* [http://www.jboss.org]. Although you can view the *trunk* [http:/

/anonsvn.jboss.org/repos/modeshape/trunk/] of the Subversion repository directly (or using *FishEye* [http://fisheye.jboss.org/browse/modeshape/trunk]) through your browser, in order to get more than just a few files of the latest version of the source code, you probably want to have an SVN client installed. Several IDE's have SVN support included (or available as plugins), but having the command-line SVN client is recommended. See *http://subversion.tigris.org/* for downloads and instructions for your particular platform.

Here are some useful URLs for the ModeShape Subversion:

Table 2.1. SVN URLs for ModeShape

Repository	URL
Anonymous Access URL	*http://anonsvn.jboss.org/repos/modeshape/trunk/*
Secure Developer Access URL	*https://svn.jboss.org/repos/modeshape/trunk/*
FishEye Code Browser	*http://fisheye.jboss.org/browse/modeshape/trunk/*

When committing to SVN, be sure to include in a commit comment that includes the JIRA issue that the commit applies to and a very good and thorough description of what was done. It only takes a minute or two to be very clear about the change. And including the JIRA issue (e.g., "MODE-123") in the comment allows the JIRA system to track the changes that have been made for each issue.

Also, any single SVN commit should apply to one and only one JIRA issue. Doing this helps ensure that each commit is atomic and focused on a single activity. There are exceptions to this rule, but they are rare.

Sometimes you may have some local changes that you don't want to (or aren't allowed to) commit. You can make a patch file and upload it to the JIRA issue, allowing other committers to review the patch. However, to ensure that patches are easily applied, please use SVN to create the patch. To do this, simply do the following in the top of the codebase (e.g., the `trunk` directory):

```
$ svn diff . > ~/MODE-000.patch
```

where `MODE-000` represents the ModeShape issue number. Note that the above command places the patch file in your home directory, but you can place the patch file anywhere. Then, simply use JIRA to attach the patch file to the particular issue, also adding a comment that describes the version number against which the patch was created.

To apply a patch, you usually want to start with a workspace that has no changes. Download the patch file, then issue the following command (again, from the top-level of the workspace):

```
$ patch -E -p0 < ~/MODE-000.patch
```

The "-E" option specifies to delete any files that were made empty by the application of the patch, and the "-p0" option instructs the patch tool to not change any of the paths. After you run this command, your working area should have the changes defined by the patch.

2.5. Git

Several contributors are using *Git* [http://git-scm.com/] on their local development machines. This allows the developer to use Git branches, commits, merges, and other Git tools, but still using the ModeShape *Subversion* repository. For more information, see our *blog* [http://jbossmodeshape.blogspot.com/2009/05/git-and-svn-beginning.html] *posts* [http://jbossmodeshape.blogspot.com/2009/05/git-it-together-with-subversion.html] on the topic.

2.6. Maven

ModeShape uses Maven 2 for its build system, as is this example. 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* and Maven 2.0.9 (or higher).

Maven can be downloaded from *http://maven.apache.org/*, and is installed by unzipping the `maven-2.0.9-bin.zip` or `maven-2.0.11-bin.zip` file to a convenient location on your local disk. Simply add `$MAVEN_HOME/bin` to your path and add the following profile to your `~/.m2/settings.xml` file:

```
<settings>
  <profiles>
    <profile>
      <id>jboss.repository</id>
      <activation>
        <property>
          <name>!jboss.repository.off</name>
        </property>
      </activation>
      <repositories>
        <repository>
          <id>snapshots.jboss.org</id>
          <url>http://snapshots.jboss.org/maven2</url>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </repository>
        <repository>
          <id>repository.jboss.org</id>
          <url>http://repository.jboss.org/maven2</url>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>repository.jboss.org</id>
          <url>http://repository.jboss.org/maven2</url>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
        <pluginRepository>
          <id>snapshots.jboss.org</id>
          <url>http://snapshots.jboss.org/maven2</url>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
</settings>
```

This profile informs Maven of the two JBoss repositories (*snapshots* [http://repository.jboss.org/maven2] and *releases* [http://snapshots.jboss.org/maven2]) that contain all of the JARs for ModeShape and all dependent libraries.

While you're adding `$MAVEN_HOME/bin` to your path, you should also set the `$MAVEN_OPTS` environment variable to "`-Xmx384m`". If you don't do this, you'll likely see an `java.lang.OutOfMemoryError` sometime during a full build.

Note

The JBoss Maven repository provides a central location for not only the artifacts produced by the JBoss.org projects (well, at least those that use Maven), but also is where those projects can place the artifacts that they depend on. ModeShape has a policy that the *source code and JARs* for *all* dependencies *must* be loaded into the JBoss Maven repository. It may be a little bit more work for the developers, but it does help ensure that developers have easy access to the source and that the project (and dependencies) can always be rebuilt when needed.

For more information about the JBoss Maven repository, including instructions for adding source and JAR artifacts, see the *JBoss.org Wiki* [http://wiki.jboss.org/wiki/Maven].

2.6.1. Building

There are just a few commands that are useful for building ModeShape (and it's *subprojects*). Usually, these are issued while at the top level of the code (usually just below `trunk/`), although issuing them inside a subproject just applies to that subproject.

Table 2.2. Useful Maven commands

Command	Description
`mvn clean`	Clean up all built artifacts (e.g., the `target/` directory in each project)
`mvn clean install`	Called the "quick build". Clean up all produced artifacts; compile the source code and test cases; run all of the unit tests; and install the resulting JAR artifact(s) into your local Maven repository (e.g, usually `~/.m2/repository`). This is often what developers run prior to checking in changes, since it generally runs quickly. Note that no integration tests are performed, and HSQLDB is used when a database is needed.
`mvn clean install -Ddatabase=dbprofile`	Same as the "quick build", except that it specifies the database management system that is to be used by the tests. Options for "`dbprofile`" values are: "`hsqldb`", "`h2`", "`postgresql_local`", "`postgresql8`", "`mysql5`", "`oracle9i`", "`oracle10g`", "`oracle11g`", "`db2v8`", "`db2v9`", "`sybase15`", and "`mssql2005`". The database connection information for these database profiles are in the parent "pom.xml" file, and most of these are configured to use database instances within the JBoss Quality Assurance lab and are accessible only to Red Hat employees. However, feel free to add your own profiles or even change the settings in the POM file to suit your needs.

Command	Description
`mvn -P integration clean install`	This "integration build" does everything the "quick" build does plus it compiles and runs the integration tests, which take several extra minutes to run. Also, HSQLDB is used when a database is needed.
`mvn -P integration clean install -Ddatabase=dbprofile`	This does the same as the "integration build", except that it specifies the database management system that is to be used by the unit and integration tests. Options for the "`dbprofile`" values are the same as listed above.
`mvn -P assembly clean install`	This runs a builds all source code, documentation, JavaDoc, runs all unit and integration tests, and produces all assemblies (e.g., zip files). HSQLDB is used when a database is needed.

2.7. Continuous integration with Hudson

ModeShape's continuous integration is done with several Hudson jobs on *JBoss.org* [http://www.jboss.org]. These jobs run periodically and basically run the Maven build process. Any build failures or test failures are reported, as are basic statistics and history for each job.

Table 2.3. Continuous integration jobs

Job	Description
Continuous [http://hudson.jboss.org/hudson/job/ModeShape%20continuous/]	Continuous build that runs an integration build after changes are committed to SVN. SVN is polled every 15 minutes.
Nightly [http://hudson.jboss.org/hudson/job/ModeShape%20nightly%20integration/]	Integration build that runs every night (usually around 2 a.m. EDT), regardless of whether changes have been committed to SVN since the previous night.

2.8. Eclipse IDE

Many of the ModeShape committers use the Eclipse IDE, and all project files required by Eclipse are committed in SVN, making it pretty easy to get an Eclipse workspace running with all of the ModeShape projects.

We're using the latest released version of Eclipse, available from *Eclipse.org* [http://www.eclipse.org/]. Simply follow the instructions for your platform.

After Eclipse is installed, create a new workspace. Before importing the ModeShape projects, import (via **File->Import->Preferences**) the subset of the Eclipse preferences by importing the `eclipse-preferences.epf` file (located under `trunk`). Then, open the Eclipse preferences and open the **Java->Code Style-> Formatter** preference page, and press the "Import" button and choose the `eclipse-code-formatter-profile.xml` file (also located under `trunk`). This will load the code formatting preferences for the ModeShape project.

Then install Eclipse plugins for SVN and Maven. (Remember, you will have to restart Eclipse after installing them.) We use the following plugins:

Table 2.4. Eclipse Subversion Plugins

Eclipse Plugins	Update Site URLs
Subversive SVN Client	*http://www.polarion.org/projects/subversive/download/eclipse/2.0/update-site/* *http://www.polarion.org/projects/subversive/download/integrations/update-site/*
Maven Integration for Eclipse	*http://m2eclipse.sonatype.org/update/*

After you check out the ModeShape codebase, you can import the ModeShape Maven projects into Eclipse as Eclipse projects. To do this, go to "File->Import->Existing Projects", navigate to the `trunk/` folder in the import wizard, and then check each of the *subprojects* that you want to have in your workspace. Don't forget about the projects under `extensions/` or `docs/`.

2.9. Releasing

This section outlines the basic process of releasing ModeShape. This **must** be done either by the project lead or only after communicating with the project lead.

Before continuing, your local workspace should contain no changes and should be a perfect reflection of Subversion. You can verify this by getting the latest from Subversion

```
$ svn update
```

and ensuring that you have no additional changes with

```
$ svn status
```

You may also want to note the revision number for use later on in the process. The release number is returned by the `svn update` command, but may also be found using

```
$ svn info
```

At this point, you're ready to verify that everything builds normally.

2.9.1. Building all artifacts and assemblies

By default, the project's Maven build process does *not* build the documentation, JavaDocs, or assemblies. These take extra time, and most of our builds don't require them. So the first step of releasing ModeShape is to use Maven to build all of regular artifacts (e.g., JARs) and these extra documents and assemblies.

> **Note**
>
> Before running Maven commands to build the releases, increase the memory available to Maven with this command: `$ export MAVEN_OPTS=-Xmx512m`

To perform this complete build, issue the following command while in the `trunk/` directory:

```
$ mvn -P assembly clean install
```

This command runs the "clean" and "install" goals using the "assembly" profile, which adds the production of JavaDocs, the Getting Started document, the Reference Guide document, the Getting Started examples, integration tests, and several ZIP archives. The order of the goals is important.

After this build has completed, verify that the assemblies under `target/` have actually been created and that they contain the correct information. At this point, we know that the actual Maven build process is building everything we want and will complete without errors. We can now proceed with preparing for the release.

2.9.2. Determine the version to be released

The version being released should match the *JIRA* [http://jira.jboss.org/jira/browse/MODE] road map. Make sure that all issues related to the release are closed. The project lead should be notified and approve that the release is taking place.

2.9.3. Release dry run

The next step is to ensure that all information in the POM is correct and contains all the information required for the release process. This is called a *dry run*, and is done with the Maven "release" plugin:

```
$ mvn -P assembly release:prepare -DdryRun=true
```

This may download a lot of Maven plugins if they already haven't been downloaded, but it will eventually prompt you for the release version of each of the Maven projects, the tag name for the release, and the next development versions (again for each of the Maven projects). The default values are probably acceptable; if not, then check that the "`<version>`" tags in each of the POM files is correct and end with "-SNAPSHOT".

After the dry run completes you should clean up the files that the release plugin created in the dry run:

```
$ mvn -P assembly release:clean
```

2.9.4. Prepare for the release

Run the prepare step (without the `dryRun` option):

```
$ mvn -P assembly release:prepare
```

You will again be prompted for the release versions and tag name. These should be the same as what was used during the dry run. This will run the same steps as the dry run, with the additional step of tagging the release in SVN.

If there are any problems during this step, you should go back and try the dry run option. But after this runs successfully, the release will be tagged in SVN, and the `pom.xml` files in SVN under `/trunk` will have the next version in the "<version>" values. However, the artifacts for the release are not yet published. That's the next step.

2.9.5. Perform the release

At this point, the release's artifacts need to be published to the JBoss Maven repository. This next command check outs the files from the release tag created earlier (into a `trunk/target/checkout` directory), runs a build, and then deploys the generated artifacts. Note that this ensures that the artifacts are built from the tagged code.

```
$ mvn release:perform -DuseReleaseProfile=false
```

Note

If during this process you get an error finding the released artifacts in your local Maven repository, you may need to go into the `trunk/target/checkout` folder and run `$ mvn install`. This is a simple workaround to make the artifacts available locally. Another option to try is adding `-Dgoals=install,assembly` to the `$ mvn release:perform...` command above.

The release has been performed, but we still need to build and deploy the *real* artifacts to the JBoss Maven repository. To do this, go to a working area and check out the recently-produced SVN tag (using the correct `{release-number}`):

```
$ svn checkout https://anonsvn.jboss.org/repos/modeshape/tags/modeshape-{release-number}/
```

Then, go into the new directory, and perform a Maven deploy:

```
$ mvn clean deploy
```

This will rebuild all the artifacts (from your local copy of the *tagged* source) and deploy them to the local file system, which is comprised of a local checkout of the JBoss Maven2 repository in a location specified by a combination of the `<distributionManagement>` section of several `pom.xml` files and your personal `settings.xml` file. Once this Maven command completes, you will need to commit the new files after they are deployed. For more information, see the *JBoss wiki* [http://wiki.jboss.org/wiki/Maven].

At this point, the software has been released and tagged, and it's been deployed to a local checked-out copy of the ModeShape Maven 2 repository (via the "<distribution>" section of the pom.xml files). Those need to be committed into the Maven 2 repository using SVN. And finally, the last thing is to publish the release onto the project's *downloads* [http://www.jboss.org/modeshape/downloads.html] and *documentation* [http://www.modeshape.org//docs] pages.

The assemblies of the source, binaries, etc. also need to be published onto the http://www.jboss.org/modeshape/downloads.html area of the *the project page* [http://www.modeshape.org]. This process is expected to change, as *JBoss.org* [http://www.jboss.org] improves its infrastructure.

2.10. Summary

In this chapter, we described the various aspects of developing code for the ModeShape project. Next, we must discuss the testing practices for ModeShape project. This is the topic of the *next chapter*.

Testing

The ModeShape project uses automated testing to verify that the software is doing what it's supposed to and not doing what it shouldn't do. These automated tests are run continuously and also act as regression tests, ensuring that we know if any problems we find and fix reappear later. All of our tests are executed as part of our *Maven* build process, and the entire build process (including the tests) is automatically run using *Hudson* continuous integration system.

3.1. Unit tests

Unit tests verify the behavior of a single class (or small set of classes) in isolation from other classes. We use the JUnit 4.4 testing framework, which has significant improvements over earlier versions and makes it very easy to quickly write unit tests with little extra code. We also frequently use the Mockito library to help create mock implementations of other classes that are not under test but are used in the tests.

Unit tests should generally run quickly and should not require large assemblies of components. Additionally, they may rely upon the file resources included in the project, but these tests should require no external resources (like databases or servers). Note that our unit tests are run during the "test" phase of the standard *Maven lifecycle* [http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html]. This means that they are executed against the raw .class files created during compilation.

Developers are expected to run all of the ModeShape unit tests in their local environment before committing changes to SVN. So, if you're a developer and you've made changes to your local copy of the source, you can run those tests that are related to your changes using your IDE or with Maven (or any other mechanism). But before you commit your changes, you are expected to run a full Maven build using `mvn clean install` (in the "trunk/" directory). Please do *not* rely upon continuous integration to run all of the tests for you - the CI system is there to catch the occasional mistakes and to also run the *integration tests*.

3.2. Integration tests

While *unit tests* test individual classes in (relative) isolation, the purpose of **integration tests** are to verify that assemblies of classes and components are behaving correctly. These assemblies are often the same ones that end users will actually use. In fact, integration tests are executed during the "integration-test" phase of the standard *Maven lifecycle* [http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html], meaning they are executed against the packaged JARs and artifacts of the project.

Integration tests also use the JUnit 4.4 framework, so they are again easy to write and follow the same pattern as unit tests. However, because they're working with larger assemblies of components, they often will take longer to set up, longer to run, and longer to tear down. They also may require initializing "external resources", like databases or servers.

Note, that while external resources may be required, care should be taken to minimize these dependencies and to ensure that most (if not all) integration tests may be run by anyone who downloads the source code. This means that these external resources should be available and set up within the tests. For example, use in-memory databases where possible. Or, if a database is required, use an open-source database (e.g., MySQL or PostgreSQL). And when these external resources are not available, it should be obvious from the test class names and/or test method names that it involved an external resource (e.g., "`MySqlConnectorIntegrationTest.shouldFindNodeStoredInDatabase()`").

3.3. Writing tests

As mentioned in *the introduction*, the ModeShape project doesn't follow any one methodology or process. Instead, we simply have a goal that as much code as possible is tested to ensure it behaves as expected. Do we expect 100% of the code is covered by automated tests? No, but we do want to test as much as we can. Maybe a simple JavaBean class doesn't need many tests, but any class with non-trivial logic should be tested.

We do encourage writing tests either before or while you write the code. Again, we're not blindly following a methodology. Instead, there's a very practical reason: writing the tests early on helps you write classes that are testable. If you wait until after the class (or classes) are done, you'll probably find that it's not easy to test all of the logic (especially the complicated logic).

Another suggestion is to write tests so that they specify and verify the behavior that is expected from a class or component. One challenge developers often have is knowing what they should even test and what the tests should look like. This is where ***Behavior-driven development (BDD)*** **[http://behaviour-driven.org/]** helps out. If you think about what a class' behaviors are supposed to be (e.g., requirements), simply capture those requirements as test methods (with no implementations). For example, a test class for sequencer implementation might have a test method `shouldNotThrowAnErrorWhenTheSuppliedStreamIsNull() { }`. Then, after you enumerate all the requirements you can think of, go back and start implementing the test methods.

If you look at the existing test cases, you'll find that the names of the unit and integration tests in ModeShape follow a naming style, where the test method names are readable sentences. Actually, we try to name the test methods *and* the test classes such that they form a concisely-worded requirement. For example,

```
InMemorySequencerTest.shouldNotThrowAnErrorWhenTheSuppliedStreamIsNull()
```

is easily translated into a readable requirement:

```
InMemorySequencer should not throw an error when the supplied stream is null.
```

In fact, at some point in the future, we'd like to process the source to automatically generate a list of the behavior specifications that are asserted by the tests.

But for now, we write tests - a lot of them. And by following a few simple conventions and practices, we're able to do it quickly and in a way that makes it easy to understand what the code is supposed to do (or not do).

3.4. Technology Compatibility Kit (TCK) tests

Many Java specifications provide TCK test suites that can be used to check or verify that an implementation correctly implements the API or SPI defined by the specification. These TCK tests vary by technology, but *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] does provide TCK tests that ensure that a JCR repository implementation exhibits the correct and expected behavior.

ModeShape has implemented all of the JCR Level 1 and Level 2 features, along with the optional locking and observation features. The JCR-SQL optional feature has already been deprecated for *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] and will not be implemented. The only optional feature left to be implemented is versioning, and that will be coming soon.

The ModeShape project also frequently runs the JCR TCK unit tests from the reference implementation. (Those these tests are not the official TCK, they apparently are used within the official TCK.) These unit tests are set up in the `modeshape-jcr-tck` project.

The 1.0.0.Final release passes all of the JCR TCK tests, but still needs to be certified before being considered fully compliant with JCR 1.0. The ModeShape project plans to focus on attaining this certification in the very near future.

Part II. ModeShape Core

The ModeShape project organizes the codebase into a number of subprojects. The most fundamental are those *core libraries*, including the graph API, connector framework, sequencing framework, as well as the configuration and engine in which all the components run. These are all topics covered in this part of the document.

The ModeShape implementation of the *JCR API* [http://www.jcp.org/en/jsr/detail?id=170] as well as some other JCR-related components are covered in the *next part*.

Execution Context

The various components of ModeShape are designed as plain old Java objects, or POJOs. And rather than making assumptions about their environment, each component instead requires that any external dependencies necessary for it to operate must be supplied to it. This pattern is known as Dependency Injection, and it allows the components to be simpler and allows for a great deal of flexibility and customization in how the components are configured.

The approach that ModeShape takes is simple: a simple POJO that represents everything about the environment in which components operate. Called `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html], it contains references to most of the essential facilities, including: security (authentication and authorization); namespace registry; name factories; factories for properties and property values; logging; and access to class loaders (given a classpath). Most of the ModeShape components require an `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] and thus have access to all these facilities.

The `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] is a concrete class that is instantiated with the no-argument constructor:

```
public class ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] implements ClassLoaderFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/component/ClassLoaderFactory.html] {

    /**
     * Create an instance of an execution context, with default implementations for all components.
     */
     public ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]() { ... }

    /**
     * Get the factories that should be used to create values for {@link Property properties}.
     * @return the property value factory; never null
     */
    public ValueFactories [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactories.html] getValueFactories() {...}

    /**
     * Get the namespace registry for this context.
     * @return the namespace registry; never null
     */
     public NamespaceRegistry [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/
graph/property/NamespaceRegistry.html] getNamespaceRegistry() {...}

    /**
     * Get the factory for creating {@link Property} objects.
     * @return the property factory; never null
     */
   public PropertyFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/PropertyFactory.html] getPropertyFactory() {...}

    /**
     * Get the security context for this environment.
     * @return the security context; never null
     */
   public SecurityContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
SecurityContext.html] getSecurityContext() {...}

    /**
    * Return a logger associated with this context. This logger records only those activities within the
    * context and provide a way to capture the context-specific activities. All log messages are also
     * sent to the system logger, so classes that log via this mechanism should
<i>not</i>
 also
     * {@link Logger#getLogger(Class) obtain a system logger}.
     * @param clazz the class that is doing the logging
     * @return the logger, named after clazz; never null
     */
     public Logger [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/util/
Logger.html] getLogger( Class<?> clazz ) {...}

    /**
    * Return a logger associated with this context. This logger records only those activities within the
    * context and provide a way to capture the context-specific activities. All log messages are also
    * sent to the system logger, so classes that log via this mechanism should
<i>not</i>
 also
    * {@link Logger#getLogger(Class) obtain a system logger}.
     * @param name the name for the logger
     * @return the logger, named after clazz; never null
     */
     public Logger [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/util/
Logger.html] getLogger( String name ) {...}

   ...
```

```
}
```

The fact that so many of the ModeShape components take `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] instances gives us some interesting possibilities. For example, one execution context instance can be used as the highest-level (or "application-level") context for all of the services (e.g., `RepositoryService` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html], `SequencingService` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/sequencer/SequencingService.html], etc.). Then, an execution context could be created for each user that will be performing operations, and that user's context can be passed around to not only provide security information about the user but also to allow the activities being performed to be recorded for user feedback, monitoring and/or auditing purposes.

As mentioned above, the starting point is to create a default execution context, which will have all the default components:

```
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] context = new ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]();
```

Once you have this top-level context, you can start creating *subcontexts* with different components, and different security contexts. (Of course, you can create a subcontext from any instance.) To create a subcontext, simply use one of the `with(...)` methods on the parent context. We'll show examples later on in this chapter.

4.1. Security

ModeShape uses a simple abstraction layer to isolate it from the security infrastructure used within an application. A *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] represents the context of an authenticated user, and is defined as an interface:

```
public interface SecurityContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] {

    /**
     * Get the name of the authenticated user.
     * @return the authenticated user's name
     */
    String getUserName();

    /**
     * Determine whether the authenticated user has the given role.
     * @param roleName the name of the role to check
     * @return true if the user has the role and is logged in; false otherwise
     */
    boolean hasRole( String roleName );

    /**
     * Logs the user out of the authentication mechanism.
     * For some authentication mechanisms, this will be implemented as a no-op.
     */
    void logout();
}
```

Every `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] has a *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] instance, though the top-level (default) execution context does not represent an authenticated user. But you can create a subcontext for a user authenticated via JAAS:

```
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] context = ...
String username = ...
char[] password = ...
String jaasRealm = ...
SecurityContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] securityContext = new JaasSecurityContext(jaasRealm, username,
 password);
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] userContext = context.with(securityContext);
```

In the case of JAAS, you might not have the password but would rather prompt the user. In that case, simply create a subcontext with a different security context:

```
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] context = ...
String jaasRealm = ...
```

```
CallbackHandler [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/callback/CallbackHandler.html] callbackHandler = ...
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] userContext = context.with(new JaasSecurityContext(jaasRealm, callbackHandler);
```

Of course if your application has a non-JAAS authentication and authorization system, you can simply provide your own implementation of *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html]:

```
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] context = ...
SecurityContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] mySecurityContext = ...
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] myAppContext = context.with(mySecurityContext);
```

These `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]s then represent the authenticated user in any component that uses the context.

4.1.1. JAAS

One of the *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] implementations provided by ModeShape is the `JaasSecurityContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/JaasSecurityContext.html], which delegates any authentication or authorization requests to a *Java Authentication and Authorization Service (JAAS)* [http://java.sun.com/javase/technologies/security/] provider. This is the standard approach for authenticating and authorizing in Java.

There are quite a few JAAS providers available, but one of the best and most powerful providers is *JBoss Security* [http://www.jboss.org/jbosssecurity/], the open source security framework used by JBoss. JBoss Security offers a number of JAAS login modules, including:

- **User-Roles Login Module** is a simple `javax.security.auth.login.LoginContext` implementation that uses usernames and passwords stored in a properties file.
- **Client Login Module** prompts the user for their username and password.
- **Database Server Login Module** uses a JDBC database to authenticate principals and associate them with roles.
- **LDAP Login Module** uses an LDAP directory to authenticate principals. Two implementations are available.

- **Certificate Login Module** authenticates using X509 certificates, obtaining roles from either property files or a JDBC database.
- **Operating System Login Module** authenticates using the operating system's mechanism.

and many others. Plus, JBoss Security also provides other capabilities, such as using XACML policies or using federated single sign-on. For more detail, see the *JBoss Security* [http://www.jboss.org/jbosssecurity/] project.

4.1.2. Web application security

If ModeShape is being used within a web application, then it is probably desirable to reuse the security infrastructure of the application server. This can be accomplished by implementing the *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] interface with an implementation that delegates to the HttpServletRequest. Then, for each request, create a `SecurityContextCredentials` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/SecurityContextCredentials.html] instance around your *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html], and use that credentials to obtain a JCR Session.

Here is an example of the *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] implementation that uses the servlet request:

```
@Immutable
public class ServletSecurityContext implements SecurityContext [http://docs.jboss.org/
modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] {

    private final String userName;
    private final HttpServletRequest request;

    /**
     * Create a {@link ServletSecurityContext} with the supplied
     * {@link HttpServletRequest servlet information}.
     *
     * @param request the servlet request; may not be null
     */
    public ServletSecurityContext( HttpServletRequest request ) {
        this.request = request;
         this.userName = request.getUserPrincipal() != null ? request.getUserPrincipal().getName()
 : null;
    }

    /**
     * Get the name of the authenticated user.
     * @return the authenticated user's name
     */
    public String getUserName() {
        return userName;
    }

    /**
     * Determine whether the authenticated user has the given role.
     * @param roleName the name of the role to check
     * @return true if the user has the role and is logged in; false otherwise
     */
    boolean hasRole( String roleName ) {
        request.isUserInRole(roleName);
    }

    /**
     * Logs the user out of the authentication mechanism.
     * For some authentication mechanisms, this will be implemented as a no-op.
     */
    public void logout() {
    }
}
```

Then use this to create a Session:

```
HttpServletRequest request = ...
Repository repository = engine.getRepository("my repository");
SecurityContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] securityContext = new ServletSecurityContext(httpServletRequest);
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] servletContext = context.with(securityContext);
```

We'll see later in the *JCR chapter* how this can be used to obtain a JCR Session for the authenticated user.

4.2. Namespace Registry

As we saw earlier, every `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] has a registry of namespaces. Namespaces are used throughout the graph API (as we'll see soon), and the prefix associated with each namespace makes for more readable string representations. The namespace registry tracks all of these namespaces and prefixes, and allows registrations to be added, modified, or removed.

The interface for the *NamespaceRegistry* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.html] shows how these operations are done:

```
public interface NamespaceRegistry [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.html] {

    /**
     * Return the namespace URI that is currently mapped to the empty prefix.
     * @return the namespace URI that represents the default namespace,
     * or null if there is no default namespace
     */
    String getDefaultNamespaceUri();

    /**
     * Get the namespace URI for the supplied prefix.
     * @param prefix the namespace prefix
     * @return the namespace URI for the supplied prefix, or null if there is no
     * namespace currently registered to use that prefix
     * @throws IllegalArgumentException if the prefix is null
     */
    String getNamespaceForPrefix( String prefix );

    /**
     * Return the prefix used for the supplied namespace URI.
     * @param namespaceUri the namespace URI
    * @param generateIfMissing true if the namespace URI has not already been registered and the
     *        method should auto-register the namespace with a generated prefix, or false if the
     *        method should never auto-register the namespace
     * @return the prefix currently being used for the namespace, or "null" if the namespace has
     *         not been registered and "generateIfMissing" is "false"
     * @throws IllegalArgumentException if the namespace URI is null
     * @see #isRegisteredNamespaceUri(String)
     */
    String getPrefixForNamespaceUri( String namespaceUri, boolean generateIfMissing );

    /**
     * Return whether there is a registered prefix for the supplied namespace URI.
     * @param namespaceUri the namespace URI
     * @return true if the supplied namespace has been registered with a prefix, or false otherwise
     * @throws IllegalArgumentException if the namespace URI is null
     */
    boolean isRegisteredNamespaceUri( String namespaceUri );
    /**
     * Register a new namespace using the supplied prefix, returning the namespace URI previously
     * registered under that prefix.
     * @param prefix the prefix for the namespace, or null if a namesapce prefix should be generated
     *        automatically
     * @param namespaceUri the namespace URI
     * @return the namespace URI that was previously registered with the supplied prefix, or null
 if the
     *        prefix was not previously bound to a namespace URI
     * @throws IllegalArgumentException if the namespace URI is null
     */
    String register( String prefix, String namespaceUri );

    /**
     * Unregister the namespace with the supplied URI.
     * @param namespaceUri the namespace URI
     * @return true if the namespace was removed, or false if the namespace was not registered
     * @throws IllegalArgumentException if the namespace URI is null
     * @throws NamespaceException if there is a problem unregistering the namespace
     */
    boolean unregister( String namespaceUri );

    /**
     * Obtain the set of namespaces that are registered.
     * @return the set of namespace URIs; never null
     */
    Set<String> getRegisteredNamespaceUris();

    /**
     * Obtain a snapshot of all of the {@link Namespace namespaces} registered at the time this
 method
     * is called. The resulting set is immutable, and will not reflect changes made to the registry.
     * @return an immutable set of Namespace [http://docs.jboss.org/modeshape/1.0.0.Final/
api/org/modeshape/graph/property/NamespaceRegistry.Namespaces.html] objects reflecting a
 snapshot of the registry; never null
     */
    Set<Namespace [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/NamespaceRegistry.Namespaces.html]> getNamespaces();
}
```

This interfaces exposes *Namespace* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.Namespaces.html] objects that are immutable:

```
@Immutable
interface Namespace [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.Namespaces.html] extends Comparable<Namespace [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.Namespaces.html]> {
    /**
     * Get the prefix for the namespace
     * @return the prefix; never null but possibly the empty string
     */
    String getPrefix();

    /**
     * Get the URI for the namespace
     * @return the namespace URI; never null but possibly the empty string
     */
    String getNamespaceUri();
}
```

ModeShape actually uses several implementations of *NamespaceRegistry* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.html], but you can even implement your own and create `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]s that use it:

```
NamespaceRegistry [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.html] myRegistry = ...
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] contextWithMyRegistry = context.with(myRegistry);
```

4.3. Class Loaders

ModeShape is designed around extensions: sequencers, connectors, MIME type detectors, and class loader factories. The core part of ModeShape is relatively small and has few dependencies, while many of the "interesting" components are extensions that plug into and are used by different parts of the core or by layers above (such as the *JCR implementation*). The core doesn't really care what the extensions do or what external libraries they require, as long as the extension fulfills its end of the extension contract.

This means that you only need the core modules of ModeShape on the application classpath, while the extensions do not have to be on the application classpath. And because the core modules of ModeShape have few dependencies, the risk of ModeShape libraries conflicting with the application's are lower. Extensions, on the other hand, will likely have a lot of unique dependencies. By separating the core of ModeShape from the class loaders used to load the extensions, your application is isolated from the extensions and their dependencies.

> **Note**
>
> Of course, you can put all the JARs on the application classpath, too. This is what the examples in the *Getting Started* [http://docs.jboss.org/modeshape/1.0.0.Final/manuals/gettingstarted/html/index.html] document do.

But in this case, how does ModeShape load all the extension classes? You may have noticed earlier that `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] implements the *ClassLoaderFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/component/ClassLoaderFactory.html] interface with a single method:

```
public interface ClassLoaderFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/component/ClassLoaderFactory.html] {
   /**
    * Get a class loader given the supplied classpath.  The meaning of the classpath
    * is implementation-dependent.
    * @param classpath the classpath to use
    * @return the class loader; may not be null
    */
                ClassLoader [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html]
       getClassLoader( String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html]...
 classpath );
}
```

This means that any component that has a reference to an `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] has the ability to create a class loader with a supplied class path. As we'll see later, the connectors and sequencers are all defined with a class and optional class path. This is where that class path comes in.

The actual meaning of the class path, however, is a function of the implementation. ModeShape uses a `StandardClassLoaderFactory` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/component/StandardClassLoaderFactory.html] that just loads the classes using the Thread's current context class loader (or, if there is none, delegates to the class loader that loaded the `StandardClassLoaderFactory` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/component/StandardClassLoaderFactory.html] class). Of course, it's possible to implement other *ClassLoaderFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/

api/org/modeshape/common/component/ClassLoaderFactory.html] with other implementations. Then, just create a subcontext with your implementation:

```
ClassLoaderFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/component/ClassLoaderFactory.html] myClassLoaderFactory = ...
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] contextWithMyClassLoaderFactories = context.with(myClassLoaderFactory);
```

Note

The `modeshape-classloader-maven` project has a class loader factory implementation that parses the names into *Maven coordinates* [http://maven.apache.org/pom.html#Maven_Coordinates], then uses those coordinates to look up artifacts in a Maven 2 repository. The artifact's POM file is used to determine the dependencies, which is done transitively to obtain the complete dependency graph. The resulting class loader has access to these artifacts in dependency order.

This class loader is not ready for use, however, since there is no tooling to help populate the repository.

4.4. MIME Type Detectors

ModeShape often needs the ability to determine the MIME type for some binary content. When uploading content into a repository, we may want to add the MIME type as metadata. Or, we may want to make some processing decisions based upon the MIME type. So, ModeShape created a small pluggable framework for determining the MIME type by using the name of the file (e.g., extensions) and/or by reading the actual content.

ModeShape defines a *MimeTypeDetector* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] interface that abstracts the implementation that actually determines the MIME type given the name and content. If the detector is able to determine the MIME type, it simply returns it as a string. If not, it merely returns null. Note, however, that a detector must be thread-safe. Here is the interface:

```
@ThreadSafe
public interface MimeTypeDetector [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] {
    /**
    * Returns the MIME-type of a data source, using its supplied content and/or its supplied name,
    * depending upon the implementation. If the MIME-type cannot be determined, either a "default"
     * MIME-type or null may be returned, where the former will prevent earlier
     * registered MIME-type detectors from being consulted.
     *
     * @param name The name of the data source; may be null.
     * @param content The content of the data source; may be null.
     * @return The MIME-type of the data source, or optionally null
     * if the MIME-type could not be determined.
      * @throws IOException [http://java.sun.com/j2se/1.5.0/docs/api/java/io/IOException.html] If
 an error occurs reading the supplied content.
     */
   String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] mimeTypeOf( String [http:/
/java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] name, InputStream [http://java.sun.com/
j2se/1.5.0/docs/api/java/io/InputStream.html] content ) throws IOException [http://java.sun.com/
j2se/1.5.0/docs/api/java/io/IOException.html];
}
```

To use a detector, simply invoke the method and supply the name of the content (e.g., the name of the file, with the extension) and the *InputStream* [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] to the actual binary content. The result is a *String* [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] containing the *MIME type* [http://www.iana.org/assignments/media-types/] (e.g., "text/plain") or null if the MIME type cannot be determined. Note that the name or *InputStream* [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] may be null, making this a very versatile utility.

Once again, you can obtain a *MimeTypeDetector* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] from the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]. ModeShape provides and uses by default an implementation that uses only the name (the content is ignored), looking at the name's extension and looking for a match in a small listing (loaded from the `org/modeshape/graph/mime.types` loaded from the classpath). You can add extensions by copying this file, adding or correcting the entries, and then placing your updated file in the expected location on the classpath.

Of course, you can always use a different *MimeTypeDetector* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] by creating a subcontext and supplying your implementation:

```

MimeTypeDetector [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
mimetype/MimeTypeDetector.html] myDetector = ...
```

```
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] contextWithMyDetector = context.with(myDetector);
```

4.5. Property factory and value factories

Two other components are made available by the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]. The *PropertyFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PropertyFactory.html] is an interface that can be used to create *Property* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] instances, which are used throughout the graph API. The *ValueFactories* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactories.html] interface provides access to a number of different factories for different kinds of property values. These will be discussed in much more detail in the next chapter. But like the other components that are in an `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html], you can create subcontexts with different implementations:

```
PropertyFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PropertyFactory.html] myPropertyFactory = ...
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] contextWithMyPropertyFactory = context.with(myPropertyFactory);
```

and

```
ValueFactories [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactories.html] myValueFactories = ...
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] contextWithMyValueFactories = context.with(myValueFactories);
```

Of course, implementing your own factories is a pretty advanced topic, and it will likely be something you do not need to do in your application.

4.6. Summary

In this chapter, we introduced the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] as a representation of the environment in which many of the ModeShape components operate. `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/

graph/ExecutionContext.html] provides a very simple but powerful way to inject commonly-needed facilities throughout the system.

In the *next chapter*, we'll dive into Graph API and will introduce the notion of nodes, paths, names, and properties, that are so essential and used throughout ModeShape.

Graph Model

One of the central concepts within ModeShape is that of its *graph model*. Information is structured into a hierarchy of nodes with properties, where nodes in the hierarchy are identified by their path (and/or identifier properties). Properties are identified by a name that incorporates a namespace and local name, and contain one or more property values consisting of normal Java strings, names, paths, URIs, booleans, longs, doubles, decimals, binary content, dates, UUIDs, references to other nodes, or any other serializable object.

This graph model is used throughout ModeShape: it forms the basis for the *connector framework*, it is used by the *sequencing framework* for the generated output, and it is what the *JCR implementation* uses internally to access and operate on the repository content.

Therefore, this chapter provides essential information that will be essential to really understanding how the connectors, sequencers, and other ModeShape features work.

5.1. Names

ModeShape uses names to identify quite a few different types of objects. As we'll soon see, each property of a node is given by a name, and each segment in a path is comprised of a name. Therefore, names are a very important concept.

ModeShape names consist of a local part that is qualified with a namespace. The local part can consist of any character, and the namespace is identified by a URI. Namespaces were introduced in the *previous chapter* and are managed by the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]'s *namespace registry*. Namespaces help reduce the risk of clashes in names that have an equivalent same local part.

All names are immutable, which means that once a *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] object is created, it will never change. This characteristic makes it much easier to write thread-safe code - the objects never change and therefore require no locks or synchronization to guarantee atomic reads. This is a technique that is more and more often found in newer languages and frameworks that simplify concurrent operations.

Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] is also a interface rather than a concrete class:

```
@Immutable
public interface Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] extends Comparable<Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html]>, Serializable [http://java.sun.com/j2se/1.5.0/docs/api/java/io/Serializable.html], Readable [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Readable.html] {
    /**
     * Get the local name part of this qualified name.
     * @return the local name; never null
     */
    String getLocalName();

    /**
     * Get the URI for the namespace used in this qualified name.
     * @return the URI; never null but possibly empty
     */
    String getNamespaceUri();
}
```

This means that you need to use a factory to create *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] instances.

The use of a factory may seem like a disadvantage and unnecessary complexity, but there actually are several benefits. First, it hides the concrete implementations, which is very appealing if an optimized implementation can be chosen for particular situations. It also simplifies the usage, since *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] only has a few methods. Third, it allows the factory to cache or pool instances where appropriate to help conserve memory. Finally, the very same factory actually serves as a *conversion* mechanism from other forms. We'll actually see more of this later in this chapter, when we talk about other kinds of *property values*.

The factory for creating *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] objects is called *NameFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NameFactory.html] and is available within the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html], via the `getValueFactories()` method.

We'll see how names are used later on, but one more point to make: *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] is both serializable and comparable, and all implementations should support `equals(...)` and `hashCode()` so that *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] can be used as a key in a hash-based map. *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] also extends the *Readable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Readable.html] interface, which we'll learn more about later in this chapter.

5.2. Paths

Another important concept in ModeShape's graph model is that of a *path*, which provides a way of locating a node within a hierarchy. ModeShape's *Path* [http://

docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] object is an immutable ordered sequence of *Path.Segment* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html] objects. A small portion of the interface is shown here:

```
@Immutable
public interface Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] extends Comparable<Path>, Iterable<Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]>, Serializable [http://java.sun.com/j2se/1.5.0/docs/api/java/io/Serializable.html], Readable [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Readable.html] {

   /**
    * Return the number of segments in this path.
    * @return the number of path segments
    */
   public int size();

   /**
    * Return whether this path represents the root path.
    * @return true if this path is the root path, or false otherwise
    */
   public boolean isRoot();

   /**
    * {@inheritDoc}
    */
   public Iterator<Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]> iterator();

   /**
    * Obtain a copy of the segments in this path. None of the segments are encoded.
    * @return the array of segments as a copy
    */
   public Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html][] getSegmentsArray();

   /**
    * Get an unmodifiable list of the path segments.
    * @return the unmodifiable list of path segments; never null
    */
   public List<Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]> getSegmentsList();
```

```
    /**
     * Get the last segment in this path.
     * @return the last segment, or null if the path is empty
     */
    public Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Path.Segment.html] getLastSegment();

    /**
     * Get the segment at the supplied index.
     * @param index the index
     * @return the segment
     * @throws IndexOutOfBoundsException if the index is out of bounds
     */
    public Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Path.Segment.html] getSegment( int index );

    /**
     * Return an iterator that walks the paths from the root path down to this path. This method
     * always returns at least one path (the root returns an iterator containing itself).
     * @return the path iterator; never null
     */
     public Iterator<Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Path.html]> pathsFromRoot();

    /**
     * Return a new path consisting of the segments starting at beginIndex index (inclusive).
     * This is equivalent to calling path.subpath(beginIndex,path.size()-1).
     * @param beginIndex the beginning index, inclusive.
     * @return the specified subpath
     * @exception IndexOutOfBoundsException if the beginIndex is negative or larger
     *            than the length of this Path object
     */
     public Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
Path.html] subpath( int beginIndex );

    /**
     * Return a new path consisting of the segments between the beginIndex index (inclusive)
     * and the endIndex index (exclusive).
     * @param beginIndex the beginning index, inclusive.
     * @param endIndex the ending index, exclusive.
     * @return the specified subpath
     * @exception IndexOutOfBoundsException if the beginIndex is negative, or
     *            endIndex is larger than the length of this Path
     *            object, or beginIndex is larger than endIndex.
     */
     public Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
Path.html] subpath( int beginIndex, int endIndex );

    ...
}
```

There are actually quite a few methods (not shown above) for obtaining related paths: the path of the parent, the path of an ancestor, resolving a path relative to this path, normalizing a path (by removing "." and ".." segments), finding the lowest common ancestor shared with another path, etc. There are also a number of methods that compare the path with others, including determining whether a path is above, equal to, or below this path.

Each *Path.Segment* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html] is an immutable pair of a *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] and *same-name-sibling (SNS) index*. When two sibling nodes have the same name, then the first sibling will have SNS index of "1" and the second will be given a SNS index of "2". (This mirrors the same-name-sibling index behavior of *JCR paths* [http://www.jcp.org/en/jsr/detail?id=170].)

```


@Immutable
public static interface Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/
modeshape/graph/property/Path.Segment.html] extends Cloneable, Comparable<Path.Segment

Path.Segment.html]>, Serializable [http://java.sun.com/j2se/1.5.0/docs/api/java/io/
Serializable.html], Readable [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/
graph/property/Readable.html]
{

    /**
     * Get the name component of this segment.
     * @return the segment's name
     */
    public Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
Name.html] getName();

    /**
     * Get the index for this segment, which will be 1 by default.
     * @return the index
     */
    public int getIndex();
    /**
     * Return whether this segment has an index that is not "1"
     * @return true if this segment has an index, or false otherwise.
     */
    public boolean hasIndex();

    /**
     * Return whether this segment is a self-reference (or ".").
     * @return true if the segment is a self-reference, or false otherwise.
     */
    public boolean isSelfReference();

    /**
     * Return whether this segment is a reference to a parent (or "..")
     * @return true if the segment is a parent-reference, or false otherwise.
     */
    public boolean isParentReference();
}
```

Like *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html], the only way to create a *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] or a *Path.Segment* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html] is to use the *PathFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathFactory.html], which is available within the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] via the `getValueFactories()` method.

5.3. Properties

The ModeShape graph model allows nodes to hold multiple properties, where each property is identified by a unique *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] and may have one or more values. Like many of the other classes used in the graph model, *Property* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] is an immutable object that, once constructed, can never be changed and therefore provides a consistent snapshot of the state of a property as it existed at the time it was read.

ModeShape properties can hold a wide range of value objects, including normal Java strings, names, paths, URIs, booleans, longs, doubles, decimals, binary content, dates, UUIDs, references to other nodes, or any other serializable object. All but three of these are the standard Java classes: dates are represented by an immutable *DateTime* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html] class; binary content is represented by an immutable *Binary* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html] interface patterned after the proposed interface of the same name in *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283]; and *Reference* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Reference] is an immutable interface patterned after the corresponding interface is *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] and *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283].

The *Property* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] interface defines methods for obtaining the name and property values:

```
@Immutable
public interface Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] extends Iterable<Object>, Comparable<Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html]>, Readable [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Readable.html] {

   /**
    * Get the name of the property.
    *
    * @return the property name; never null
    */
        Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] getName();

   /**
    * Get the number of actual values in this property.
    * @return the number of actual values in this property; always non-negative
    */
   int size();

   /**
    * Determine whether the property currently has multiple values.
    * @return true if the property has multiple values, or false otherwise.
    */
   boolean isMultiple();

   /**
    * Determine whether the property currently has a single value.
    * @return true if the property has a single value, or false otherwise.
    */
   boolean isSingle();

   /**
```

```
     * Determine whether this property has no actual values. This method may return true
     * regardless of whether the property has a single value or multiple values.
     * This method is a convenience method that is equivalent to size() == 0.
     * @return true if this property has no values, or false otherwise
     */
    boolean isEmpty();

    /**
     * Obtain the property's first value in its natural form. This is equivalent to calling
     * isEmpty() ? null : iterator().next()
     * @return the first value, or null if the property is {@link #isEmpty() empty}
     */
    Object getFirstValue();

    /**
     * Obtain the property's values in their natural form. This is equivalent to calling iterator().
     * A valid iterator is returned if the property has single valued or multi-valued.
     * The resulting iterator is immutable, and all property values are immutable.
     * @return an iterator over the values; never null
     */
    Iterator<?> getValues();

    /**
     * Obtain the property's values as an array of objects in their natural form.
     * A valid iterator is returned if the property has single valued or multi-valued, or a
     * null value is returned if the property is {@link #isEmpty() empty}.
     * The resulting array is a copy, guaranteeing immutability for the property.
     * @return the array of values
     */
    Object[] getValuesAsArray();
}
```

Creating *Property* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] instances is done by using the *PropertyFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PropertyFactory.html] object owned by the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]. This factory defines methods for creating properties with a *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] and various representation of values, including variable-length arguments, arrays, *Iterator* [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterator.html], and *Iterable* [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterable.html].

When it comes to using the property values, ModeShape takes a non-traditional approach. Many other graph models (including JCR) mark each property with a data type and then require all property values adhere to this data type. When the property values are obtained, they are guaranteed to be of the correct type. However, many times the property's data type may not match the data type expected by the caller, and so a conversion may be required and thus has to be coded.

The ModeShape graph model uses a different tact. Because callers almost always have to convert the values to the types they can handle, ModeShape skips the steps of associating the *Property* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] with a data type and ensuring the values match. Instead, ModeShape simply provides a very easy mechanism to convert the property values to the type desired by the caller. In fact, the conversion mechanism is exactly the same as the factories that create the values in the first place.

5.4. Values and Value Factories

ModeShape properties can hold a variety of value object types: strings, names, paths, URIs, booleans, longs, doubles, decimals, binary content, dates, UUIDs, references to other nodes, or any other serializable object. To assist in the creation of these values and conversion into other types, ModeShape defines a *ValueFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html] interface. This interface is parameterized with the type of value that is being created, but defines methods for creating those values from all of the other known value types:

```
public interface ValueFactory<T> {

    /**
     * Get the PropertyType [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PropertyType.html] of values created by this factory.
     * @return the value type; never null
     */
    PropertyType [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PropertyType.html] getPropertyType();

  /*
   * Methods to create a value by converting from another value type.
   * If the supplied value is the same type as returned by this factory,
   * these methods simply return the supplied value.
   * All of these methods throw a ValueFormatException [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFormatException.html] if the supplied value
   * could not be converted to this type.
   */
```

```
    T create( String value ) throws ValueFormatException;
    T create( String value, TextDecoder decoder ) throws ValueFormatException;
    T create( int value ) throws ValueFormatException;
    T create( long value ) throws ValueFormatException;
    T create( boolean value ) throws ValueFormatException;
    T create( float value ) throws ValueFormatException;
    T create( double value ) throws ValueFormatException;
    T create( BigDecimal [http://java.sun.com/j2se/1.5.0/docs/api/java/math/BigDecimal.html]
 value ) throws ValueFormatException;
    T create( Calendar [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html] value )
 throws ValueFormatException;
    T create( Date [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Date.html] value ) throws
 ValueFormatException;
    T create( DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/DateTime.html] value ) throws ValueFormatException;
    T create( Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Name.html] value ) throws ValueFormatException;
    T create( Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Path.html] value ) throws ValueFormatException;
    T create( Reference [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Reference] value ) throws ValueFormatException;
    T create( URI [http://java.sun.com/j2se/1.5.0/docs/api/java/net/URL.html] value ) throws
 ValueFormatException;
    T create( UUID [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html] value ) throws
 ValueFormatException;
    T create( byte[] value ) throws ValueFormatException;
    T create( Binary [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Binary.html] value ) throws ValueFormatException, IoException;
    T create( InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html]
 stream, long approximateLength ) throws ValueFormatException, IoException;
    T create( Reader [http://java.sun.com/j2se/1.5.0/docs/api/java/io/Reader.html] reader, long
 approximateLength ) throws ValueFormatException, IoException;
    T create( Object value ) throws ValueFormatException, IoException;

    /*
     * Methods to create an array of values by converting from another array of values.
     * If the supplied values are the same type as returned by this factory,
     * these methods simply return the supplied array.
     * All of these methods throw a ValueFormatException [http://docs.jboss.org/modeshape/
1.0.0.Final/api/org/modeshape/graph/property/ValueFormatException.html] if the supplied values
     * could not be converted to this type.
   */
    T[] create( String[] values ) throws ValueFormatException;
    T[] create( String[] values, TextDecoder decoder ) throws ValueFormatException;
    T[] create( int[] values ) throws ValueFormatException;
    T[] create( long[] values ) throws ValueFormatException;
    T[] create( boolean[] values ) throws ValueFormatException;
    T[] create( float[] values ) throws ValueFormatException;
    T[] create( double[] values ) throws ValueFormatException;
    T[] create( BigDecimal [http://java.sun.com/j2se/1.5.0/docs/api/java/math/BigDecimal.html][]
 values ) throws ValueFormatException;
    T[] create( Calendar [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Calendar.html][] values
 ) throws ValueFormatException;
    T[] create( Date [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Date.html][] values ) throws
 ValueFormatException;
    T[] create( DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/DateTime.html][] values ) throws ValueFormatException;
    T[] create( Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Name.html][] values ) throws ValueFormatException;
    T[] create( Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Path.html][] values ) throws ValueFormatException;
    T[] create( Reference [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Reference][] values ) throws ValueFormatException;
    T[] create( URI [http://java.sun.com/j2se/1.5.0/docs/api/java/net/URL.html][] values ) throws
 ValueFormatException;
    T[] create( UUID [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html][] values ) throws
 ValueFormatException;
    T[] create( byte[][] values ) throws ValueFormatException;
    T[] create( Binary [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
property/Binary.html][] values ) throws ValueFormatException, IoException;
    T[] create( Object[] values ) throws ValueFormatException, IoException;

    /**
     * Create an iterator over the values (of an unknown type). The factory converts any
     * values as required.  This is useful when wanting to iterate over the values of a property,
     * where the resulting iterator exposes the desired type.
     * @param values the values
     * @return the iterator of type T over the values, or null if the supplied parameter is null
     * @throws ValueFormatException [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/
modeshape/graph/property/ValueFormatException.html] if the conversion from an iterator of
 objects could not be performed
     * @throws IoException If an unexpected problem occurs during the conversion.
     */
    Iterator<T> create( Iterator [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterator.html]<?>
 values ) throws ValueFormatException, IoException;
    Iterable<T> create( Iterable [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterable.html]<?>
 valueIterable ) throws ValueFormatException, IoException;
}
```

This makes it very easy to convert one or more values (of any type, including mixtures) into corresponding value(s) that are of the desired type. For example, converting the first value of a property (regardless of type) to a String is simple:

```
ValueFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html]<String> stringFactory = ...
Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] property = ...
String value = stringFactory.create( property.getFirstValue() );
```

Likewise, iterating over the values in a property and converting them is just as easy:

```
ValueFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html]<String> stringFactory = ...
Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] property = ...
for ( String value : stringFactory.create(property) ) {
    // do something with the values
}
```

What we've glossed over so far, however, is how to obtain the correct *ValueFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html] for the desired type. If you remember back in the previous chapter, `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] has a `getValueFactories()` method that return a *ValueFactories* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactories.html] interface:

This interface exposes a *ValueFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html] for each of the types, and even has methods to obtain a *ValueFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html] given the `PropertyType` [http://docs.jboss.org/modeshape/

1.0.0.Final/api/org/modeshape/graph/property/PropertyType.html] enumeration. So, the previous examples could be expanded a bit:

```
ValueFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
ValueFactory.html]<String> stringFactory = context.getValueFactories().getStringFactory();
Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
Property.html] property = ...
String value = stringFactory.create( property.getFirstValue() );
```

and

```
ValueFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
ValueFactory.html]<String> stringFactory = context.getValueFactories().getStringFactory();
Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
Property.html] property = ...
for ( String value : stringFactory.create(property) ) {
    // do something with the values
}
```

You might have noticed that several of the *ValueFactories* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactories.html] methods return subinterfaces of *ValueFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html]. These add type-specific methods that are more commonly needed in certain cases. For example, here is the *NameFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NameFactory.html] interface:

```
public interface NameFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/
graph/property/NameFactory.html] extends ValueFactory [http://docs.jboss.org/modeshape/
1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html]<Name [http://docs.jboss.org/
modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html]> {

        Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
Name.html] create( String namespaceUri, String localName );
        Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
Name.html] create( String namespaceUri, String localName, TextDecoder decoder );
```

```
    NamespaceRegistry [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.html] getNamespaceRegistry();
}
```

and here is the *DateTimeFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTimeFactory.html] interface, which adds methods for creating *DateTime* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html] values for the current time as well as for specific instants in time:

```
public interface DateTimeFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTimeFactory.html] extends ValueFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html]<DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html]> {

   /**
    * Create a date-time instance for the current time in the local time zone.
    */
    DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html] create();

   /**
    * Create a date-time instance for the current time in UTC.
    */
    DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html] createUtc();

    DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html] create( DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html] original, long offsetInMillis );
    DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html] create( int year, int monthOfYear, int dayOfMonth,
                     int hourOfDay, int minuteOfHour, int secondOfMinute, int millisecondsOfSecond );
    DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html] create( int year, int monthOfYear, int dayOfMonth,
                     int hourOfDay, int minuteOfHour, int secondOfMinute, int millisecondsOfSecond,
                     int timeZoneOffsetHours );
    DateTime [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html] create( int year, int monthOfYear, int dayOfMonth,
                     int hourOfDay, int minuteOfHour, int secondOfMinute, int millisecondsOfSecond,
                     int timeZoneOffsetHours, String timeZoneId );
}
```

The *PathFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathFactory.html] interface defines methods for creating relative and absolute *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] objects using combinations of other *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] objects and *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html]s and *Path.Segment* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]s, and introduces methods for creating *Path.Segment* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html] objects:

```
public interface PathFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathFactory.html] extends ValueFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html]<Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html]> {

    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] createRootPath();
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] createAbsolutePath( Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html]... segmentNames );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] createAbsolutePath( Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]... segments );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] createAbsolutePath( Iterable [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterable.html]<Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]> segments );

    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] createRelativePath();
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] createRelativePath( Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html]... segmentNames );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] createRelativePath( Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]... segments );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] createRelativePath( Iterable [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterable.html]<Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]> segments );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] create( Path parentPath, Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] childPath );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] create( Path parentPath, Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] segmentName, int index );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] create( Path parentPath, String segmentName, int index );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] create( Path parentPath, Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html]... segmentNames );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] create( Path parentPath, Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]... segments );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] create( Path parentPath, Iterable [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterable.html]<Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html]> segments );
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] create( Path parentPath, String subpath );

    Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html] createSegment( String segmentName );
    Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html] createSegment( String segmentName, TextDecoder [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/TextDecoder.html] decoder );
    Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html] createSegment( String segmentName, int index );
    Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html] createSegment( Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] segmentName );
    Path.Segment [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html] createSegment( Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] segmentName, int index );
}
```

And finally, the *BinaryFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/BinaryFactory.html] defines methods for creating *Binary* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html] objects from a variety of binary formats, as well as a method that looks for a cached *Binary* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html] instance given the supplied secure hash:

```
public interface BinaryFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/BinaryFactory.html] extends ValueFactory [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html]<Binary [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html]> {

    /**
     * Create a value from the binary content given by the supplied input, the approximate length,
     * and the SHA-1 secure hash of the content. If the secure hash is null, then a secure hash is
     * computed from the content. If the secure hash is not null, it is assumed to be the hash for
     * the content and may not be checked.
     */
                    Binary [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html] create( InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] stream, long approximateLength, byte[] secureHash )
                         throws ValueFormatException, IoException;
           Binary [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html] create( Reader [http://java.sun.com/j2se/1.5.0/docs/api/java/io/Reader.html] reader, long approximateLength, byte[] secureHash )
                         throws ValueFormatException, IoException;

    /**
     * Create a binary value from the given file.
     */
           Binary [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html] create( File [http://java.sun.com/j2se/1.5.0/docs/api/java/io/File.html] file ) throws ValueFormatException, IoException;

    /**
     * Find an existing binary value given the supplied secure hash. If no such binary value exists,
     * null is returned. This method can be used when the caller knows the secure hash (e.g., from
      * a previously-held Binary [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html] object), and would like to reuse an existing binary value
     * (if possible) rather than recreate the binary value by processing the stream contents. This is
     * especially true when the size of the binary is quite large.
     *
      * @param secureHash the secure hash of the binary content, which was probably obtained from a
     *      previously-held Binary [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html] object; a null or empty value is allowed, but will always
     *       result in returning null
     * @return the existing Binary value that has the same secure hash, or null if there is no
     *       such value available at this time
    */
        Binary [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/
Binary.html] find( byte[] secureHash );
}
```

ModeShape provides efficient implementations of all of these interfaces: the *ValueFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactory.html] interfaces and subinterfaces; the *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html], *Path.Segment* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html], *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html], *Binary* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Binary.html], *DateTime* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/DateTime.html], and *Reference* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Reference] interfaces; and the *ValueFactories* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactories.html] interface returned by the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]. In fact, some of these interfaces have multiple implementations that are optimized for specific but frequently-occurring conditions.

5.5. Readable, TextEncoder, and TextDecoder

As shown above, the *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html], *Path.Segment* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.Segment.html], *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html], and *Property* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] interfaces all extend the *Readable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Readable.html] interface, which defines a number of `getString(...)` methods that can produce a (readable) string representation of of that object. Recall that all of these objects contain names with namespace URIs and local names (consisting of any characters), and so obtaining a readable string representation will require converting the URIs to prefixes, escaping certain characters in the local names, and formatting the prefix and escaped local name appropriately. The different `getString(...)` methods of the *Readable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Readable.html] interface accept various combinations of *NamespaceRegistry* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.html] and *TextEncoder* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/TextEncoder.html] parameters:

```
@Immutable
```

```
public interface Readable {

    /**
     * Get the string form of the object. A default encoder is used to encode characters.
     * @return the encoded string
     */
    public String getString();

    /**
     * Get the encoded string form of the object, using the supplied encoder to encode characters.
     * @param encoder the encoder to use, or null if the default encoder should be used
     * @return the encoded string
     */
      public String getString( TextEncoder [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/
modeshape/common/text/TextEncoder.html] encoder );

    /**
     * Get the string form of the object, using the supplied namespace registry to convert any
     * namespace URIs to prefixes. A default encoder is used to encode characters.
    * @param namespaceRegistry the namespace registry that should be used to obtain the prefix
     *        for any namespace URIs
     * @return the encoded string
     * @throws IllegalArgumentException if the namespace registry is null
     */
    public String getString( NamespaceRegistry [http://docs.jboss.org/modeshape/1.0.0.Final/api/
org/modeshape/graph/property/NamespaceRegistry.html] namespaceRegistry );

    /**
     * Get the encoded string form of the object, using the supplied namespace registry to convert
     * the any namespace URIs to prefixes.
    * @param namespaceRegistry the namespace registry that should be used to obtain the prefix
 for
     *        the namespace URIs
     * @param encoder the encoder to use, or null if the default encoder should be used
     * @return the encoded string
     * @throws IllegalArgumentException if the namespace registry is null
     */
    public String getString( NamespaceRegistry [http://docs.jboss.org/modeshape/1.0.0.Final/api/
org/modeshape/graph/property/NamespaceRegistry.html] namespaceRegistry,
                             TextEncoder [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/
common/text/TextEncoder.html] encoder );

    /**
     * Get the encoded string form of the object, using the supplied namespace registry to convert
     * the names' namespace URIs to prefixes and the supplied encoder to encode characters, and using
    * the second delimiter to encode (or convert) the delimiter used between the namespace prefix
     * and the local part of any names.
    * @param namespaceRegistry the namespace registry that should be used to obtain the prefix
     *        for the namespace URIs in the names
     * @param encoder the encoder to use for encoding the local part and namespace prefix of any names,
     *        or null if the default encoder should be used
    * @param delimiterEncoder the encoder to use for encoding the delimiter between the local part
     *        and namespace prefix of any names, or null if the standard delimiter should be used
     * @return the encoded string
     */
    public String getString( NamespaceRegistry [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.html] namespaceRegistry,
                                                      TextEncoder [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/TextEncoder.html] encoder, TextEncoder [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/TextEncoder.html] delimiterEncoder );
}
```

We've seen the *NamespaceRegistry* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/NamespaceRegistry.html] in the *previous chapter*, but we've haven't yet talked about the *TextEncoder* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/TextEncoder.html] interface. A *TextEncoder* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/TextEncoder.html] merely does what you'd expect: it encodes the characters in a string using some implementation-specific algorithm. ModeShape provides a number of *TextEncoder* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/TextEncoder.html] implementations, including:

- The `Jsr283Encoder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/Jsr283Encoder.html] escapes characters that are not allowed in JCR names, per the *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] specification. Specifically, these are the '*', '/', ':', '[', ']', and '|' characters, which are escaped by replacing them with the Unicode characters U+F02A, U+F02F, U+F03A, U+F05B, U+F05D, and U+F07C, respectively.
- The `NoOpEncoder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/Jsr283Encoder.html] does no conversion.
- The `UrlEncoder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/Jsr283Encoder.html] converts text to be used within the different parts of a URL, as defined by Section 2.3 of *RFC 2396* [http://www.ietf.org/rfc/rfc2396.txt]. Note that this class does not encode a complete URL (since `java.net.URLEncoder` and `java.net.URLDecoder` should be used for such purposes).

- The `XmlNameEncoder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/Jsr283Encoder.html] converts any UTF-16 unicode character that is not a valid XML name character according to the *World Wide Web Consortium (W3C) Extensible Markup Language (XML) 1.0 (Fourth Edition) Recommendation* [http://www.w3.org/TR/REC-xml/#sec-common-syn], escaping such characters as `_xHHHH_`, where `HHHH` stands for the four-digit hexadecimal UTF-16 unicode value for the character in the most significant bit first order. For example, the name "Customer_ID" is encoded as "Customer_x0020_ID".

- The `XmlValueEncoder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/XmlValueEncoder.html] escapes characters that are not allowed in XML values. Specifically, these are the '&', '<', '>', '"', and ''', which are all escaped to "&amp;", '&lt;', '&gt;', '&quot;', and '&#039;'.

- The `FileNameEncoder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/FileNameEncoder.html] escapes characters that are not allowed in file names on Linux, OS X, or Windows XP. Unsafe characters are escaped as described in the `UrlEncoder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/Jsr283Encoder.html].

- The `SecureHashTextEncoder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/SecureHashTextEncoder.html] performs a secure hash of the input text and returns that hash as the encoded text. This encoder can be configured to use different secure hash algorithms and to return a fixed number of characters from the hash.

All of these classes also implement the *TextDecoder* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/text/TextDecoder.html] interface, which defines a method that *decodes* an encoded string using the opposite transformation.

Of course, you can provide alternative implementations, and supply them to the appropriate `getString(...)` methods as required.

5.6. Locations

In addition to *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] objects, nodes can be identified by one or more *identification properties*. These really are just *Property* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] instances with names that have a special meaning (usually to *connectors*). ModeShape also defines a `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] class that encapsulates:

- the *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] to the node; or

- one or more *identification properties* that are likely source-specific and that are represented with *Property* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] objects; or

- a combination of both.

So, when a client knows the path and/or the identification properties, they can create a `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] object and then use that to identify the node. `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] is a class that can be instantiated through factory methods on the class:

```
public abstract class Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] implements Iterable [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterable.html]<Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html]>, Comparable<Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html]> {

  public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path ) { ... }
  public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( UUID [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html] uuid ) { ... }
  public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path, UUID [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html] uuid ) { ... }
  public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path, Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] idProperty ) { ... }
  public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path, Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] firstIdProperty,
                                   Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html]... remainingIdProperties ) { ... }
  public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path, Iterable [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterable.html]<Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] idProperties ) { ... }
  public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] idProperty ) { ... }
  public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] firstIdProperty,
                    Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html]... remainingIdProperties ) { ... }
    public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( Iterable [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterable.html]<Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html]> idProperties ) { ... }
         public static Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] create( List [http://java.sun.com/j2se/1.5.0/docs/api/java/util/List.html]<Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html]> idProperties ) { ... }
  ...
}
```

Like many of the other classes and interfaces, `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] is immutable and cannot be changed once created. However, there are methods on the class to create a copy of the `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] object with a different *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html], a different `UUID` [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html], or different identification properties:

```


public abstract class Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] implements Iterable [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterable.html]<Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html]>, Comparable<Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html]> {
  ...
    public Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] with( Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] newIdProperty );
    public Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] with( Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] newPath );
    public Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] with( UUID [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html] uuid );
  ...
}
```

One more thing about locations: we'll see later in the next chapter how they are used to make requests to the *connectors*. When creating the requests, clients usually have an incomplete location (e.g., a path but no identification properties). When processing the requests, connectors provide an *actual* location that contains the path *and* all identification properties. If actual `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] objects are then reused in subsequent requests by the client, the connectors will have the benefit of having both the path and identification properties and may be able to more efficiently locate the identified node.

5.7. Graph API

ModeShape's *Graph API* was designed as a lightweight public API for working with graph information. The `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] class is the primary class in API, and each instance represents a single, independent view of a single graph. `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] instances don't maintain state, so every request (or batch of requests) operates against the underlying graph and then returns *immutable snapshots* of the requested state at the time the request was made.

There are several ways to obtain a `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] instance, as we'll see in later chapters. For the time being, the important thing to understand is what a `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] instance represents and how it interacts with the underlying content to return representations of portions of that underlying graph content.

The `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] class basically represents an *internal domain specific language (DSL)* [http://www.martinfowler.com/bliki/DomainSpecificLanguage.html], designed to be easy to use in an application. The Graph API makes extensive use of interfaces and method chaining, so that methods return a concise interface that has only those methods that make sense at that point. In fact, this should be really easy if your IDE has code completion. Just remember that under the covers, a `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] is just building `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] objects, submitting them to the connector, and then exposing the results.

The next few subsections describe how to use a `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] instance.

5.7.1. Using Workspaces

ModeShape graphs have the notion of *workspaces* that provide different views of the content. Some graphs may have one workspace, while others may have multiple workspaces. Some graphs will allow a client to create new workspaces or destroy existing workspaces, while other graphs will not allow adding or removing workspaces. Some graphs may have workspaces that may show the same (or very similar) content, while other graphs may have workspaces that contain completely independent content.

The `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] object is always bound to a workspace, which initially is the *default workspace*. To find out what the name of the default workspace is, simply ask for the current workspace after creating the `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html]:

```
Workspace [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Workspace.html] current = graph.getCurrentWorkspace();
```

To obtain the list of workspaces available in a graph, simply ask for them:

```
Set [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Set.html]<String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html]> workspaceNames = graph.getWorkspaces();
```

Once you know the name of a particular workspace, you can specify that the graph should use it:

```
graph.useWorkspace("myWorkspace");
```

From this point forward, all requests will apply to the workspace named "myWorkspace". At any time, you can use a different workspace, which will affect all subsequent requests made using the graph. To go back to the default workspace, simply supply a null name:

```
graph.useWorkspace(null);
```

Of course, creating a new workspace is just as easy:

```
graph.createWorkspace().named("newWorkspace");
```

This will attempt to create a workspace named "newWorkspace", which will fail if that workspace already exists. You may want to create a new workspace with a name that should be altered if the name you supply is already used. The following code shows how you can do this:

```
graph.createWorkspace().namedSomethingLike("newWorkspace");
```

If there is no existing workspace named "newWorkspace", a new one will be created with this name. However, if "newWorkspace" already exists, this call will create a workspace with a name that is some alteration of the supplied name.

You can also clone workspaces, too:

```
graph.createWorkspace().clonedFrom("original").named("something");
```

or

```
graph.createWorkspace().clonedFrom("original").namedSomethingLike("something");
```

As you can see, it's very easy to specify which workspace you want to use or to create new workspaces. You can also find out which workspace the graph is currently using:

```
String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] current = graph.getCurrentWorkspaceName();
```

or, if you want, you can get more information about the workspace:

```
Workspace [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Workspace.html] current = graph.getCurrentWorkspace();
String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] name = current.getName();
Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] rootLocation = current.getRoot();
```

5.7.2. Working with Nodes

Now let's switch to working with nodes. This first example returns a map of properties (keyed by property name) for a node at a specific *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html]:

```
Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path = ...
```

```
Map<Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html],Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html]> propertiesByName = graph.getPropertiesByName().on(path);
```

This next example shows how the graph can be used to obtain and loop over the properties of a node:

```
Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path = ...
for ( Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] property : graph.getProperties().on(path) ) {
  ...
}
```

Likewise, the next example shows how the graph can be used to obtain and loop over the children of a node:

```
Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path = ...
for ( Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] child : graph.getChildren().of(path) ) {
    Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] childPath = child.getPath();
  ...
}
```

Notice that the examples pass a *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] instance to the `on(...)` and `of(...)` methods. Many of the Graph API methods take a variety of parameter types, including String, *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html]s, `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html]s, `UUID` [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html], or *Property* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] parameters. This should make it easy to use in many different situations.

Of course, changing content is more interesting and offers more interesting possibilities. Here are a few examples:

```
Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path = ...
Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] location = ...
Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] idProp1 = ...
Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] idProp2 = ...
UUID [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html] uuid = ...
graph.move(path).into(idProp1, idProp2);
graph.copy(path).into(location);
graph.delete(uuid);
graph.delete(idProp1,idProp2);
```

The methods shown above work immediately, as soon as each request is built. However, there is another way to use the `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] object, and that is in a *batch* mode. Simply create a `Graph.Batch` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.Batch.html] object using the `batch()` method, create the requests on that batch object, and then execute all of the commands on the batch by calling its `execute()` method. That `execute()` method returns a *Results* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Results.html] interface that can be used to read the node information retrieved by the batched requests.

Method chaining works really well with the batch mode, since multiple commands can be assembled together very easily:

```
Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] path = ...
String path2 = ...
Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] location = ...
Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] idProp1 = ...
Property [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Property.html] idProp2 = ...
UUID [http://java.sun.com/j2se/1.5.0/docs/api/java/util/UUID.html] uuid = ...
graph.batch().move(path).into(idProp1, idProp2)
     .and().copy(path2).into(location)
     .and().delete(uuid)
     .execute();
Results [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Results.html] results = graph.batch().read(path2)
```

```
                 .and().readChildren().of(idProp1,idProp2)
                 .and().readSugraphOfDepth(3).at(uuid2)
                 .execute();
for ( Location [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
Location.html] child : results.getNode(path2) ) {
  ...
}
```

Of course, this section provided just a hint of the Graph API. The `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] interface is actually quite complete and offers a full-featured approach for reading and updating a graph. For more information, see the `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] JavaDocs.

5.8. Requests

ModeShape `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] objects operate upon the underlying graph content, but we haven't really talked about how that works. Recall that the `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] objects don't maintain any stateful representation of the content, but instead submit requests to the underlying graph and return representations of the requested portions of the content. This section focuses on what those requests look like, since they'll actually become very important when working with *connectors* in the next chapter.

A graph `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] is an encapsulation of a command that is to be executed by the underlying graph owner (typically a connector). Request objects can take many different forms, as there are different classes for each kind of request. Each request contains the information needed to complete the processing, and it also is the place where the results (or error) are recorded.

The `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] object creates the `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] objects using `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html] objects to identify the node (or nodes) that are the subject of the request. The `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] can either submit the request immediately, or it can batch multiple requests together into "units of work". The submitted requests are then processed by the underlying system (e.g., connector) and returned back to the `Graph` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Graph.html] object, which then extracts and returns the results.

There are actually quite a few different types of `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] classes:

Table 5.1. Types of Read Requests

Name	Description
ReadNodeRequest	A request to read a node's properties and children from the named workspace in the source. The node may be specified by path and/or by identification properties. The connector returns all properties and the locations for all children, or sets a *`PathNotFoundException`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node did not exist in the workspace. If the node is found, the connector sets on the request the actual location of the node (including the path and identification properties). The connector sets a *`InvalidWorkspaceException`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
VerifyNodeExistsRequest	A request to verify the existence of a node at the specified location in the named workspace of the source. The connector returns all the actual location for the node if it exists, or sets a *`PathNotFoundException`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node does not exist in the workspace. The connector sets a *`InvalidWorkspaceException`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
ReadAllPropertiesRequest	A request to read all of the properties of a node from the named workspace in the source. The node may be specified by path and/or by identification properties.

Name	Description
	The connector returns all properties that were found on the node, or sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node did not exist in the workspace. If the node is found, the connector sets on the request the actual location of the node (including the path and identification properties). The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
ReadPropertyRequest	A request to read a single property of a node from the named workspace in the source. The node may be specified by path and/or by identification properties, and the property is specified by name. The connector returns the property if found on the node, or sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node or property did not exist in the workspace. If the node is found, the connector sets on the request the actual location of the node (including the path and identification properties). The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
ReadAllChildrenRequest	A request to read all of the children of a node from the named workspace in the source. The node may be specified by path and/or by identification properties.

Name	Description
	The connector returns an ordered list of locations for each child found on the node, an empty list if the node had no children, or sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node did not exist in the workspace. If the node is found, the connector sets on the request the actual location of the parent node (including the path and identification properties). The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
ReadBlockOfChildrenRequest	A request to read a block of children of a node, starting with the nth child from the named workspace in the source. This is designed to allow paging through the children, which is much more efficient for large numbers of children. The node may be specified by path and/or by identification properties, and the block is defined by a starting index and a count (i.e., the block size). The connector returns an ordered list of locations for each of the node's children found in the block, or an empty list if there are no children in that range. The connector also sets on the request the actual location of the parent node (including the path and identification properties) or sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the parent node did not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/

Name	Description
	InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
ReadNextBlockOfChildrenRequest	A request to read a block of children of a node, starting with the children that immediately follow a previously-returned child from the named workspace in the source. This is designed to allow paging through the children, which is much more efficient for large numbers of children. The node may be specified by path and/ or by identification properties, and the block is defined by the location of the node immediately preceding the block and a count (i.e., the block size). The connector returns an ordered list of locations for each of the node's children found in the block, or an empty list if there are no children in that range. The connector also sets on the request the actual location of the parent node (including the path and identification properties) or sets a `PathNotFoundException` [http:// docs.jboss.org/modeshape/1.0.0.Final/ api/org/modeshape/graph/property/ PathNotFoundException.html] error on the request if the parent node did not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http:// docs.jboss.org/modeshape/1.0.0.Final/ api/org/modeshape/graph/request/ InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
ReadBranchRequest	A request to read a portion of a subgraph that has as its root a particular node, up to a maximum depth. This request is an efficient mechanism when a branch (or part of a branch) is to be navigated and processed, and replaces some non-trivial code to read the branch iteratively using multiple `ReadNodeRequests`. The connector reads the branch to the specified maximum depth,

Name	Description
	returning the properties and children for all nodes found in the branch. The connector also sets on the request the actual location of the branch's root node (including the path and identification properties). The connector sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node at the top of the branch does not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.

`ChangeRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html] is a subclass of `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] that provides a base class for all the requests that request a change be made to the content. As we'll see later, these `ChangeRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html] objects also get reused by the *observation* system.

Table 5.2. Types of Change Requests

Name	Description
CreateNodeRequest	A request to create a node at the specified location and setting on the new node the properties included in the request. The connector creates the node at the desired location, adjusting any same-name-sibling indexes as required. (If an SNS index is provided in the new node's location, existing children with the same name after that SNS index will have their SNS indexes adjusted. However, if the requested location does not include a SNS index, the new node is added after all existing children, and it's SNS index is set accordingly.) The connector also sets on the request the actual location of the new node (including the path and

Name	Description
	identification properties).. The connector sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the parent node does not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
RemovePropertiesRequest	A request to remove a set of properties on an existing node. The request contains the location of the node as well as the names of the properties to be removed. The connector performs these changes and sets on the request the actual location (including the path and identification properties) of the node. The connector sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node does not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
UpdatePropertiesRequest	A request to set or update properties on an existing node. The request contains the location of the node as well as the properties to be set and those to be deleted. The connector performs these changes and sets on the request the actual location (including the path and identification properties) of the node. The connector sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/

Name	Description
	api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node does not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
RenameNodeRequest	A request to change the name of a node. The connector changes the node's name, adjusts all SNS indexes accordingly, and returns the actual locations (including the path and identification properties) of both the original location and the new location. The connector sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node does not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
CopyBranchRequest	A request to copy a portion of a subgraph that has as its root a particular node, up to a maximum depth. The request includes the name of the workspace where the original node is located as well as the name of the workspace where the copy is to be placed (these may be the same, but may be different). The connector copies the branch from the original location, up to the specified maximum depth, and places a copy of the node as a child of the new location. The connector also sets on the request the actual location (including the path and identification properties) of the original location as

Name	Description
	well as the location of the new copy. The connector sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node at the top of the branch does not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if one of the named workspaces does not exist.
MoveBranchRequest	A request to move a subgraph that has a particular node as its root. The connector moves the branch from the original location and places it as child of the specified new location. The connector also sets on the request the actual location (including the path and identification properties) of the original and new locations. The connector will adjust SNS indexes accordingly. The connector sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node that is to be moved or the new location do not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
DeleteBranchRequest	A request to delete an entire branch specified by a single node's location. The connector deletes the specified node and all nodes below it, and sets the actual location, including the path and identification properties, of the node that was deleted. The

Name	Description
	connector sets a `PathNotFoundException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/PathNotFoundException.html] error on the request if the node being deleted does not exist in the workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
CompositeRequest	A request that actually comprises multiple requests (none of which will be a composite). The connector simply processes all of the requests in the composite request, but should set on the composite request any error (usually the first error) that occurs during processing of the contained requests.

There are also requests that deal with workspaces:

Table 5.3. Types of Workspace Read Requests

Name	Description
GetWorkspacesRequest	A request to obtain the names of the existing workspaces that are accessible to the caller.
VerifyWorkspaceRequest	A request to verify that a workspace with a particular name exists. The connector returns the actual location for the root node if the workspace exists, as well as the actual name of the workspace (e.g., the default workspace name if a null name is supplied).

And there are also requests that deal with changing workspaces (and thus extend `ChangeRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html]):

Table 5.4. Types of Workspace Change Requests

Name	Description
CreateWorkspaceRequest	A request to create a workspace with a particular name. The connector returns the actual location for the root node if

Name	Description
	the workspace exists, as well as the actual name of the workspace (e.g., the default workspace name if a null name is supplied). The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace already exists.
DestroyWorkspaceRequest	A request to destroy a workspace with a particular name. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.
CloneWorkspaceRequest	A request to clone one named workspace as another new named workspace. The connector sets a `InvalidWorkspaceException` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the original workspace does not exist, or if the new workspace already exists.

Although there are over a dozen different kinds of requests, we do anticipate adding more in future releases. For example, ModeShape has recently added support for searching repository content in sources through an additional subclass of `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html]. Getting the version history for a node will likely be another kind of request added in an upcoming release.

This section covered the different kinds of `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] classes. The next section provides a easy way to encapsulate how a component should responds to these requests, and after that we'll see how these `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] objects are also used in the *observation* framework.

5.9. Request processors

ModeShape connectors are typically the components that receive these `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] objects.

We'll dive deep into connectors in the *next chapter*, but before we do there is one more component related to `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html]s that should be discussed.

The `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] class is an abstract class that defines a `process(...)` method for each concrete `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] subclass. In other words, there is a `process(CompositeRequest)` method, a `process(ReadNodeRequest)` method, and so on. This makes it easy to implement behavior that responds to the different kinds of `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] classes: simply subclass the `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html], override all of the abstract methods, and optionally overriding any of the other methods that have a default implementation.

Note

The `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] abstract class contains default implementations for quite a few of the `process(...)` methods, and these will be *sufficient* but probably not efficient or optimum. If you can provide a more efficient implementation given your source, feel free to do so. However, if performance is not a big issue, all of the concrete methods will provide the correct behavior. Keep things simple to start out - you can always provide better implementations later.

5.10. Observation

The ModeShape graph model also incorporates an observation framework that allows components to register and be notified when changes occur within the content owned by a graph.

Many event frameworks define the listeners and sources as interfaces. While this is often useful, it requires that the implementations properly address the thread-safe semantics of managing and calling the listeners. The ModeShape observation framework uses abstract or concrete classes to minimize the effort required for implementing `ChangeObserver` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObserver.html] or *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html]. These abstract classes provide implementations for a number of utility methods (such as the `unregister()` method on `ChangeObserver` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObserver.html]) that also save effort and code.

However, one of the more important reasons for providing classes is that `ChangeObserver` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObserver.html] uses weak references to track the

Observable [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html] instances, and the `ChangeObservers` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObservers.html] class uses weak references for the listeners. This means that an observer does not prevent *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html] instances from being garbage collected, nor do observers prevent *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html] instances from being garbage collected. These abstract class provide all this functionality for free.

5.10.1. Observable

Any component that can have changes and be observed can implement the *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html] interface. This interface allows *Observer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observer.html]s to register (or be registered) to receive notifications of the changes. However, a concrete and thread-safe implementation of this interface, called `ChangeObservers` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObservers.html], is available and should be used where possible, since it automatically manages the registered `ChangeObserver` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObserver.html] instances and properly implements the register and unregister mechanisms.

5.10.2. Observers

Components that are to recieve notifications of changes are called *observers*. To create an observer, simply extend the `ChangeObserver` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObserver.html] abstract class and provide an implementation of the `notify(Changes [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Changes.html])` method. Then, register the observer with an *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html] using its `register(ChangeObserver [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObserver.html])` method. The observer's `notify(Changes [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Changes.html])` method will then be called with the changes that have been made to the *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html].

When an observer is no longer needed, it should be unregistered from all *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html] instances with which it was registered. The `ChangeObserver` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObserver.html] class automatically tracks which *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html] instances it is registered with, and calling the observer's `unregister()` will unregister the observer from all of these *Observable* [http://docs.jboss.org/

modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html]s. Alternatively, an observer can be unregistered from a single *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html] using the *Observable* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observable.html]'s `unregister(`*`ChangeObserver`* `[http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/ChangeObserver.html])` method.

5.10.3. Changes

The *`Changes`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Changes.html] class represents the set of individual changes that have been made during a single, atomic operation. Each *`Changes`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Changes.html] instance has information about the source of the changes, the timestamp at which the changes occurred, and the individual changes that were made. These individual changes take the form of *`ChangeRequest`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html] objects, which we'll see more of in the next chapter. Each request is frozen, meaning it is immutable and will not change. Also none of the change requests will be marked as cancelled.

Using the actual *`ChangeRequest`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html] objects as the "events" has a number of advantages. First, the existing *`ChangeRequest`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html] subclasses already contain the information to accurately and completely describe the operation. Reusing these classes means we don't need a duplicate class structure or come up with a generic event class.

Second, the requests have all the state required for an event, plus they often will have more. For example, the *`DeleteBranchRequest`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/DeleteBranchRequest.html] has the actual location of the branch that was deleted (and in this way is not much different than a more generic event), but the *`CreateNodeRequest`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/CreateNodeRequest.html] has the actual location of the created node along with the properties of that node. Additionally, the *`RemovePropertyRequest`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/RemovePropertyRequest.html] has the actual location of the node along with the name of the property that was removed. In many cases, these requests have all the information a more general event class might have but then hopefully enough information for many observers to use directly without having to read the graph to decide what actually changed.

Third, the requests that make up a *`Changes`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Changes.html] instance can actually be replayed. Consider the case of a cache that is backed by a *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html], which might use an observer to keep the cache in sync. As the cache is notified of *`Changes`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Changes.html], the cache can simply replay the changes against its source.

As we'll see in the *next chapter*, each connector is responsible for propagating the `ChangeRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html] objects to the connector's *Observer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observer.html]. But that's not the only use of *Observer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observer.html]s. We'll also see later how the *sequencing system* uses *Observer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observer.html]s to monitor for changes in the graph content to determine which, if any, sequencers should be run. And, the *JCR implementation* also uses the observation framework to propagate those changes to JCR clients.

5.11. Summary

In this chapter, we introduced ModeShape's *graph model* and showed the different kinds of objects used to represent nodes, paths, names, and properties. We saw how all of these objects are actually immutable, and how the low-level Graph API uses this characteristic to provide a stateless and thread-safe interface for working with repository content using the *request model* used to read, update, and change content.

Next, we'll dive into the *connector framework*, which builds on top of the graph model and request model, allowing ModeShape to access the graph content stored in many different kinds of systems.

Connector Framework

There is a lot of information stored in many of different places: databases, repositories, SCM systems, registries, file systems, services, etc. The purpose of the federation engine is to allow applications to use the JCR API to access that information as if it were all stored in a single JCR repository, but to really leave the information where it is.

Why not just copy or move the information into a JCR repository? Moving it is probably pretty difficult, since most likely there are existing applications that rely upon that information being where it is. All of those applications would break or have to change. And copying the information means that we'd have to continually synchronize the changes. This not only is a lot of work, but it often makes it difficult to know whether information is accurate and "the master" data.

ModeShape lets us leave information where it is, yet access it through the JCR API as if it were in one big repository. One major benefit is that existing applications that use the information in the original locations don't break, since they can keep using the information. But now our JCR clients can also access all the information, too. And if our federating ModeShape repository is configured to allow updates, JCR client applications can change the information in the repository and ModeShape will propagate those changes down to the original source, making those changes visible to all the other applications.

In short, all clients see the correct information, even when it changes in the underlying systems. But the JCR clients can get to all of the information in one spot, using one powerful standard API.

6.1. Connectors

With ModeShape, your applications use the *JCR API* [http://www.jcp.org/en/jsr/detail?id=170] to work with the repository, but the ModeShape repository transparently fetches the information from different kinds of repositories and storage systems, not just a single purpose-built store. 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 graph-based *connector* system. Essentially, ModeShape's JCR implementation uses a single connector to access all content:

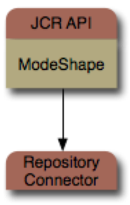

Figure 6.1. ModeShape's JCR implementation delegates to a connector

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 system to present its 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)

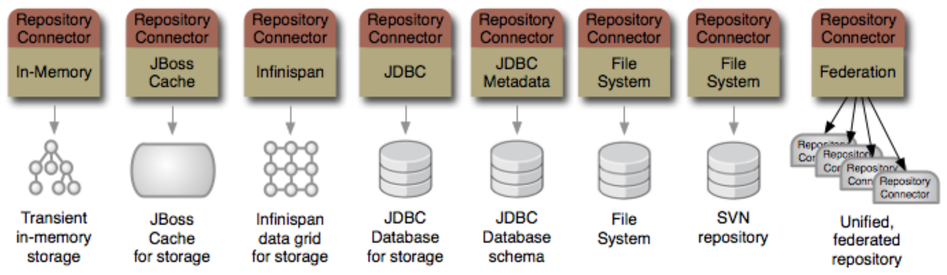

Figure 6.2. ModeShape can put JCR on top of multiple kinds of systems

Really, the federated connector gives us all kinds of possibilities, since we can use that connector on top of lots of connectors to other individual sources. This simple connector architecture is fundamentally what makes ModeShape so powerful and flexible. Along with a good library of connectors, which is what we're planning to create.

For instance, we want to build a connector to *other JCR repositories* [http://jira.jboss.org/jira/browse/MODE-39], and another to access *existing databases* [http://jira.jboss.org/jira/browse/MODE-199] so that some or all of the existing data (in whatever structure) can be accessed through JCR. For more information, check out our *roadmap* [http://jira.jboss.org/jira/browse/MODE?report=com.atlassian.jira.plugin.system.project:roadmap-panel]. Of course, if we don't have a connector to suit your needs, you can *write your own*.

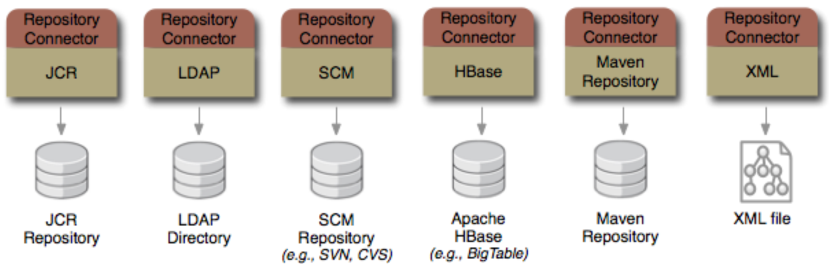

Figure 6.3. Future ModeShape connectors

It's even possible to put a different API layer on top of the connectors. For example, the new *New I/O (JSR-203)* [http://www.jcp.org/en/jsr/detail?id=203] API offers the opportunity to build new file system providers. This would be very straightforward to put on top of a JCR implementation, but it could be made even simpler by putting it on top of a ModeShape connector. In both cases, it'd be a trivial mapping from nodes that represent files and folders into JSR-203 files and directories, and events on those nodes could easily be translated into JSR-203 watch events. Then, simply choose a ModeShape connector and configure it to use the source you want to use.

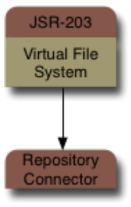

Figure 6.4. Virtual File System with ModeShape

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/1.0.0.Final/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/1.0.0.Final/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/1.0.0.Final/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/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] instance and a *`Request`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] object. The *`ExecutionContext`* [http://docs.jboss.org/modeshape/1.0.0.Final/

api/org/modeshape/graph/ExecutionContext.html] object defines the environment in which the processing is occurring, while the *`Request`* [http://docs.jboss.org/modeshape/1.0.0.Final/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/1.0.0.Final/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 if we wanted ModeShape to give us access through JCR to the 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 `JdbcAccessSource` Java class that implements *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/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/jbossmodeshape/latest/manuals/reference/html/provided-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.

Our new connector might also have a `JdbcAccessConnection` Java class that implements the *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] interface. This class would probably wrap a JDBC database connection, and would implement the `execute(...)` method such that the nodes exposed by the connector describe the database tables and their contents. For example, the connector might represent each database table as a node with the table's name, with properties that describe the table (e.g., the description, whether it's a temporary table), and with child nodes that represent rows in the table.

To use our connector in an application that uses ModeShape, we would need to create an instance of the `JdbcAccessSource` for each database instance that we want to access. If we have 3 MySQL databases, 9 Oracle databases, and 4 PostgreSQL databases, then we'd need to create a total of 16 `JdbcAccessSource` instances, each with the properties describing a single database instance. Those sources are then available for use by ModeShape components, including the *JCR* implementation.

So, we've so far learned what a connector is and how they're used to establish connections to the underlying sources and access the content in those sources. Next we'll show how connectors expose the notion of workspaces, and describe how to *create your own connectors*.

6.2. Out-of-the-box connectors

A number of connectors are already available in ModeShape, and are outlined in detail *later in the document*. Note that we do want to build *more connectors* [https://jira.jboss.org/jira/secure/IssueNavigator.jspa?reset=true&mode=hide&pid=12310520&sorter/order=DESC&sorter/field=priority&resolution=-1&component=12311441] in the upcoming releases.

6.3. Writing custom connectors

There may come a time when you want to tackle creating your own connector. Maybe the connectors we provide out-of-the-box don't work with your source. Maybe you want to use a different cache system. Maybe you have a system that you want to make available through a ModeShape repository. Or, maybe you're a contributor and want to help us round out our library with a new connector. No matter what the reason, creating a new connector is pretty straightforward, as we'll see in this section.

Creating a custom connector involves the following steps:

1. Create a Maven 2 project for your connector;

2. Implement the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] interface, using JavaBean properties for each bit of information the implementation will need to establish a connection to the source system. Then, implement the *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] interface with a class that represents a connection to the source. The `execute(`*`ExecutionContext`* `[http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html],` *`Request`* `[http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html])` method should process any and all requests that may come down the pike, and the results of each request can be put directly on that request. This approach is pretty straightforward, and gives you ultimate freedom in terms of your class structure.

   Alternatively, an easier way to get a complete read-write connector would be to extend one of our two abstract *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] implementations. If the content your connector exposes has unique keys (such as a unique string, UUID or other identifier), consider implementing *`MapRepositorySource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/map/MapRepositorySource.html], subclassing *`MapRepository`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/map/PathRepository.html], and using the existing *`MapRepositoryConnection`* [http://

docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/map/MapRepositoryConnection.html] implementation. This `MapRepositoryConnection` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/map/MapRepositoryConnection.html] does most of the work already, relying upon your `MapRepository` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/map/PathRepository.html] subclass for anything that might be source-specific. (See the *JavaDoc* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/map/package-summary.html] for details.) Or, if the content your connector exposes is simply path-based, consider implementing `PathRepositorySource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/path/PathRepositorySource.html], subclassing `PathRepository` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/path/PathRepository.html], and using the existing `PathRepositoryConnection` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/path/PathRepositoryConnection.html] implementation. Again, `PathRepositoryConnection` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/path/PathRepositoryConnection.html] class does almost all of the work and delegates to your `PathRepository` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/path/PathRepository.html] subclass for anything that might be source-specific. (See the *JavaDoc* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/path/package-summary.html] for details.)

Don't forget unit tests that verify that the connector is doing what it's expected to do. (If you'll be committing the connector code to the ModeShape project, please ensure that the unit tests can be run by others that may not have access to the source system. In this case, consider writing integration tests that can be easily configured to use different sources in different environments, and try to make the failure messages clear when the tests can't connect to the underlying source.)

3. Configure ModeShape to use your connector. This may involve just registering the source with the `RepositoryService` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html], or it may involve adding a source to a configuration repository used by the *federated repository*.

4. Deploy the JAR file with your connector (as well as any dependencies), and make them available to ModeShape in your application.

Let's go through each one of these steps in more detail.

6.3.1. Creating the Maven 2 project

The first step is to create the Maven 2 project that you can use to compile your code and build the JARs. Maven 2 automates a lot of the work, and since you're already *set up to use Maven*, using Maven for your project will save you a lot of time and effort. Of course, you don't have to use Maven 2, but then you'll have to get the required libraries and manage the compiling and building process yourself.

Note

ModeShape may provide in the future a Maven archetype for creating connector projects. If you'd find this useful and would like to help create it, please *join the community*.

In lieu of a Maven archetype, you may find it easier to start with a small existing connector project. The **modeshape-connector-filesystem** project is small and provides good example of implementing a path-based repository. See the subversion repository: *http://anonsvn.jboss.org/repos/modeshape/trunk/extensions/modeshape-connector-filesystem/*

You can create your Maven project any way you'd like. For examples, see the *Maven 2 documentation* [http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project]. Once you've done that, just add the dependencies in your project's `pom.xml` dependencies section:

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>1.0.0.Final</version>
</dependency>
```

This is the only dependency required for compiling a connector - Maven pulls in all of the dependencies needed by the 'modeshape-graph' artifact. Of course, you'll still have to add dependencies for any library your connector needs to talk to its underlying system.

As for testing, you probably will want to add more dependencies, such as those listed here:

```
<!-- ModeShape-related unit testing utilities and classes -->
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>1.0.0.Final</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-common</artifactId>
  <version>1.0.0.Final</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<!-- Unit testing -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.4</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-library</artifactId>
  <version>1.1</version>
  <scope>test</scope>
</dependency>
<!-- Logging with Log4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.5.8</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.14</version>
  <scope>test</scope>
</dependency>
```

Testing ModeShape connectors does not require a JCR repository or the ModeShape services. (For more detail, see the *testing section*.) However, if you want to do integration testing with a JCR repository and the ModeShape services, you'll need additional dependencies (e.g., `modeshape-repository` and any other extensions).

At this point, your project should be set up correctly, and you're ready to move on to *writing the Java implementation* for your connector.

6.3.2. Implementing a `RepositorySource`

As mentioned earlier, a *connector* consists of the Java code that is used to access content from a system. Perhaps the most important class that makes up a connector is the implementation of the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html]. This class is analogous to JDBC's DataSource in that it is instantiated to represent a single instance of a system that will be accessed, and it contains enough information (in the form of JavaBean properties) so that it can create connections to the source.

Why is the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] implementation a JavaBean? Well, this is the class that is instantiated, usually reflectively, and so a no-arg constructor is required. Using JavaBean properties makes it possible to reflect upon the object's class to determine the properties that can be set (using setters) and read (using getters). This means that an administrative application can instantiate, configure, and manage the objects that represent the actual sources, without having to know anything about the actual implementation.

So, your connector will need a public class that implements *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] and provides JavaBean properties for any kind of inputs or options required to establish a connection to and interact with the underlying source. Most of the semantics of the class are defined by the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] and inherited interface. However, there are a few characteristics that are worth mentioning here.

6.3.2.1. Workspaces

The *previous chapter* talked about how connector expose their information through the graph language of ModeShape. This is true, except that we didn't dive into too much of the detail. ModeShape graphs have the notion of *workspaces* in which the content appears, and its very easy for clients using the graph to switch between workspaces. In fact, workspaces differ from each other in that they provide different views of the same information.

Consider a source control system, like SVN or CVS. These systems provide different views of the source code: a mainline development branch as well as other branches (or tags) commonly used for releases. So, just like one source file might appear in the mainline branch as well as the previous two release branches, a node in a repository source might appear in multiple workspaces.

However, each connector can kind of decide how (or whether) it uses workspaces. For example, there may be no overlap in the content between workspaces. Or a connector might only expose a single workspace (in other words, there's only one "default" workspace).

6.3.2.2. Broadcasting events

When your *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] instance is put into the library within a running ModeShape system, the `initialize(`*`RepositoryContext`* `[http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryContext.html])` method will be called on the instance. The supplied *RepositoryContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/

modeshape/graph/connector/RepositoryContext.html] object represents the context in which the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] is running, and provides access to an `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html], a *RepositoryConnectionFactory* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnectionFactory.html] that can be used to obtain connections to other sources, and an *Observer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observer.html] of your source that should be called with events describing the `Changes` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Changes.html] being made within the source, either as a result of `ChangeRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html] operations being performed on this source, or as a result of operations being performed on the content from outside the source.

6.3.2.3. Cache policy

Each connector is responsible for determining whether and how long ModeShape is to cache the content made available by the connector. This is referred to as the *caching policy*, and consists of a *time to live* value representing the number of milliseconds that a piece of data may be cached. After the TTL has passed, the information is no longer used.

ModeShape allows a connector to use a flexible and powerful caching policy. First, each connection returns the *default* caching policy for all information returned by that connection. Often this policy can be configured via properties on the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] implementation. This is optional, meaning the connector can return `null` if it does not wish to have a default caching policy.

Second, the connector is able to override its default caching policy on individual requests (which we'll cover in the *next section*). Again, this is optional, meaning that a null caching policy on a request implies that the request has no overridden caching policy.

Third, if the connector has no default caching policy and none is set on the individual requests, ModeShape uses whatever caching policy is set up for that component using the connector. For example, the federating connector allows a default caching policy to be specified, and this policy is used should the sources being federated not define their own caching policy.

In summary, a connector has total control over whether and for how long the information it provides is cached.

Note

At this time, not every connector takes advantage of cache policies. However, it is anticipated that this will change.

6.3.2.4. Leveraging JNDI

Sometimes it is necessary (or easier) for a *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] implementation to look up an object in JNDI. One example of this is the JBoss Cache connector: while the connector can instantiate a new JBoss Cache instance, more interesting use cases involve JBoss Cache instances that are set up for clustering and replication, something that is generally difficult to configure in a single JavaBean. Therefore the `JBossCacheSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/jbosscache/JBossCacheSource.html] has optional JavaBean properties that define how it is to look up a JBoss Cache instance in JNDI.

This is a simple pattern that you may find useful in your connector. Basically, if your source implementation can look up an object in JNDI, simply use a single JavaBean String property that defines the full name that should be used to locate that object in JNDI. Usually it's best to include "Jndi" in the JavaBean property name so that administrative users understand the purpose of the property. (And some may suggest that any optional property also use the word "optional" in the property name.)

6.3.2.5. Capabilities

Another characteristic of a *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] implementation is that it provides some hint as to whether it supports several features. This is defined on the interface as a method that returns a `RepositorySourceCapabilities` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySourceCapabilities.html] object. This class currently provides methods that say whether the connector supports updates, whether it supports same-name-siblings (SNS), and whether the connector supports listeners and events.

Note that these may be hard-coded values, or the connector's response may be determined at runtime by various factors. For example, a connector may interrogate the underlying system to decide whether it can support updates.

The `RepositorySourceCapabilities` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySourceCapabilities.html] can be used as is (the class is immutable), or it can be subclassed to provide more complex behavior. It is important, however, that the capabilities remain constant throughout the lifetime of the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] instance.

Note

Why a concrete class and not an interface? By using a concrete class, connectors inherit the default behavior. If additional capabilities need to be added to the class in future releases, connectors may not have to override the defaults. This provides some insulation against future enhancements to the connector framework.

6.3.2.6. Security and authentication

As we'll see in the next section, the main method connectors have to process requests takes an `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html], which contains the JAAS security information of the subject performing the request. This means that the connector can use this to determine authentication and authorization information for each request.

Sometimes that is not sufficient. For example, it may be that the connector needs its own authorization information so that it can establish a connection (even if user-level privileges still use the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] provided with each request). In this case, the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] implementation will probably need JavaBean properties that represent the connector's authentication information. This may take the form of a username and password, or it may be properties that are used to delegate authentication to JAAS. Either way, just realize that it's perfectly acceptable for the connector to require its own security properties.

6.3.3. Implementing a `RepositoryConnection`

One job of the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] implementation is to create connections to the underlying sources. Connections are represented by classes that implement the *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] interface, and creating this class is the next step in writing a connector. This is what we'll cover in this section.

The *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] interface is pretty straightforward:

```
/**
 * A connection to a repository source.
 *
 * These connections need not support concurrent operations by multiple threads.
 */
@NotThreadSafe
public interface RepositoryConnection [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] {

    /**
     * Get the name for this repository source. This value should be the same as that returned
     * by the same RepositorySource [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] that created this connection.
     *
     * @return the identifier; never null or empty
     */
    String getSourceName();

    /**
     * Return the transactional resource associated with this connection. The transaction manager
     * will use this resource to manage the participation of this connection in a distributed transaction.
     *
     * @return the XA resource, or null if this connection is not aware of distributed transactions
     */
    XAResource getXAResource();

    /**
     * Ping the underlying system to determine if the connection is still valid and alive.
     *
     * @param time the length of time to wait before timing out
     * @param unit the time unit to use; may not be null
     * @return true if this connection is still valid and can still be used, or false otherwise
     * @throws InterruptedException if the thread has been interrupted during the operation
     */
    boolean ping( long time, TimeUnit [http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/
TimeUnit.html] unit ) throws InterruptedException;

    /**
     * Get the default cache policy for this repository. If none is provided, a global cache policy
     * will be used.
     *
     * @return the default cache policy
     */
    CachePolicy [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/cache/
CachePolicy.html] getDefaultCachePolicy();

    /**
     * Execute the supplied commands against this repository source.
     *
     * @param context the environment in which the commands are being executed; never null
     * @param request the request to be executed; never null
     * @throws RepositorySourceException if there is a problem loading the node data
     */
    void execute( ExecutionContext [http://docs.jboss.org/modeshape/
1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] context, Request [http://
docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] request
 ) throws RepositorySourceException [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/
modeshape/graph/connector/RepositorySourceException.html];
```

```
    /**
     * Close this connection to signal that it is no longer needed and that any accumulated
     * resources are to be released.
     */
    void close();
}
```

While most of these methods are straightforward, a few warrant additional information. The `ping(...)` method allows ModeShape to check the connection to see if it is alive. This method can be used in a variety of situations, ranging from verifying that a *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html]'s JavaBean properties are correct to ensuring that a connection is still alive before returning the connection from a connection pool.

The most important method on this interface, though, is the `execute(...)` method, which serves as the mechanism by which the component using the connector access and manipulates the content exposed by the connector. The first parameter to this method is the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html], which contains the information about environment as well as the subject performing the request. This was discussed *earlier*.

The second parameter, however, represents a `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] that is to be processed by the connector. Request objects can take many different forms, as there are different classes for each kind of request (see the *previous chapter* for details). Each request contains the information a connector needs to do the processing, and it also is the place where the connector places the results (or the error, if one occurs).

A connector is technically free to implement the `execute(...)` method in any way, as long as the semantics are maintained. But as discussed in the *previous chapter*, ModeShape provides a `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] class that can simplify writing your own connector and at the same time help insulate your connector from new kinds of requests that may be added in the future. The `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] is an abstract class that defines a `process(...)` method for each concrete `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] subclass. In other words, there is a `process(CompositeRequest)` method, a `process(ReadNodeRequest)` method, and so on.

To use this in your connector, simply create a subclass of `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html], overriding all of the abstract methods and optionally overriding any of the other methods that have a default implementation.

Note

The `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] abstract class contains default implementations for quite a few of the `process(...)` methods, and these will be *sufficient* but probably not efficient or optimum. If you can provide a more efficient implementation given your source, feel free to do so. However, if performance is not a big issue, all of the concrete methods will provide the correct behavior. Keep things simple to start out - you can always provide better implementations later.

Also, make sure your `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] is properly *broadcasting the changes* made during execution. The `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] class has a `recordChange(ChangeRequest [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html])` that can be called from each of the `process(...)` methods that take a `ChangeRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html]. The `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] enqueues these requests, and when the `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] is closed, the default implementation is to send a `Changes` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Changes.html] to the *Observer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observer.html] supplied into the constructor.

Then, in your connector's `execute(ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html], Request [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html])` method, instantiate your `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] subclass and call its `process(Request [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html]) method, passing in the execute(...) method's Request [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] parameter`. The `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] will determine the appropriate method given the actual `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] object and will then invoke that method:

```
public void execute( final ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] context,
                     final Request [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] request ) throws RepositorySourceException {
    String sourceName = // from the RepositorySource [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html]
    Observer [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observer.html] observer = // from the RepositoryContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryContext.html]
    RequestProcessor [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] processor = new CustomRequestProcessor(sourceName,context,observer);
  try {
      processor.process(request);
  } finally {
      processor.close();  // sends the accumulated ChangeRequest [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html]s as a Changes [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Changes.html] to the Observer [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/observe/Observer.html]
  }
}
```

If you do this, the bulk of your connector implementation may be in the `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] implementation methods. This not only is pretty maintainable, it also lends itself to easier testing. And should any new request types be added in the future, your connector may work just fine without any changes. In fact, if the `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] class can implement meaningful methods for those new request types, your connector may "just work". Or, at least your connector will still be binary compatible, even if your connector won't support any of the new features.

Finally, how should the connector handle exceptions? As mentioned above, each `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] object has a slot where the connector can set any exception encountered during processing. This not only handles the exception, but in the case of `CompositeRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/CompositeRequest.html]s it also correctly associates the problem with the request. However, it is perfectly acceptable to throw an exception if the connection becomes invalid (e.g., there is a communication failure) or if a fatal error would prevent subsequent requests from being processed.

6.3.4. Testing custom connectors

Testing connectors is not really that much different than testing other classes. Using mocks may help to isolate your instances so you can create more unit tests that don't require the underlying source system.

However, there may be times when you have to use the underlying source system in your tests. If this is the case, we recommend using Maven integration tests, which run at a different point in the Maven lifecycle. The benefit of using integration tests is that by convention they're able to rely upon external systems. Plus, your unit tests don't become polluted with slow-running tests that break if the external system is not available.

6.4. Summary

In this chapter, we covered all the aspects of ModeShape connectors, including the connector API, how ModeShape's JCR implementation works with connectors, what connectors are available (and how to use them), and how to write your own connector. So now that you know how to set up and use ModeShape repositories, the *next chapter* describes the sequencing framework and how to build your own custom sequencers. After that, we'll get into *how to configure ModeShape and use JCR*.

Sequencing framework

Many repositories are used (at least in part) to manage files and other artifacts, including service definitions, policy files, images, media, documents, presentations, application components, reusable libraries, configuration files, application installations, databases schemas, management scripts, and so on. Unlocking the information buried within all of those files is what ModeShape sequencing is all about. As files are loaded into the repository, you ModeShape instance can automatically sequence these files to extract from their content meaningful information that can be stored in the repository, where it can then be searched, accessed, and analyzed using the JCR API.

7.1. Sequencers

Sequencers are just POJOs that implement a specific interface, and their job is to process a stream of data (supplied by ModeShape) to extract meaningful content that usually takes the form of a structured graph. Exactly what content is up to each sequencer implementation. For example, ModeShape comes with an *image sequencer* that extracts the simple metadata from different kinds of image files (e.g., JPEG, GIF, PNG, etc.). Another example is the *Compact Node Definition (CND) sequencer* that processes the CND files to extract and produce a structured representation of the node type definitions, property definitions, and child node definitions contained within the file.

Sequencers are configured to identify the kinds of nodes that the sequencers can work against. When content in the repository changes, ModeShape looks to see which (if any) sequencers might be able to run on the changed content. If any sequencer configurations do match, those sequencers are run against the content, and the structured graph output of the sequencers is then written back into the repository (at a location dictated by the sequencer configuration). And once that information is in the repository, it can be easily found and accessed via the standard JCR API.

In other words, ModeShape uses sequencers to help you extract more meaning from the artifacts you already are managing, and makes it much easier for applications to find and use all that valuable information. All without your applications doing anything extra.

7.2. Stream Sequencers

The *StreamSequencer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencer.html] interface defines the single method that must be implemented by a sequencer:

```
public interface StreamSequencer [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencer.html] {

    /**
     * Sequence the data found in the supplied stream, placing the output
     * information into the supplied map.
     *
     * @param stream the stream with the data to be sequenced; never null
     * @param output the output from the sequencing operation; never null
     * @param context the context for the sequencing operation; never null
     */
                    void sequence( InputStream [http://java.sun.com/j2se/1.5.0/
docs/api/java/io/InputStream.html] stream, SequencerOutput [http://docs.jboss.org/
modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/SequencerOutput.html] output,
    StreamSequencerContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/
graph/sequencer/StreamSequencerContext.html] context );
}
```

A new instance is created for each sequencing operation, so there is no need for the class to be synchronized or thread-safe. Additionally, when a sequencer configuration includes properties (see *configuring a sequencer*), ModeShape will set those properties on the *StreamSequencer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencer.html] implementation using JavaBean-style setter methods. This makes it easy to define sequencer-specific properties on the sequencer configurations, while making it easy to implement with JavaBean-style setter methods.

Implementations are responsible for processing the content in the supplied *InputStream* [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] content and generating structured content using the supplied *SequencerOutput* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/SequencerOutput.html] interface. The *StreamSequencerContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencerContext.html] provides additional details about the information that is being sequenced, including the location and properties of the node being sequenced, the MIME type of the node being sequenced, and a *Problems* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/collection/Problems.html] object where the sequencer can record problems that aren't severe enough to warrant throwing an exception. The *StreamSequencerContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencerContext.html] also provides access to the *ValueFactories* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/ValueFactories.html] that can be used to create *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html], *Name* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html], and any other value objects.

The *SequencerOutput* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/SequencerOutput.html] interface is fairly easy to use, and its job is to hide from the sequencer all the specifics about where the output is being written. Therefore, the interface has only a few methods for implementations to call. Two methods set the property values on a node, while the other sets references to other nodes in the repository. Use these methods to describe the properties of the nodes you want to create, using relative paths for the nodes and valid JCR property names for properties and references. ModeShape will ensure that nodes are created or updated whenever they're needed.

```
public interface SequencerOutput [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/SequencerOutput.html] {

  /**
   * Set the supplied property on the supplied node.  The allowable
   * values are any of the following:
   *   - primitives (which will be autoboxed)
   *   - String instances
   *   - String arrays
   *   - byte arrays
   *   - InputStream instances
   *   - Calendar instances
   *
   * @param nodePath the path to the node containing the property;
   * may not be null
   * @param property the name of the property to be set
   * @param values the value(s) for the property; may be empty if
   * any existing property is to be removed
   */
  void setProperty( String nodePath, String property, Object... values );
    void setProperty( Path [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] nodePath, Name [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Name.html] property, Object... values );

  /**
   * Set the supplied reference on the supplied node.
   *
   * @param nodePath the path to the node containing the property;
   * may not be null
   * @param property the name of the property to be set
   * @param paths the paths to the referenced property, which may be
   * absolute paths or relative to the sequencer output node;
   * may be empty if any existing property is to be removed
   */
  void setReference( String nodePath, String property, String... paths );
}
```

Note

ModeShape will create nodes of type `nt:unstructured` unless you specify the value for the `jcr:primaryType` property. You can also specify the values for the `jcr:mixinTypes` property if you want to add mixins to any node.

7.3. Path Expressions

Each sequencer must be configured to describe the areas or types of content that the sequencer is capable of handling. This is done by specifying these patterns using path expressions that identify the nodes (or node patterns) that should be sequenced and where to store the output generated by the sequencer. We'll see how to fully configure a sequencer in the *next chapter*, but before then let's dive into path expressions in more detail.

A path expression consist of two parts: a selection criteria (or an input path) and an output path:

```
inputPath => outputPath
```

The *inputPath* part defines an expression for the path of a node that is to be sequenced. Input paths consist of '`/`' separated segments, where each segment represents a pattern for a single node's name (including the same-name-sibling indexes) and '`@`' signifies a property name.

Let's first look at some simple examples:

Table 7.1. Simple Input Path Examples

Input Path	Description
/a/b	Match node "b" that is a child of the top level node "a". Neither node may have any same-name-sibilings.
/a/*	Match any child node of the top level node "a".
/a/*.txt	Match any child node of the top level node "a" that also has a name ending in "`.txt`".
/a/*.txt	Match any child node of the top level node "a" that also has a name ending in "`.txt`".
/a/b@c	Match the property "c" of node "`/a/b`".
/a/b[2]	The second child named "b" below the top level node "a".
/a/b[2,3,4]	The second, third or fourth child named "b" below the top level node "a".

Input Path	Description
/a/b[*]	Any (and every) child named "b" below the top level node "a".
//a/b	Any node named "b" that exists below a node named "a", regardless of where node "a" occurs. Again, neither node may have any same-name-sibilings.

With these simple examples, you can probably discern the most important rules. First, the '*' is a wildcard character that matches any character or sequence of characters in a node's name (or index if appearing in between square brackets), and can be used in conjunction with other characters (e.g., "`*.txt`").

Second, square brackets (i.e., '`[`' and '`]`') are used to match a node's same-name-sibiling index. You can put a single non-negative number or a comma-separated list of non-negative numbers. Use '0' to match a node that has no same-name-sibilings, or any positive number to match the specific same-name-sibling.

Third, combining two delimiters (e.g., "`//`") matches any sequence of nodes, regardless of what their names are or how many nodes. Often used with other patterns to identify nodes at any level matching other patterns. Three or more sequential slash characters are treated as two.

Many input paths can be created using just these simple rules. However, input paths can be more complicated. Here are some more examples:

Table 7.2. More Complex Input Path Examples

Input Path	Description
/a/(b|c|d)	Match children of the top level node "a" that are named "b", "c" or "d". None of the nodes may have same-name-sibling indexes.
/a/b[c/d]	Match node "b" child of the top level node "a", when node "b" has a child named "c", and "c" has a child named "d". Node "b" is the selected node, while nodes "c" and "d" are used as criteria but are not selected.
/a(/(b|c|d|)/e)[f/g/@something]	Match node "`/a/b/e`", "`/a/c/e`", "`/a/d/e`", or "`/a/e`" when they also have a child "`f`" that itself has a child "`g`" with property "`something`". None of the nodes may have same-name-sibling indexes.

These examples show a few more advanced rules. Parentheses (i.e., '`(`' and '`)`') can be used to define a set of options for names, as shown in the first and third rules. Whatever part of the selected node's path appears between the parentheses is captured for use within the output path.

Thus, the first input path in the previous table would match node "`/a/b`", and "b" would be captured and could be used within the output path using "`$1`", where the number used in the output path identifies the parentheses.

Square brackets can also be used to specify criteria on a node's properties or children. Whatever appears in between the square brackets does not appear in the selected node.

Let's go back to the previous code fragment and look at the first path expression:

```
//(*.(jpg|jpeg|gif|bmp|pcx|png)[*])/jcr:content[@jcr:data] => /images/$1
```

This matches a node named "`jcr:content`" with property "`jcr:data`" but no siblings with the same name, and that is a child of a node whose name ends with "`.jpg`", "`.jpeg`", "`.gif`", "`.bmp`", "`.pcx`", or "`.png`" that may have any same-name-sibling index. These nodes can appear at any level in the repository. Note how the input path capture the filename (the segment containing the file extension), including any same-name-sibling index. This filename is then used in the output path, which is where the sequenced content is placed.

7.4. Out-of-the-box Sequencers

A number of sequencers are already available in ModeShape, and are outlined in detail *later in the document*. Note that we do want to build *more sequencers* [https://jira.jboss.org/jira/secure/IssueNavigator.jspa?reset=true&mode=hide&pid=12310520&sorter/order=DESC&sorter/field=priority&resolution=-1&component=12311441] in the upcoming releases.

7.5. Creating Custom Sequencers

The current release of ModeShape comes with eleven sequencers. However, it's very easy to create your own sequencers and to then configure ModeShape to use them in your own application.

Creating a custom sequencer involves the following steps:

1. Create a Maven 2 project for your sequencer;
2. Implement the *StreamSequencer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencer.html] interface with your own implementation, and create unit tests to verify the functionality and expected behavior;
3. Add the sequencer configuration to the ModeShape `SequencingService` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/sequencer/SequencingService.html] in your application as described in the *previous chapter*; and
4. Deploy the JAR file with your implementation (as well as any dependencies), and make them available to ModeShape in your application.

It's that simple.

7.5.1. Creating the Maven 2 project

The first step is to create the Maven 2 project that you can use to compile your code and build the JARs. Maven 2 automates a lot of the work, and since you're already *set up to use Maven*, using Maven for your project will save you a lot of time and effort. Of course, you don't have to use Maven 2, but then you'll have to get the required libraries and manage the compiling and building process yourself.

> **Note**
>
> ModeShape may provide in the future a Maven archetype for creating sequencer projects. If you'd find this useful and would like to help create it, please *join the community*.
>
> In lieu of a Maven archetype, you may find it easier to start with a small existing sequencer project. The **modeshape-sequencer-images** project is a small, self-contained sequencer implementation that has only the minimal dependencies. See the subversion repository: *http://anonsvn.jboss.org/repos/modeshape/trunk/extensions/modeshape-sequencer-images/*

You can create your Maven project any way you'd like. For examples, see the *Maven 2 documentation* [http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project]. Once you've done that, just add the dependencies in your project's `pom.xml` dependencies section:

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>1.0.0.Final</version>
</dependency>
```

These are minimum dependencies required for compiling a sequencer. Of course, you'll have to add other dependencies that your sequencer needs.

As for testing, you probably will want to add more dependencies, such as those listed here:

```
<!-- ModeShape-related unit testing utilities and classes -->
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>1.0.0.Final</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-common</artifactId>
  <version>1.0.0.Final</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<!-- Unit testing -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.4</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-library</artifactId>
  <version>1.1</version>
  <scope>test</scope>
</dependency>
<!-- Logging with Log4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.5.8</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.14</version>
  <scope>test</scope>
</dependency>
```

Testing ModeShape sequencers does not require a JCR repository or the ModeShape services. (For more detail, see the *testing section*.) However, if you want to do integration testing with a JCR repository and the ModeShape services, you'll need additional dependencies for these libraries.

```
<!-- ModeShape JCR Repository -->
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-jcr</artifactId>
  <version>1.0.0.Final</version>
  <scope>test</scope>
</dependency>
<!-- Java Content Repository API -->
<dependency>
  <groupId>javax.jcr</groupId>
  <artifactId>jcr</artifactId>
  <version>1.0.1</version>
  <scope>test</scope>
</dependency>
```

At this point, your project should be set up correctly, and you're ready to move on to write your custom implementation of the *StreamSequencer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencer.html] interface. As stated earlier, this should be fairly straightforward: process the stream and generate the output that's appropriate for the kind of file being sequenced.

Let's look at an example. Here is the complete code for the `ImageMetadataSequencer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/image/ImageMetadataSequencer.html] implementation:

```
public class ImageMetadataSequencer [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/
modeshape/sequencer/image/ImageMetadataSequencer.html] implements StreamSequencer

StreamSequencer.html] {

    public static final String METADATA_NODE = "image:metadata";
    public static final String IMAGE_PRIMARY_TYPE = "jcr:primaryType";
    public static final String IMAGE_MIXINS = "jcr:mixinTypes";
    public static final String IMAGE_MIME_TYPE = "jcr:mimeType";
    public static final String IMAGE_ENCODING = "jcr:encoding";
    public static final String IMAGE_FORMAT_NAME = "image:formatName";
    public static final String IMAGE_WIDTH = "image:width";
    public static final String IMAGE_HEIGHT = "image:height";
    public static final String IMAGE_BITS_PER_PIXEL = "image:bitsPerPixel";
    public static final String IMAGE_PROGRESSIVE = "image:progressive";
    public static final String IMAGE_NUMBER_OF_IMAGES = "image:numberOfImages";
    public static final String IMAGE_PHYSICAL_WIDTH_DPI = "image:physicalWidthDpi";
    public static final String IMAGE_PHYSICAL_HEIGHT_DPI = "image:physicalHeightDpi";
    public static final String IMAGE_PHYSICAL_WIDTH_INCHES = "image:physicalWidthInches";
    public static final String IMAGE_PHYSICAL_HEIGHT_INCHES = "image:physicalHeightInches";

    /**
     * {@inheritDoc}
     */
    public void sequence( InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] stream, SequencerOutput [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/SequencerOutput.html] output,
                          SequencerContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/SequencerContext.html] context ) {
        ImageMetadata [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/image/ImageMetadata.html] metadata = new ImageMetadata [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/image/ImageMetadata.html]();
        metadata.setInput(stream);
        metadata.setDetermineImageNumber(true);
        metadata.setCollectComments(true);

        // Process the image stream and extract the metadata ...
        if (!metadata.check()) {
            metadata = null;
        }
        // Generate the output graph if we found useful metadata ...
        if (metadata != null) {
            // Place the image metadata into the output map ...
            output.setProperty(METADATA_NODE, IMAGE_PRIMARY_TYPE, "image:metadata");
            // output.psetProperty(METADATA_NODE, IMAGE_MIXINS, "");
            output.setProperty(METADATA_NODE, IMAGE_MIME_TYPE, metadata.getMimeType());
            // output.setProperty(METADATA_NODE, IMAGE_ENCODING, "");
            output.setProperty(METADATA_NODE, IMAGE_FORMAT_NAME, metadata.getFormatName());
            output.setProperty(METADATA_NODE, IMAGE_WIDTH, metadata.getWidth());
            output.setProperty(METADATA_NODE, IMAGE_HEIGHT, metadata.getHeight());
            output.setProperty(METADATA_NODE, IMAGE_BITS_PER_PIXEL, metadata.getBitsPerPixel());
            output.setProperty(METADATA_NODE, IMAGE_PROGRESSIVE, metadata.isProgressive());
            output.setProperty(METADATA_NODE, IMAGE_NUMBER_OF_IMAGES,
                               metadata.getNumberOfImages());
            output.setProperty(METADATA_NODE, IMAGE_PHYSICAL_WIDTH_DPI,
      metadata.getPhysicalWidthDpi());
```

```
            output.setProperty(METADATA_NODE, IMAGE_PHYSICAL_HEIGHT_DPI,
      metadata.getPhysicalHeightDpi());
            output.setProperty(METADATA_NODE, IMAGE_PHYSICAL_WIDTH_INCHES,
      metadata.getPhysicalWidthInch());
            output.setProperty(METADATA_NODE, IMAGE_PHYSICAL_HEIGHT_INCHES,
      metadata.getPhysicalHeightInch());
        }
    }
}
```

Notice how the image metadata is extracted and the output graph is generated. A single node is created with the name `image:metadata` and with the `image:metadata` node type. No mixins are defined for the node, but several properties are set on the node using the values obtained from the image metadata. After this method returns, the constructed graph will be saved to the repository in all of the places defined by its configuration. (This is why only relative paths are used in the sequencer.)

7.5.2. Testing custom sequencers

The sequencing framework was designed to make testing sequencers much easier. In particular, the *StreamSequencer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencer.html] interface does not make use of the JCR API. So instead of requiring a fully-configured JCR repository and ModeShape system, unit tests for a sequencer can focus on testing that the content is processed correctly and the desired output graph is generated.

Note

For a complete example of a sequencer unit test, see the `ImageMetadataSequencerTest` unit test in the `org.modeshape.sequencer.images` package of the `modeshape-sequencers-image` project.

The following code fragment shows one way of testing a sequencer, using JUnit 4.4 assertions and some of the classes made available by ModeShape. Of course, this example code does not do any error handling and does not make all the assertions a real test would.

```
StreamSequencer [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencer.html] sequencer = new ImageMetadataSequencer [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/image/ImageMetadataSequencer.html]();
MockSequencerOutput [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/MockSequencerOutput.html] output = new MockSequencerOutput [http://
docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/
MockSequencerOutput.html]();
MockSequencerContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
sequencer/MockSequencerContext.html] context = new MockSequencerContext [http://
docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/
MockSequencerContext.html]();
InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] stream = null;
try {
   stream = this.getClass().getClassLoader().getResource("caution.gif").openStream();
   sequencer.sequence(stream,output,context);   // writes to 'output'
   assertThat(output.getPropertyValues("image:metadata", "jcr:primaryType"),
              is(new Object[] {"image:metadata"}));
   assertThat(output.getPropertyValues("image:metadata", "jcr:mimeType"),
              is(new Object[] {"image/gif"}));
   // ... make more assertions here
   assertThat(output.hasReferences(), is(false));
} finally {
   stream.close();
}
```

It's also useful to test that a sequencer produces no output for something it should not understand:

```
Sequencer [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/
sequencer/Sequencer.html] sequencer = new ImageMetadataSequencer [http://docs.jboss.org/
modeshape/1.0.0.Final/api/org/modeshape/sequencer/image/ImageMetadataSequencer.html]();
MockSequencerOutput [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
sequencer/MockSequencerOutput.html] output = new MockSequencerOutput [http://
docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/
MockSequencerOutput.html]();
MockSequencerContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/
sequencer/MockSequencerContext.html] context = new MockSequencerContext [http://
docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/
MockSequencerContext.html]();
InputStream [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] stream = null;
try {
   stream = this.getClass().getClassLoader().getResource("caution.pict").openStream();
   sequencer.sequence(stream,output,context);   // writes to 'output'
   assertThat(output.hasProperties(), is(false));
   assertThat(output.hasReferences(), is(false));
} finally {
   stream.close();
}
```

These are just two simple tests that show ways of testing a sequencer. Some tests may get quite involved, especially if a lot of output data is produced.

It may also be useful to create some integration tests that *configure ModeShape* to use a custom sequencer, and to then upload content using the JCR API, verifying that the custom sequencer did run. However, remember that ModeShape runs sequencers asynchronously in the background, and you must synchronize your tests to ensure that the sequencers have a chance to run before checking the results.

7.6. Summary

In this chapter, we described how ModeShape sequences files as they're uploaded into a repository. We've also learned in previous chapters about the ModeShape *execution contexts*, *graph model*, and *connectors*. In the *next part* we'll put all these pieces together to learn how to set up a ModeShape repository and access it using the JCR API.

Part III. ModeShape JCR

The ModeShape project provides an implementation of the *JCR API* [http://www.jcp.org/en/jsr/detail?id=170], which is built on top of the *core libraries* discussed earlier. This implementation as well as a number of JCR-related components are described in this part of the document. But before talking about how to use the JCR API with a ModeShape repository, first we need to show how to set up a ModeShape engine.

Configuring and Using ModeShape

Using ModeShape within your application is actually quite straightforward. As you'll see in this chapter, the first step is setting up ModeShape and starting the `JcrEngine`. After that, you obtain the `javax.jcr.Repository` instance for a named repository and just use the standard JCR API throughout your application.

8.1. ModeShape's JcrEngine

ModeShape encapsulates everything necessary to run one or more JCR repositories into a single *`JcrEngine`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] instance. This includes all underlying repository sources, the pools of connections to the sources, the sequencers, the MIME type detector(s), and the Repository implementations.

Obtaining a *`JcrEngine`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] instance is very easy - assuming that you have a valid *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance. We'll see how to get one of those in a little bit, but if you have one then all you have to do is build and start the engine:

```
JcrConfiguration config = ...
JcrEngine engine = config.build();
engine.start();
```

Obtaining a JCR Repository instance is a matter of simply asking the engine for it by the name defined in the configuration:

```
javax.jcr.Repository repository = engine.getRepository("Name of repository");
```

At this point, your application can proceed by working with the JCR API.

And, once you're finished with the *`JcrEngine`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html], you should shut it down:

```
engine.shutdown();
engine.awaitTermination(3,TimeUnit.SECONDS);    // optional
```

When the `shutdown()` method is called, the Repository instances managed by the engine are marked as being shut down, and they will not be able to create new Sessions. However, any existing Sessions or ongoing operations (e.g., event notifications) present at the time of the `shutdown()` call will be allowed to finish. In essence, `shutdown()` is a *graceful* request, and since it may take some time to complete, you can wait until the shutdown has completed by simply calling `awaitTermination(...)` as shown above. This method will block until the engine has indeed shutdown or until the supplied time duration has passed (whichever comes first). And, yes, you can call the `awaitTermination(...)` method repeatedly if needed.

8.2. JcrConfiguration

The previous section assumed the existence of a *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html]. It's not really that creating an instance is all that difficult. In fact, there's only one no-argument constructor, so actually creating the instance is a piece of cake. What can be a little more challenging, though, is setting up the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance, which must define the following components:

- **`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/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] classes are analogous to JDBC's *`DataSource`* [http://java.sun.com/j2se/1.5.0/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/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] instance is analogous to a *`DataSource`* [http://java.sun.com/j2se/1.5.0/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/1.0.0.Final/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 instance from the *`JcrEngine`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html]'s `getRepository(String)` method, 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.

There really are three options:

- **`Load from a file`** is conceptually the easiest and requires the least amount of Java code, but it now requires a configuration file.
- **`Load from a configuration repository`** is not much more complicated than loading from a file, but it does allow multiple *`JcrEngine`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] instances (usually in different processes perhaps on different machines) to easily access their (shared) configuration. And technically, loading the configuration from a file really just creates an *`InMemoryRepositorySource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/inmemory/InMemoryRepositorySource.html], imports the configuration file into that source, and then proceeds with this approach.
- **`Programmatic configuration`** is always possible, even if the configuration is loaded from a file or repository. Using the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html]'s API, you can define (or update or remove) all of the definitions that make up a configuration.

Each of these approaches has their obvious advantages, so the choice of which one to use is entirely up to you.

8.2.1. Loading from a Configuration File

Loading the ModeShape configuration from a file is actually very simple:

```
JcrConfiguration config = new JcrConfiguration();
configuration.loadFrom(file);
```

where the `file` parameter can actually be a *`File`* [http://java.sun.com/j2se/1.5.0/docs/api/java/io/File.html] instance, a *`URL`* [http://java.sun.com/j2se/1.5.0/docs/api/java/net/URL.html] to the file, an *InputStream* [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html] containing the contents of the file, or even a *String* [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] containing the contents of the file.

Note

The `loadFrom(...)` method can be called any number of times, but each time it is called it completely wipes out any current notion of the configuration and replaces it with the configuration found in the file.

There is an optional second parameter that defines the *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] within the configuration file identifying the parent node of the various configuration nodes. If not specified, it assumes "/". This makes it possible for the configuration content to be located at a different location in the hierarchical structure. (This is not often required, but when it is required this second parameter is very useful.)

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.
       This of course is optional, since we could access the same content through 'vehicles'.
       -->
       <mode:repository jcr:name="car repository" mode:source="Cars">
          <mode:options jcr:primaryType="mode:options">
             <jaasLoginConfigName jcr:primaryType="mode:option" mode:value="modeshape-
jcr"/>
          </mode:options>
          <mode:descriptors>
            <!--
               This adds a JCR Repository descriptor named "myDescriptor" with a value of "foo".
               So this code:
               Repository repo = ...;
               System.out.println(repo.getDescriptor("myDescriptor");
                Will now print out "foo".
              -->
              <myDescriptor mode:value="foo" />
            </mode:descriptors>
            <!--
                Import the custom node types defined in the named resource (a file at a
                classpath-relative path).  If there was more than one file with custom node
                types, we could either add successive <jcr:nodeTypes ... /> elements or just
                add all of the files as a comma-delimited string in the mode:resource property.
              -->
            <jcr:nodeTypes mode:resource="/tck/tck_test_types.cnd" />
          </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"/>
        <mode:source jcr:name="Aircraft"

mode:classname="org.modeshape.graph.connector.inmemory.InMemoryRepositorySource">
          <!-- Define the name of the workspace used by default.  Optional, but convenient. -->
          <defaultWorkspaceName>workspace2</defaultWorkspaceName>
        </mode:source>
      </mode:sources>
      <!--
       Define the sequencers. This is an optional section. For this example, we're not using any
sequencers.
      -->
      <mode:sequencers>
        <!--mode:sequencer jcr:name="Image Sequencer">
          <mode:classname>
            org.modeshape.sequencer.image.ImageMetadataSequencer
          </mode:classname>
          <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>
      <mode:mimeTypeDetectors>
```

```
    <mode:mimeTypeDetector jcr:name="Detector"
                           mode:description="Standard extension-based MIME type detector"/>
  </mode:mimeTypeDetectors>
</configuration>
```

8.2.2. Loading from a Configuration Repository

Loading the ModeShape configuration from an existing repository is also pretty straightforward. Simply create and configure the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] instance to point to the desired repository, and then call the `loadFrom(RepositorySource [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] source)` method:

```
RepositorySource configSource = ...
JcrConfiguration config = new JcrConfiguration();
configuration.loadFrom(configSource);
```

This really is a more advanced way to define your configuration, so we won't go into how you configure a *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html].

Note

The `loadFrom(...)` method can be called any number of times, but each time it is called it completely wipes out any current notion of the configuration and replaces it with the configuration found in the file.

There is an optional second parameter that defines the name of the workspace in the supplied source where the configuration content can be found. It is not needed if the workspace is the source's default workspace. There is an optional third parameter that defines the *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] within the configuration repository identifying the parent node of the various configuration nodes. If not specified, it assumes "/". This makes it possible for the configuration content to be located at a different location in the hierarchical structure. (This is not often required, but when it is required this second parameter is very useful.)

8.2.3. Programmatic Configuration

Defining the configuration programmatically is not terribly complicated, and it for obvious reasons results in more verbose Java code. But this approach is very useful and often the easiest approach when the configuration must change or is a reflection of other dynamic information.

The `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] class was designed to have an easy-to-use API that makes it easy to configure each of the different kinds of components, especially when using an IDE with code completion. Here are several examples:

8.2.3.1. Repository Sources

Each repository source definition must include the name of the *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] class as well as each bean property that should be set on the object:

```
JcrConfiguration config = ...
config.repositorySource("source A")
      .usingClass(InMemoryRepositorySource.class)
      .setDescription("The repository for our content")
      .setProperty("defaultWorkspaceName", workspaceName);
```

This example defines an in-memory source with the name "source A", a description, and a single "defaultWorkspaceName" bean property. Different *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] implementations will the bean properties that are required and optional. Of course, the class can be specified as Class reference or a string (followed by whether the class should be loaded from the classpath or from a specific classpath).

Note

Each time `repositorySource(String)` is called, it will either load the existing definition with the supplied name or will create a new definition if one does not already exist. To remove a definition, simply call `remove()` on the result of `repositorySource(String)`. The set of existing definitions can be accessed with the `repositorySources()` method.

8.2.3.2. Repositories

Each repository must be defined to use a named repository source, but all other aspects (e.g., namespaces, node types, options) are optional.

```
JcrConfiguration config = ...
config.repository("repository A")
      .addNodeTypes("myCustomNodeTypes.cnd")
      .setSource("source 1")
      .registerNamespace("acme","http://www.example.com/acme")
      .setOption(JcrRepository.Option.JAAS_LOGIN_CONFIG_NAME, "modeshape-jcr");
```

This example defines a repository that uses the "source 1" repository source (which could be a federated source, an in-memory source, a database store, or any other source). Additionally, this example adds the node types in the "myCustomNodeTypes.cnd" file as those that will be made available when the repository is accessed. It also defines the "http://www.example.com/acme" namespace, and finally sets the "JAAS_LOGIN_CONFIG_NAME" option to define the name of the JAAS login configuration that should be used by the ModeShape repository.

Note

Each time `repository(String)` is called, it will either load the existing definition with the supplied name or will create a new definition if one does not already exist. To remove a definition, simply call `remove()` on the result of `repository(String)`. The set of existing definitions can be accessed with the `repositories()` method.

8.2.3.3. Sequencers

Each defined sequencer must specify the name of the *StreamSequencer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencer.html] implementation class as well as the path expressions defining which nodes should be sequenced and the output paths defining where the sequencer output should be placed (often as a function of the input path expression).

```
JcrConfiguration config = ...
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");
```

This shows an example of a sequencer definition named "Image Sequencer" that uses the `ImageMetadataSequencer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/image/ImageMetadataSequencer.html] class (loaded from the classpath), that is to sequence the "jcr:data" property on any new or changed nodes that are named "jcr:content" below a parent node with a name ending in ".jpg", ".jpeg", ".gif", ".bmp", ".pcx", ".iff", ".ras", ".pbm", ".pgm", ".ppm" or ".psd". The output of the sequencing operation should be placed at the "/images/$1" node, where the "$1" value is captured as the name of the parent node. (The capture groups work the same way as regular expressions.) Of course, the class can be specified as Class reference or a string (followed by whether the class should be loaded from the classpath or from a specific classpath).

Note

Each time `sequencer(String)` is called, it will either load the existing definition with the supplied name or will create a new definition if one does not already exist. To remove a definition, simply call `remove()` on the result of `sequencer(String)`. The set of existing definitions can be accessed with the `sequencers()` method.

Note that in addition to including a description for the configuration, it is also possible to set sequencer-specific properties using the `setProperty(String,String[])` method. When ModeShape uses this configuration to set up a sequencing operation, it will instantiate the *StreamSequencer* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/sequencer/StreamSequencer.html] class and will call a JavaBean-style setter method for each property. For example, calling `setProperty("foo","val1")` on the sequencer configuration will mean that ModeShape will instantiate the sequencer implementation and will look for a `setFoo(String)` method on the sequencer implementation class, and use that method (if found) to pass the "val1" value to the instance.

8.2.3.4. MIME Type Detectors

Each defined MIME type detector must specify the name of the *MimeTypeDetector* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] implementation class as well as any other bean properties required by the implementation.

```
JcrConfiguration config = ...
config.mimeTypeDetector("Extension Detector")
```

```
    .usingClass(org.modeshape.graph.mimetype.ExtensionBasedMimeTypeDetector.class);
```

Of course, the class can be specified as Class reference or a string (followed by whether the class should be loaded from the classpath or from a specific classpath).

Note

Each time `mimeTypeDetector(String)` is called, it will either load the existing definition with the supplied name or will create a new definition if one does not already exist. To remove a definition, simply call `remove()` on the result of `mimeTypeDetector(String)`. The set of existing definitions can be accessed with the `mimeTypeDetectors()` method.

8.2.4. Storing Configuration

Regardless of how the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] is loaded, it can also be stored to a file or stream in an XML format that can then be reloaded in the future to recreate the configuration. This makes it very easy to programmatically generate a configuration file once while being able to load that same configuration at a later time (or on a different instance).

```
JcrConfiguration config = ...
String pathToFile = ...

// Save any changes before this point in the configuration repository ...
configuration.save();
// And now write out the configuration repository to a file ...
configuration.storeTo(pathToFile);
```

This will create a file at `pathToFile` that contains the current configuration in XML format. Any changes made after the most recent call to the `save()` method on the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] object will not be saved in the configuration repository, and thus will not be in the generated file. The generated XML will not be formatted to maximize human readability.

8.3. Deploying ModeShape via JNDI

Sometimes your applications can simply define a *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] and instantiate the

`JcrEngine` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] instance directly. This is very straightforward, and this is what the *ModeShape examples* do.

Web applications are a different story. Often, you may not want your web application to contain the code that initializes a ModeShape engine. Or, you may want the same `JcrEngine` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] instance to be reused in multiple web applications deployed to the same web/application server. In these cases, it is possible to configure the web/app server's JNDI instance to instantiate the `JcrEngine` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html], meaning the web applications need only use the standard JNDI and JCR APIs.

8.3.1. Example application using JCR and JNDI

Here's an example of how such a web application would obtain a JCR Repository instance, use it to create a `JcrSession` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrSession.html], and then close the session when completed.

```
Session session = null;

try {
  // Look up the JCR Repository object ...
    InitialContext initCtx = new InitialContext();
    Context envCtx = (Context) initCtx.lookup("java:comp/env");
     Repository repo = (Repository) envCtx.lookup("jcr/local");       // name in JNDI is defined by
 configuration

    // Obtain a JCR Session using simple authentication
    // (or use anonymous authentication if desired)
    session = repo.login(new SimpleCredentials("username", "password".toCharArray()));

    // Use the JCR Session to do something interesting

} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    if (session != null) session.logout();
}
```

Note that the location of the Repository instance in JNDI depends upon the configuration. In this example, we used "`jcr/local`", but the only requirement is that it match the location where it was placed in JNDI.

We showed how web applications can use an existing Repository instance. In the next section, we describe how to configure the web server so that the Repository instance is available in JNDI.

8.3.2. Configuring JCR and JNDI

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/1.0.0.Final/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/1.0.0.Final/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 `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] 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/1.0.0.Final/api/org/modeshape/jcr/JcrRepository.html] ensures that only one `JcrEngine` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] is instantiated, but that a Repository 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.

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 object, create and use a Session, and then close the Session. 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
   String configFile = "security/jaas.conf.xml";
   IDTrustConfiguration idtrustConfig = new IDTrustConfiguration();
   try {
      idtrustConfig.config(configFile);
   } 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. Moving the security folder into the `conf` directory does not allow those files to be visible to the JSP page.)

Note

If you use an application server such as *JBoss EAP* [http://www.jboss.com/products/platforms/application/], you could just configure the JAAS realm as part of the server configuration and be done with it.

8.4. Using ModeShape via Maven

ModeShape is a Maven-based project. If your application is using Maven, it is very easy to add a dependency on ModeShape's JCR library (plus any extensions), and Maven will ensure your application has access to all of the ModeShape artifacts and all 3rd-party libraries upon which ModeShape depends. Simply add a dependency in your application's POM:

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-jcr</artifactId>
  <version>1.0.0.Final</version>
</dependency>
```

plus dependencies for each optional extension (sequencers, connectors, MIME type detectors, etc.):

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-store-jpa</artifactId>
  <version>1.0.0.Final</version>
</dependency>
...
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-java</artifactId>
  <version>1.0.0.Final</version>
</dependency>
```

Then, continue by defining a `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] and building the engine, as discussed *earlier*. This is very straightforward, and this is exactly what the *ModeShape examples* do.

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!

8.5. Migrating from JBoss DNA to ModeShape

If you're currently using JBoss DNA, we highly recommend migrating to ModeShape. The process is actually pretty straightforward, since most of your code is probably using the JCR API. (Thank goodness for standards!)

First, change over to use the ModeShape artifacts in Maven. In your application's POM:

- change the group ID from "`org.jboss.dna`" to "`org.modeshape`";
- change the artifact IDs to use the "`modeshape-`" prefix instead of "`dna-`"; and
- change the version numbers to "`1.0.0.Beta1`".

Your Maven dependencies should then look like what is shown in the *Maven* chapter.

Note

None of the third-party dependencies have changed from JBoss DNA 0.7 to ModeShape 1.0.0.Final.

Second, change your Java code to use the ModeShape classes. Wherever your code is using JBoss DNA classes, simply replace the "`org.jboss.dna...`" imports with "`org.modeshape...`". All of the method names are the same, and nearly all of the class names are the same. (The names of some internal classes did begin with "Dna", and these were changed to begin with "ModeShape". But you're not likely using any of these internal classes anyway.)

Third, in your configuration files, change the JBoss DNA namespace URIs to the ModeShape namespace URIs. In other words, any namespace URI starting with "`http://www.jboss.org/`

`dna`" should be changed to start with "`http://www.modeshape.org`". We also suggest changing the prefixes to "`mode`", but this is technically optional.

What about persisted data? ModeShape will automatically convert any names or paths that use the old DNA namespaces to instead use the corresponding ModeShape namespaces. And, when that data is modified and the node updated, the ModeShape namespaces will be saved in the store.

Note

Be aware that any string property values that contain the DNA namespaces and that are treated by your application as strings will not be changed. The automatic conversion only happens when Name or Path objects are created, or when string values are converted to Name or Path objects.

Note

The search indexes are *not* automatically converted, so the easiest solution is to simply rebuild the indexes.

One final note. If you've begun using the "Simple" JPA connector model just recently introduced in JBoss DNA 0.7, please be aware that we've changed the names of the database tables. We could have kept using the "DNA_" prefix, but we thought it best to change the name because the "Simple" model is so new and will be with ModeShape for a long time to come.

8.6. What's next

This chapter outlines 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* talks about using the JCR API with your ModeShape repository.

Using the JCR API with ModeShape

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] (JCR 1.0) and has been revised under *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283]. ModeShape provides a partial JCR 1.0 implementation that allows you to work with the contents of a repository using the JCR API. For information about how to use the JCR API, please see the *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] specification.

Note

In the interests of brevity, this chapter does not attempt to reproduce the JSR-170 specification nor provide an exhaustive definition of ModeShape JCR capabilities. Rather, this chapter will describe any deviations from the specification as well as any ModeShape-specific public APIs and configuration.

Using ModeShape within your application is actually quite straightforward. As you'll see in this chapter, the first step is setting up ModeShape and starting the `JcrEngine`. After that, you obtain the `javax.jcr.Repository` instance for a named repository and just use the standard JCR API throughout your application.

9.1. Obtaining JCR Repositories

Once you've obtained a reference to a `JcrEngine` as described in *the previous chapter*, obtaining a repository is as easy as calling the `getRepository(String)` method with the name of the repository that you just configured.

```
String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] repositoryName = ...;
JcrEngine [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] jcrEngine = ...;
Repository repository = jcrEngine.getRepository(repositoryName);
```

At this point, your application can proceed by working with the JCR API.

9.2. Creating JCR Sessions

Once you have obtained a reference to the JCR Repository, you can create a JCR session using one of its `login(...)` methods. The *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] specification provides four login methods, but the behavior of these methods depends on the kind of authentication system your application is using.

9.2.1. Using JAAS

The `login()` method allows the implementation to choose its own security context to create a session in the default workspace for the repository. The ModeShape JCR implementation uses the security context from the current JAAS *`AccessControlContext`* [http://java.sun.com/j2se/1.5.0/docs/api/java/security/AccessController.html]. This implies that this method will throw a LoginException if it is not executed as a *`PrivilegedAction`* [http://java.sun.com/j2se/1.5.0/docs/api/java/security/PrivilegedAction.html] (AND the `JcrRepository.Options.ANONYMOUS_USER_ROLES` option does not allow access - *see below* for an example of how to configure guest user access). Here is one example of how this might work:

```
Subject subject = ...;
Session session = Subject.doAsPrivileged(subject, new PrivilegedExceptionAction<Session>() {
    public Session run() throws Exception {
        return repository.login();
    }
}, AccessController.getContext());
```

Another variant of this is to use the AccessControlContext directly, which then operates against the current Subject:

```
Session session = AccessController.doPrivileged( new PrivilegedExceptionAction<Session>() {
    public Session run() throws Exception {
        return repository.login();
    }
});
```

Either of these approaches will yield a session with the same user name and roles as `subject`. The `login(String workspaceName)` method is comparable and allows the workspace to be specified by name:

```
Subject subject = ...;
final String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] workspaceName = ...;
Session session = (Session) Subject.doAsPrivileged(subject, new
 PrivilegedExceptionAction<Session>() {
    public Session run() throws Exception {
        return repository.login(workspaceName);
    }}, AccessController.getContext());
```

The JCR API also allows supplying a JCR Credentials object directly as part of the login process, although ModeShape imposes some requirements on what types of Credentials may be supplied. The simplest way is to provide a JCR SimpleCredentials object. These credentials will be validated against the JAAS realm named "modeshape-jcr", unless another realm name is provided as an option during the JCR repository configuration. For example:

```
String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] userName = ...;
char[] password = ...;
Session session = repository.login(new SimpleCredentials(userName, password));
```

Similarly, the `login(Credentials credentials, String workspaceName)` method enables passing the credentials and a workspace name:

```
String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] userName = ...;
char[] password = ...;
String [http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html] workspaceName = ...;
Session session = repository.login(new SimpleCredentials(userName, password),
 workspaceName);
```

If a `LoginContext` [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html] is available for the user, that can be used as part of the credentials to authenticate the user with ModeShape instead. This snippet uses an anonymous class to provide the login context, but any class with a `LoginContext [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html] getLoginContext()` method can be used as well.

```
final LoginContext [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html] loginContext = ...;
Session session = repository.login(new Credentials() {
    LoginContext [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/login/LoginContext.html] loginContext getLoginContext() {
   return loginContext;
 }
}, workspaceName);
```

9.2.2. Using Custom Security

Not all applications can or want to use JAAS for their authentication system, so ModeShape provides a way to integrate your own custom security provider. The first step is to provide a custom implementation of *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] that integrates with your application security, allowing ModeShape to discover the authenticated user's name, determine whether the authenticated user has been assigned particular roles (see the *JCR Security section*), and to notify your application security system that the authenticated session (for JCR) has ended.

The next step is to wrap your *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] instance within an instance of `SecurityContextCredentials` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/SecurityContextCredentials.html], and pass it as the Credentials parameter in one of the two `login(...)` methods:

```
SecurityContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] securityContext = new CustomSecurityContext(...);
Session session = repository.login(new SecurityContextCredentials [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/SecurityContextCredentials.html](securityContext));
```

Once the Session is obtained, the repository content can be accessed and modified like any other JCR repository.

9.2.3. Using HTTP Servlet security

Servlet-based applications can make use of the servlet's existing authentication mechanism from `HttpServletRequest` [http://java.sun.com/javaee/5/docs/api/javax/servlet/http/HttpServletRequest.html]. Please note that the example below assumes that the servlet has a security constraint that prevents unauthenticated access.

```
HttpServletRequest [http://java.sun.com/javaee/5/docs/api/javax/servlet/http/HttpServletRequest.html] request = ...;
SecurityContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] securityContext = new ServletSecurityContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ServletSecurityContext.html](request);
Session session = repository.login(new SecurityContextCredentials [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/SecurityContextCredentials.html](securityContext));
```

You'll note that this is just a specialization of the *custom security context* approach, since the *ServletSecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ServletSecurityContext.html] just implements the *SecurityContext* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/SecurityContext.html] interface and delegates to the `HttpServletRequest` [http://java.sun.com/javaee/5/docs/api/javax/servlet/http/HttpServletRequest.html]. Feel free to use this class in your servlet-based applications.

9.2.4. Guest (Anonymous) User Access

By default, ModeShape allows guest users full administrative access. This is done to make it easier to get started with ModeShape. Of course, this is clearly not an appropriate security model for a production system.

To modify the roles granted to guest users, change the `JcrRepository.Options.ANONYMOUS_USER_ROLES` option for your repository to have a different value, like "" (to disable guest access entirely) or "readonly" (to give guests read-only access to all repositories). The value of this option can be any pattern that matches those described in the *table below*.

Note

The Using ModeShape chapter of the Getting Started Guide provides examples of modifying this option through programmatic configuration or in an XML configuration file.

9.3. JCR Specification Support

We believe that ModeShape JCR implementation is JCR-compliant, but we are awaiting final certification of compliance. Additionally, the JCR specification allows some latitude to implementors for some implementation details. The sections below clarify ModeShape's current and planned behavior. *As always, please consult the current list of known issues and bugs [http://jira.jboss.org/jira/browse/MODE?report=com.atlassian.jira.plugin.system.project:roadmap-panel].*

9.3.1. Level 1 and Level 2 (Required) Features

ModeShape currently supports all Level 1 and Level 2 features defined by the *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] specification. This release adds support for referential integrity for `REFERENCE` properties.

9.3.2. Optional Features

ModeShape also supports the optional JCR locking and observation features, though the ModeShape behavior with regard to events upon deletion follows the updated behavior outline in the *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] specification (namely that when a subgraph is deleted, ModeShape generates only one event for the top-level node in that subgraph).

ModeShape does not support the optional SQL query feature as defined by *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] and the JCR-SQL query language. Instead, ModeShape already supports its replacement, the *JCR-SQL2* query language defined by the *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] specification. JCR-SQL2 is much improved and much more capable. For details, see the chapter on *queries and search languages*.

ModeShape does not yet support the optional versioning feature of JSR-170, but our goal is to support it in the next release.

9.3.3. JCR Security

Although the *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] specification requires implementation of the `Session.checkPermission(String, String)` method, it allows implementors to choose the granularity of their access controls. ModeShape supports coarse-grained, role-based access control at the repository and workspace level.

ModeShape has extended the set of JCR-defined actions ("add_node", "set_property", "remove", and "read") with additional actions ("register_type", "register_namespace", and "unlock_any"). The register_type and register_namespace permissions restrict the ability to register (and unregister) node types and namespaces, respectively. The unlock_any permission grants the user the ability to unlock any locked node or branch (as opposed to users without that permission who can only unlock nodes or branches that they have locked themselves or for which they hold the lock token). Permissions to perform these actions are aggregated in roles that can be assigned to users.

ModeShape currently defines three roles: `readonly`, `readwrite`, and `admin`. If the Credentials passed into `Repository.login(...)` (or the *`Subject`* [http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/Subject.html] from the *`AccessControlContext`* [http://java.sun.com/j2se/1.5.0/docs/api/java/security/AccessController.html], if one of the no-credential `login` methods were used) have any of these roles, the session will have the corresponding access to all workspaces within the repository. The mapping from the roles to the actions that they allow is provided below, for any values of `path`.

Table 9.1. Role / Action Mapping

Action Name	readonly	readwrite	admin
read	Allows	Allows	Allows
add_node		Allows	Allows
set_property		Allows	Allows
remove		Allows	Allows
register_namespace			Allows
register_type			Allows

Action Name	readonly	readwrite	admin
unlock_any			Allows

Note

In this release, ModeShape does not check that the `actions` parameter passed into `Session.checkPermission(...)` contains only valid actions. This check may be added in a future release.

It is also possible to grant access only to one or more repositories on a single ModeShape server or to one or more named workspaces within a repository. The format for role names is defined below:

Table 9.2. Role Formats

Role Pattern	Examples	Description
ROLE_NAME	`readonly`, `admin`	Grants the named role to the assigned user on every workspace in any repository on the ModeShape server.
ROLE_NAME.REPOSITORY_NAME	`readonly.modeshape_repo`, `admin.localRepository`	Grants the named role to the assigned user on every workspace in the named repository on the ModeShape server.
ROLE_NAME.REPOSITORY_NAME.WORKSPACE_NAME	`readonly.modeshape_repo.jsmith`, `admin.localRepository.default`	Grants the named role to the assigned user on the named workspace in the named repository on the ModeShape server.

It is also possible to grant more than one role to the same user. For example, the user jsmith could be granted the roles readonly.production, readwrite.production.jsmith, and readwrite.staging to allow read-only access to any workspace on a production repository, read/write access to a personal workspace on the same production repository, and read/write access to any workspace in a staging repository.

As a final note, the ModeShape JCR implementation may have additional security roles added prior to the 1.0 release. A `CONNECT` role is already being used by the ModeShape REST Server to control whether users have access to the repository through that means.

9.3.4. Built-In Node Types

ModeShape supports all of the built-in node types described in the JSR-170 specification. However, several of these node types (mix:versionable, nt:version, nt:versionLabels, nt:versionHistory, and nt:frozenNode) are semantically meaningless as ModeShape does not yet support the versioning optional feature.

Although ModeShape does define some custom node types in the `modeshape` namespace, none of these node types except for `mode:resource` are intended to be used by developers integrating with ModeShape and may be changed or removed at any time.

9.3.5. Custom Node Type Registration

Although the *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] specification does not require support for registration and unregistration of custom types, ModeShape supports this extremely useful feature. Custom node types can be added at startup, as noted above or at runtime through a ModeShape-specific interface. ModeShape supports defining node types either through a *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283]-like template approach or through the use of *Compact Node Definition* [http://jackrabbit.apache.org/node-type-notation.html] (CND) files. Both type registration mechanisms are supported equally within ModeShape, although the CND approach for defining node types is recommended.

Note

ModeShape also supports defining custom node types to load at startup. This is discussed in more detail in the *previous chapter*.

The JSR-283 specification provides a useful means of programmatically defining JCR node types. ModeShape supports a comparable node type definition API that implements the functionality from the specification, albeit with interfaces in the `org.modeshape.jcr.nodetype` package. The intent is to deprecate these classes and replace their usage with the JSR-283 equivalents when ModeShape fully supports the JSR-283 final adopted specification in a future release.

Node types can be defined like so:

```
Session session = ... ;
Workspace [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Workspace.html] workspace = session.getWorkspace();

// Obtain the ModeShape-specific node type manager ...
JcrNodeTypeManager [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrNodeTypeManager.html] nodeTypeManager = (JcrNodeTypeManager)
 workspace.getNodeTypeManager();

// Declare a mixin node type named "searchable" (with no namespace)
NodeTypeTemplate [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/nodetype/NodeTypeTemplate.html] nodeType = nodeTypeManager.createNodeTypeTemplate();
```

```
nodeType.setName("searchable");
nodeType.setMixin(true);

// Add a mandatory child named "source" with a required primary type of "nt:file"
NodeDefinitionTemplate [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/nodetype/NodeDefinitionTemplate.html] childNode = nodeTypeManager.createNodeDefinitionTemplate();
childNode.setName("source");
childNode.setMandatory(true);
childNode.setRequiredPrimaryTypes(new String[] { "nt:file" });
childNode.setDefaultPrimaryType("nt:file");
nodeType.getNodeDefinitionTemplates().add(childNode);

// Add a multi-valued STRING property named "keywords"
PropertyDefinitionTemplate [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/nodetype/PropertyDefinitionTemplate.html] property = nodeTypeManager.createPropertyDefinitionTemplate();
property.setName("keywords");
property.setMultiple(true);
property.setRequiredType(PropertyType.STRING);
nodeType.getPropertyDefinitionTemplates().add(property);

// Register the custom node type
nodeTypeManager.registerNodeType(nodeType,false);
```

Residual properties and child node definitions can also be defined simply by not calling `setName` on the template.

Custom node types can be defined more succinctly through the *Compact Node Definition* [http://jackrabbit.apache.org/node-type-notation.html] file format. In fact, this is how JBoss ModeShape defines its built-in node types. An example CND file that declares the same node type as above would be:

```


[searchable] mixin
- keywords (string) multiple
+ source (nt:file) = nt:file
```

This definition could then be registered as part of the repository configuration, using the `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] class (see the *previous chapter*). Or, you can also use a Session to declare the node types in a CDN file, but this also requires ModeShape-specific interfaces and classes:

```
String pathToCndFileInClassLoader = ...;
CndNodeTypeSource [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/CndNodeTypeSource.html] nodeTypeSource = new CndNodeTypeSource [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/CndNodeTypeSource.html](pathToCndFileInClassLoader);

for (Problem [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/common/collection/Problem.html] problem : nodeTypeSource.getProblems()) {
    System.err.println(problem);
}
if (!nodeTypeSource.isValid()) {
    throw new IllegalStateException("Problems loading node types");
}

Session session = ... ;
// Obtain the ModeShape-specific node type manager ...
Workspace [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Workspace.html] workspace = session.getWorkspace();
JcrNodeTypeManager [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrNodeTypeManager.html] nodeTypeManager = (JcrNodeTypeManager) workspace.getNodeTypeManager();
nodeTypeManager.registerNodeTypes(nodeTypeSource);
```

The `CndNodeTypeSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/CndNodeTypeSource.html] class actually implements the *JcrNodeTypeSource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrNodeTypeSource.html] interface, so other implementations can actually be defined. For more information, see the JavaDoc for *JcrNodeTypeSource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrNodeTypeSource.html].

ModeShape also supports a simple means of unregistering types, although it is not possible to unregister types that are currently being used by nodes or as required primary types or supertypes of other types. Unused node types can be unregistered with the following code:

```
String unusedNodeTypeName = ...;

Session session = ... ;
// Obtain the ModeShape-specific node type manager ...
Workspace [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Workspace.html] workspace = session.getWorkspace();
```

```
JcrNodeTypeManager [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrNodeTypeManager.html] nodeTypeManager = (JcrNodeTypeManager) workspace.getNodeTypeManager();
nodeTypeManager.unregisterNodeType(Collections.singleton(unusedNodeTypeName));
```

9.4. Summary

In this chapter, we covered how to use JCR with ModeShape and learned about how it implements the JCR specification. Now that you know how ModeShape repositories work and how to use JCR to work with ModeShape repositories, we'll move on in the *next chapter* to show how you can use the RESTful web service to provide access to the content in a JCR repository to clients.

Querying and Searching using JCR

The JCR API defines a way to query a repository for content that meets user-defined criteria. The JCR API actually makes it possible for implementations to support multiple query languages, but the only language required by *JCR 1.0* [http://www.jcp.org/en/jsr/detail?id=170] is a subset of XPath. The 1.0 specification also defines a SQL-like query language, but supporting it is optional.

ModeShape now supports this query feature, including the required XPath language and two other languages not defined in the *JCR 1.0* [http://www.jcp.org/en/jsr/detail?id=170] specification. This chapter describes how your applications can use queries to search your repositories, and defines the three query languages that are available with ModeShape.

10.1. JCR Query API

Like most operations in the JCR API, querying is done through a Session instance, from which can be obtained the QueryManager that defines methods for creating Query objects, storing queries as Nodes in the repository, and reconstituting queries that were stored on Nodes. Thus, querying a repository generally follows this pattern:

```

// Obtain the query manager for the session ...
javax.jcr.query.QueryManager queryManager = session.getWorkspace().getQueryManager();

// Create a query object ...
String language = ...
String expression = ...
javax.jcr.Query query = queryManager.createQuery(expression,language);

// Execute the query and get the results ...
javax.jcr.QueryResult result = query.execute();

// Iterate over the nodes in the results ...
javax.jcr.NodeIterator nodeIter = result.getNodes();
while ( nodeIter.hasNext() ) {
   javax.jcr.Node node = nodeIter.nextNode();
      ...
}

// Or iterate over the rows in the results ...
String[] columnNames = result.getColumnNames();
javax.jcr.query.RowIterator rowIter = result.getRows();
while ( rowIter.hasNext() ) {
   javax.jcr.query.Row row = rowIter.nextRow();
```

```
    // Iterate over the column values in each row ...
    javax.jcr.Value[] values = row.getValues();
    for ( javax.jcr.Value value : values ) {
            ...
    }
    // Or access the column values by name ...
    for ( String columnName : columnNames ) {
        javax.jcr.Value value = row.getValue(columnName);
            ...
    }
}

// When finished, close the session ...
session.logout();
```

For more detail about these methods or about how to use other facets of the JCR query API, please consult Section 6.7 of the *JCR 1.0 specification* [http://www.jcp.org/en/jsr/detail?id=170].

10.2. JCR XPath Query Language

The *JCR 1.0 specification* [http://www.jcp.org/en/jsr/detail?id=170] uses the XPath query language because node structures in JCR are very analogous to the structure of an XML document. Thus, XPath provides a useful language for selecting and searching workspace content. And since JCR 1.0 defines a mapping between XML and a workspace view called the "document view", adapting XPath to workspace content is quite natural.

A JCR XPath query specifies the subset of nodes in a workspace that satisfy the constraints defined in the query. Constraints can limit the nodes in the results to be those nodes with a specific (primary or mixin) node type, with properties having particular values, or to be within a specific subtree of the workspace. The query also defines how the nodes are to be returned in the result sets using column specifiers and ordering specifiers.

Note

As an aside, ModeShape actually implements XPath queries by transforming them into the equivalent JCR-SQL2 representation. And the JCR-SQL2 language, although often more verbose, is much more capable of representing complex queries with multiple combinations of type, property, and path constraints.

10.2.1. Column Specifiers

JCR 1.0 specifies that support is required only for returning column values based upon single-valued, non-residual properties that are declared on or inherited by the node types specified in the type constraint. ModeShape follows this requirement, and does not specifying residual properties. However, ModeShape does allow multi-valued properties to be specified as result columns. And as per the specification, ModeShape always returns the "`jcr:path`" and "`jcr:score`" pseudo-columns.

ModeShape uses the last location step with an attribute axis to specify the properties that are to be returned as result columns. Multiple properties are specified with a union. For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

Table 10.1. Specifying result set columns

XPath	JCR-SQL2
`//*`	SELECT * FROM [nt:base]
`//element(*,my:type)`	SELECT * FROM [my:type]
`//element(*,my:type)/@my:title`	SELECT [my:title] FROM [my:type]
`//element(*,my:type)/(@my:title | @my:text)`	SELECT [my:title], [my:text] FROM [my:type]
`//element(*,my:type)/(@my:title union @my:text)`	SELECT [my:title], [my:text] FROM [my:type]

10.2.2. Type Constraints

JCR 1.0 specifies that support is required only for specifying constraints of one primary type, and it is optional to support specifying constraints on one (or more) mixin types. The specification also defines that the XPath `element` test be used to test against node types, and that it is optional to support `element` tests on location steps other than the last one. Type constraints are inherently inheritance-sensitive, in that a constraint against a particular node type 'X' will be satisfied by nodes explicitly declared to be of type 'X' or of subtypes of 'X'.

ModeShape does support using the `element` test to test against primary or mixin type. ModeShape also only supports using an `element` test on the last location step. For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

Table 10.2. Specifying type constraints

XPath	JCR-SQL2
`//*`	

XPath	JCR-SQL2
	SELECT * FROM [nt:base]
`//element(*,my:type)`	SELECT * FROM [my:type]

`/jcr:root/nodes/element(*,my:type)`	SELECT * FROM [my:type]
	WHERE PATH([my:type])> LIKE '/nodes/%'
	AND DEPTH([my:type]) = CAST(2 AS LONG)

`/jcr:root/nodes//element(*,my:type)`	SELECT * FROM [my:type]
	WHERE PATH([my:type]) LIKE '/nodes/%'

`/jcr:root/nodes//element(ex:nodeName,my:type)`	SELECT * FROM [my:type]
	WHERE PATH([my:type]) LIKE '/nodes/%'
	AND NAME([my:type]) = 'ex:nodeName'

Note that the JCR-SQL2 language supported by ModeShape is far more capable of joining multiple sets of nodes with different type, property and path constraints.

10.2.3. Property Constraints

JCR 1.0 specifies that attribute tests on the last location step is required, but that predicate tests on any other location steps are optional.

ModeShape does support using attribute tests on the last location step to specify property constraints, as well as supporting axis and filter predicates on other location steps. For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

Table 10.3. Specifying property constraints

XPath	JCR-SQL2

`//*[@prop1]`	SELECT * FROM [nt:base]
	WHERE [nt:base].prop1 IS NOT NULL

`//element(*,my:type)[@prop1]`	SELECT * FROM [my:type]
	WHERE [my:type].prop1 IS NOT NULL

XPath	JCR-SQL2

`//element(*,my:type)[@prop1=xs:boolea`	SELECT * FROM [my:type]
	WHERE [my:type].prop1 = CAST('true' AS
	 BOOLEAN)

`//element(*,my:type)[@id<1 and @name='john']`	SELECT * FROM [my:type]
	WHERE id < 1 AND name = 'john'

`//element(*,my:type)[a/b/@id]`	SELECT * FROM [my:type]
	JOIN [nt:base] as nodeSet1
	  ON ISCHILDNODE(nodeSet1,[my:type])
	JOIN [nt:base] as nodeSet2
	  ON ISCHILDNODE(nodeSet2,nodeSet1)
	WHERE (NAME(nodeSet1) = 'a'
	  AND NAME(nodeSet2) = 'b')
	  AND nodeSet2.id IS NOT NULL

`//element(*,my:type)[./*/*/@id]`	SELECT * FROM [my:type]
	JOIN [nt:base] as nodeSet1
	  ON ISCHILDNODE(nodeSet1,[my:type])
	JOIN [nt:base] as nodeSet2
	  ON ISCHILDNODE(nodeSet2,nodeSet1)
	WHERE nodeSet2.id IS NOT NULLL

`//element(*,my:type)[.//@id]`	SELECT * FROM [my:type]
	JOIN [nt:base] as nodeSet1
	  ON
	 ISDESCENDANTNODE(nodeSet1,[my:type])
	WHERE nodeSet2.id IS NOT NULLL

Section 6.6.3.3 of the JCR 1.0 specification contains an in-depth description of property value constraints using various comparison operators.

10.2.4. Path Constraints

JCR 1.0 specifies that exact, child node, and descendants-or-self path constraints be supported on the location steps in an XPath query.

ModeShape does support the four kinds of path constraints. For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

Table 10.4. Specifying path constraints

XPath	JCR-SQL2
`/jcr:root/a/b[*]`	`SELECT * FROM [nt:base] WHERE PATH([nt:base]) = '/a/b'`
`/jcr:root/a[1]/b[*]`	`SELECT * FROM [nt:base] WHERE PATH([nt:base]) = '/a/b'`
`/jcr:root/a[2]/b`	`SELECT * FROM [nt:base] WHERE PATH([nt:base]) = '/a[2]/b'`
`/jcr:root/a/b[2]//c[4]`	`SELECT * FROM [my:type] WHERE PATH([nt:base]) = '/a/b[2]/c[4]' OR PATH(nodeSet1) LIKE '/a/b[2]/%/c[4]'`
`/jcr:root/a/b//c//d`	`SELECT * FROM [my:type] WHERE PATH([nt:base]) = '/a/b/c/d' OR PATH([nt:base]) LIKE '/a/b/%/c/d' OR PATH([nt:base]) LIKE '/a/b/c/%/d' OR PATH([nt:base]) LIKE '/a/b/%/c/%/d'`
`//element(*,my:type)[@id<1 and @name='john']`	`SELECT * FROM [my:type] WHERE id < 1 AND name = 'john'`
`/jcr:root/a/b//element(*,my:type)`	`SELECT * FROM [my:type] WHERE PATH([my:type]) = '/a/b/%'`

Note that the JCR-SQL2 language supported by ModeShape is capable of representing a wider combination of path constraints.

10.2.5. Ordering Specifiers

JCR 1.0 extends the XPath grammar to add support for ordering the results according to the natural ordering of the values of one or more properties on the nodes.

ModeShape does support zero or more ordering specifiers, including whether each specifier is ascending or descending. If no ordering specifiers are defined, the ordering of the results is not predefined and may vary (though ordering by score is often the approach). For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

Table 10.5. Specifying result ordering

XPath	JCR-SQL2
`//element(*,*) order by @title`	`SELECT nodeSet1.title FROM [nt:base] AS nodeSet1 ORDER BY nodeSet1.title`
`//element(*,*) order by @title, @jcr:score`	`SELECT nodeSet1.title FROM [nt:base] AS nodeSet1 ORDER BY nodeSet1.title, SCORE([nt:base])`

Note that the JCR-SQL2 language supported by ModeShape has a far richer `ORDER BY` clause, allowing the use of any kind of dynamic operand, including ordering upon arithmetic operations of multiple dynamic operands.

10.2.6. Miscellaneous

JCR 1.0 defines a number of other optional and required features, and these are summarized in this section.

- Only abbreviated XPath syntax is supported.
- Only the `child` axis (the default axis, represented by '/' in abbreviated syntax), `descendant-or-self` axis (represented by '//' in abbreviated syntax), `self` axis (represented by '.' in abbreviated syntax), and `attribute` axis (represent by '@' in abbreviated syntax) are supported.
- The `text()` node test is *not* supported.
- The `element()` node test is supported.
- The `jcr:like()` function is supported.
- The `jcr:contains()` function is supported.
- The `jcr:score()` function is supported.
- The `jcr:deref()` function is *not* supported.

10.3. JCR-SQL Query Language

The JCR-SQL query language is defined by the *JCR 1.0 specification* [http://www.jcp.org/en/jsr/detail?id=170] as a way to express queries using strings that are similar to SQL. Support for the language is optional, and in fact this language was deprecated in the *JCR 2.0 specification* [http://www.jcp.org/en/jsr/detail?id=283] in favor of the improved and more powerful (and more SQL-like) *JCR-SQL2* language, which is covered in the next section. As such, **ModeShape does not support the original JCR-SQL language**.

10.4. JCR-SQL2 Query Language

The JCR-SQL2 query language is defined by the *JCR 2.0 specification* [http://www.jcp.org/en/jsr/detail?id=283] as a way to express queries using strings that are similar to SQL. This query language is an improvement over the earlier JCR-SQL language, which has been deprecated in JCR 2.0 (see previous section).

ModeShape includes full support for the complete JCR-SQL2 query language. However, ModeShape adds several extensions to make it even more powerful:

- Support for the "`FULL OUTER JOIN`" and "`CROSS JOIN`" join types, in addition to the "`LEFT OUTER JOIN`", "`RIGHT OUTER JOIN`" and "`INNER JOIN`" types defined by JCR-SQL2. Note that "`JOIN`" is a shorthand for "`INNER JOIN`". For detail, see the *grammar for joins*.
- Support for the `UNION`, `INTERSECT`, and `EXCEPT` set operations on multiple result sets to form a single result set. As with standard SQL, the result sets being combined must have the same columns. The `UNION` operator combines the rows from two result sets, the `INTERSECT` operator returns the difference between two result sets, and the `EXCEPT` operator returns the rows that are common to two result sets. Duplicate rows are removed unless the operator is followed by the `ALL` keyword. For detail, see the *grammar for set queries*.
- Removal of duplicate rows in the results, using "`SELECT DISTINCT ...`". For detail, see the *grammar for queries*.
- Limiting the number of rows in the result set with the "`LIMIT count`" clause, where `count` is the maximum number of rows that should be returned. This clause may optionally be followed by the "`OFFSET number`" clause to specify the number of initial rows that should be skipped. For detail, see the *grammar for limits and offsets*.
- Additional dynamic operands "`DEPTH([<selectorName>])`" and "`PATH([<selectorName>])`" that enable placing constraints on the node depth and path, respectively. These dynamic operands can be used in a manner similar to "`NAME([<selectorName>])`" and "`LOCALNAME([<selectorName>])`" that are defined by JCR-SQL2. Note in each of these cases, the selector name is optional if there is only one selector in the query. For detail, see the *grammar for dynamic operands*.
- Additional dynamic operand "`REFERENCE([<selectorName>.]<propertyName>)`" and "`REFERENCE([<selectorName>])`" that enables placing constraints on one or any of the reference properties, respectively, and which can be used in a manner similar to "`PropertyValue([<selectorName>.]<propertyName>)`". Note in each of these cases, the selector name is optional if there is only one selector in the query, and that the property name can be excluded if the constraint should apply to all reference properties. For detail, see the *grammar for dynamic operands*.

- Support for the `IN` and `NOT IN` clauses to more easily and concisely supply multiple of discrete static operands. For example, "`WHERE ... [my:type].[prop1] IN (3,5,7,10,11,50) ...`". For detail, see the *grammar for set constraints*.
- Support for the `BETWEEN` clause to more easily and concisely supply a range of discrete operands. For example, "`WHERE ... [my:type].[prop1] BETWEEN 3 EXCLUSIVE AND 10 ...`". For detail, see the *grammar for between constraints*.
- Support for simple arithmetic in numeric-based criteria and order-by clauses. For example, "`... WHERE SCORE(type1) + SCORE(type2) > 1.0`" or "`... ORDER BY (SCORE(type1) * SCORE(type2)) ASC, LENGTH(type2.property1) DESC`". For detail, see the *grammar for order-by clauses*.

The grammar for the JCR-SQL2 query language is actually a superset of that defined by the *JCR 2.0 specification* [http://www.jcp.org/en/jsr/detail?id=283], and as such the complete grammar is included here.

Note

The grammar is presented using the same EBNF nomenclature as used in the JCR 2.0 specification. Terms are surrounded by '[' and ']' denote optional terms that appear zero or one times. Terms surrounded by '{' and '}' denote terms that appear zero or more times. Parentheses are used to identify groups, and are often used to surround possible values. Literals (or keywords) are denoted by single-quotes.

10.4.1. Queries

```
QueryCommand ::= Query | SetQuery

SetQuery ::= Query ('UNION'|'INTERSECT'|'EXCEPT') ['ALL'] Query
                 { ('UNION'|'INTERSECT'|'EXCEPT') ['ALL'] Query }

Query ::= 'SELECT' ['DISTINCT'] columns
          'FROM' Source
          ['WHERE' Constraint]
          ['ORDER BY' orderings]
          [Limit]
```

10.4.2. Sources

```
Source ::= Selector | Join

Selector ::= nodeTypeName ['AS' selectorName]

nodeTypeName ::= Name
```

10.4.3. Joins

```
Join ::= left [JoinType] 'JOIN' right 'ON' JoinCondition
        // If JoinType is omitted INNER is assumed.

left ::= Source
right ::= Source

JoinType ::= Inner | LeftOuter | RightOuter | FullOuter | Cross

Inner ::= 'INNER' ['JOIN']

LeftOuter ::= 'LEFT JOIN' | 'OUTER JOIN' | 'LEFT OUTER JOIN'

RightOuter ::= 'RIGHT OUTER' ['JOIN']

RightOuter ::= 'FULL OUTER' ['JOIN']

RightOuter ::= 'CROSS' ['JOIN']

JoinCondition ::= EquiJoinCondition | SameNodeJoinCondition |
                ChildNodeJoinCondition | DescendantNodeJoinCondition
```

10.4.4. Equi-Join Conditions

```
EquiJoinCondition ::= selector1Name'.'property1Name '=' selector2Name'.'property2Name

selector1Name ::= selectorName
selector2Name ::= selectorName
```

```
property1Name ::= propertyName
property2Name ::= propertyName
```

10.4.5. Same-Node Join Conditions

```
SameNodeJoinCondition ::= 'ISSAMENODE(' selector1Name ',' selector2Name [',' selector2Path]
 ')'

selector2Path ::= Path
```

10.4.6. Child-Node Join Conditions

```
ChildNodeJoinCondition ::= 'ISCHILDNODE(' childSelectorName ',' parentSelectorName ')'

childSelectorName ::= selectorName
parentSelectorName ::= selectorName
```

10.4.7. Descendant-Node Join Conditions

```
DescendantNodeJoinCondition ::= 'ISDESCENDANTNODE(' descendantSelectorName
                                ',' ancestorSelectorName ')'
descendantSelectorName ::= selectorName
ancestorSelectorName ::= selectorName
```

10.4.8. Constraints

```
Constraint ::= ConstraintItem | '(' ConstraintItem ')'

ConstraintItem ::= And | Or | Not | Comparison | Between | PropertyExistence |
                   SetConstraint | FullTextSearch | SameNode | ChildNode | DescendantNode
```

10.4.9. And Constraints

```
And ::= constraint1 'AND' constraint2

constraint1 ::= Constraint
constraint2 ::= Constraint
```

10.4.10. Or Constraints

```
Or ::= constraint1 'OR' constraint2
```

10.4.11. Not Constraints

```
Not ::= 'NOT' Constraint
```

10.4.12. Comparison Constraints

```
Comparison ::= DynamicOperand Operator StaticOperand
```

```
Operator ::= '=' | '!=' | '<' | '<=' | '>' | '>=' | 'LIKE'
```

10.4.13. Between Constraints

```
Between ::= DynamicOperand ['NOT'] 'BETWEEN' lowerBound ['EXCLUSIVE']
                    'AND' upperBound ['EXCLUSIVE']

lowerBound ::= StaticOperand
upperBound ::= StaticOperand
```

10.4.14. Property Existence Constraints

```
PropertyExistence ::= selectorName'.'propertyName 'IS' ['NOT'] 'NULL' |
             propertyName 'IS' ['NOT'] 'NULL' /* If only one selector exists in this query */
```

10.4.15. Set Constraints

```
SetConstraint ::= selectorName'.'propertyName ['NOT'] 'IN' |
             propertyName ['NOT'] 'IN' /* If only one selector exists in this query */
             '(' firstStaticOperand {',' additionalStaticOperand } ')'
firstStaticOperand ::= StaticOperand
additionalStaticOperand ::= StaticOperand
```

10.4.16. Full-text Search Constraints

```
FullTextSearch ::= 'CONTAINS(' ([selectorName'.']propertyName | selectorName'.*')
                            ',' ''' fullTextSearchExpression''' ')'
                    /* If only one selector exists in this query, explicit specification of the selectorName
                       preceding the propertyName is optional */

fullTextSearchExpression ::= FulltextSearch
```

where `FulltextSearch` is defined by the following, and is the same as the *full-text search language* supported by ModeShape:

```
FulltextSearch ::= Disjunct {Space 'OR' Space Disjunct}

Disjunct ::= Term {Space Term}

Term ::= ['-'] SimpleTerm

SimpleTerm ::= Word | '"' Word {Space Word} '"'

Word ::= NonSpaceChar {NonSpaceChar}

Space ::= SpaceChar {SpaceChar}

NonSpaceChar ::= Char - SpaceChar /* Any Char except SpaceChar */

SpaceChar ::= ' '

Char ::= /* Any character */
```

10.4.17. Same-Node Constraint

```
SameNode ::= 'ISSAMENODE(' [selectorName ','] Path ')'
                    /* If only one selector exists in this query, explicit specification of the selectorName
                       preceding the path is optional */
```

10.4.18. Child-Node Constraints

```
ChildNode ::= 'ISCHILDNODE(' [selectorName ','] Path ')'
                  /* If only one selector exists in this query, explicit specification of the selectorName
                     preceding the path is optional */
```

10.4.19. Descendant-Node Constraints

```
DescendantNode ::= 'ISDESCENDANTNODE(' [selectorName ','] Path ')'
                  /* If only one selector exists in this query, explicit specification of the selectorName
                     preceding the propertyName is optional */
```

10.4.20. Paths and Names

```
Name ::= '[' quotedName ']' | '[' simpleName ']' | simpleName

quotedName ::= /* A JCR Name (see the JCR specification) */
simpleName ::= /* A JCR Name that contains only SQL-legal
                 characters (namely letters, digits, and underscore) */

Path ::= '[' quotedPath ']' | '[' simplePath ']' | simplePath

quotedPath ::= /* A JCR Path that contains non-SQL-legal characters */
simplePath ::= /* A JCR Path (rather Name) that contains only SQL-legal
                 characters (namely letters, digits, and underscore) */
```

10.4.21. Static Operands

```
StaticOperand ::= Literal | BindVariableValue

Literal
Literal ::= CastLiteral | UncastLiteral

CastLiteral ::= 'CAST(' UncastLiteral ' AS ' PropertyType ')'

PropertyType ::= 'STRING' | 'BINARY' | 'DATE' | 'LONG' | 'DOUBLE' | 'DECIMAL' |
                'BOOLEAN' | 'NAME' | 'PATH' | 'REFERENCE' | 'WEAKREFERENCE' | 'URI'
                /* 'WEAKREFERENCE' is not currently supported in JCR 1.0 */

UncastLiteral ::= UnquotedLiteral | ''' UnquotedLiteral ''' | '"' UnquotedLiteral '"'

UnquotedLiteral ::= /* String form of a JCR Value, as defined in the JCR specification */
```

10.4.22. Bind Variables

```
BindVariableValue ::= '$'bindVariableName

bindVariableName ::= /* A string that conforms to the JCR Name syntax, though the prefix
                      does not need to be a registered namespace prefix. */
```

10.4.23. Dynamic Operands

```
DynamicOperand ::= PropertyValue | ReferenceValue | Length | NodeName | NodeLocalName |
 NodePath | NodeDepth |
                  FullTextSearchScore | LowerCase | UpperCase | Arithmetic |
                  '(' DynamicOperand ')'

PropertyValue ::= [selectorName'.'] propertyName
                  /* If only one selector exists in this query, explicit specification of the selectorName
                     preceding the propertyName is optional */

ReferenceValue ::= 'REFERENCE(' selectorName '.' propertyName ')' |
                  'REFERENCE(' selectorName ')' |
                  'REFERENCE()' |
                  /* If only one selector exists in this query, explicit specification of the selectorName
                    preceding the propertyName is optional. Also, the property name may be excluded
                     if the constraint should apply to any reference property. *&#47;

Length ::= 'LENGTH(' PropertyValue ')'

NodeName ::= 'NAME(' [selectorName] ')'
                  /* If only one selector exists in this query, explicit specification of the selectorName
                     is optional */

NodeLocalName ::= 'LOCALNAME(' [selectorName] ')'
                  /* If only one selector exists in this query, explicit specification of the selectorName
                     is optional */

NodePath ::= 'PATH(' [selectorName] ')'
                  /* If only one selector exists in this query, explicit specification of the selectorName
                     is optional */

NodeDepth ::= 'DEPTH(' [selectorName] ')'
                  /* If only one selector exists in this query, explicit specification of the selectorName
                     is optional */

FullTextSearchScore ::= 'SCORE(' [selectorName] ')'
                  /* If only one selector exists in this query, explicit specification of the selectorName
                     is optional */

LowerCase ::= 'LOWER(' DynamicOperand ')'

UpperCase ::= 'UPPER(' DynamicOperand ')'

Arithmetic ::= DynamicOperand ('+'|'-'|'*'|'/') DynamicOperand
```

10.4.24. Ordering

```
orderings ::= Ordering {',' Ordering}

Ordering ::= DynamicOperand [Order]
```

```
Order ::= 'ASC' | 'DESC'
```

10.4.25. Columns

```
columns ::= (Column ',' {Column}) | '*'

Column ::= ([selectorName'.']propertyName ['AS' columnName]) | (selectorName'.*')
                   /* If only one selector exists in this query, explicit specification of the selectorName
                      preceding the propertyName is optional */
selectorName ::= Name
propertyName ::= Name
columnName ::= Name
```

10.4.26. Limit and Offset

```
Limit ::= 'LIMIT' count [ 'OFFSET' offset ]
count ::= /* Positive integer value */
offset ::= /* Non-negative integer value */
```

10.5. Full-Text Search Language

There are times when a formal structured query language is overkill, and the easiest way to find the right content is to perform a search, like you would with a search engine such as Google or Yahoo! This is where ModeShape's **full-text search language** comes in, because it allows you to use the JCR query API but with a far simpler, Google-style search grammar.

This query language is actually defined by the *JCR 2.0 specification* [http://www.jcp.org/en/jsr/detail?id=283] as the *full-text search expression grammar* used in the second parameter of the `CONTAINS(...)` function of the JCR-SQL2 language. We just pulled it out and made it available as a first-class query language.

This language allows a JCR client to construct a query to find nodes with property values that match the supplied terms. Nodes that "best" match the terms are returned before nodes that have a lesser match. Of course, ModeShape uses a complex system to analyze the node content and the query terms, and may perform a number of optimizations, such as (but not limited to) eliminating stop words (e.g., "the", "a", "and", etc.), treating terms independent of case, and converting words to base forms using a process called *stemming* (e.g., "running" into "run", "customers" into "customer").

Search terms can also include phrases by simply wrapping the phrase with double-quotes. For example, the search term '`table "customer invoice"`' would rank higher those nodes with properties containing the phrase "customer invoice" than nodes with properties containing just "customer" or "invoice".

Term in the query are implicitly AND-ed together, meaning that the matches occur when a node has property values that match *all* of the terms. However, it is also possible to put an "OR" in between two terms where either of those terms may occur.

It is also possible to specify that terms should *not* appear in the results. This is called a *negative term*, and it reduces the rank of any node whose property values contain the the value. To specify a negative term, simply prefix the term with a hyphen ('-').

The grammar for this full-text search language is specified in Section 6.7.19 of the *JCR 2.0 specification* [http://www.jcp.org/en/jsr/detail?id=283], but it is also included here as a convenience.

Note

The grammar is presented using the same EBNF nomenclature as used in the JCR 2.0 specification. Terms are surrounded by '[' and ']' denote optional terms that appear zero or one times. Terms surrounded by '{' and '}' denote terms that appear zero or more times. Parentheses are used to identify groups, and are often used to surround possible values.

10.5.1. Full-text Search Expressions

```
FulltextSearch ::= Disjunct {Space 'OR' Space Disjunct}

Disjunct ::= Term {Space Term}

Term ::= ['-'] SimpleTerm

SimpleTerm ::= Word | '"' Word {Space Word} '"'

Word ::= NonSpaceChar {NonSpaceChar}

Space ::= SpaceChar {SpaceChar}
```

```
NonSpaceChar ::= Char - SpaceChar /* Any Char except SpaceChar */

SpaceChar ::= ' '

Char ::= /* Any character */
```

As you can see, this is a pretty simple and straightforward query language. But this language makes it extremely easy to find all the nodes in the repository that match a set of terms.

When using this query language, the QueryResult always contains the "jcr:path" and "jcr:score" columns.

The ModeShape RESTful Web Service

ModeShape now provides a RESTful interface to its JCR implementation that allows HTTP-based access and updating of content. Although the initial version of this REST server only supports the ModeShape JCR implementation, it has been designed to make integration with other JCR implementors easy. This chapter describes how to configure and deploy the REST server.

11.1. Supported Resources and Methods

The REST Server currently supports the URIs and HTTP methods described below. The URI patterns assume that the REST server is deployed at its conventional location of "/resources". These URI patterns would change if the REST server were deployed under a different web context and URI patterns below would change accordingly. Currently, only JSON-encoded responses are provided.

Table 11.1. Supported URIs for the ModeShape REST Server

URI Pattern	HTTP Method(s)	HTTP Description
/resources	Returns a list of accessible repositories	GET
/resources/{repositoryName}	Returns a list of accessible workspaces within that repository	GET
/resources/{repositoryName}/{workspaceName}	Returns a list of available operations within the workspace	GET
/resources/{repositoryName}/{workspaceName}/item/{path}	Accesses the item (node or property) at the path	ALL

Note that this approach supports dynamic discovery of the available repositories on the server. A typical conversation might start with a request to the server to check the available repositories.

```
GET http://www.example.com/resources
```

This request would generate a response that mapped the names of the available repositories to metadata information about the repositories like so:

```
{
 "modeshape%3arepository" : {
  "repository" : {
   "name" : "modeshape%3arepository",
   "resources" : { "workspaces":"/resources/modeshape%3arepository" }
  }
 }
}
```

The actual response wouldn't be pretty-printed like the example, but the format would be the same. The name of the repository ("mode:repository" URL-encoded) is mapped to a repository object that contains a name (the redundant "mode:repository") and a list of available resources within the repository and their respective URIs. Note that ModeShape supports deploying multiple JCR repositories side-by-side on the same server, so this response could easily contain multiple repositories in a real deployment.

The only thing that you can do with a repository through the REST interface at this time is to get a list of its workspaces. A request to do so can be built up from the previous response like this:

```
GET http://www.example.com/resources/modeshape%3arepository
```

This request (and all of the following requests) actually create a JCR Session to service the request and require that security be configured. This process is described in more detail in *a later section*. Assuming that security has been properly configured, the response would look something like this:

```
{
 "default" : {
  "workspace" : {
   "name" : "default",
   "resources" : { "items":"/resources/modeshape%3arepository/default/items" }
  }
 }
}
```

Like the first response, this response consists of a list of workspace names mapped to metadata about the workspaces. The example above only lists one workspace for simplicity, but there could be many different workspaces returned in a real deployment. Note that the "items" resource builds the full URI to the root of the items hierarchy, including the encoding of the repository name and the workspace name.

Now a request can be built to retrieve the root item of the repository.

```
GET http://www.example.com/resources/modeshape%3arepository/default/items
```

Any other item in the repository could be accessed by appending its path to the URI above. In a default repository with no content, this would return the following response:

```
{
 "properties": {
  "jcr:primaryType": "mode:root",
  "jcr:uuid": "97d7e2ef-996e-4d99-8ec2-dc623e6c2239"
 },
 "children": ["jcr:system"]
```

The response contains a mapping of property names to their values and an array of child names. Had one of the properties been multi-valued, the values for that property would have been provided as an array as well, as will shortly be shown.

The items resource also contains an option query parameter: `mode:depth`. This parameter, which defaults to 1, controls how deep the hierarchy of returned nodes should be. Had the request had the parameter:

```
GET http://www.example.com/resources/modeshape%3arepository/default/items?mode:depth=2
```

Then the response would have contained details for the children of the root node as well.

```
{
 "properties": {
  "jcr:primaryType": "mode:root",
  "jcr:uuid": "163bc5e5-3b57-4e63-b2ae-ededf43d3445"
 },
 "children": {
  "jcr:system": {
   "properties": {"jcr:primaryType": "mode:system"},
      "children": ["mode:namespaces"]
  }
 }
}
```

It is also possible to use the RESTful API to add, modify and remove repository content. Removes are simple - a DELETE request with no body returns a response with no body.

```
DELETE http://www.example.com/resources/modeshape%3arepository/default/items/path/to/deletedNode
```

Adding content simply requires a POST to the name of the *relative* root node of the content that you wish to add and a request body in the same format as the response from a GET. Adding multiple nodes at once is supported, as shown below.

```
POST http://www.example.com/resources/modeshape%3arepository/default/items/newNode

{
 "properties": {
  "jcr:primaryType": "nt:unstructured",
  "jcr:mixinTypes": "mix:referenceable",
  "someProperty": "foo"
 },
 "children": {
  "newChildNode": {
   "properties": {"jcr:primaryType": "nt:unstructured"}
  }
 }
}
```

Note that protected properties like jcr:uuid are not provided but that the primary type and mixin types are provided as properties. The REST server will translate these into the appropriate calls behind the scenes. The response from the request will be empty by convention.

The PUT method allows for updates of nodes and properties. If the URI points to a property, the body of the request should be the new JSON-encoded value for the property, which includes the property name (allowing proper determination of whether the values are binary; see the *next section'*").

```
PUT http://www.example.com/resources/modeshape%3arepository/default/items/newNode/someProperty

{
 "someProperty" : "bar"
}
```

Setting multiple properties at once can be performed by providing a URI to a node instead of a property. The body of the request should then be a JSON object that maps property names to their new values.

```
PUT http://www.example.com/resources/modeshape%3arepository/default/items/newNode

{
 "someProperty": "foobar",
 "someOtherProperty": "newValue"
}
```

Note

The PUT method doesn't currently support adding or removing mixin types. This will be corrected in the future. A *JIRA issue* [https://jira.jboss.org/jira/browse/ModeShape-447] has been created to help track this issue.

11.1.1. Binary properties

Binary property values are included in any of the the responses or requests, but are represented string values containing the *Base 64 encoding* [http://en.wikipedia.org/wiki/Base64] of the binary content. Any such property is explicitly annotated such that "/base64/" is appended to the property name. First of all, this makes it very clear to the client and service which properties are encoded, allowing them to properly decode the values before use. Secondly, the "/base64/" suffix was carefully chosen because it cannot be used in a real property name (without escaping). Here's an example of a node containing a "jcr:primaryType" property with a single string value, a "jcr:uuid" property with another single UUID value, another "options" property that has two integer values, and a fourth "content" property that has a single binary value:

```
{
 "properties": {
  "jcr:primaryType": "nt:unstructured",
  "jcr:uuid": "163bc5e5-3b57-4e63-b2ae-ededf43d3445"
  "options": [ "1", "2" ]
  "content/base64/":
 "TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1dCBieSB0aGlz
IHNpbmd1bGFyIHBhc3Npb24gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaCBpcyBhIGx1c3Qgb2Yg
dGhlIG1pbmQsIHRoYXQgYnkgYSBwZXJzZXZlcmFuY2Ugb2YgZGVsaWdodCBpbiB0aGUgY29udGlu
dWVkIGFuZCBpbmRlZmF0aWdhYmxlIGdlbmVyYXRpb24gb2Yga25vd2xlZGdlLCBleGNlZWRzIHRo
ZSBzaG9ydCB2ZWhlbWVuY2Ugb2YgYW55IGNhcm5hbCBwbGVhc3VyZS4="
 },
}
```

All values of a property will always be Base 64 encoded if at least one of the values is binary. If there are multiple values, then they will be separated by commas and will appear within '[' and ']' characters (just like other properties).

11.2. Configuring the ModeShape REST Server

The ModeShape REST server is deployed as a WAR and configured mostly through its web configuration file (web.xml). Here is an example web configuration that is used for integration testing of the ModeShape REST server along with an explanation of its parts.

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
                         "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>ModeShape JCR RESTful Interface</display-name>
```

This first section is largely boilerplate and should look familiar to anyone who has deployed a servlet-based application before. The display-name can be customized, of course.

The next stanza configures the *repository provider*.

```
<!--
  This parameter provides the fully-qualified name of a class that implements
  the o.m.web.jcr.rest.spi.RepositoryProvider interface.  It is required
  by the ModeShapeJcrDeployer that controls the lifecycle for the ModeShape REST server.
-->
<context-param>
  <param-name>org.modeshape.web.jcr.rest.REPOSITORY_PROVIDER</param-name>
      <param-value>org.modeshape.web.jcr.rest.spi.ModeShapeJcrRepositoryProvider</param-
value>
</context-param>
```

As noted above, this parameter informs the `ModeShapeJcrDeployer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/web/jcr/rest/ModeShapeJcrDeployer.html] of the specific repository provider in use. Unless you are using the ModeShape REST server to connect to a different JCR implementation, this should never change.

Next we configure the ModeShape `JcrEngine` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] itself.

```
<!--
  This parameter, specific to the ModeShapeJcrRepositoryProvider implementation, specifies
  the name of the configuration file to initialize the repository or repositories.
  This configuration file must be on the classpath and is given as a classpath-relative
  directory.
-->
<context-param>
  <param-name>org.modeshape.web.jcr.rest.CONFIG_FILE</param-name>
  <param-value>/configRepository.xml</param-value>
</context-param>
```

If you are not familiar with the file format for a `JcrEngine` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] configuration file, you can build one programatically with the `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] class and call `save(...)` instead of `build()` to output the configuration file that equates to the configuration.

This is followed by a bit of RESTEasy and JAX-RS boilerplate.

```
<!--
```

```
  This parameter defines the JAX-RS application class, which is really just a metadata class
  that lets the JAX-RS engine (RESTEasy in this case) know which classes implement pieces
  of the JAX-RS specification like exception handling and resource serving.

  This should not be modified.
-->
<context-param>
  <param-name>javax.ws.rs.Application</param-name>
  <param-value>org.modeshape.web.jcr.rest.JcrApplication</param-value>
</context-param>

<!-- Required parameter for RESTEasy - should not be modified -->
<listener>
  <listener-class>org.jboss.resteasy.plugins.server.servlet.ResteasyBootstrap</listener-class>
</listener>

<!-- Required parameter for ModeShape REST - should not be modified -->
<listener>
  <listener-class>org.modeshape.web.jcr.rest.ModeShapeJcrDeployer</listener-class>
</listener>

<!-- Required parameter for RESTEasy - should not be modified -->
<servlet>
  <servlet-name>Resteasy</servlet-name>
 <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
</servlet>

<!-- Required parameter for ModeShape REST - should not be modified -->
<servlet-mapping>
  <servlet-name>Resteasy</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

In general, this part of the web configuration file should not be modified.

Finally, security must be configured for the REST server.

```

<!--
   The ModeShape REST implementation leverages the HTTP credentials to for authentication
and
  authorization within the JCR repository.  It makes no sense to try to log into the JCR
  repository without credentials, so this constraint helps lock down the repository.
```

```
    This should generally not be modified.
  -->
  <security-constraint>
    <display-name>ModeShape REST</display-name>
    <web-resource-collection>
      <web-resource-name>RestEasy</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
            <!--
       A user must be assigned this role to connect to any JCR repository, in addition to needing the
           READONLY or READWRITE roles to actually read or modify the data.  This is not used
internally,
         so another role could be substituted here.
      -->
      <role-name>connect</role-name>
    </auth-constraint>
  </security-constraint>

  <!--
    Any auth-method will work for ModeShape.  BASIC is used this example for simplicity.
  -->
  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>

  <!--
    This must match the role-name in the auth-constraint above.
  -->
  <security-role>
    <role-name>connect</role-name>
  </security-role>
</web-app>
```

As noted above, the REST server will not function properly unless security is configured. All authorization methods supported by the Servlet specification are supported by ModeShape and can be used interchangeable, as long as authenticated users have the connect role listed above.

11.3. Deploying the ModeShape REST Server

Deploying the ModeShape REST server only requires three steps: *preparing the web configuration*, configuring the users and their roles in your web container (outside the scope of this document), and assembling the WAR. This section describes the requirements for assembling the WAR.

If you are using Maven to build your projects, the WAR can be built from a POM. Here is a portion of the POM used to build the ModeShape REST Server integration subproject.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0    http://maven.apache.org/maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <artifactId>modeshape</artifactId>
      <groupId>org.modeshape</groupId>
      <version>1.0.0.Final</version>
      <relativePath>../..</relativePath>
   </parent>
   <artifactId>modeshape-web-jcr-rest-war</artifactId>
   <packaging>war</packaging>
   <name>ModeShape JCR REST Servlet</name>
   <description>ModeShape servlet that provides RESTful access to JCR items</description>
   <url>http://www.modeshape.org</url>
   <dependencies>
      <dependency>
         <groupId>org.modeshape</groupId>
         <artifactId>modeshape-web-jcr-rest</artifactId>
         <version>1.0.0.Final</version>
      </dependency>

      <dependency>
         <groupId>org.slf4j</groupId>
         <artifactId>slf4j-log4j12</artifactId>
         <version>1.5.8</version>
         <scope>runtime</scope>
      </dependency>

      <dependency>
         <groupId>org.jboss.resteasy</groupId>
         <artifactId>resteasy-client</artifactId>
         <version>1.1.GA</version>
      </dependency>
   </dependencies>
```

```
</project>
```

If you use this approach, make sure that web configuration file is in the `/src/main/webapp/WEB-INF` directory.

The JBoss REST Server WAR is still easy enough to build if you are not using Maven. Simply construct a WAR with the following contents:

```

+ /WEB-INF
 + /classes
 | + configRepository.xml
 | + log4j.properties (Optional)
 + /lib
 | + activation-1.1.jar
 | + commons-codec-1.2.jar
 | + commons-httpclient-3.1.jar
 | + commons-logging-1.0.4.jar
 | + google-collections-1.0.0.Final.jar
 | + hamcrest-core-1.1.jar
 | + httpclient-4.0.jar
 | + httpcore-4.0.1.jar
 | + jakarta-regexp-1.4.jar
 | + javassist-3.6.0.GA.jar
 | + jaxb-api-2.1.jar
 | + jaxb-impl-2.1.12.jar
 | + jaxrs-api-1.2.1-GA.jar
 | + jcip-annotations-1.0.jar
 | + jcl-over-slf4j-1.5.8.jar
 | + jcr-1.0.1.jar
 | + jettison-1.1.jar
 | + joda-time-1.6.jar
 | + jsr250-api-1.0.jar
 | + junit-dep-4.4.jar
 | + lucene-analyzers-3.0.0.jar
 | + lucene-core-3.0.0.jar
 | + lucene-regex-3.0.0.jar
 | + lucene-snowball-3.0.0.jar
 | + modeshape-cnd-1.0.0.Final.jar
 | + modeshape-common-1.0.0.Final.jar
 | + modeshape-graph-1.0.0.Final.jar
 | + modeshape-jcr-1.0.0.Final.jar
 | + modeshape-repository-1.0.0.Final.jar
 | + modeshape-search-lucene-1.0.0.Final.jar
 | + modeshape-web-jcr-rest-1.0.0.Final.jar
 | + resteasy-jaxb-provider-1.2.1-GA.jar
 | + resteasy-jettison-provider-1.2.1-GA.jar
 | + scannotation-1.0.2.jar
 | + servlet-api-2.5.jar
 | + sjsxp-1.0.1.jar
 | + slf4j-api-1.5.8.jar
 | + slf4j-log4j12-1.5.8.jar
 | + slf4j-simple-1.5.8.jar
 | + stax-api-1.0-2.jar
 + web.xml
```

If you are using sequencers or any connectors other than the in-memory or federated connector, you will also have to add the JARs for those dependencies into the `WEB-INF/lib` directory as well. You will also have to change the version numbers on the JARs to reflect the current version of ModeShape.

This WAR can be deployed into your servlet container.

11.4. Repository Providers

The ModeShape REST server can also be used as an interface to to other JCR repositories by creating an implementation of the *`RepositoryProvider`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/web/jcr/rest/spi/RepositoryProvider.html] interface that connects to the other repository.

The *`RepositoryProvider`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/web/jcr/rest/spi/RepositoryProvider.html] only has a few methods that must be implemented. When the *`ModeShapeJcrDeployer`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/web/jcr/rest/ModeShapeJcrDeployer.html] starts up, it will dynamically load the *`RepositoryProvider`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/web/jcr/rest/spi/RepositoryProvider.html] implementation (as noted above) and call the `startup(ServletContext)` method on the provider. The provider can use this method to load any required configuration parameters from the web configuration (web.xml) and initialize the repository.

As an example, here's the ModeShape JCR provider implementation of this method with exception handling omitted for brevity.

```
public void startup( ServletContext context ) {
    String configFile = context.getInitParameter(CONFIG_FILE);
```

```
     InputStream configFileInputStream = getClass().getResourceAsStream(configFile);
     jcrEngine = new JcrConfiguration().loadFrom(configFileInputStream).build();
     jcrEngine.start();
}
```

As you can see, the name of configuration file for the `JcrEngine` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrEngine.html] is read from the servlet context and used to initialize the engine. Once the repository has been started, it is now ready to accept the main methods that provide the interface to the repository.

The first method returns the set of repository names supported by this REST server.

```


public Set<String> getJcrRepositoryNames() {
    return new HashSet<String>(jcrEngine.getRepositoryNames());
}
```

The ModeShape JCR repository does support multiple repositories on the same server. Other JCR implementations that don't support multiple repositories are free to return a singleton set containing any string from this method.

The other required method returns an open JCR Session for the user from the current request in a given repository and workspace. The provider can use the `HttpServletRequest` [http://java.sun.com/javaee/5/docs/api/javax/servlet/http/HttpServletRequest.html] to get the authentication credentials for the HTTP user.

```


public Session getSession( HttpServletRequest request,
                           String repositoryName,
                           String workspaceName ) throws RepositoryException {
    Repository repository = getRepository(repositoryName);

 SecurityContext context = new ServletSecurityContext(request);
 Credentials credentials = new SecurityContextCredentials(context);
    return repository.login(credentials, workspaceName);
}
```

The `getSession(...)` method is used by most of the REST server methods to access the JCR repository and return results as needed.

Finally, the `shutdown()` method signals that the web context is being undeployed and the JCR repository should shutdown and clean up any resources that are in use.

11.5. ModeShape REST Client API

The ModeShape REST Client API provides a POJO way of using the ModeShape REST web service to publish (upload) and unpublish (delete) files from ModeShape repositories. Java objects open the HTTP connection, create the HTTP request URLs, attach the payload associated with `PUT` and `POST` requests, parse the HTTP JSON response back into Java objects, and close the HTTP connection.

Here are the Java business objects you will need (all found in the `org.modeshape.web.jcr.rest.client.domain` package):

- `Server` - hosts one or more ModeShape JCR repositories,
- `Repository` - a ModeShape JCR repository containing one or more workspaces, and
- `Workspace` - a ModeShape JCR repository workspace.

Along with the POJOs above, an `org.modeshape.web.jcr.rest.client.IRestClient` is needed. The `IRestClient` is responsible for executing the publishing and unpublishing operations. You can also use the `IRestClient` to find out what repositories and workspaces are available on a ModeShape server.

Note

The only implementation of `IRestClient` is `JsonRestClient` as JSON-encoded responses are all that are currently available.

Here's a code snippet that publishes (uploads) a file:

```
// Setup POJOs
Server server = new Server("http://localhost:8080", "username", "password");
Repository repository = new Repository("repositoryName", server);
Workspace workspace = new Workspace("workspaceName", repository);

// Publish
File file = new File("/path/to/file");
IRestClient restClient = new JsonRestClient();
Status status = restClient.publish(workspace, "/workspace/path/", file);

if (status.isError() {
   // Handle error here
}
```

Successfully executing the above code results in the creation a JCR folder node (`nt:folder`) for each segment of the workspace path (if the folder didn't already exist). Also, a JCR file node (a node with primary type `nt:file`) is created or updated under the last folder node and the file contents are encoded and uploaded into a child node of that file node.

Part IV. Connector Library

The ModeShape project provides a number of *connectors* out-of-the-box. These are ready to be used by simply including them in the classpath and *configuring* them as a repository source.

In-Memory Connector

The in-memory repository connector is a simple connector that creates a transient, in-memory repository. This repository is used as a very simple in-memory cache or as a standalone transient repository. This connector works well for a readable and writable repository source with small to moderate sized content that need not be permanently saved.

The *`InMemoryRepositorySource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/inmemory/InMemoryRepositorySource.html] class provides a number of JavaBean properties that control its behavior:

Table 12.1. *`InMemoryRepositorySource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/inmemo properties

Property	Description
defaultCachePolicy	Optional property that, if used, defines the default for how long this information provided by this source may to be cached by other, higher-level components. The default value of null implies that this source does not define a specific duration for caching information provided by this repository source.
defaultWorkspaceName	Optional property that is initialized to an empty string and which defines the name for the workspace that will be used by default if none is specified.
jndiName	Optional property that, if used, specifies the name in JNDI where an *`InMemoryRepository`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/inmemory/InMemoryRepository.html] instance can be found. This is an advanced property that is infrequently used.
name	The name of the repository source, which is used by the *`RepositoryService`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] by name.

Property	Description
rootNodeUuid	Optional property that, if used, defines the UUID of the root node in the in-memory repository. If not used, then a new UUID is generated.
retryLimit	Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.

One way to configure the in-memory connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the `InMemoryRepositorySource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/inmemory/InMemoryRepositorySource.html] class. For example:

```
JcrConfiguration config = ...
config.repositorySource("IMR Store")
      .usingClass(InMemoryRepositorySource.class)
      .setDescription("The repository for our content")
      .setProperty("defaultWorkspaceName", workspaceName);
```

Another way to configure the in-memory connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance and load an XML configuration file that contains a repository source that uses the `InMemoryRepositorySource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/inmemory/InMemoryRepositorySource.html] class. For example a file named configRepository.xml can be created with these contents:

```
<?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 sources for the content.  These sources are directly accessible using the
   ModeShape-specific Graph API.  In fact, this is how the ModeShape JCR implementation works.
  You
```

```
    can think of these as being similar to JDBC DataSource objects, except that they expose
    graph content via the Graph API instead of records via SQL or JDBC.
    -->
    <mode:sources jcr:primaryType="nt:unstructured">
        <!--
        The 'IMR Store' repository is an in-memory source with a single default workspace (though
        others could be created, too).
        -->
        <mode:source jcr:name="IMR Store"

 mode:classname="org.modeshape.graph.connector.inmemory.InMemoryRepositorySource"
                    mode:description="The repository for our content"
                    mode:defaultWorkspaceName="default"/>
    </mode:sources>

    <!-- MIME type detectors and JCR repositories would be defined below -->
</configuration>
```

The configuration can then be loaded from Java like this:

```
JcrConfiguration config = new JcrConfiguration().loadFrom("/configRepository.xml");
```

File System Connector

This connector exposes an area of the local file system as a graph of "nt:file" and "nt:folder" nodes. The connector can be configured so that the workspace name is either a path to the directory on the file system that represents the root of that workspace *or* the name of subdirectory within a root directory (see the `workspaceRootPath` property below). Each connector can define whether it allows new workspaces to be created, but if so the names of the new workspaces must represent valid paths to existing directories.

Note

The file nodes returned by this connector will have a primary type of `nt:file` and a child node named `jcr:content`. The `jcr:content` node will have a primary type of `mode:resource`. The `mode:resource` node type is equivalent to the built-in `nt:resource` node type in all ways except one: it does not extend `mix:referenceable`. This is because ModeShape cannot assign a persistent UUID to the files in the file system or guarantee that no other process will move or delete the files outside of ModeShape. The `mix:referenceable` node type cannot be implemented if either of these conditions cannot be met. Additional properties (including mixin types) can be added by setting the `customPropertiesFactory` property to point to an implementation of the `CustomPropertiesFactory` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/filesystem/CustomPropertiesFactory] interface.

The `FileSystemSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/filesystem/FileSystemSource.html] class provides a number of JavaBean properties that control its behavior:

Table 13.1. `FileSystemSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/filesystem/Fil properties

Property	Description
cachePolicy	Optional property that, if used, defines the cache policy for this repository source. When not used, this source will not define a specific duration for caching information.
creatingWorkspaceAllowed	Optional property that defines whether clients can create additional workspaces. The default value is "true".
defaultWorkspaceName	Optional property that is initialized to `"default"` and which defines the name for

Property	Description
	the workspace that will be used by default if none is specified.
name	The name of the repository source, which is used by the `RepositoryService` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] by name.
predefinedWorkspaceNames	Optional property that, if used, defines names of the workspaces that are predefined and need not be created before being used. This can be coupled with a "false" value for the "creatingWorkspaceAllowed" property to allow only the use of only predefined workspaces.
retryLimit	Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.
rootNodeUuid	Optional property that, if used, specifies the UUID that should be used for the root node of each workspace. If no value is specified, a default UUID is used.
updatesAllowed	Determines whether the content in the file system can be updated ("true"), or if the content may only be read ("false"). The default value is "false" to avoid unintentional security vulnerabilities.
workspaceRootPath	Optional property that, if used, specifies a path on the local file system to the root of all workspaces. The source will will use the name of the workspace as a relative path from the `workspaceRootPath` to determine the path for a particular workspace. If no value (or a `null` value) is specified, the source will use the name of the workspace

Property	Description
	as a relative path from the current working directory of this virtual machine (as defined by `new File(".")`. As an example for a workspace named `"default/foo"`, the source will use `new File(workspaceRootPath, "default/foo")` as the source directory for the connector if `workspaceRootPath` is set to a non-null value, or `new File(".", "default/foo")` as the source directory for the connector if `workspaceRootPath` is set to `null`.
customPropertiesFactory	Specifies the *`CustomPropertiesFactory`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/filesystem/CustomPropertiesFactory] implementation that should be used to augment the default properties available on each node.

One way to configure the file system connector is to create *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the *`FileSystemSource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/filesystem/FileSystemSource.html] class. For example:

```
JcrConfiguration config = ...
config.repositorySource("FS Store")
      .usingClass(FileSystemSource.class)
      .setDescription("The repository for our content")
      .setProperty("workspaceRootPath", "/home/content/someApp")
      .setProperty("defaultWorkspaceName", "prod")
      .setProperty("predefinedWorkspaceNames", new String[] { "staging", "dev"})
      .setProperty("rootNodeUuid", UUID.fromString("fd129c12-81a8-42ed-aa4b-820dba49e6f0")
      .setProperty("updatesAllowed", "true")
      .setProperty("creatingWorkspaceAllowed", "false");
```

Another way to configure the file system connector is to create *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance and load an XML configuration file that contains a repository source that uses the *`FileSystemSource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/

filesystem/FileSystemSource.html] class. For example a file named configRepository.xml can be created with these contents:

```
<?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 sources for the content.  These sources are directly accessible using the
  ModeShape-specific Graph API. In fact, this is how the ModeShape JCR implementation works.
 You can
  think of these as being similar to JDBC DataSource objects, except that they expose graph
  content via the Graph API instead of records via SQL or JDBC.
  -->
  <mode:sources jcr:primaryType="nt:unstructured">
    <!--
    The 'FS Store' repository is a file system source with a three predefined workspaces
    ("prod", "staging", and "dev").
    -->
    <mode:source jcr:name="FS Store"
      mode:classname="org.modeshape.connector.filesystem.FileSystemSource"
      mode:description="The repository for our content"
      mode:workspaceRootPath="/home/content/someApp"
      mode:defaultWorkspaceName="prod"
      mode:predefinedWorkspaceNames="staging, dev"
      mode:rootNodeUuid="fd129c12-81a8-42ed-aa4b-820dba49e6f0"
      mode:creatingWorkspacesAllowed="false"
      mode:updatesAllowed="true" >
      <!--
     If desired, specify a cache policy that caches items in memory for 5 minutes (300000 ms).
      This fragment can be left out if the connector should not cache any content.
      -->
      <mode:cachePolicy jcr:name="cachePolicy"

 mode:classname="org.modeshape.graph.connector.path.cache.InMemoryWorkspaceCache$InMemoryCachePoli
        mode:timeToLiveInMilliseconds="300000" />
    </mode:source>
  </mode:sources>

  <!-- MIME type detectors and JCR repositories would be defined below -->
</configuration>
```

The configuration can then be loaded from Java like this:

```
JcrConfiguration config = new JcrConfiguration().loadFrom("/configRepository.xml");
```

JPA Connector

This connector stores a graph of any structure or size in a relational database, using a JPA provider on top of a JDBC driver. Currently this connector relies upon some Hibernate-specific capabilities. The schema of the database is dictated by this connector and is optimized for storing a graph structure. (In other words, this connector does not expose as a graph the data in an existing database with an arbitrary schema.)

The `JpaSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/store/jpa/JpaSource.html] class provides a number of JavaBean properties that control its behavior:

Table 14.1. `JpaSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/store/jpa/JpaS properties

Property	Description
autoGenerateSchema	Sets the Hibernate setting dictating what it does with the database schema upon first connection. Valid values are as follows (though the value is not checked): • "`create`" - Create the database schema objects when the `EntityManagerFactory` [http://java.sun.com/j2se/1.5.0/docs/api/javax/persistence/EntityManagerFactory.html] is created (actually when Hibernate's SessionFactory is created by the entity manager factory). If a file named "import.sql" exists in the root of the class path (e.g., '/import.sql') Hibernate will read and execute the SQL statements in this file after it has created the database objects. Note that Hibernate first delete all tables, constraints, or any other database object that is going to be created in the process of building the schema. • "`create-drop`" - Same as "`create`", except that the schema will be dropped after the `EntityManagerFactory` [http://java.sun.com/j2se/1.5.0/docs/api/javax/persistence/EntityManagerFactory.html] is closed.

Property	Description
	• "`update`" - Attempt to update the database structure to the current mapping (but does not read and invoke the SQL statements from "import.sql"). *Use with caution.* • "`validate`" - Validates the existing schema with the current entities configuration, but does not make any changes to the schema (and does not read and invoke the SQL statements from "import.sql"). This is often the proper setting to use in production, and thus this is the default value.
cacheTimeToLiveInMilliseconds	Optional property that, if used, defines the maximum time in milliseconds that any information returned by this connector is allowed to be cached before being considered invalid. When not used, this source will not define a specific duration for caching information. The default value is "600000" milliseconds, or 10 minutes.
compressData	An advanced boolean property that dictates whether large binary and string values should be stored in a compressed form. This is enabled by default. Setting this value only affects how new records are stored; records can always be read regardless of the value of this setting. The default value is "true".
creatingWorkspaceAllowed	Optional property that defines whether clients can create additional workspaces. The default value is "true".
dialect	Required property that defines the dialect of the database. This must match one of the Hibernate dialect names, and must correspond to the type of driver being used.
dataSourceJndiName	The JNDI name of the JDBC DataSource instance that should be used. If not specified, the other driver properties must be set.
driverClassloaderName	The name of the class loader or classpath that should be used to load the JDBC driver class. This is not required if the DataSource is found in JNDI.

Property	Description
driverClassName	The name of the JDBC driver class. This is not required if the DataSource is found in JNDI, but is required otherwise.
idleTimeInSecondsBeforeTestingConnections	The number of seconds after a connection remains in the pool that the connection should be tested to ensure it is still valid. The default is 180 seconds (or 3 minutes).
largeValueSizeInBytes	An advanced boolean property that controls the size of property values at which they are considered to be "large values". Depending upon the model, large property values may be stored in a centralized area and keyed by a secure hash of the value. This is an space and performance optimization that stores each unique large value only once. The default value is "1024" bytes, or 1 kilobyte.
maximumConnectionsInPool	The maximum number of connections that may be in the connection pool. The default is "5".
maximumConnectionIdleTimeInSeconds	The maximum number of seconds that a connection should remain in the pool before being closed. The default is "600" seconds (or 10 minutes).
maximumSizeOfStatementCache	The maximum number of statements that should be cached. Statement caching can be disabled by setting to "0". The default is "100".
minimumConnectionsInPool	The minimum number of connections that will be kept in the connection pool. The default is "0".
model	An advanced property that dictates the type of storage schema that is used. Currently, the only supported values are "Basic" and "Simple". The Basic model supports a read-only mode (q.v., the "updatesAllowed" property) and database-level enforcement of referential integrity (q.v., the "referentialIntegrityEnforced" property above), but does not fully support all JCR functions. As a result, the Simple model is now the default model, but ModeShape repositories that were created under the

Property	Description
	Basic model will continue to use the "Basic" model regardless of the value of this property. Repositories can be converted from the Basic model to the Simple model by exporting them to an XML file as a system view through the JCR interface and then importing them into a new repository created with the model property set to "Simple".
name	The name of the repository source, which is used by the `RepositoryService` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] by name.
nameOfDefaultWorkspace	Optional property that is initialized to an empty string and which defines the name for the workspace that will be used by default if none is specified.
numberOfConnectionsToAcquireAsNeeded	The number of connections that should be added to the pool when there are not enough to be used. The default is "1".
password	The password that should be used when creating JDBC connections using the JDBC driver class. This is not required if the DataSource is found in JNDI.
predefinedWorkspaceNames	Optional property that, if used, defines names of the workspaces that are predefined and need not be created before being used. This can be coupled with a "false" value for the "creatingWorkspaceAllowed" property to allow only the use of only predefined workspaces.
referentialIntegrityEnforced (Basic Model Only)	An advanced boolean property that dictates whether the database's referential integrity should be enabled, or false if this checking is not to be used. While referential integrity does help to ensure the consistency of the records, it does add work to update operations and can impact performance. The Simple Model

Property	Description
	(q.v., the "model" property below) ignores this property and does not support this feature. The default value is "true".
retryLimit	Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.
rootNodeUuid	Optional property that, if used, defines the UUID of the root node in the repository. If not used, then a new UUID is generated.
updatesAllowed	Determines whether the content in the database is can be updated ("true"), or if the content may only be read ("false"). The default value is "true".
url	The URL that should be used when creating JDBC connections using the JDBC driver class. This is not required if the DataSource is found in JNDI.
username	The username that should be used when creating JDBC connections using the JDBC driver class. This is not required if the DataSource is found in JNDI.

One way to configure the JPA connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the `JpaSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/store/jpa/JpaSource.html] class. For example:

```
JcrConfiguration config = ...
config.repositorySource("JPA Store")
      .usingClass(JpaSource.class)
      .setDescription("The database store for our content")
      .setProperty("dialect", "org.hibernate.dialect.MySQLDialect")
      .setProperty("dataSourceJndiName", "java:/MyDataSource")
      .setProperty("defaultWorkspaceName", "My Default Workspace")
      .setProperty("autoGenerateSchema", "validate");
```

Of course, setting other more advanced properties would entail calling `setProperty(...)` for each. Since almost all of the properties have acceptable default values, however, we don't need to set very many of them.

Another way to configure the JPA connector is to create *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance and load an XML configuration file that contains a repository source that uses the *`JpaSource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/store/jpa/JpaSource.html] class. For example a file named configRepository.xml can be created with these contents:

```
<?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 sources for the content.  These sources are directly accessible using the
   ModeShape-specific Graph API.  In fact, this is how the ModeShape JCR implementation works.
  You
    can think of these as being similar to JDBC DataSource objects, except that they expose
    graph content via the Graph API instead of records via SQL or JDBC.
    -->
    <mode:sources jcr:primaryType="nt:unstructured">
        <!--
        The 'JPA Store' repository is an JPA source with a single default workspace (though
        others could be created, too).
        -->
        <mode:source jcr:name="JPA Store"
                    mode:classname="org.modeshape.graph.connector.store.jpa.JpaSource"
                    mode:description="The database store for our content"
                    mode:dialect="org.hibernate.dialect.MySQLDialect"
                    mode:dataSourceJndiName="java:/MyDataSource"
                    mode:defaultWorkspaceName="default"
                    mode:autoGenerateSchema="validate"/>
    </mode:sources>

    <!-- MIME type detectors and JCR repositories would be defined below -->
</configuration>
```

The configuration can then be loaded from Java like this:

```
JcrConfiguration config = new JcrConfiguration().loadFrom("/configRepository.xml");
```

ModeShape users who prefer not to give DDL privileges to the ModeShape database user for this connector can use the ModeShape JPA DDL generation tool to create the proper DDL files for their database dialect. This tool is packaged as an executable jar in the utils/modeshape-jpa-ddl-gen subproject and can be executed with the following syntax:

```
java -jar <jar_name> -dialect <dialect name> -model <model_name> [-out <path to output
 directory>]
```

The dialect and model parameters should match the value of the `dialect` and `model` properties specified for the JPA connector.

Running this executable will create two files in the output directory (or the current directory if no output directory was specified): create.modeshape-jpa-connector.ddl and drop.modeshape-jpa-connector.ddl. The former contains the DDL to create or replace the tables, foreign keys, indices, and sequences needed by the JPA connector and the latter contains the DDL to drop any tables, foreign keys, indices, and sequences needed by the JPA connector.

14.1. Basic Model

This database schema model stores node properties as opaque records and children as transparent records. Large property values are stored separately.

The set of tables used in this model includes:

- Workspaces - the set of workspaces and their names.
- Namespaces - the set of namespace URIs used in paths, property names, and property values.
- Properties - the properties for each node, stored in a serialized (and optionally compressed) form.
- Large values - property values larger than a certain size will be broken out into this table, where they are tracked by their SHA-1 has and shared by all properties that have that same value. The values are stored in a binary (and optionally compressed) form.
- Children - the children for each node, where each child is represented by a separate record. This approach makes it possible to efficiently work with nodes containing large numbers of children, where adding and removing child nodes is largely independent of the number of children. Also, working with properties is also completely independent of the number of child nodes.
- ReferenceChanges - the references from one node to another

- Subgraph - a working area for efficiently computing the space of a subgraph; see below
- Options - the parameters for this store's configuration (common to all models)

This database model contains two tables that are used in an efficient mechanism to find all of the nodes in the subgraph below a certain node. This process starts by creating a record for the subgraph query, and then proceeds by executing a join to find all the children of the top-level node, and inserting them into the database (in a working area associated with the subgraph query). Then, another join finds all the children of those children and inserts them into the same working area. This continues until the maximum depth has been reached, or until there are no more children (whichever comes first). All of the nodes in the subgraph are then represented by records in the working area, and can be used to quickly and efficient work with the subgraph nodes. When finished, the mechanism deletes the records in the working area associated with the subgraph query.

This subgraph query mechanism is extremely efficient, performing one join/insert statement *per level of the subgraph*, and is completely independent of the number of nodes in the subgraph. For example, consider a subgraph of node A, where A has 10 children, and each child contains 10 children, and each grandchild contains 10 children. This subgraph has a total of 1111 nodes (1 root + 10 children + 10*10 grandchildren + 10*10*10 great-grandchildren). Finding the nodes in this subgraph would normally require 1 query per node (in other words, 1111 queries). But with this subgraph query mechanism, all of the nodes in the subgraph can be found with 1 insert plus 4 additional join/inserts.

This mechanism has the added benefit that the set of nodes in the subgraph are kept in a working area in the database, meaning they don't have to be pulled into memory.

Subgraph queries are used to efficiently process a number of different requests, including `ReadBranchRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ReadBranchRequest.html], `DeleteBranchRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/DeleteBranchRequest.html], `MoveBranchRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/MoveBranchRequest.html], and `CopyBranchRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/CopyBranchRequest.html]. Processing each of these kinds of requests requires knowledge of the subgraph, and in fact all but the `ReadBranchRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ReadBranchRequest.html] need to know the complete subgraph.

14.2. Simple Model

This database schema model stores node properties as opaque records in the same row as transparent values like the node's namespace, local name, and same-name-sibling index. Large property values are stored separately. It is a small evolution of the design from the Basic model.

The set of tables used in this model includes:

- Workspaces - the set of workspaces and their names.

- Namespaces - the set of namespace URIs used in paths, property names, and property values.
- Nodes - the nodes in the repository, where each node and its properties are represented by a single record. This approach makes it possible to efficiently work with nodes containing large numbers of children, where adding and removing child nodes is largely independent of the number of children. Since the primary consumer of ModeShape graph information is the JCR layer, and the JCR layer always retrieves the nodes' properties for retrieved nodes, the properties have been moved in-row with the nodes. Properties are still store in an opaque, serialized (and optionally compressed) form.
- Large values - property values larger than a certain size will be broken out into this table, where they are tracked by their SHA-1 has and shared by all properties that have that same value. The values are stored in a binary (and optionally compressed) form. This is equivalent to the Basic model's approach for storing large values.
- Subgraph - a working area for efficiently computing the space of a subgraph; see below
- Options - the parameters for this store's configuration (common to all models)

Just like the Basic model, this model contains two tables that are used in an efficient mechanism to find all of the nodes in the subgraph below a certain node. The subgraph tables work so similarly in the Simple model that the description from the Basic model still applies.

In the Simple model, subgraph queries are used to efficiently process a number of different requests, including *`ReadBranchRequest`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ReadBranchRequest.html] and *`DeleteBranchRequest`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/DeleteBranchRequest.html]. Processing each of these kinds of requests requires knowledge of the subgraph, and in fact all but the *`ReadBranchRequest`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ReadBranchRequest.html] need to know the complete subgraph.

Federation Connector

The federated repository source provides a unified repository consisting of information that is dynamically federated from multiple other *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] instances. This is a very powerful repository source that appears to be a single repository, when in fact the content is stored and managed in multiple other systems. Each `FederatedRepositorySource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/federation/FederatedRepositorySource.html] is typically configured with the name of another *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] that should be used as the local, unified cache of the federated content. The `FederatedRepositorySource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/federation/FederatedRepositorySource.html] then looks in the configuration repository to determine the various workspaces and how other sources are projected into each workspace.

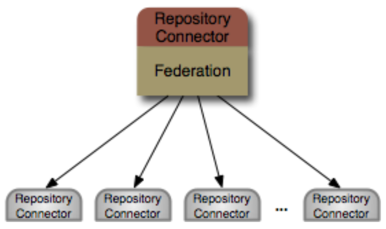

Figure 15.1. Federating multiple sources using the Federated Repository Connector

15.1. Projections

Each federated repository source provides a unified repository consisting of information that is dynamically federated from multiple other *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] instances. The connector is configured with a number of **projections** that each describe where in the unified repository the federated connector should place the content from another source. Projections consist of the name of the source containing the content and a number of **rules** that define the path mappings, where each rule is defined as a string with this format:

```
pathInFederatedRepository => pathInSourceRepository
```

Here, the `pathInFederatedRepository` is the string representation of the path in the unified (or federated) repository, and `pathInSourceRepository` is the string representation of the path of the actual content in the underlying source. For example:

```
/ => /
```

is a trivial rule that states that all of the content in the underlying source should be mapped into the unified repository such that the locations are the same. Therefore, a node at `/a/b/c` in the source would appear in the unified repository at `/a/b/c`. This is called a **mirror projection**, since the unified repository mirrors the underlying source repository.

Another example is an **offset projection**, which is similar to the mirror projection except that the federated path includes an offset not found in the source:

```
/alpha/beta => /
```

Here, a node at `/a/b/c` in the source would actually appear in the unified repository at `/alpha/beta/a/b/c`. The offset path (`/alpha/beta` in this example) can have 1 or more segments. (If there are no segments, then it reduces to a mirror projection.)

Often a rule will map a path in one source into another path in the unified source:

```
/alpha/beta => /foo/bar
```

Here, the content at `/foo/bar` is projected in the unified repository under `/alpha/beta`, meaning that the `/foo/bar` prefix never even appears in the unified repository. So the node at `/foo/bar/baz/raz` would appear in the unified repository at `/alpha/beta/baz/raz`. Again, the size of the two paths in the rule don't matter.

15.2. Multiple Projections

Federated repositories that use a single projection are useful, but they aren't as interesting or powerful as those that use multiple projections. Consider a federated repository that is defined by two projections:

```

/ => /                          for source "S1"
/alpha => /foo/bar              for source "S2"
```

And consider that S1 contains the following structure:

```

+- a
```

```
| +- i
| +- j
+- b
  +- k
  +- m
  +- n
```

and S2 contains the following:

```
+- foo
  +- bar
  | +- baz
  | | +- taz
  | | +- zaz
  | +- raz
  +- bum
    +- bot
```

The unified repository would then have this structure:

```
+- a
| +- i
| +- j
+- b
| +- k
| +- m
| +- n
+- alpha
  +- baz
    +- taz
    | +- zaz
    +- raz
```

Note how the `/foo/bum` branch does not even appear in the unified repository, since it is outside of the branch being projected. Also, the `/alpha` node doesn't exist in S1 or S2; it's what is called a **placeholder** node that exists purely so that the nodes below it have a place to exist. Placeholders are somewhat special: they allow any structure below them (including other placeholder nodes or real projected nodes), but they cannot be modified.

Even more interesting are cases that involve more projections. Consider a federated repository that contains information about different kinds of automobiles, aircraft, and spacecraft, except that the information about each kind of vehicle exists in a different source (and possibly a different *kind* of source, such as a database, or file, or web service).

First, the sources. The "Cars" source contains the following structure:

```
+- Cars
   +- Hybrid
   |  +- Toyota Prius
   |  +- Toyota Highlander
   |  +- Nissan Altima
   +- Sports
   |  +- Aston Martin DB9
   |  +- Infinity G37
   +- Luxury
   |  +- Cadillac DTS
   |  +- Bentley Continental
   |  +- Lexus IS350
   +- Utility
      +- Land Rover LR2
      +- Land Rover LR3
      +- Hummer H3
      +- Ford F-150
```

The "Aircraft" source contains the following structure:

```
+- Aviation
   +- Business
   |  +- Gulfstream V
   |  +- Learjet 45
   +- Commercial
   |  +- Boeing 777
   |  +- Boeing 767
   |  +- Boeing 787
   |  +- Boeing 757
   |  +- Airbus A380
   |  +- Airbus A340
   |  +- Airbus A310
   |  +- Embraer RJ-175
   +- Vintage
```

```
  |  +- Fokker Trimotor
  |  +- P-38 Lightning
  |  +- A6M Zero
  |  +- Bf 109
  |  +- Wright Flyer
  +- Homebuilt
     +- Long-EZ
     +- Cirrus VK-30
     +- Van's RV-4
```

Finally, our "Spacecraft" source contains the following structure:

```
+- Space Vehicles
  +- Manned
  |  +- Space Shuttle
  |  +- Soyuz
  |  +- Skylab
  |  +- ISS
  +- Unmanned
  |  +- Sputnik
  |  +- Explorer
  |  +- Vanguard
  |  +- Pioneer
  |  +- Marsnik
  |  +- Mariner
  |  +- Mars Pathfinder
  |  +- Mars Observer
  |  +- Mars Polar Lander
  +- Launch Vehicles
  |  +- Saturn V
  |  +- Aries
  |  +- Delta
  |  +- Delta II
  |  +- Orion
  +- X-Prize
     +- SpaceShipOne
     +- WildFire
     +- Spirit of Liberty
```

So, we can define our unified "Vehicles" source with the following projections:

```
/Vehicles => /                          for source "Cars"
/Vehicles/Aircraft => /Aviation          for source "Aircraft"
/Vehicles/Spacecraft => /Space Vehicles     for source "Spacecraft"
```

The result is a unified repository with the following structure:

```
+- Vehicles
   +- Cars
   |  +- Hybrid
   |  |  +- Toyota Prius
   |  |  +- Toyota Highlander
   |  |  +- Nissan Altima
   |  +- Sports
   |  |  +- Aston Martin DB9
   |  |  +- Infinity G37
   |  +- Luxury
   |  |  +- Cadillac DTS
   |  |  +- Bentley Continental
   |  +- Lexus IS350
   |  +- Utility
   |     +- Land Rover LR2
   |     +- Land Rover LR3
   |     +- Hummer H3
   |     +- Ford F-150
   +- Aircraft
   |  +- Business
   |  |  +- Gulfstream V
   |  |  +- Learjet 45
   |  +- Commercial
   |  |  +- Boeing 777
   |  |  +- Boeing 767
   |  |  +- Boeing 787
   |  |  +- Boeing 757
   |  |  +- Airbus A380
   |  |  +- Airbus A340
   |  |  +- Airbus A310
   |  |  +- Embraer RJ-175
   |  +- Vintage
   |  |  +- Fokker Trimotor
   |  |  +- P-38 Lightning
   |  |  +- A6M Zero
   |  |  +- Bf 109
   |  |  +- Wright Flyer
   |  +- Homebuilt
   |     +- Long-EZ
   |     +- Cirrus VK-30
   |     +- Van's RV-4
   +- Spacecraft
      +- Manned
      |  +- Space Shuttle
      |  +- Soyuz
      |  +- Skylab
      |  +- ISS
      +- Unmanned
      |  +- Sputnik
      |  +- Explorer
      |  +- Vanguard
      |  +- Pioneer
      |  +- Marsnik
      |  +- Mariner
      |  +- Mars Pathfinder
      |  +- Mars Observer
      |  +- Mars Polar Lander
      +- Launch Vehicles
      |  +- Saturn V
      |  +- Aries
      |  +- Delta
      |  +- Delta II
      |  +- Orion
      +- X-Prize
         +- SpaceShipOne
         +- WildFire
         +- Spirit of Liberty
```

Other combinations are of course possible.

15.3. Processing flow

This connector executes `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html]s against the federated repository by projecting them into requests against the underlying sources that are being federated.

One important design of the connector framework is that requests can be submitted in a batch, which may be processed more efficiently than if each request was submitted one at a time.

This connector design accomplishes this by projecting the incoming requests into requests against each source, then submitting the batch of projected requests to each source, and then transforming the results of the projected requests back into original requests.

This is accomplished using a three-step process:

1. Process the incoming requests and for each generate the appropriate request(s) against the sources (dictated by the workspace's projections). These "projected requests" are then enqueued for each source.

2. Submit each batch of projected requests to the appropriate source, in parallel where possible. Note that the requests are still ordered correctly for each source.

3. Accumulate the results for the incoming requests by post-processing the projected requests and transforming the source-specific results back into the federated workspace (again, using the workspace's projections).

This process is a form of the *fork-join* divide-and-conquer algorithm, which involves splitting a problem into smaller parts, forking new subtasks to execute each smaller part, joining on the subtasks (waiting until all have finished), and then composing the results. Technically, Step 2 performs the fork and join operations, but this class uses `RequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/processor/RequestProcessor.html] implementations to do Step 1 and 3 (called `ForkRequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/federation/ForkRequestProcessor.html] and `JoinRequestProcessor` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/federation/JoinRequestProcessor.html], respectively).

Such fork-join style techniques are well-suited to *parallel processing*. This connector uses an *ExecutorService* [http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ExecutorService.html] to allow these different processors to operate concurrently. This can greatly improve the performance as perceived by the clients, since indeed much of the operations on the different sources are occurring at the same time.

It is also possible that not every incoming `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] get projected to all sources. Indeed, many operations can effectively be mapped to a *single projection*. In such cases, the overhead of the federated connector is quite minimal.

Note

`Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html]s that include the *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html] within the request's `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/

modeshape/graph/Location.html] can be very quickly mapped to the correct projection, and thus such federated requests can be processed with very little overhead. However, when requests contain `Location` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/Location.html]s that only contain identification properties (e.g., UUIDs), the connector may not be able to determine the correct projection(s), and may have to simply forward the request to all of the projections. This is obviously less desirable, so when possible ensure that the `Request` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/Request.html] objects include the *Path* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/property/Path.html].

15.4. Update operations

The federated connector behavior for *read-only* requests is fairly obvious. In the best case, the connector determines the appropriate projections, forwards the request into the appropriate sources, and then combines the results. But what happens with *change requests*?

Currently, the federated connector requires that each `ChangeRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html] be mapped to *one and only one* projection. However, when a single projection cannot be determined for a `ChangeRequest` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/request/ChangeRequest.html], the connector throws an error.

This is thought to be a minimal problem that will not actually be an issue in most uses of the federated connector. If you find that your usage does indeed fall into this category, please let us know via the *mailing lists* [http://www.modeshape.org/lists.html] or log an enhancement request in *JIRA* [http://jira.jboss.org/jira/browse/MODE]. Be sure to include as much detail as possible about the scenario, the problem condition, and the desired behavior.

15.5. Configuration

The federated repository uses other *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html]s that are to be federated and a *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] that is to be used as the cache of the unified contents. These are configured in another *RepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositorySource.html] that is treated as a configuration repository, which should contain information about the workspaces and how other sources are projected:

```
<!-- Define the federation configuration. -->
<mode:workspaces>
  <mode:workspace jcr:name="default">
    <!-- Define how the content in the two sources maps to the federated/unified repository.
         This example puts the 'Cars' and 'Aircraft' content underneath '/vehicles', but the
         'Configuration' content (which is defined by this file) will appear under '/'. -->
    <mode:projections>
      <!-- Project the 'Cars' content, starting with the '/Cars' node. -->
      <mode:projection jcr:name="Cars projection"
                       mode:source="Cars"
                       mode:workspaceName="workspace1">
          <mode:projectionRules>/Vehicles/Cars => /Cars</mode:projectionRules>
      </mode:projection>
      <!-- Project the 'Aicraft' content, starting with the '/Aircraft' node. -->
      <mode:projection jcr:name="Aircarft projection"
                       mode:source="Aircraft"
                       mode:workspaceName="workspace2">
          <mode:projectionRules>/Vehicles/Aircraft => /Aircraft</mode:projectionRules>
      </mode:projection>
      <!-- Project the 'System' content. Only needed when this source is accessed through JCR. -->
                    <mode:projection jcr:name="System projection" mode:source="System"
 mode:workspaceName="default">
          <mode:projectionRules>/jcr:system => /</mode:projectionRules>
      </mode:projection>
    </mode:projections>
  </mode:workspace>
</mode:workspaces>
```

Note

We're using XML to represent a graph structure, since the two map pretty well. Each XML element represents a node and XML attributes represent properties on a node. The name of the node is defined by either the `jcr:name` attribute (if it exists) or the name of the XML element. And we use XML namespaces to define the namespaces used in the node and property names. As an aside, this is exactly how the XML graph importer works.

15.6. Repository Source properties

While the majority of the configuration is defined using the configuration source (as discussed above), the *`FederatedRepositorySource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/federation/FederatedRepositorySource.html] class have have a few JavaBean properties:

Table 15.1. *FederatedRepositorySource* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/federat properties

Property	Description
name	The name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] by name.
retryLimit	Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.

Subversion Connector

This connector provides read and write access to the directories and folders within a Subversion repository, providing that content in the form of `nt:file` and `nt:folder` nodes. This source considers a workspace name to be the path to the directory on the repository's root directory location that represents the root of that workspace (e.g., "trunk" or "branches"). New workspaces can be created, as long as the names represent valid existing directories within the SVN repository.

The *`SVNRepositorySource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/svn/SVNRepositorySource.html] class provides a number of JavaBean properties that control its behavior:

Table 16.1. *`SVNRepositorySource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/svn/SVNRepo properties

Property	Description
cachePolicy	Optional property that, if used, defines the cache policy to use for caching information in the repository. When not used, this source will not define a specific duration for caching information.
creatingWorkspaceAllowed	Optional property that defines whether clients can create additional workspaces. The default value is "true".
defaultWorkspaceName	Optional property that, if used, specifies the name of the workspace to use when no workspace name is specified in an operation. Each workspace name is treated as a path relative to the SVN repository being exposed (e.g., a workspace name of "trunk" will map to the URL "http://acme.com/repo/trunk" if the repository root URL is "http://acme.com/repo/ ").
name	The name of the repository source, which is used by the *`RepositoryService`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] by name.

Property	Description
password	The password that should be used to establish a connection to the repository. This is not required if the URL represents an anonymous SVN repository address.
predefinedWorkspaceNames	Optional property that, if used, defines names of the workspaces that are predefined and need not be created before being used. Each workspace name is treated as a path relative to the SVN repository being exposed (e.g., a workspace name of "trunk" will map to the URL "http://acme.com/repo/trunk" if the repository root URL is "http://acme.com/repo/ "). This can be coupled with a "false" value for the "creatingWorkspaceAllowed" property to allow only the use of predefined workspaces.
repositoryRootURL	Required property that should be set with the URL to the Subversion repository.
retryLimit	Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.
username	The username that should be used to establish a connection to the repository.

One way to configure the Subversion connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the `SVNRepositorySource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/svn/SVNRepositorySource.html] class. For example:

```
JcrConfiguration config = ...
config.repositorySource("SVN Store")
      .usingClass(SVNRepositorySource.class)
      .setDescription("The ModeShape SVN repository (anonymous access)")
      .setProperty("repositoryRootUrl", "http://anonsvn.jboss.org/repos/modeshape");
      .setProperty("defaultWorkspaceName", "trunk");
      .setProperty("predefinedWorkspaceNames", new String[] {"trunk });
```

Another way to configure the Subversion connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance and load an XML configuration file that contains a repository source that uses the `SVNRepositorySource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/svn/SVNRepositorySource.html] class. For example a file named configRepository.xml can be created with these contents:

```
<?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 sources for the content.  These sources are directly accessible using the
   ModeShape-specific Graph API.  In fact, this is how the ModeShape JCR implementation works.
  You
    can think of these as being similar to JDBC DataSource objects, except that they expose
    graph content via the Graph API instead of records via SQL or JDBC.
    -->
    <mode:sources jcr:primaryType="nt:unstructured">
        <!--
         The 'SVN Store' repository is an Subversion source with one workspace (although others
 could
        be defined).
        -->
        <mode:source jcr:name="SVN Store"
                    mode:classname="org.modeshape.graph.connector.svn.SVNRepositorySource"
                    mode:description="The ModeShape SVN repository (anonymous access)"
                    mode:repositoryRootUrl="http://anonsvn.jboss.org/repos/modeshape"
                    mode:defaultWorkspaceName="trunk"
                    mode:predefinedWorkspaceNames="trunk"
                    mode:defaultWorkspaceName="default" >
          <!--
          If desired, specify a cache policy that caches items in memory for 5 minutes (300000 ms).
          This fragment can be left out if the connector should not cache any content.
          -->
          <mode:cachePolicy jcr:name="cachePolicy"

 mode:classname="org.modeshape.graph.connector.path.cache.InMemoryWorkspaceCache$InMemoryCachePoli
           mode:timeToLiveInMilliseconds="300000" />
        </mode:source>
    </mode:sources>

    <!-- MIME type detectors and JCR repositories would be defined below -->
</configuration>
```

The configuration can then be loaded from Java like this:

```
JcrConfiguration config = new JcrConfiguration().loadFrom("/configRepository.xml");
```

JBoss Cache Connector

The JBoss Cache repository connector allows a *JBoss Cache* [http://www.jboss.org/jbosscache/] instance to be used as a ModeShape (and thus JCR) repository. This provides a repository that is an effective, scalable, and distributed cache, and can be *federated* with other repository sources to provide a distributed repository.

The `JBossCacheSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/jbosscache/JBossCacheSource.html] class provides a number of JavaBean properties that control its behavior:

Table 17.1. `JBossCacheSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/jbosscache/J properties

Property	Description
cacheConfigurationName	Optional property that, if used, specifies the name of the configuration that is supplied to the cache factory when creating a new JBoss Cache instance.
cacheFactoryJndiName	Optional property that, if used, specifies the name in JNDI where an existing JBoss Cache Factory instance can be found. That factory would then be used if needed to create a JBoss Cache instance. If no value is provided, then the JBoss Cache `DefaultCacheFactory` class is used.
cacheJndiName	Optional property that, if used, specifies the name in JNDI where an existing JBoss Cache instance can be found. This should be used if your application already has a cache that is used, or if you need to configure the cache in a special way.
creatingWorkspacesAllowed	Optional property that is by default 'true' that defines whether clients can create new workspaces.
defaultCachePolicy	Optional property that, if used, defines the default for how long this information provided by this source may to be cached by other, higher-level components. The default value of null implies that this source does not define a specific duration for caching information provided by this repository source.

Property	Description
defaultWorkspaceName	Optional property that is initialized to an empty string and which defines the name for the workspace that will be used by default if none is specified.
name	The name of the repository source, which is used by the `RepositoryService` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] by name.
predefinedWorkspaceNames	Optional property that defines the names of the workspaces that exist and that are available for use without having to create them.
retryLimit	Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.
rootNodeUuid	Optional property that, if used, specifies the UUID that should be used for the root node of each workspace. If no value is specified, a random UUID is generated each time that the repository is started.
updatesAllowed	Determines whether the content in the connector is can be updated ("true"), or if the content may only be read ("false"). The default value is "true".
uuidPropertyName	Optional property that, if used, defines the property that should be used to find the UUID value for each node in the cache. "`mode:uuid`" is the default.

One way to configure the JBoss Cache connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the `JBossCacheSource` [http://docs.jboss.org/modeshape/

1.0.0.Final/api/org/modeshape/connector/jbosscache/JBossCacheSource.html] class. For example:

```
JcrConfiguration config = ...
config.repositorySource("Store")
      .usingClass(JBossCacheSource.class)
      .setDescription("The repository for our content")
      .setProperty("defaultWorkspaceName", "prod")
      .setProperty("rootNodeUuid", UUID.fromString("12083e7e-2b55-4c8d-954d-627a9f5c45c2"))
      .setProperty("predefinedWorkspaceNames", new String[] { "staging", "dev"});
```

Another way to configure the JBoss Cache connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance and load an XML configuration file that contains a repository source that uses the `JBossCacheSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/jbosscache/JBossCacheSource.html] class. For example a file named configRepository.xml can be created with these contents:

```
<?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 sources for the content.  These sources are directly accessible using the
   ModeShape-specific Graph API.  In fact, this is how the ModeShape JCR implementation works.
  You
    can think of these as being similar to JDBC DataSource objects, except that they expose
    graph content via the Graph API instead of records via SQL or JDBC.
    -->
    <mode:sources jcr:primaryType="nt:unstructured">
        <!--
        The 'Store' repository is a JBoss Cache repository with a single default workspace (though
        others could be created, too).
        -->
        <mode:source jcr:name="Store"
               mode:classname="org.modeshape.graph.connector.jbosscache.JBossCacheSource"
                    mode:description="The repository for our content"
                    mode:defaultworkspaceName="prod"
                    mode:rootNodeUuid="12083e7e-2b55-4c8d-954d-627a9f5c45c2"
                    mode:predefinedWorkspaceNames="staging,dev"/>
    </mode:sources>

    <!-- MIME type detectors and JCR repositories would be defined below -->
</configuration>
```

The configuration can then be loaded from Java like this:

```
JcrConfiguration config = new JcrConfiguration().loadFrom("/configRepository.xml");
```

Infinispan Connector

The Infinispan repository connector allows a *Infinispan* [http://www.jboss.org/infinispan/] instance to be used as a ModeShape (and thus JCR) repository. This provides a way for the content in a repository to be stored in an effective, scalable, and distributed data grid, and can be *federated* with other repository sources to provide a distributed repository.

The `InfinispanSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/infinispan/InfinispanSource.html] class provides a number of JavaBean properties that control its behavior:

Table 18.1. `InfinispanSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/infinispan/Infi properties

Property	Description
cacheManagerJndiName	Optional property that, if used, specifies the name in JNDI where an existing Infinispan Cache Manager instance can be found. That factory would then be used if needed to create an Infinispan Cache instance. If no value is provided, then the Infinispan `DefaultCacheManager` class is used.
cacheConfigurationName	Optional property that, if used, specifies the name of the configuration that is supplied to the cache manager when creating a new Infinispan CacheManager instance.
defaultCachePolicy	Optional property that, if used, defines the default for how long this information provided by this source may to be cached by other, higher-level components. The default value of null implies that this source does not define a specific duration for caching information provided by this repository source.
defaultWorkspaceName	Optional property that is initialized to an empty string and which defines the name for the workspace that will be used by default if none is specified.
name	The name of the repository source, which is used by the `RepositoryService` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html] when obtaining a

Property	Description
	RepositoryConnection [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] by name.
predefinedWorkspaceNames	Optional property that defines the names of the workspaces that exist and that are available for use without having to create them.
retryLimit	Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.
rootNodeUuid	Optional property that, if used, specifies the UUID that should be used for the root node of each workspace. If no value is specified, a random UUID is generated each time that the repository is started.
updatesAllowed	Determines whether the content in the connector is can be updated ("true"), or if the content may only be read ("false"). The default value is "true".

One way to configure the Infinispan connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the `InfinispanSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/infinispan/InfinispanSource.html] class. For example:

```
JcrConfiguration config = ...
config.repositorySource("Infinispan Store")
      .usingClass(InfinispanSource.class)
      .setDescription("The repository for our content")
      .setProperty("defaultWorkspaceName", "prod")
      .setProperty("rootNodeUuid", UUID.fromString("84b73fc8-81a8-42ed-aa4b-3905094966f0")
      .setProperty("predefinedWorkspaceNames", new String[] { "staging", "dev"});
```

Another way to configure the Infinispan connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance and load an XML configuration file that contains a repository source that uses the `InfinispanSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/infinispan/InfinispanSource.html] class. For example a file named configRepository.xml can be created with these contents:

```
<?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 sources for the content.  These sources are directly accessible using the
   ModeShape-specific Graph API.  In fact, this is how the ModeShape JCR implementation works.
  You
    can think of these as being similar to JDBC DataSource objects, except that they expose
    graph content via the Graph API instead of records via SQL or JDBC.
    -->
    <mode:sources jcr:primaryType="nt:unstructured">
        <!--
        The 'Infinispan Store' repository is a Infinispan repository with a single default
        workspace (though others could be created, too).
        -->
        <mode:source jcr:name="Infinispan Store"
                    mode:classname="org.modeshape.graph.connector.infinispan.InfinispanSource"
                    mode:description="The repository for our content"
                    mode:defaultworkspaceName="prod"
                    mode:rootNodeUuid="84b73fc8-81a8-42ed-aa4b-3905094966f0"
                    mode:predefinedWorkspaceNames="staging,dev"/>
    </mode:sources>

    <!-- MIME type detectors and JCR repositories would be defined below -->
</configuration>
```

The configuration can then be loaded from Java like this:

```
JcrConfiguration config = new JcrConfiguration().loadFrom("/configRepository.xml");
```

JDBC Metadata Connector

This connector provides read-only access to the metadata (e.g., catalogs, schemas, table structures) of a relational database. The connector yields a content graph that looks like this:

```
/ (root node)
  + <catalog name> - one node for each accessible catalog in the database.
    + <schema name> - one node for each accessible schema in the catalog.
      + tables - a single node that is the parent of all tables in the schema.
      |  + <table name> - one node for each table in the schema.
      |    + <column name> - one node for each column in the table.
      + procedures - a single node that is the parent of all procedures in the schema.
         + <procedure name> - one node for each procedure in the schema.
```

The root, table, column, and procedure nodes contain additional properties that correspond to the metadata provide by the `DatabaseMetaData` [http://java.sun.com/j2se/1.5.0/docs/api/java/sql/DatabaseMetaData.html] class. In databases that do not support catalogs or schemas (or allow the empty string as a valid catalog or schema name, the value of the `defaultCatalogName` and/or `defaultSchemaName` properties will be used instead when determining the graph name.

Note

This connector has currently been tested successfully against Oracle 10g, Oracle 11g, Microsoft SQL Server 2008 (with the Microsoft JDBC driver), IBM DB2 v9, Sybase ASE 15, MySQL 5 (with the InnoDB engine), PostgreSQL 8, and HSQLDB. As JDBC driver implementations of the `DatabaseMetaData` [http://java.sun.com/j2se/1.5.0/docs/api/java/sql/DatabaseMetaData.html] interface tend to vary widely, other databases may or may not work with the default `MetadataCollector` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/meta/jdbc/MetadataCollector] implementation. As one example, the `metadataCollectorClassName` property must be set to `org.modeshape.connector.meta.jdbc.SqlServerMetadataConnector` if the Microsoft JDBC driver is used. This is to work around a known bug where that driver returns a list of users from a call to `DatabaseMetaData` [http://java.sun.com/j2se/1.5.0/docs/api/java/sql/DatabaseMetaData.html].getSchemas() instead of a list of schemas.

To use this connector with the ModeShape JCR layer, you must import the JCR node types that this connector uses. These are bundled in the JAR for this connector at the path `/org/modeshape/connector/meta/jdbc/nodeTypes.cnd`. Please see the *Getting Started* [http://docs.jboss.org/

modeshape/1.0.0.Final/manuals/gettingstarted/html/index.html] Guide for detailed examples of how to import custom JCR node types.

The `JdbcMetadataSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/meta/jdbc/JdbcMetadataSource.html] class provides a number of JavaBean properties that control its behavior:

Table 19.1. `JdbcMetadataSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/meta/jdbc/Jdb properties

Property	Description
cachePolicy	Optional property that, if used, defines the cache policy to use for this repository source. When not used, this source will not define a specific duration for caching information.
dataSourceJndiName	The JNDI name of the JDBC DataSource instance that should be used. If not specified, the other driver properties must be set.
defaultCatalogName	The name to use for the catalog name if the database does not support catalogs or the database has a catalog with the empty string as a name. The default value is "default".
defaultSchemaName	The name to use for the schema name if the database does not support schemas or the database has a schema with the empty string as a name. The default value is "default".
driverClassloaderName	The name of the class loader or classpath that should be used to load the JDBC driver class. This is not required if the DataSource is found in JNDI.
driverClassName	The name of the JDBC driver class. This is not required if the DataSource is found in JNDI, but is required otherwise.
idleTimeInSecondsBeforeTestingConnections	The number of seconds after a connection remains in the pool the connection should be tested to ensure it is still valid. The default is 180 seconds (or 3 minutes).
maximumConnectionsInPool	The maximum number of connections that may be in the connection pool. The default is "5".
maximumConnectionIdleTimeInSeconds	

Property	Description
	The maximum number of seconds that a connection should remain in the pool before being closed. The default is "600" seconds (or 10 minutes).
maximumSizeOfStatementCache	The maximum number of statements that should be cached. Statement caching can be disabled by setting to "0". The default is "100".
metadataCollectorClassName	The name of a custom class to use for metadata collection. The class must implement the `MetadataCollector` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/meta/jdbc/MetadataCollector] interface. If a null value is specified for this property, a default `MetadataCollector` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/meta/jdbc/MetadataCollector] implementation will be used that relies on the `DatabaseMetaData` [http://java.sun.com/j2se/1.5.0/docs/api/java/sql/DatabaseMetaData.html] provided by the JDBC driver for the connection. This property is provided as a means for connecting to databases with a JDBC driver that provides a non-standard `DatabaseMetaData` [http://java.sun.com/j2se/1.5.0/docs/api/java/sql/DatabaseMetaData.html] implementation or no `DatabaseMetaData` [http://java.sun.com/j2se/1.5.0/docs/api/java/sql/DatabaseMetaData.html] implementation at all.
minimumConnectionsInPool	The minimum number of connections that will be kept in the connection pool. The default is "0".
name	The name of the repository source, which is used by the `RepositoryService` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/

Property	Description
	graph/connector/RepositoryConnection.html] by name.
nameOfDefaultWorkspace	Optional property that is initialized to an empty string and which defines the name for the workspace that will be used by default if none is specified.
numberOfConnectionsToAcquireAsNeeded	The number of connections that should be added to the pool when there are not enough to be used. The default is "1".
retryLimit	Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.
password	The password that should be used when creating JDBC connections using the JDBC driver class. This is not required if the DataSource is found in JNDI.
rootNodeUuid	Optional property that, if used, defines the UUID of the root node in the repository. If not used, then a new UUID is generated.
url	The URL that should be used when creating JDBC connections using the JDBC driver class. This is not required if the DataSource is found in JNDI.
username	The username that should be used when creating JDBC connections using the JDBC driver class. This is not required if the DataSource is found in JNDI.

One way to configure the JDBC metadata connector is to create `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the `JdbcMetadataSource` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/meta/jdbc/JdbcMetadataSource.html] class. For example:

```
JcrConfiguration config = ...
config.repositorySource("Meta Store")
```

```
     .usingClass(JdbcMetadataSource.class)
     .setDescription("The database source for our content")
     .setProperty("dataSourceJndiName", "java:/MyDataSource")
     .setProperty("nameOfDefaultWorkspace", "default");
```

Of course, setting other more advanced properties would entail calling `setProperty(...)` for each. Since almost all of the properties have acceptable default values, however, we don't need to set very many of them.

Another way to configure the JDBC metadata connector is to create *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] instance and load an XML configuration file that contains a repository source that uses the *`JdbcMetadataSource`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/connector/meta/jdbc/JdbcMetadataSource.html] class. For example a file named configRepository.xml can be created with these contents:

```
<?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 sources for the content.  These sources are directly accessible using the
  ModeShape-specific Graph API.  In fact, this is how the ModeShape JCR implementation works.
 You
  can think of these as being similar to JDBC DataSource objects, except that they expose
  graph content via the Graph API instead of records via SQL or JDBC.
  -->
  <mode:sources jcr:primaryType="nt:unstructured">
    <!--
    The 'Meta Store' repository is a JDBC metadata repository with a single default
    workspace (though others could be created, too).
    -->
    <mode:source jcr:name="Meta Store"
        mode:classname="org.modeshape.graph.connector.meta.jdbc.JdbcMetadataSource"
           mode:description="The database source for our content"
           mode:dataSourceJndiName="java:/MyDataSource"
           mode:defaultworkspaceName="default" >
     <!--
     If desired, specify a cache policy that caches items in memory for 5 minutes (300000 ms).
     This fragment can be left out if the connector should not cache any content.
     -->
     <mode:cachePolicy jcr:name="cachePolicy"
 mode:classname="org.modeshape.graph.connector.path.cache.InMemoryWorkspaceCache$InMemoryCachePoli
           mode:timeToLiveInMilliseconds="300000" />
       </mode:source>
    </mode:sources>

    <!-- MIME type detectors and JCR repositories would be defined below -->
</configuration>
```

The configuration can then be loaded from Java like this:

```
JcrConfiguration config = new JcrConfiguration().loadFrom("/configRepository.xml");
```

Part V. Sequencer Library

The ModeShape project provides a number of *sequencers* out-of-the-box. These are ready to be used by simply including them in the classpath and *configuring* them appropriately.

Compact Node Type (CND) Sequencer

This sequencer processes JCR Compact Node Definition (CND) files to extract the node definitions with their property definitions, and inserts these into the repository using JCR built-in types. The node structure generated by this sequencer is equivalent to the node structure used in `/jcr:system/jcr:nodeTypes`.

This sequencer can be added to the repository configuration like so:

```
JcrConfiguration config = ...

config.sequencer("CND Sequencer")
      .usingClass("org.modeshape.sequencer.cnd.CndSequencer")
      .loadedFromClasspath()
      .setDescription("Sequences CND files to extract the node type definitions")
      .sequencingFrom("//(*.cnd[*])/jcr:content[@jcr:data]")
      .andOutputtingTo("/nodeTypes/$1");
```

XML Document Sequencer

This sequencer stores the structure and data of an XML file into the repository. DTD, entity, comments, and other content are maintained by the sequencer in the output structure.

```
JcrConfiguration config = ...

config.sequencer("XML Sequencer")
      .usingClass("org.modeshape.sequencer.xml.XmlSequencer")
      .loadedFromClasspath()
      .setDescription("Sequences XML documents and maps their data into the repository")
      .sequencingFrom("//(*.xml[*])/jcr:content[@jcr:data]")
      .andOutputtingTo("/xml/$1");
```

ZIP File Sequencer

The ZIP file sequencer is included in ModeShape and extracts the files and folders contained in the ZIP archive file, extracting the files and folders into the repository using JCR's `nt:file` and `nt:folder` built-in node types. The structure of the output thus matches the logical structure of the contents of the ZIP file.

To use this sequencer, simply include the `modeshape-sequencer-zip` JAR in your application and configure the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] to use this sequencer using something similar to:

```
JcrConfiguration config = ...

config.sequencer("ZIP Sequencer")
      .usingClass("org.modeshape.sequencer.zip.ZipSequencer")
      .loadedFromClasspath()
      .setDescription("Sequences compressed files to extract the internal file and folder structure")
      .sequencingFrom("//(*.(zip|gz|jar|war|ear)[*])/jcr:content[@jcr:data]")
      .andOutputtingTo("/zips/$1");
```

Microsoft Office Document Sequencer

This sequencer is included in ModeShape and processes Microsoft Office documents, including Word documents, Excel spreadsheets, and PowerPoint presentations. With documents, the sequencer attempts to infer the internal structure from the heading styles. With presentations, the sequencer extracts the slides, titles, text and slide thumbnails. With spreadsheets, the sequencer extracts the names of the sheets. And, the sequencer extracts for all the files the general file information, including the name of the author, title, keywords, subject, comments, and various dates.

To use this sequencer, simply include the `modeshape-sequencer-msoffice` JAR and all of the *POI* [http://poi.apache.org/] JARs in your application and configure the `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] to use this sequencer using something similar to:

```
JcrConfiguration config = ...

config.sequencer("Microsoft Office Document Sequencer")
    .usingClass("org.modeshape.sequencer.msoffice.MSOfficeMetadataSequencer")
    .loadedFromClasspath()
   .setDescription("Sequences MS Office documents, including spreadsheets and presentations")
    .sequencingFrom("//(*.(*.(doc|docx|ppt|pps|xls)[*])/jcr:content[@jcr:data]")
    .andOutputtingTo("/msoffice/$1");
```

Java Source File Sequencer

One of the sequencers that included in ModeShape is the **modeshape-sequencer-java** subproject. This sequencer parses Java source code added to the repository and extracts the basic structure of the classes and enumerations defined in the code. This structure includes: the package structures, class declarations, class and member attribute declarations, class and member method declarations with signature (but not implementation logic), enumerations with each enumeration literal value, annotations, and JavaDoc information for all of the above. After extracting this information from the source code, the sequencer then writes this structure into the repository, where it can be further processed, analyzed, searched, navigated, or referenced.

As noted previously, the `JavaMetadataSequencer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/java/JavaMetadataSequencer.html] class provides a pair of JavaBean properties that can be used to specify a custom `SourceFileRecorder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/java/SourceFileRecorder.html] implementation to use to map the extracted metadata to an output location:

Table 24.1. `JavaMetadataSequencer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/java/JavaMeta properties

Property	Description
sourceFileRecorder	Optional property that, if set, provides an instance of the `SourceFileRecorder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/java/SourceFileRecorder.html] interface that will be used for all subsequent sequencing activity for this sequencer. If this property is set to null, a default implementation will be used. The default value of this property is null.
sourceFileRecorderClassName	Optional property that, if set, provides the name of a class that provides a custom implementation of the `SourceFileRecorder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/java/SourceFileRecorder.html] interface. This class must have a no-argument, public constructor. If set, an instance of this class will be created immediately and reused for all subsequent sequencing activity for this sequencer. If this property is set to null, a

Property	Description
	default implementation will be used. The default value of this property is null.

The default `SourceFileRecorder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/java/SourceFileRecorder.html] generates output that is compatible with previous versions of the `JavaMetadataSequencer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/java/JavaMetadataSequencer.html]. To generated sequenced output that is identical to the output generated by the `ClassFileSequencer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/classfile/ClassFileSequencer.html], set the `sourceFileRecorderClassName` property to "org.modeshape.sequencer.java.ClassSourceFileRecorder".

To use this sequencer, simply include the `modeshape-sequencer-java` JAR (plus all of the JARs that it is dependent upon) in your application and configure the `JcrConfiguration` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] to use this sequencer using something similar to:

```
JcrConfiguration config = ...

config.sequencer("Java Sequencer")
      .usingClass("org.modeshape.sequencer.java.JavaMetadataSequencer")
      .loadedFromClasspath()
      .setDescription("Sequences java files to extract the characteristics of the Java source")
      .sequencingFrom("//(*.(java)[*])/jcr:content[@jcr:data]")
      .andOutputtingTo("/java/$1");
```

Java Class File Sequencer

The Java class file sequencer parses Java class file to extract metadata for the class, its methods, its fields, and its annotations. The output of the sequencer can be customized by using the `classFileRecorder` or `classFileRecorderClassName` properties to provide a custom implementation of the *`ClassFileRecorder`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/classfile/ClassFileRecorder.html] interface. A default implementation (*`DefaultClassFileRecorder`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/classfile/DefaultClassFileRecorder.html]) is provided that records all extracted metadata to the output location.

As noted previously, the *`ClassFileSequencer`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/classfile/ClassFileSequencer.html] class provides a pair of JavaBean properties that can be used to specify a custom *`ClassFileRecorder`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/classfile/ClassFileRecorder.html] implementation to use to map the extracted metadata to an output location:

Table 25.1. *`ClassFileSequencer`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/classfile/Clas properties

Property	Description
classFileRecorder	Optional property that, if set, provides an instance of the *`ClassFileRecorder`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/classfile/ClassFileRecorder.html] interface that will be used for all subsequent sequencing activity for this sequencer. If this property is set to null, a default implementation will be used. The default value of this property is null.
classFileRecorderClassName	Optional property that, if set, provides the name of a class that provides a custom implementation of the *`ClassFileRecorder`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/classfile/ClassFileRecorder.html] interface. This class must have a no-argument, public constructor. If set, an instance of this class will be created immediately and reused for all subsequent sequencing activity for this sequencer. If this property is set to null, a

Property	Description
	default implementation will be used. The default value of this property is null.

The default class file recorder creates a subgraph rooted at the output location that takes the following form:

```
<nt:unstructured jcr:name="packageName1">
   ...
   <nt:unstructured jcr:name="packageNameN">
      <class:class jcr:name="ClassName">
         <class:annotations jcr:name="class:annotations">
            <class:annotation jcr:name="AnnotationName1"/>
            ...
            <class:annotation jcr:name="AnnotationNameN"/>
         </class:annotations>
         <class:constructors jcr:name="class:constructors">
            <class:constructor jcr:name="constructor parameters">
               <class:annotation jcr:name="AnnotationName1"/>
               ...
               <class:annotation jcr:name="AnnotationNameN"/>
            </class:constructor>
         </class:constructors>
         <class:methods jcr:name="class:methods">
            <class:method jcr:name="methodName(parameters)">
               <class:annotation jcr:name="AnnotationName1"/>
               ...
               <class:annotation jcr:name="AnnotationNameN"/>
            </class:method>
         </class:methods>
         <class:fields jcr:name="class:fields">
            <class:field jcr:name="fieldName">
               <class:annotation jcr:name="AnnotationName1"/>
               ...
               <class:annotation jcr:name="AnnotationNameN"/>
            </class:field>
         </class:fields>
       </class:class>
   </nt:unstructured>
   ...
</nt:unstructured>
```

The compact node definitions for the class:* types is provided below. *Please note that these definitions may change in a future release.*

```
[class:annotationMember]
- class:name (string) mandatory
- class:value (string)

[class:annotation]
- class:name (string) mandatory
+ * (class:annotationMember) = class:annotationMember

[class:annotations]
+ * (class:annotation) = class:annotation

[class:field]
- class:name (string) mandatory
- class:typeClassName (string) mandatory
- class:visibility (string) mandatory < 'public', 'protected', 'package', 'private'
- class:static (boolean) mandatory
- class:final (boolean) mandatory
- class:transient (boolean) mandatory
- class:volatile (boolean) mandatory
+ class:annotations (class:annotations) = class:annotations

[class:fields]
+ * (class:field) = class:field

[class:interfaces]
- * (string)

[class:parameters]
- * (string)

[class:method]
- class:name (string) mandatory
- class:returnTypeClassName (string) mandatory
- class:visibility (string) mandatory < 'public', 'protected', 'package', 'private'
- class:static (boolean) mandatory
- class:final (boolean) mandatory
- class:abstract (boolean) mandatory
- class:strictFp (boolean) mandatory
- class:native (boolean) mandatory
- class:synchronized (boolean) mandatory
- class:parameters (string) multiple
+ class:annotations (class:annotations) = class:annotations

[class:methods]
+ * (class:method) = class:method

[class:constructors]
+ * (class:method) = class:method

[class:class]
- class:name (string) mandatory
- class:superClassName (string)
- class:visibility (string) mandatory < 'public', 'protected', 'package', 'private'
- class:abstract (boolean) mandatory
- class:interface (boolean) mandatory
- class:final (boolean) mandatory
- class:strictFp (boolean) mandatory
- class:interfaces (string) multiple
+ class:annotations (class:annotations) = class:annotations
+ class:constructors (class:constructors) = class:constructors
+ class:methods (class:methods) = class:methods
+ class:fields (class:fields) = class:fields

[class:enum] > class:class
- class:enumValues (string) mandatory multiple
```

To use this sequencer, simply include the `modeshape-sequencer-classfile` JAR in your application and configure the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] to use this sequencer using something similar to:

```
JcrConfiguration config = ...

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");
```

Image Sequencer

The `ImageMetadataSequencer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/image/ImageMetadataSequencer.html] sequencer extracts metadata from JPEG, GIF, BMP, PCX, PNG, IFF, RAS, PBM, PGM, PPM and PSD image files. This sequencer extracts the file format, image resolution, number of bits per pixel and optionally number of images, comments and physical resolution, and then writes this information into the repository using the following structure:

- **image:metadata** node of type `image:metadata`
- - **jcr:mimeType** - optional string property for the mime type of the image
  - **jcr:encoding** - optional string property for the encoding of the image
  - **image:formatName** - string property for the name of the format
  - **image:width** - optional integer property for the image's width in pixels
  - **image:height** - optional integer property for the image's height in pixles
  - **image:bitsPerPixel** - optional integer property for the number of bits per pixel
  - **image:progressive** - optional boolean property specifying whether the image is stored in a progressive (i.e., interlaced) form
  - **image:numberOfImages** - optional integer property for the number of images stored in the file; defaults to 1
  - **image:physicalWidthDpi** - optional integer property for the physical width of the image in dots per inch
  - **image:physicalHeightDpi** - optional integer property for the physical height of the image in dots per inch
  - **image:physicalWidthInches** - optional double property for the physical width of the image in inches
  - **image:physicalHeightInches** - optional double property for the physical height of the image in inches

This structure could be extended in the future to add EXIF and IPTC metadata as child nodes. For example, EXIF metadata is structured as tags in directories, where the directories form something like namespaces, and which are used by different camera vendors to store custom metadata. This structure could be mapped with each directory (e.g. "EXIF" or "Nikon Makernote" or "IPTC") as the name of a child node, with the EXIF tags values stored as either properties or child nodes.

To use this sequencer, simply include the `modeshape-sequencer-images` JAR in your application and configure the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] to use this sequencer using something similar to:

```
JcrConfiguration config = ...
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");
```

MP3 Sequencer

Another sequencer that is included in ModeShape is the **modeshape-sequencer-mp3** sequencer project. This sequencer processes MP3 audio files added to a repository and extracts the *ID3* [http://www.id3.org/] metadata for the file, including the track's title, author, album name, year, and comment. After extracting this information from the audio files, the sequencer then writes this structure into the repository, where it can be further processed, analyzed, searched, navigated, or referenced.

To use this sequencer, simply include the `modeshape-sequencer-mp3` JAR and the *JAudioTagger* [http://www.jthink.net/jaudiotagger/] library in your application and configure the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] to use this sequencer using something similar to:

```
JcrConfiguration config = ...

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");
```

DDL File Sequencer

The DDL file sequencer included in ModeShape is capable of parsing the more important DDL statements from SQL-92, Oracle, Derby, and PostgreSQL, and constructing a graph structure containing a structured representation of these statements. The resulting graph structure is largely the same for all dialects, though some dialects have non-standard additions to their grammar, and thus require dialect-specific additions to the graph structure.

The sequencer is designed to behave as intelligently as possible with as little configuration. Thus, the sequencer automatically determines the dialect used by a given DDL stream. This can be tricky, of course, since most dialects are very similar and the distinguishing features of a dialect may only be apparent in some of the statements.

To get around this, the sequencer uses a "best fit" algorithm: run the DDL stream through the parser for each of the dialects, and determine which parser was able to successfully read the greatest number of statements and tokens.

Note

It is possible to define which DDL dialects (or grammars) should be considered during sequencing using the "grammars" property in the sequencer configuration. Set the values of this property to the names of the grammars (e.g., "oracle", "postgres", "standard", or "derby"), specified in the order they should be used. To use a custom DDL parser not provided by ModeShape, simply provide the fully-qualified class name of the *DdlParser* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/ddl/DdlParser] implementation class.

One very interesting capability of this sequencer is that, although only a subset of the (more common) DDL statements are supported, the sequencer is still extremely functional since it does still add all statements into the output graph, just without much detail other than just the statement text and the position in the DDL file. Thus, if a DDL file contains statements the sequencer understands and statements the sequencer does not understand, the graph will still contain all statements, where those statements understood by the sequencer will have full detail. Since the underlying parsers are able to operate upon a single statement, it is possible to go back later (after the parsers have been enhanced to support additional DDL statements) and re-parse only those incomplete statements in the graph.

At this time, the sequencer supports SQL-92 standard DDL as well as dialects from Oracle, Derby, and PostgreSQL. It supports:

- Detailed parsing of CREATE SCHEMA, CREATE TABLE and ALTER TABLE.
- Partial parsing of DROP statements

- General parsing of remaining schema definition statements (i.e. CREATE VIEW, CREATE DOMAIN, etc.

Note that the sequencer does *not* perform detailed parsing of SQL (i.e. SELECT, INSERT, UPDATE, etc....) statements.

Caution

The DDL sequencer is being included as a Technology Preview. It is fully functional for the dialects listed above, and may indeed work on certain DDL files that use other dialects. But we would like to have feedback from users, test against more DDL examples, support additional dialects, and support more kinds of DDL statements. As such, the output format and node types associated with the `DefaultClassFileRecorder` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/classfile/DefaultClassFileRecorder.html] may change in future versions.

28.1. Example

Sequencing results in graph nodes basically representing the BNF structure of each DDL statement. Below is an example DDL schema definition statement containing table and view definition statements.

```
CREATE SCHEMA hollywood
   CREATE TABLE films (title varchar(255), release date, producerName varchar(255))
   CREATE VIEW winners AS SELECT title, release FROM films WHERE producerName IS NOT
 NULL;
```

The resulting graph structure contains the raw statement expression, pertinent table, column and key reference information and position of the statement in the text stream (e.g., line number, column number and character index) so the statement can be tied back to the original DDL:

```
<nt:unstructured jcr:name="statements" ddl:parserId="POSTGRES">
  <nt:unstructured jcr:name="hollywood" jcr:mixinTypes="ddl:createSchemaStatement"
                    ddl:startLineNumber="1"
                  ddl:startColumnNumber="1"
                  ddl:expression="CREATE SCHEMA hollywood"
                  ddl:startCharIndex="0">
    <nt:unstructured jcr:name="films" jcr:mixinTypes="ddl:createTableStatement"
                  ddl:startLineNumber="2"
                  ddl:startColumnNumber="5"
```

```
             ddl:expression="CREATE TABLE films (title varchar(255), release date, producerName
varchar(255))"
                 ddl:startCharIndex="28"/>
   <nt:unstructured jcr:name="title" jcr:mixinTypes="ddl:columnDefinition"
                 ddl:datatypeName="VARCHAR"
                 ddl:datatypeLength="255"/>
   <nt:unstructured jcr:name="release" jcr:mixinTypes="ddl:columnDefinition"
                 ddl:datatypeName="DATE"/>
   <nt:unstructured jcr:name="producerName" jcr:mixinTypes="ddl:columnDefinition"
                 ddl:datatypeName="VARCHAR"
                 ddl:datatypeLength="255"/>
  <nt:unstructured jcr:name="winners" jcr:mixinTypes="ddl:createViewStatement"
                 ddl:startLineNumber="3"
                 ddl:startColumnNumber="5"
                    ddl:expression="CREATE VIEW winners AS SELECT title, release FROM films
WHERE producerName IS NOT NULL;"
                  ddl:queryExpression="SELECT title, release FROM films WHERE producerName
IS NOT NULL"
                 ddl:startCharIndex="113"/>
</nt:unstructured>
```

Note that all nodes are of type `nt:unstructured` while the type of statement is identified using mixins. Also, each of the nodes representing a statement contain: a `ddl:expression` property with the exact statement as it appeared in the original DDL stream; a `ddl:startLineNumber` and `ddl:startColumnNumber` property defining the position in the original DDL stream of the first character in the expression; and a `ddl:startCharIndex` property that defines the integral index of the first character in the expression as found in the DDL stream. All of these properties make sure the statement can be traced back to its location in the original DDL.

To use this sequencer, simply include the `modeshape-sequencer-ddl` JAR in your application and configure the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] to use this sequencer using something similar to:

```


JcrConfiguration config = ...

config.sequencer("DDL Sequencer")
      .usingClass("org.modeshape.sequencer.ddl.DdlSequencer")
      .loadedFromClasspath()
        .setDescription("Sequences DDL files to extract individual statements and accompanying
statement properties and values")
      .sequencingFrom("//(*.(ddl)[*])/jcr:content[@jcr:data]")
      .andOutputtingTo("/ddls/$1");
```

This will use all of the built-in grammars (e.g., "standard", "oracle", "postgres", and "derby"). To specify a different order or subset of the grammars, use the `setProperty(...)` method. Here's an example that just uses the standard grammar followed by the PostgreSQL grammar:

```
config.sequencer("DDL Sequencer")
      .usingClass("org.modeshape.sequencer.ddl.DdlSequencer")
      .loadedFromClasspath()
        .setDescription("Sequences DDL files to extract individual statements and accompanying
 statement properties and values")
      .setProperty("grammar","standard","postgres")
      .sequencingFrom("//(*.(ddl)[*])/jcr:content[@jcr:data]")
      .andOutputtingTo("/ddls/$1");
```

And, to use a custom implementation of *DdlParser* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/ddl/DdlParser], simply use the fully-qualified name of the implementation class (which must have a no-arg constructor) as the name of the grammar:

```
config.sequencer("DDL Sequencer")
      .usingClass("org.modeshape.sequencer.ddl.DdlSequencer")
      .loadedFromClasspath()
        .setDescription("Sequences DDL files to extract individual statements and accompanying
 statement properties and values")
      .setProperty("grammar","standard","postgres","org.example.ddl.MyCustomDdlParser")
      .sequencingFrom("//(*.(ddl)[*])/jcr:content[@jcr:data]")
      .andOutputtingTo("/ddls/$1");
```

Text Sequencers

The text sequencers extract data from text streams. There are separate sequencers for character-delimited sequencing and fixed width sequencing, but both treat the incoming text stream as a series of rows (separated by line-terminators, as defined in `BufferedReader` [http://java.sun.com/j2se/1.5.0/docs/api/java/io/BufferedReader.html].readLine() with each row consisting of one or more columns. As noted above, each text sequencer provides its own mechanism for splitting the row into columns.

The `AbstractTextSequencer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/text/AbstractTextSequencer.html] class provides a number of JavaBean properties that are common to both of the concrete text sequencer classes:

Table 29.1. `AbstractTextSequencer` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/text/Abstract properties

Property	Description
commentMarker	Optional property that, if set, indicates that any line beginning with *exactly this string* should be treated as a comment and should not be processed further. If this value is null, then all lines will be sequenced. The default value for this property is null.
maximumLinesToRead	Optional property that, if set, limits the number of lines that will be read during sequencing. Additional lines will be ignored. If this value is non-positive, all lines will be read and sequenced. Comment lines are not counted towards this total. The default value of this property is -1 (indicating that all lines should be read and sequenced).
rowFactoryClassName	Optional property that, if set, provides the name of a class that provides a custom implementation of the `RowFactory` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/text/RowFactory] interface. This class must have a no-argument, public constructor. If set, an instance of this class will be created each time that the sequencer sequences an input stream and will be used to provide the output structure of the graph. If this property is set

Property	Description
	to null, a default implementation will be used. The default value of this property is null.

The default row factory creates one node in the output location for each row sequenced from the source and adds each column with the row as a child node of the row node. The output graph takes the following form (all nodes have primary type `nt:unstructured`:

```
<graph root>
    + text:row[1]
    |   + text:column[1] (jcr:mixinTypes = text:column, text:data = <column1 data>)
    |   + ...
    |   + text:column[n] (jcr:mixinTypes = text:column, text:data = <columnN data>)
    + ...
    + text:row[m]
        + text:column[1] (jcr:mixinTypes = text:column, text:data = <column1 data>)
        + ...
        + text:column[n] (jcr:mixinTypes = text:column, text:data = <columnN data>)
```

29.1. Delimited Text Sequencer

The *`DelimitedTextSequencer`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/text/DelimitedTextSequencer] splits rows into columns based on a regular expression pattern. Although the default pattern is a comma, any regular expression can be provided allowing for more sophisticated splitting patterns.

The *`DelimitedTextSequencer`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/text/DelimitedTextSequencer] class provides an additional JavaBean property to override the default regular expression pattern:

Table 29.2. *`DelimitedTextSequencer`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/text/Delimited properties

Property	Description
splitPattern	Optional property that, if set, sets the regular expression pattern that is used to split each row into columns. This property may not be set to null and defaults to ",".

To use this sequencer, simply include the `modeshape-sequencer-text` JAR in your application and configure the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] to use this sequencer using something similar to:

```
JcrConfiguration config = ...

config.sequencer("Delimited Text Sequencer")
      .usingClass("org.modeshape.sequencer.text.DelimitedTextSequencer")
      .loadedFromClasspath()
      .setDescription("Sequences delimited files to extract values")
      .sequencingFrom("//(*.(txt)[*])/jcr:content[@jcr:data]")
      .setProperty("splitPattern", "|")
      .andOutputtingTo("/txt/$1");
```

29.2. Fixed Width Text Sequencer

The *`FixedWidthTextSequencer`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/text/FixedWidthTextSequencer] splits rows into columns based on predefined positions. The default setting is to have a single column per row. It also provides an additional JavaBean property to override the default start positions for each column.

Table 29.3. *`FixedWidthTextSequencer`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/sequencer/text/FixedWid properties

Property	Description
columnStartPositions	Optional property that, if set, provides the start position of each column after the first. The start positions are concatenated into a single, comma-delimited string. The default value is the empty string (implying that each row should be treated as a single column). This property may not be set to null. There is an implicit column start position of 0 that never needs to be specified.

To use this sequencer, simply include the `modeshape-sequencer-text` JAR in your application and configure the *`JcrConfiguration`* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/jcr/JcrConfiguration.html] to use this sequencer using something similar to:

```
JcrConfiguration config = ...

config.sequencer("Fixed Width Text Sequencer")
      .usingClass("org.modeshape.sequencer.text.FixedWidthTextSequencer")
      .loadedFromClasspath()
      .setDescription("Sequences fixed width files to extract values")
      .sequencingFrom("//(*.(txt)[*])/jcr:content[@jcr:data]")
      .setProperty("columnStartPositions", "3,6,15")
      .andOutputtingTo("/txt/$1");
```

Part VI. MIME Type Detector Library

The ModeShape project provides a number of *MIME type detectors* out-of-the-box. These are ready to be used by simply including them in the classpath and *setting up the `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html]* appropriately.

Aperture MIME type detector

The `AperatureMimeTypeDetector` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/mimetype/aperture/ApertureMimeTypeDetector.html] class is an implementation of *MimeTypeDetector* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] that uses the *Aperture* [http://aperture.sourceforge.net/] open-source library, which is a very capable utility for determining the MIME type for a wide range of file types, using both the file name and the actual content.

To use, simply include the `modeshape-mime-type-detector-aperture.jar` file on the classpath and create a new `ExecutionContext` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] subcontext with it:

```
MimeTypeDetector [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] myDetector = new ApertureMimeTypeDetector();
ExecutionContext [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/ExecutionContext.html] contextWithMyDetector = context.with(myDetector);
```

Writing custom detectors

Creating a custom detector involves the following steps:

- Create a Maven 2 project for your detector;
- Implement the *MimeTypeDetector* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] interface with your own implementation, and create unit tests to verify the functionality and expected behavior;
- Add a *MimeTypeDetectorConfig* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/mimetype/MimeTypeDetectorConfig.html] to the `MimeType` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/mimetype/MimeType.html] class in your application as described *earlier*; and
- Deploy the JAR file with your implementation (as well as any dependencies), and make them available to ModeShape in your application.

It's that simple.

The first step is to create the Maven 2 project that you can use to compile your code and build the JARs. Maven 2 automates a lot of the work, and since you're already *set up to use Maven*, using Maven for your project will save you a lot of time and effort. Of course, you don't have to use Maven 2, but then you'll have to get the required libraries and manage the compiling and building process yourself.

Note

ModeShape may provide in the future a Maven archetype for creating detector projects. If you'd find this useful and would like to help create it, please *join the community*.

Note

The **modeshape-mimetype-detector-aperture** project is a small, self-contained detector implementation that that you can use to help you get going. Starting with this project's source and modifying it to suit your needs may be the easiest way to get started. See the subversion repository: *http://anonsvn.jboss.org/repos/modeshape/trunk/sequencers/modeshape-mimetype-detector-aperture/* [http://anonsvn.jboss.org/repos/modeshape/trunk/extensions/modeshape-mimetype-detector-aperture/]

You can create your Maven project any way you'd like. For examples, see the *Maven 2 documentation* [http://maven.apache.org/guides/getting-started/

index.html#How_do_I_make_my_first_Maven_project]. Once you've done that, just add the dependencies in your project's `pom.xml` dependencies section:

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-common</artifactId>
  <version>0.1</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>0.1</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
</dependency>
```

These are minimum dependencies required for compiling a detector. Of course, you'll have to add other dependencies that your sequencer needs.

As for testing, you probably will want to add more dependencies, such as those listed here:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.4</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-library</artifactId>
  <version>1.1</version>
  <scope>test</scope>
</dependency>
<!-- Logging with Log4J -->
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.4.3</version>
  <scope>test</scope>
</dependency>
<dependency>
```

```
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.14</version>
  <scope>test</scope>
</dependency>
```

After you've created the project, simply implement the *MimeTypeDetector* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] interface. And testing should be quite straightforward, MIME type detectors don't require any other components. In your tests, simply instantiate your *MimeTypeDetector* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] implementation, supply various combinations of names and/or *InputStream* [http://java.sun.com/j2se/1.5.0/docs/api/java/io/InputStream.html]s, and verify the output is what you expect.

To use in your application, create a *MimeTypeDetectorConfig* [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/mimetype/MimeTypeDetectorConfig.html] object with the name, description, and class information for your detector, and add to the `MimeType` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/mimetype/MimeType.html] class using the `addDetector(MimeTypeDetectorConfig [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/mimetype/MimeTypeDetectorConfig.html] config)` method. Then, just use the `MimeType` [http://docs.jboss.org/modeshape/1.0.0.Final/api/org/modeshape/repository/mimetype/MimeType.html] class.

Looking to the future

ModeShape adds a lot of new features and capabilities. It introduced an initial RESTful server that makes JCR repositories accessible over HTTP to clients. The JCR implementation was enhanced to support more features, including the ability to define and register node types using the Compact Node Definition (CND) format. A new configuration system was added, making it very easy to configure and manage the ModeShape JCR engine. An observation framework was added to the graph API. The federation connector was rewritten to improve performance and correct several issues. And quite a few issues were fixed.

What's next for ModeShape? Passing all of the JCR API compatibility tests for Level 1 and Level 2, plus some of the optional features, is the primary focus for the next release. Of course, there are a handful of improvements we'd like to make under the covers, and a few outstanding issues that we'll address. Farther out on our *roadmap* [http://jira.jboss.org/jira/browse/MODE?report=com.atlassian.jira.plugin.system.project:roadmap-panel] are the development of additional connectors and sequencers, some Eclipse tooling for publishing artifacts to a repository, and quite a few other interesting features.

We're always looking for suggestions and contributors. If you'd like to get involved on ModeShape, the first step is joining the *mailing lists* [http://www.modeshape.org/lists.html] or hopping into our chat room on IRC (at irc.freenode.net#jbossmodeshape). You can also *download the code* [http://www.modeshape.org/subversion.html] and get it building, and start looking for simple issues or bugs in our *JIRA issue management system* [http://jira.jboss.org/jira/browse/MODE].

But if nothing else, please contact us and let us know how you're using ModeShape and what we can do to make it even better.

And, if you haven't already, check out our *Getting Started* [http://docs.jboss.org/modeshape/1.0.0.Final/manuals/gettingstarted/html/index.html] guide, which has examples that you can build and run to see ModeShape in action.