ModeShape

Reference Guide

2.6.0.Beta2

by Randall M. Hauch and Brian Carothers

Target audience

This reference guide is for application developers that want a better understanding of how ModeShape works, how to take advantage of its advanced features, and how to extend the functionality. This document is also very valuable for community developers because it covers the design and implementation of most of the components that make up ModeShape.

For a higher-level introduction to ModeShape, see the *Getting Started* [http://docs.jboss.org/modeshape/2.6.0.Beta2/manuals/gettingstarted/html/index.html] document.

If you have any questions or comments, please feel free to use ModeShape's *user mailing list* [mailto:modeshape-users@lists.jboss.org] or *user forums* [http://community.jboss.org/community/modeshape]. We welcome all who want to *get involved* [http://www.modeshape.org/community.html]. 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* [http://www.jcp.org/en/jsr/detail?id=283] 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 2.0 API (a.k.a. *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283]), 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 far less useful.

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

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

and other features that help to improve performance and usefulness. Plus, Hibernate is open-source (with support offered by *JBoss* [http://www.jboss.com]).

While relational databases and JPA are solutions that work well for many applications, they are more limited in cases when the information structure is highly flexible, the structure is not known *a priori*, or that structure is subject to frequent change and customization. In these situations, *content repositories* may offer a better choice for persistence. Content repositories offer the storage capabilities of relational databases with the flexibility offered by other systems, such as using files. Content repositories also typically provide other capabilities as well, including hierarchical organization, versioning, indexing, search, access control, transactions, and observation. Content repositories are often used by content management systems (CMS), document management systems (DMS), and other applications that manage electronic files (e.g., documents, images, multi-media, web content, etc.) and metadata associated with them (e.g., author, date, status, security information, etc.). The *Content Repository for Java technology API* [http://www.jcp.org/en/jsr/detail?id=283] provides a standard Java API for working with content repositories. Abbreviated "JCR", this API was developed through the Java Community Process originally under *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] (as "JCR 1.0"), but has since been revised and improved as "JCR 2.0" under *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283].

The JCR 2.0 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 structured, semi-structured, and 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.

ModeShape implements the JCR 2.0 API, including many of the optional features.
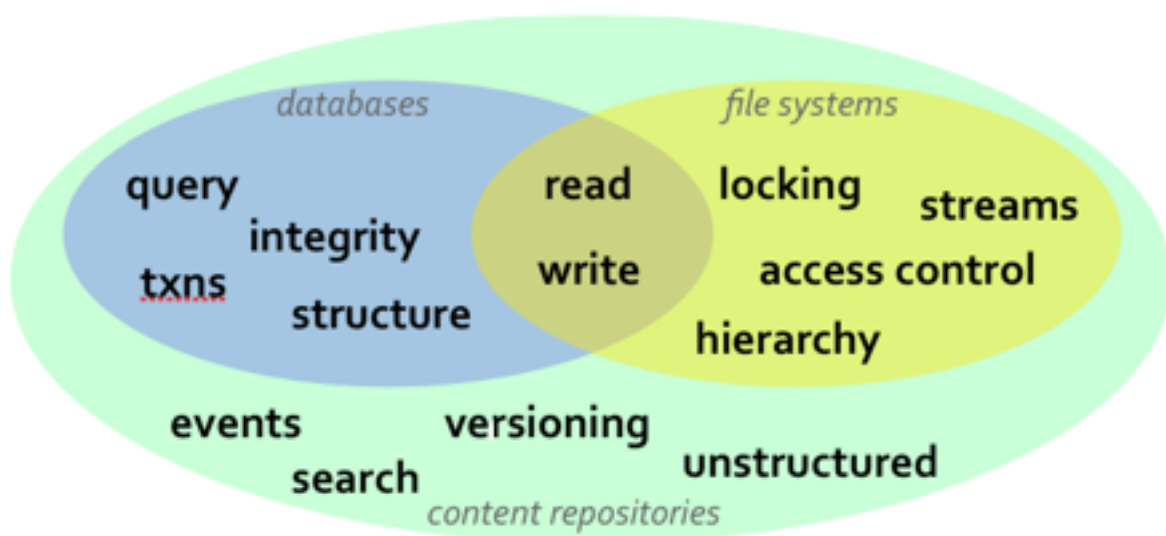
Figure 1.1. JCR API features

1.4. Project roadmap

The ModeShape open source project uses its *JIRA instance* [http://jira.jboss.org/browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:summary-panel] to track issues for tasks, requirements, bugs, and other activities. The *roadmap report* [http://jira.jboss.org/browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:roadmap-panel] shows how each of these issues are targeted to the upcoming releases, while the *change log report* [http://jira.jboss.org/browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:changelog-panel] shows all of the issues that were fixed in each of the past 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/browse/MODE/fixforversion/12314367] 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 quite a few separate modules. Just a few of these make up the essential core components of the system:

- **modeshape-jcr** contains ModeShape's implementation of the JCR 2.0 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 as easy as defining a *configuration*, obtaining the JCR *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] object for your repository using the *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html], and then using the standard JCR API. This module also uses the JCR unit tests from the reference implementation to verify the behavior of the ModeShape implementation.

- **modeshape-jcr-api** defines a number of interfaces that extend several of the JCR API interfaces. For example, this module defines a `Repositories` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/api/Repositories.html] interface that defines a way to look up `javax.jcr.Repository` instances by name, and that is implemented by the ModeShape `JcrEngine`. It also defines several new interfaces that extend the JCR 2.0 API's Query Object Model with additional behavior, including more criteria options (such as `BETWEEN`, the `mode:depth` and `jcr:path` pseudo-columns, and the `REFERENCE` function), formal `LIMIT` and `OFFSET` clauses, and a *set query operators* for unions, intersects, and difference queries. This module is very small, only depends upon the JCR API, and was designed so that client applications can depend only upon this module without having to depend on the `modeshape-jcr` interfaces or its dependencies. For example, this module defines a `Repositories` [http://

docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/api/Repositories.html] interface that defines a way to look up `javax.jcr.Repository` instances by name, and that is implemented by the ModeShape `JcrEngine`, allowing client applications to dependency on this module to look up repositories by name *without* having to depend on the

Several other modules are also essential, but for the most part are hidden to client applications as they provide components used within the JCR 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.

Most of the ModeShape modules, however, are optional extensions. Many of these depend on third party libraries, so you will probably want to include only those modules that provide functionality you'll use in your repository. These modules are located in the source under the `extensions/` directory.

- **modeshape-clustering** contains ModeShape's clustering components and are needed only when two or more ModeShape engines are to be clustered together (so listeners in one session get notifications made from within any of the engines). ModeShape clustering uses the powerful, flexible and mature *JGroups* [http://jgroups.org] reliable multicast communication library. Simply *enable clustering* in ModeShape's configuration, include this library, and start your cluster. Engines can be dynamically added and removed from the cluster.

- **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-jcr** is a ModeShape repository connector that accesses and stores content in an external JCR 2.0 repository. This allows ModeShape to integrate with other JCR implementations and even federate multiple JCR repositories into a single unified repository. Any differences in namespaces are automatically handled, although node types used by the content in the external JCR repository must also be registered into the ModeShape repository using the connector. *Note that this connector is currently a technical preview, and we're seeking feedback and assistance in identifying the required functionality.*

- **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-xsd** is a ModeShape sequencer that extracts the structure and content from XML Schema Definition (XSD) files.

- **modeshape-sequencer-wsdl** is a ModeShape sequencer that extracts the structure and content from Web Service Definition Language (WSDL) 1.1 files.

- **modeshape-sequencer-sramp** is a library with reusable node types patterned after the core model of S-RAMP, and used by other ModeShape sequencers.

- **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-teiid** contains two sequencers. ModelSequencer extracts the structured data model contained with a Teiid relational XMI model, including the catalogs, schemas, tables, views, columns, primary keys, foreign keys, indexes, procedures, procedure parameters, procedure results, logical relationships, and the JDBC source from which the model was imported. Teiid VDB files contain several models, so the VdbSequencer extracts the virtual database metadata and the structured data model from each of the models contained within the VDB.

- **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/2.6.0.Beta2/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/2.6.0.Beta2/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-extractor-tika** is a *TextExtractor* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/graph/text/TextExtractor.html] implementation that uses the *Apache Tika* [http://aperture.sourceforge.net/] parsing library to extract from binary content text that can be used for indexing the content.

- **modeshape-classloader-maven** is a small library that provides a *ClassLoaderFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ common/component/ClassLoaderFactory.html] implementation that can create *ClassLoader* [http://java.sun.com/javase/6/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). You may be able to use these artifacts "out of the box", but more likely the configuration defined in the WAR files will not be exactly what you want for your environment. In this case, you can replicate one of our "-war" modules and customize the configuration settings to easily assembly a custom WAR.

- **modeshape-web-jcr-webdav** provides a WebDAV 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 WebDAV 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 WebDAV server WAR.

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

- **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.

- **modeshape-web-jcr** provides a reusable library for web applications using JCR, and is used by the `modeshape-web-jcr-rest` and `modeshape-web-jcr-webdav` modules.

ModeShape recently added several modules that make it very easy to *deploy ModeShape in JBoss AS or EAP* as a full-fledged, central, shared service that can be monitored and administered using the embedded console and used directly by web applications deployed to the application server. Our Maven build produces a "kit" ZIP file that can be unzipped into a JBoss AS profile. When your server restarts, ModeShape will be running with a very simple configuration (although that can be easily changed).

The modules that make up the JBoss AS deployment kit are located in the source under the "`deploy/jbossas` directory":

- **modeshape-jbossas-service** provides several components that are deployed through the microcontainer in JBoss AS, registered in JNDI, and exposed through the Profile Service for monitoring and management. This service leverages the JAAS support within the application server.

- **modeshape-jbossas-console** defines the plugin for *RHQ* [http://support.rhq-project.org/ display/RHQ/Home] that enables administration, monitoring, alerting, operational control and configuration. All of the major components within a ModeShape engine are exposed as RHQ resources, and the plugin provides a number of metrics and administrative operations as well as exposing most configuration properties. *(We plan to add more metrics and operations over the next few releases, as we gain more experience using the ModeShape RHQ plugin.)*

- **modeshape-jbossas-web-rest-war** defines a variant of the more general `modeshape-web-rest-war` that is tailored for deployment on JBoss AS, since it reuses the same ModeShape service deployed into the application server.

- **modeshape-jbossas-web-webdav-war** defines a variant of the more general `modeshape-web-webdav-war` that is tailored for deployment on JBoss AS, since it reuses the same ModeShape service deployed into the application server.

There are also modules for ModeShape's documentation (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.

There are several utility modules:

- **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. This is also useful for users who prefer not to give DDL privileges to the ModeShape database user.

- **modeshape-jdbc-local** provides a JDBC driver implementation that allows JDBC clients to query the contents of a *local* JCR repository using *JCR-SQL2*. The driver even supports JDBC metadata, making it possible to dynamically discover the tables and columns available for querying (which are determined from the node types). It can be configured as a data source in JBoss AS, and can even leverage the ModeShape service, allowing JDBC-based access by clients deployed to that JBoss AS instance to query the repository content. This library is very lightweight and fast, since it directly accesses the repository using the JCR API.

- **modeshape-jdbc** provides a JDBC driver implementation that allows JDBC clients to query the contents of a *local or remote* JCR repository using *JCR-SQL2*. The driver even supports JDBC metadata, making it possible to dynamically discover the tables and columns available for querying (which are determined from the node types). It can be configured as a data source in JBoss AS, and can even leverage the ModeShape service, allowing JDBC-based access to the same repository content available via the JCR API, RESTful service, or WebDAV.

There is another module that runs the full suite of JCR TCK tests, and which at the moment still contains a few failures. This module is never needed in client applications.

- **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.

Another module provides system- and integration-level tests and is never needed in client applications:

- **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).

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* [https://repository.jboss.org/nexus/content/repositories/public/org/modeshape/] in the *JBoss Maven 2 Repository* [https://repository.jboss.org/nexus/], the settings for which are described on the *JBoss.org wiki* [http://community.jboss.org/wiki/MavenGettingStarted-Users].

1.6. Compiling and building

The ModeShape source code is freely available and easy to build. For more information about this, please see our documentation that describes the *tools* [http://community.jboss.org/wiki/ModeShapeDevelopmentTools], *build commands* [http://community.jboss.org/wiki/ModeShapeandMaven], and even *how to contribute using Git* [http://community.jboss.org/wiki/ModeShapeDevelopmentWorkflow].

1.7. What's new?

ModeShape 2.6.0.Beta2 includes several improvements and minor features, and numerous fixes for issues reported against the earlier 2.x releases. For details, see the *release notes* [http://docs.jboss.org/modeshape/2.6.0.Beta2/release.html].

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

> **Note**
>
> ModeShape 2.6.0.Beta2 currently passes 1372 of the 1391 JCR TCK tests, where 17 of these 19 failures appear to be bugs in the TCK tests (see *JCR-2648* [https://issues.apache.org/jira/browse/JCR-2648], *JCR-2661* [https://issues.apache.org/jira/browse/JCR-2661], *JCR-2662* [https://issues.apache.org/jira/browse/JCR-2662], and *JCR-2663* [https://issues.apache.org/jira/browse/JCR-2663]). The remaining 2 failures are due to a known issue (see *MODE-760* [http://jira.jboss.org/browse/MODE-760]).

Part I. 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=283] 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 (Plain Old Java Objects). 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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] and thus have access to all these facilities.

The *ExecutionContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] implements ClassLoaderFactory [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/ org/modeshape/repository/RepositoryService.html], *SequencingService* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ ExecutionContext.html] context = new *ExecutionContext* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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.

2.1. 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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] has a *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] context = ...
String username = ...
char[] password = ...
String jaasRealm = ...
SecurityContext [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/SecurityContext.html] securityContext = new JaasSecurityContext(jaasRealm, username, password);
`ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] context = ...
String jaasRealm = ...

```
CallbackHandler                    [http://java.sun.com/javase/6/docs/api/javax/security/auth/callback/
CallbackHandler.html] callbackHandler = ...
ExecutionContext        [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/SecurityContext.html]:

```
ExecutionContext        [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
ExecutionContext.html] context = ...
SecurityContext           [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
SecurityContext.html] mySecurityContext = ...
ExecutionContext        [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
ExecutionContext.html] myAppContext = context.with(mySecurityContext);
```

These *ExecutionContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html]s then represent the authenticated user in any component that uses the context.

2.1.1. JAAS

One of the *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/SecurityContext.html] implementations provided by ModeShape is the *JaasSecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

2.1.2. 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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/jcr/api/SecurityContextCredentials.html] instance around your *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/SecurityContext.html], and use that credentials to obtain a JCR *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html].

Here is an example of the *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/SecurityContext.html] implementation that uses the servlet request:

```
@Immutable
public class ServletSecurityContext implements SecurityContext [http://docs.jboss.org/
modeshape/2.6.0.Beta2/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* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/
Session.html]:

```
HttpServletRequest request = ...
Repository          [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html]
 repository = engine.getRepository("my repository");
SecurityContext          [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
SecurityContext.html] securityContext = new ServletSecurityContext(httpServletRequest);
ExecutionContext          [http://docs.jboss.org/modeshape/2.6.0.Beta2/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* [http://
www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] for the authenticated user.

2.2. Namespace Registry

As we saw earlier, every `ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/NamespaceRegistry.html] shows how these operations are done:

```
public interface NamespaceRegistry [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/
api/org/modeshape/graph/property/NamespaceRegistry.Namespaces.html] objects reflecting a
 snapshot of the registry; never null
   */
    Set<Namespace [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/NamespaceRegistry.Namespaces.html]> getNamespaces();
}
```

This interfaces exposes *Namespace* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/graph/property/NamespaceRegistry.Namespaces.html] objects that are immutable:

```
@Immutable
interface Namespace [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/NamespaceRegistry.Namespaces.html] extends Comparable<Namespace [http://
docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/NamespaceRegistry.html], but you
can even implement your own and create `ExecutionContext` [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html]s that use it:

```
NamespaceRegistry [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/NamespaceRegistry.html] myRegistry = ...
ExecutionContext [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
ExecutionContext.html] contextWithMyRegistry = context.with(myRegistry);
```

2.3. 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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] implements the *ClassLoaderFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/common/component/ClassLoaderFactory.html] interface with a single method:

```
public interface ClassLoaderFactory [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/javase/6/docs/api/java/lang/ClassLoader.html]
     getClassLoader(     String      [http://java.sun.com/javase/6/docs/api/java/lang/String.html]...
 classpath );
}
```

This means that any component that has a reference to an `ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/common/component/StandardClassLoaderFactory.html] class). Of course, it's possible to implement other *ClassLoaderFactory* [http://docs.jboss.org/

modeshape/2.6.0.Beta2/api/org/modeshape/common/component/ClassLoaderFactory.html] with other implementations. Then, just create a subcontext with your implementation:

ClassLoaderFactory [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/common/ component/ClassLoaderFactory.html] myClassLoaderFactory = ...

`ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

2.4. 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 has 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/2.6.0.Beta2/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/2.6.0.Beta2/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/javase/6/docs/api/java/io/IOException.html] If an
 error occurs reading the supplied content.
 */
```

String [http://java.sun.com/javase/6/docs/api/java/lang/String.html] mimeTypeOf(*String* [http://java.sun.com/javase/6/docs/api/java/lang/String.html] name, *InputStream* [http://java.sun.com/javase/6/docs/api/java/io/InputStream.html] content) throws *IOException* [http://java.sun.com/javase/6/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/javase/6/docs/api/java/io/InputStream.html] to the actual binary content. The result is a *String* [http://java.sun.com/javase/6/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/javase/6/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/2.6.0.Beta2/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] from the *ExecutionContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] by creating a subcontext and supplying your implementation:

MimeTypeDetector [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] myDetector = ...

ExecutionContext [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ ExecutionContext.html] contextWithMyDetector = context.with(myDetector);

2.5. Text Extractors

ModeShape can store all kinds of content, and ModeShape makes it easy to perform full-text searches on that content. To support searching, ModeShape extracts the text from the various properties on each node. They way it does this for most property types (e.g., STRING, LONG, DATE, PATH, NAME, etc.) is simply to read and use the literal values. But BINARY properties are another story: there's no way to indexes the binary content directly. Instead, ModeShape has a small pluggable framework for extracting useful text from the binary content, based upon the MIME type of the content itself.

The process works like this: when a BINARY property needs to be indexed for search, ModeShape determines the MIME type of the content, determines if there is a text extractor capable of handling that MIME type, and if so it passes the content to the text extractor and gets back a string of text, and it indexes that text.

ModeShape provides two text extractors out-of-the-box. The Teiid VDB text extractor operates only upon Teiid virtual database (i.e., ".vdb") files and extracts the virtual database's logical name, description, and version, plus the logical name, description, source name, source translator name, and JNDI name for each of the virtual database's models.

The second out-of-the-box extractor is capable of extracting text from wider variety of file types, including Microsoft Office, PDF, HTML, plain text, and XML. This extractor uses the *Tika* [http:// tika.apache.org/] toolkit from Apache, so a number of other file formats are supported. However, these other file formats require additional libraries that are not included out of the box. This is discussed in more detail in a *later chapter*.

Text extraction can be an intensive process, so it is not enabled by default. But enabling the text extractors in ModeShape's configuration is actually pretty easy. When using a configuration file, simply add a "<mode:textExtractors>" fragment under the "<configuration>" root element. Within the "<mode:textExtractors>" element place one or more "<mode:textExtractor>" fragments specifying at least the extractor's name and fully-qualified Java class.

For example, here is the fragment that defines the Teiid text extractor and the Tika text extractor. Note that the Teiid text extractor has no options and is pretty simple, while the Tika extractor allows much more control over the MIME types that should be processed:

```
<mode:textExtractors>
  <mode:textExtractor jcr:name="VDB Text Extractors">
   <mode:description>Extract text from Teiid VDB files</mode:description>
   <mode:classname>org.modeshape.extractor.teiid.TeiidVdbTextExtractor</mode:classname>
  </mode:textExtractor>
```

```xml
 <mode:textExtractor jcr:name="Tika Text Extractors">
  <mode:description>Text extractors using Tika parsers</mode:description>
  <mode:classname>org.modeshape.extractor.tika.TikaTextExtractor</mode:classname>
  <!--
  A comma- or whitespace-delimited list of MIME types that are to be excluded.
  The following are excluded by default, but the default is completely overridden
  when this property is set. In other words, if you explicitly exclude any MIME types,
  be sure to list all of the MIME types you want to exclude. Exclusions always
  have a higher precedence than inclusions.
  -->
  <mode:excludedMimeTypes>
    application/x-archive,application/x-bzip,application/x-bzip2,
    application/x-cpio,application/x-gtar,application/x-gzip,
    application/x-ta,application/zip,application/vnd.teiid.vdb
  </mode:excludedMimeTypes>
  <!--
  A comma- or whitespace-delimited list of MIME types that are to be included.
  If this is used, then the extractor will include only those MIME types found
  in this list for which there is an available parser (unless the MIME type
  is also excluded). Including explicit MIME types is often easier if text is
  to be extracted for are only a few MIME types.
  -->
  <mode:includedMimeTypes>
    application/msword,application/vnd.oasis.opendocument.text
  </mode:includedMimeTypes>
 </mode:textExtractor>
 ... <!-- other extractors -->
</mode:textExtractors>
```

It's also possible to define your own text extractors by implementing the *TextExtractor* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/text/TextExtractor.html] interface:

```java
@ThreadSafe
public interface TextExtractor [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/text/TextExtractor.html] {

  /**
   * Determine if this extractor is capable of processing content with the supplied MIME type.
   *
   * @param mimeType the MIME type; never null
```

```
     * @return true if this extractor can process content with the supplied MIME type, or false
 otherwise.
    */
        boolean   supportsMimeType(   String   [http://java.sun.com/javase/6/docs/api/java/lang/
String.html] mimeType );


   /**
     * Sequence the data found in the supplied stream, placing the output information into the
 supplied map.
     *
<p>
     * ModeShape's SequencingService determines the sequencers that should be executed by
 monitoring the changes to one or more
     * workspaces that it is monitoring. Changes in those workspaces are aggregated and used
 to determine which sequencers should
     * be called. If the sequencer implements this interface, then this method is called with the
 property that is to be sequenced
    * along with the interface used to register the output. The framework takes care of all the rest.
    * </p>


    *
    * @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
    * @throws IOException if there is a problem reading the stream
    */
   void extractFrom( InputStream [http://java.sun.com/javase/6/docs/api/java/io/InputStream.html]
 stream,
           TextExtractorOutput [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/text/TextExtractorOutput.html] output,
                TextExtractorContext [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/text/TextExtractorContext.html] context ) throws IOException;


}
```

As mentioned above, the "supportsMimeType" method will be called first, and only if your implementation returns true for a given MIME type will the "extractFrom" method be called. The supplied `TextExtractorContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/ org/modeshape/graph/text/TextExtractorContext.html] object provides information about the text being processed, while the *TextExtractorOutput* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/graph/text/TextExtractorOutput.html] is a simple interface that your extractor uses to record one or more strings containing the extracted text.

If you need text extraction in sequencers or connectors, you can always get a *TextExtractor* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/graph/text/TextExtractor.html] instance from the `ExecutionContext` [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html]. That *TextExtractor* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/text/ TextExtractor.html] implementation is actually a composite of all of the text extractors defined in the configuration.

Of course, you can always use a different *TextExtractor* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/text/TextExtractor.html] by creating a subcontext and supplying your implementation:

TextExtractor [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/text/ TextExtractor.html] myExtractor = ...
`ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ ExecutionContext.html] contextWithMyExtractor = context.with(myExtractor);

2.6. Property factory and value factories

Two other components are made available by the `ExecutionContext` [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html]. The *PropertyFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ property/PropertyFactory.html] is an interface that can be used to create *Property* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ property/Property.html] instances, which are used throughout the graph API. The *ValueFactories* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html], you can create subcontexts with different implementations:

PropertyFactory [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ property/PropertyFactory.html] myPropertyFactory = ...
`ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ ExecutionContext.html] contextWithMyPropertyFactory = context.with(myPropertyFactory);

and

```
ValueFactories          [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/ValueFactories.html] myValueFactories = ...
ExecutionContext          [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

2.7. Summary

In this chapter, we introduced the *ExecutionContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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.

3.1. 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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ property/Name.html] extends Comparable<*Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/graph/property/Name.html]>, *Serializable* [http://java.sun.com/javase/6/docs/

```
api/java/io/Serializable.html], Readable [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/ modeshape/graph/property/Name.html] objects is called *NameFactory* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/NameFactory.html] and is available within the `ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/ modeshape/graph/property/Name.html] can be used as a key in a hash-based map. *Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ Name.html] also extends the *Readable* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/graph/property/Readable.html] interface, which we'll learn more about later in this chapter.

3.2. 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/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] object is an immutable ordered sequence of *Path.Segment* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/
property/Path.html] extends Comparable<Path>, Iterable<Path.Segment [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html]>, Serializable
[http://java.sun.com/javase/6/docs/api/java/io/Serializable.html], Readable [http://docs.jboss.org/
modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ property/Path.Segment.html] is an immutable pair of a *Name* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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=283].)

```
@Immutable
public static interface Path.Segment [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/Path.Segment.html] extends Cloneable, Comparable<Path.Segment

Path.Segment.html]>,         Serializable         [http://java.sun.com/javase/6/docs/api/java/io/
Serializable.html], Readable [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/ Name.html], the only way to create a *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/graph/property/Path.html] or a *Path.Segment* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html] is to use the *PathFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ PathFactory.html], which is available within the `ExecutionContext` [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] via the `getValueFactories()` method.

3.3. 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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/DateTime.html] class; binary content is represented by an immutable *Binary* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/ javax/jcr/Binary.html] interface patterned after the interface of the same name in *JSR-283* [http:// www.jcp.org/en/jsr/detail?id=283]; and *Reference* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/
graph/property/Property.html] extends Iterable<Object>, Comparable<Property [http://
docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Property.html]>, Readable [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ property/Property.html] instances is done by using the *PropertyFactory* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/PropertyFactory.html] object owned by the *ExecutionContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/ExecutionContext.html]. This factory defines methods for creating properties

with a *Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html] and various representation of values, including variable-length arguments, arrays, *Iterator* [http://java.sun.com/javase/6/docs/api/java/util/Iterator.html], and *Iterable* [http://java.sun.com/javase/6/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/2.6.0.Beta2/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.

3.4. 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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/javase/6/docs/api/java/math/BigDecimal.html]
 value ) throws ValueFormatException;
      T  create( Calendar  [http://java.sun.com/javase/6/docs/api/java/util/Calendar.html]  value  )
 throws ValueFormatException;
       T  create( Date  [http://java.sun.com/javase/6/docs/api/java/util/Date.html]  value  )  throws
 ValueFormatException;
    T create( DateTime [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/DateTime.html] value ) throws ValueFormatException;
      T  create( Name  [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/Name.html] value ) throws ValueFormatException;
        T  create( Path  [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/Path.html] value ) throws ValueFormatException;
    T create( Reference [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/Reference] value ) throws ValueFormatException;
      T  create( URI  [http://java.sun.com/javase/6/docs/api/java/net/URL.html]  value  )  throws
 ValueFormatException;
      T  create( UUID  [http://java.sun.com/javase/6/docs/api/java/util/UUID.html]  value  )  throws
 ValueFormatException;
   T create( byte[] value ) throws ValueFormatException;
     T  create( Binary  [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html]
 value ) throws ValueFormatException, IoException;
   T create( InputStream [http://java.sun.com/javase/6/docs/api/java/io/InputStream.html] stream,
 long approximateLength ) throws ValueFormatException, IoException;
      T  create( Reader  [http://java.sun.com/javase/6/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.
```

.

```
   * @throws IoException If an unexpected problem occurs during the conversion.
   */
    Iterator<T> create( Iterator [http://java.sun.com/javase/6/docs/api/java/util/Iterator.html]<?>
 values ) throws ValueFormatException, IoException;
    Iterable<T> create( Iterable [http://java.sun.com/javase/6/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/2.6.0.Beta2/api/org/modeshape/graph/property/
ValueFactory.html]<String> stringFactory = ...
Property        [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/
ValueFactory.html]<String> stringFactory = ...
Property        [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] has a `getValueFactories()` method that return a *ValueFactories* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ValueFactories.html] interface:

This interface exposes a *ValueFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/ modeshape/graph/property/ValueFactory.html] given the `PropertyType` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/PropertyType.html] enumeration. So, the previous examples could be expanded a bit:

ValueFactory [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ ValueFactory.html]<String> stringFactory = context.getValueFactories().getStringFactory();
Property [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ Property.html] property = ...
String value = stringFactory.create(property.getFirstValue());

and

ValueFactory [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ ValueFactory.html]<String> stringFactory = context.getValueFactories().getStringFactory();
Property [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/ValueFactories.html] methods return subinterfaces of *ValueFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/NameFactory.html] interface:

```
public    interface    NameFactory    [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/NameFactory.html]   extends   ValueFactory   [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ValueFactory.html]<Name    [http://
docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html]> {


      Name   [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Name.html] create( String namespaceUri, String localName );
      Name   [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Name.html] create( String namespaceUri, String localName, TextDecoder decoder );

   NamespaceRegistry [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/NamespaceRegistry.html] getNamespaceRegistry();
}
```

and here is the *DateTimeFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/graph/property/DateTimeFactory.html] interface, which adds methods for creating *DateTime* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/
modeshape/graph/property/DateTimeFactory.html]   extends   ValueFactory   [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ValueFactory.html]<DateTime

DateTime.html]> {

  /**
   * Create a date-time instance for the current time in the local time zone.
   */
    DateTime [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/
DateTime.html] createUtc();

         DateTime    [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/DateTime.html] create( DateTime [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/DateTime.html] original, long offsetInMillis );
```

```
    DateTime [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/PathFactory.html] interface defines methods for creating relative and absolute *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] objects using combinations of other *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] objects and *Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html]s and *Path.Segment* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html]s, and introduces methods for creating *Path.Segment* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html] objects:

```
public interface PathFactory [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/PathFactory.html] extends ValueFactory [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ValueFactory.html]<Path [http://
docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html]> {

    Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Path.html] createRootPath();
    Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Path.html] createAbsolutePath( Name [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/Name.html]... segmentNames );
    Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Path.html] createAbsolutePath( Path.Segment [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/
org/modeshape/graph/property/Path.Segment.html]... segments );
    Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Path.html] createAbsolutePath( Iterable [http://java.sun.com/javase/6/docs/api/java/util/
Iterable.html]<Path.Segment [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/property/Path.Segment.html]> segments );
```

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] createRelativePath();

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] createRelativePath(*Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html]... segmentNames);

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] createRelativePath(*Path.Segment* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html]... segments);

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] createRelativePath(*Iterable* [http://java.sun.com/javase/6/docs/api/java/util/Iterable.html]<*Path.Segment* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html]> segments);

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] create(Path parentPath, *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] childPath);

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] create(Path parentPath, *Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html] segmentName, int index);

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] create(Path parentPath, String segmentName, int index);

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] create(Path parentPath, *Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html]... segmentNames);

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] create(Path parentPath, *Path.Segment* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html]... segments);

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] create(Path parentPath, *Iterable* [http://java.sun.com/javase/6/docs/api/java/util/Iterable.html]<*Path.Segment* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html]> segments);

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] create(Path parentPath, String subpath);

Path.Segment [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html] createSegment(String segmentName);

Path.Segment [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html] createSegment(String segmentName, *TextDecoder* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/common/text/TextDecoder.html] decoder);

Path.Segment [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html] createSegment(String segmentName, int index);

```
          Path.Segment        [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/Path.Segment.html]  createSegment(  Name  [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html] segmentName );
          Path.Segment        [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/Path.Segment.html]  createSegment(  Name  [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html]   segmentName,   int
 index );
}
```

And finally, the *BinaryFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/property/BinaryFactory.html] defines methods for creating *Binary* [http://www.day.com/
maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html] objects from a variety of binary formats, as
well as a method that looks for a cached *Binary* [http://www.day.com/maven/javax.jcr/javadocs/
jcr-2.0/javax/jcr/Binary.html] instance given the supplied secure hash:

```
public     interface     BinaryFactory     [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/BinaryFactory.html]   extends   ValueFactory   [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ValueFactory.html]<Binary     [http://
www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/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://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html]
  create( InputStream [http://java.sun.com/javase/6/docs/api/java/io/InputStream.html]  stream,
long approximateLength, byte[] secureHash )
              throws ValueFormatException, IoException;
          Binary    [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html]
   create( Reader   [http://java.sun.com/javase/6/docs/api/java/io/Reader.html]   reader,   long
 approximateLength, byte[] secureHash )
              throws ValueFormatException, IoException;

  /**
   * Create a binary value from the given file.
   */
  Binary [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html] create( File
  [http://java.sun.com/javase/6/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://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/
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://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/
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://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html] find( byte[]
 secureHash );
}
```

ModeShape provides efficient implementations of all of these interfaces: the *ValueFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ValueFactory.html] interfaces and subinterfaces; the *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html], *Path.Segment* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html], *Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html], *Binary* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html], *DateTime* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/DateTime.html], and *Reference* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Reference] interfaces; and the *ValueFactories* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ValueFactories.html] interface returned by the `ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html]. In fact, some of these interfaces have multiple implementations that are optimized for specific but frequently-occurring conditions.

3.5. Readable, TextEncoder, and TextDecoder

As shown above, the *Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html], *Path.Segment* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.Segment.html], *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html], and *Property* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/

Property.html] interfaces all extend the *Readable* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/ api/org/modeshape/graph/property/Readable.html] interface accept various combinations of *NamespaceRegistry* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/property/NamespaceRegistry.html] and *TextEncoder* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/
org/modeshape/graph/property/NamespaceRegistry.html] namespaceRegistry,
                    TextEncoder [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/
org/modeshape/graph/property/NamespaceRegistry.html] namespaceRegistry,
                         TextEncoder [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/
org/modeshape/common/text/TextEncoder.html]  encoder,  TextEncoder  [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/common/text/TextEncoder.html] delimiterEncoder );
}
```

We've seen the *NamespaceRegistry* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/
org/modeshape/common/text/TextEncoder.html] interface. A *TextEncoder* [http://docs.jboss.org/
modeshape/2.6.0.Beta2/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/2.6.0.Beta2/
api/org/modeshape/common/text/TextEncoder.html] implementations, including:

- The `Jsr283Encoder` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/common/text/Jsr283Encoder.html] does no conversion.

- The `UrlEncoder` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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 "&", '<', '>', '"', and '''.

- The `FileNameEncoder` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/common/text/Jsr283Encoder.html].

- The `SecureHashTextEncoder` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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.

3.6. Locations

In addition to *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/Location.html] class that encapsulates:

- the *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/Location.html] object and then use that to identify the node. `Location` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/Location.html] implements Iterable [http://java.sun.com/javase/6/docs/api/java/util/Iterable.html]<Property [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Property.html]>, Comparable<Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/Location.html]> {

        public static Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/Location.html] create( Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] path ) { ... }
    public static Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/Location.html] create( UUID [http://java.sun.com/javase/6/docs/api/java/util/UUID.html] uuid ) { ... }
        public static Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/Location.html] create( Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] path, UUID [http://java.sun.com/javase/6/docs/api/java/util/UUID.html] uuid ) { ... }
        public static Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/Location.html] create( Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] path, Property [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Property.html] idProperty ) { ... }
```

```
        public   static   Location   [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/Location.html]  create(  Path  [http://docs.jboss.org/modeshape/2.6.0.Beta2/
api/org/modeshape/graph/property/Path.html]        path,        Property        [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Property.html] firstIdProperty,
                Property [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/property/Property.html]... remainingIdProperties ) { ... }
        public   static   Location   [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/Location.html]  create(  Path  [http://docs.jboss.org/modeshape/2.6.0.Beta2/
api/org/modeshape/graph/property/Path.html]    path,    Iterable    [http://java.sun.com/javase/6/
docs/api/java/util/Iterable.html]<Property   [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/Property.html] idProperties ) { ... }
    public static Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/Location.html]  create(  Property  [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/Property.html] idProperty ) { ... }
    public static Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/Location.html]  create(  Property  [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/Property.html] firstIdProperty,
                Property [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/property/Property.html]... remainingIdProperties ) { ... }
        public   static   Location   [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/Location.html] create( Iterable [http://java.sun.com/javase/6/docs/api/java/util/
Iterable.html]<Property [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/Property.html]> idProperties ) { ... }
        public   static   Location   [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/Location.html]  create(  List  [http://java.sun.com/javase/6/docs/api/java/
util/List.html]<Property  [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
property/Property.html]> idProperties ) { ... }
  ...
}
```

Like   many   of   the   other   classes   and   interfaces,   Location   [http://docs.jboss.org/
modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/
Location.html] object with a different Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/graph/property/Path.html], a different UUID [http://java.sun.com/javase/6/docs/api/
java/util/UUID.html], or different identification properties:

```
public    abstract    class    Location    [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/
org/modeshape/graph/Location.html]    implements    Iterable    [http://java.sun.com/javase/6/
docs/api/java/util/Iterable.html]<Property    [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
```

```
modeshape/graph/property/Property.html]>,     Comparable<Location     [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/Location.html]> {
   ...
      public Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
Location.html] with( Property [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/property/Property.html] newIdProperty );
      public Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
Location.html]  with( Path  [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/property/Path.html] newPath );
      public Location [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
Location.html] with( UUID [http://java.sun.com/javase/6/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/2.6.0.Beta2/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.

3.7. 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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/Graph.html] is just building `Request` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/Graph.html] instance.

3.7.1. 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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/Graph.html]:

```
Workspace            [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/javase/6/docs/api/java/util/Set.html]<String      [http://java.sun.com/javase/6/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/javase/6/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/2.6.0.Beta2/api/org/modeshape/graph/ Workspace.html] current = graph.getCurrentWorkspace();
String [http://java.sun.com/javase/6/docs/api/java/lang/String.html] name = current.getName();
`Location` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ Location.html] rootLocation = current.getRoot();

3.7.2. 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/2.6.0.Beta2/api/ org/modeshape/graph/property/Path.html]:

Path [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ Path.html] path = ...
Map<*Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ Name.html],*Property* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/ Path.html] path = ...
for (*Property* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/ Path.html] path = ...
for (`Location` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ Location.html] child : graph.getChildren().of(path)) {

```
        Path   [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Path.html] childPath = child.getPath();
   ...
}
```

Notice that the examples pass a *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/api/org/modeshape/ graph/property/Path.html]s, `Location` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/ org/modeshape/graph/Location.html]s, `UUID` [http://java.sun.com/javase/6/docs/api/java/util/ UUID.html], or *Property* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/
Path.html] path = ...
Location              [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
Location.html] location = ...
Property        [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Property.html] idProp1 = ...
Property        [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Property.html] idProp2 = ...
UUID [http://java.sun.com/javase/6/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/2.6.0.Beta2/ api/org/modeshape/graph/Graph.html] object, and that is in a *batch* mode. Simply create a `Graph.Batch` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/property/
Path.html] path = ...
String path2 = ...
Location              [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
Location.html] location = ...
Property       [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Property.html] idProp1 = ...
Property       [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/
Property.html] idProp2 = ...
UUID [http://java.sun.com/javase/6/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/Graph.html] JavaDocs.

3.8. Requests

ModeShape `Graph` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/Graph.html] object creates the _Request_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/graph/request/Request.html] objects using _Location_ [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/Graph.html] object, which then extracts and returns the results.

3.8.1. Basic Requests

There are actually quite a few different types of _Request_ [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/Request.html] classes:

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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/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. The connector returns all properties that were found on the node, or sets a *PathNotFoundException* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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. 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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/

modeshape/graph/request/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/2.6.0.Beta2/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/2.6.0.Beta2/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 `ReadNodeRequest`s. The connector reads the branch to the specified maximum depth, 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/2.6.0.Beta2/ 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/2.6.0.Beta2/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.

3.8.2. Change Requests

ChangeRequest [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ request/ChangeRequest.html] is a subclass of *Request* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ request/ChangeRequest.html] objects also get reused by the *observation* system.

There specific subclasses of *ChangeRequest* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/ org/modeshape/graph/request/ChangeRequest.html] are:

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 identification properties).. The connector sets a *PathNotFoundException* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/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 well as the location of the new copy. The connector sets a *PathNotFoundException* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/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 connector sets a *PathNotFoundException* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/api/org/ modeshape/graph/request/InvalidWorkspaceException.html] error on the request if the named workspace does not exist.

3.8.3. Workspace Read Requests

There are also requests that read information about workspaces:

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).

3.8.4. Workspace Change Requests

And there are also requests that deal with changing workspaces (and thus extend *ChangeRequest* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ request/ChangeRequest.html]):

CreateWorkspaceRequest

A request to create a workspace with a particular name. 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). The connector sets a *InvalidWorkspaceException* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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.

3.8.5. Search Requests

Several requests are designed to push searches and queries down to the connector, if connectors support such operations:

SearchRequest

A request to query a named workspace using a supplied query. The connector returns tuples containing the columns and resulting values, plus statistics about the execution of the query.

FullTextSearchRequest

A request to search a named workspace using a supplied full-text search string and optional offset and limit values. The connector returns tuples containing the columns and resulting values, plus statistics about the execution of the query.

3.8.6. Function Requests

One type of request allows a function to be passed to the connector:

FunctionRequest

A request that executes a supplied function at a particular location within a named workspace. The inputs to the function can be set on the request (as a series of name-value pairs), and when executed the function will set the outputs as name-value pairs on the request. This request is extremely useful for (complex) operations that must first read information from the workspace and then perform other actions.

This section covered the different kinds of `Request` [http://docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/api/org/modeshape/graph/request/ Request.html] objects are also used in the *observation* framework.

3.9. Request processors

ModeShape connectors are typically the components that receive these `Request` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/ modeshape/graph/request/Request.html]s that should be discussed.

The `RequestProcessor` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/request/Request.html] classes: simply subclass the `RequestProcessor` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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.

3.10. 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/2.6.0.Beta2/ api/org/modeshape/graph/observe/ChangeObserver.html] or *Observable* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/ graph/observe/ChangeObserver.html] uses weak references to track the *Observable* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/ Observable.html] instances, and the *ChangeObservers* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ observe/Observable.html] instances from being garbage collected, nor do observers prevent *Observable* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ observe/Observable.html] instances from being garbage collected. These abstract class provide all this functionality for free.

3.10.1. Observable

Any component that can have changes and be observed can implement the *Observable* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/

Observable.html] interface. This interface allows *Observer* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/ graph/observe/ChangeObserver.html] instances and properly implements the register and unregister mechanisms.

3.10.2. 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/2.6.0.Beta2/ api/org/modeshape/graph/observe/ChangeObserver.html] abstract class and provide an implementation of the `notify(`*`Changes`* `[http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/Changes.html])` method. Then, register the observer with an *Observable* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/Observable.html] using its `register(`*`ChangeObserver`* `[http://docs.jboss.org/modeshape/2.6.0.Beta2/api/ org/modeshape/graph/observe/ChangeObserver.html])` method. The observer's `notify(`*`Changes`* `[http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/ org/modeshape/graph/observe/Observable.html] instances with which it was registered. The `ChangeObserver` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/graph/observe/ChangeObserver.html] class automatically tracks which *Observable* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/observe/Observable.html]s. Alternatively, an observer can be unregistered from a single *Observable* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/Observable.html] using the *Observable* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/ Observable.html]'s `unregister(`*`ChangeObserver`* `[http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/ChangeObserver.html])` method.

3.10.3. Changes

The `Changes` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/request/ChangeRequest.html] objects to the connector's *Observer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/Observer.html]. But that's not the only use of *Observer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/Observer.html]s. We'll also see later how the *sequencing system* uses *Observer*[http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

3.11. 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.

4.1. Connectors

With ModeShape, your applications use the *JCR 2.0 API* [http://www.jcp.org/en/jsr/detail?id=283] 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:
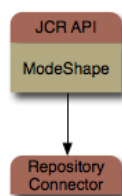
Figure 4.1. ModeShape's JCR implementation delegates to a connector

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 the content in another JCR repository

- 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)
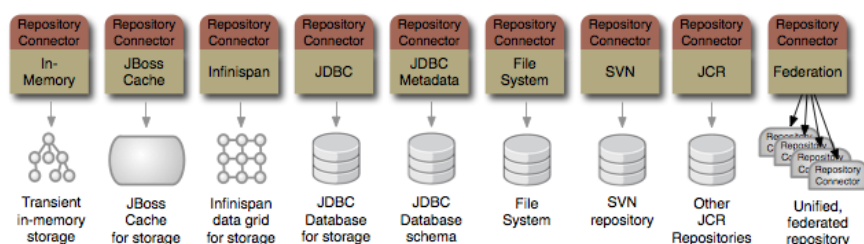
Figure 4.2. ModeShape can put JCR on top of multiple kinds of systems

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 access *existing relational databases* [http://jira.jboss.org/browse/MODE-282] 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/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*.
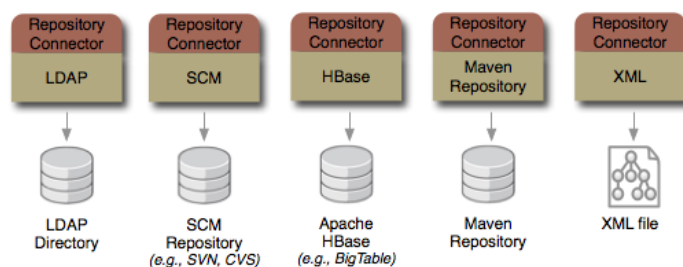
Figure 4.3. Future ModeShape connectors

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.
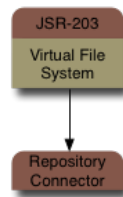
Figure 4.4. Virtual File System with ModeShape

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/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html] interface that defines methods describing the behavior and supported features and a method for establishing connections. A connector will have a class that implements this interface and that has JavaBean properties for all of the connector-specific properties required to fully describe an instance of the system. Use of JavaBean properties is not required, but it is highly recommended, as it enables reflective configuration and administration. Applications that use ModeShape create an instance of the connector's *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html] implementation and set the properties for the external source that the application wants to access with that connector.

- A repository source instance is then used to establish **connections** to that source. A connector provides an implementation of the *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositoryConnection.html] interface, which defines methods for interacting with the external system. In particular, the `execute(...)` method takes an `ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] instance and a `Request` [http://docs.jboss.org/

modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/Request.html] object. The *ExecutionContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] object defines the environment in which the processing is occurring, while the *Request* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/Request.html] object describes the requested operations on the content, with different concrete subclasses representing each type of activity. Examples of commands include (but not limited to) getting a node, moving a node, creating a node, changing a node, and deleting a node. And, if the repository source is able to participate in JTA/JTS distributed transactions, then the *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositoryConnection.html] must implement the `getXaResource()` method by returning a valid `javax.transaction.xa.XAResource` object that can be used by the transaction monitor.

As an example, consider 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/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html], and that has all of the various JavaBean properties for setting the name of the driver class, URL, username, password, and other properties. If we add a JavaBean property defining the JNDI name, our connector could look in JNDI to find a JDBC `DataSource` instance, perhaps already configured to use connection pools.

> **ℹ 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. And we've been adding new connectors with almost every release.

Our new connector might also have a `JdbcAccessConnection` Java class that implements the *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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*.

4.2. 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.

4.3. 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 3 project for your connector;

2. Implement the *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/api/ org/modeshape/graph/ExecutionContext.html],` *`Request`* `[http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/api/org/modeshape/ graph/connector/map/MapRepositorySource.html], subclassing *MapRepository* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/map/

PathRepository.html], and using the existing *MapRepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/map/MapRepositoryConnection.html] implementation. This *MapRepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/map/MapRepositoryConnection.html] does most of the work already, relying upon your *MapRepository* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/map/PathRepository.html] subclass for anything that might be source-specific. (See the *JavaDoc* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/connector/path/PathRepositorySource.html], subclassing *PathRepository* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/path/PathRepository.html], and using the existing *PathRepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/path/PathRepositoryConnection.html] implementation. Again, *PathRepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/connector/path/PathRepository.html] subclass for anything that might be source-specific. (See the *JavaDoc* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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.

4.3.1. Creating the Maven 3 project

The first step is to create the Maven 3 project that you can use to compile your code and build the JARs. Maven 3 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 3, 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 Git repository: *http://github.com/ModeShape/modeshape//tree/modeshape-2.6.0.Beta2/extensions/modeshape-connector-filesystem/*

You can create your Maven project any way you'd like. For examples, see the *Maven 3 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:

```xml
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>2.4.0.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:

```xml
<!-- ModeShape-related unit testing utilities and classes -->
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>2.4.0.Final</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
```

```xml
  <artifactId>modeshape-common</artifactId>
  <version>2.4.0.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.mockito</groupId>
   <artifactId>mockito-all</artifactId>
   <version>1.8.4</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>1.5.11</version>
 <scope>test</scope>
</dependency>
<dependency>
 <groupId>log4j</groupId>
 <artifactId>log4j</artifactId>
 <version>1.2.16</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.

4.3.2. 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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html] and inherited interface. However, there are a few characteristics that are worth mentioning here.

4.3.2.1. 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).

4.3.2.2. Broadcasting events

When your *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/connector/ RepositoryContext.html])` method will be called on the instance. The supplied *RepositoryContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/connector/RepositoryContext.html] object represents the context in which the *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/connector/RepositorySource.html] is running, and provides access to an *`ExecutionContext`* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/ExecutionContext.html], a *RepositoryConnectionFactory* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/ graph/observe/Observer.html] of your source that should be called with events describing the *`Changes`* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/observe/Changes.html] being made within the source, either as a result of *`ChangeRequest`* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

4.3.2.3. 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/2.6.0.Beta2/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.

4.3.2.4. Leveraging JNDI

Sometimes it is necessary (or easier) for a *RepositorySource* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/ 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.)

4.3.2.5. Capabilities

Another characteristic of a *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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.

4.3.2.6. 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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] provided with each request). In this case, the *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

4.3.3. Implementing a `RepositoryConnection`

One job of the *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/javase/6/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/2.6.0.Beta2/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/2.6.0.Beta2/
api/org/modeshape/graph/ExecutionContext.html] context, Request [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/Request.html] request )
    throws RepositorySourceException [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/ api/org/modeshape/graph/request/processor/RequestProcessor.html] is properly *broadcasting the changes* made during execution. The `RequestProcessor` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/processor/RequestProcessor.html] class has a `recordChange(`*ChangeRequest* `[http://docs.jboss.org/ modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/ graph/request/ChangeRequest.html]. The `RequestProcessor` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/processor/RequestProcessor.html] enqueues these requests, and when the `RequestProcessor` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/processor/RequestProcessor.html] is closed, the default implementation is to send a *Changes* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/Changes.html] to the *Observer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/ Observer.html] supplied into the constructor.

Then, in your connector's `execute(`*ExecutionContext* `[http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html],` *Request* `[http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/ Request.html])` method, instantiate your `RequestProcessor` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/processor/RequestProcessor.html] subclass and call its `process(`*Request* `[http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/graph/request/Request.html])` method, passing in

the execute(...) method's *Request* [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/Request.html] parameter.
The *RequestProcessor* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/request/processor/RequestProcessor.html] will determine the appropriate method given the
actual *Request* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/
Request.html] object and will then invoke that method:

public void execute(final *ExecutionContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/
org/modeshape/graph/ExecutionContext.html] context,
                final *Request* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/request/Request.html] request) throws RepositorySourceException {
        String sourceName = // from the *RepositorySource* [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html]
        Observer [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
observe/Observer.html] observer = // from the *RepositoryContext* [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositoryContext.html]
                RequestProcessor [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/
modeshape/graph/request/ChangeRequest.html]s as a *Changes* [http://
docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/observe/Changes.html] to
 the *Observer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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.

4.3.4. 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.

4.4. 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.

5.1. 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.

5.2. Stream Sequencers

The *StreamSequencer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/javase/6/docs/api/java/io/InputStream.html]
  stream, SequencerOutput [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/sequencer/SequencerOutput.html]    output,    StreamSequencerContext    [http://
docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/javase/6/docs/api/java/io/InputStream.html] content and generating structured content using the supplied *SequencerOutput* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/sequencer/SequencerOutput.html] interface. The *StreamSequencerContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/sequencer/StreamSequencerContext.html] also provides access to the *ValueFactories* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ValueFactories.html] that can be used to create *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html], *Name* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html], and any other value objects.

The *SequencerOutput* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/
graph/property/Path.html] nodePath, Name [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

5.3. 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 5.1. 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-sibling 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 5.2. 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.

So far, we've talked about how input paths and output paths are independent of the repository and workspace. However, there are times when it's desirable to configure sequencers to only work against content in a specific source and/or specific workspace. In these cases, it is possible to specify the repository name and workspace names before the path. For example:

Table 5.3. Input Paths with Source and Workspace Names

Input Path	Description
source:default:/a/(b|c|d)	Match nodes in the "`default`" workspace within the "`source`" source that are children of the top level node "`a`" and named "`b`", "`c`" or "`d`". None of the nodes may have same-name-sibling indexes.
:default:/a/(b|c|d)	Match nodes in the "`default`" workspace within any source source that are children of the top level node "`a`" and named "`b`", "`c`" or "`d`". None of the nodes may have same-name-sibling indexes.
source::/a/(b|c|d)	Match nodes in any workspace in the "`source`" source that are children of the top level node "`a`" and named "`b`", "`c`" or "`d`". None of the nodes may have same-name-sibling indexes.
::/a/(b|c|d)	Match nodes in any within any source source that are children of the top level node "`a`" and named "`b`", "`c`" or "`d`". None of the nodes may have same-name-sibling indexes. (This is equivalent to the path "`/a/(b|c|d)`".)

Again, the rules are pretty straightforward. You can leave off the repository name and workspace name, or you can prepend the path with "`{sourceNamePattern}:{workspaceNamePattern}:`", where "`{sourceNamePattern}` is a regular-expression pattern used to match the applicable source names, and "`{workspaceNamePattern}` is a regular-expression pattern used to match the applicable workspace names. A blank pattern implies any match, and is a shorthand notation for "`.*`". Note that the repository names may not include forward slashes (e.g., '/') or colons (e.g., ':').

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.

5.4. 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.

5.5. 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 3 project for your sequencer;

2. Implement the *StreamSequencer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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.

5.5.1. Creating the Maven 3 project

The first step is to create the Maven 3 project that you can use to compile your code and build the JARs. Maven 3 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 3, 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 Git repository: *http://github.com/ModeShape/modeshape//tree/modeshape-2.6.0.Beta2/extensions/modeshape-sequencer-images/*

You can create your Maven project any way you'd like. For examples, see the *Maven 3 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:

```xml
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>2.4.0.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:

```xml
<!-- ModeShape-related unit testing utilities and classes -->
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>2.4.0.Final</version>
  <type>test-jar</type>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-common</artifactId>
```

```xml
    <version>2.4.0.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.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.8.4</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>1.5.11</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.16</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.

```xml
<!-- ModeShape JCR Repository -->
```

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-jcr</artifactId>
  <version>2.4.0.Final</version>
  <scope>test</scope>
</dependency>
<!-- Java Content Repository API -->
<dependency>
  <groupId>javax.jcr</groupId>
  <artifactId>jcr</artifactId>
  <version>2.0</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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/sequencer/image/ImageMetadataSequencer.html] implementation:

```java
public class ImageMetadataSequencer [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/sequencer/image/ImageMetadataSequencer.html]  implements  StreamSequencer

StreamSequencer.html] {

  /**
   * {@inheritDoc}
   *
   *         *    @see    StreamSequencer#sequence(InputStream,    SequencerOutput,
   StreamSequencerContext)
   */
  public void sequence( InputStream stream,
              SequencerOutput output,
              StreamSequencerContext context ) {

      ImageMetadata metadata = new ImageMetadata();
      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) {
        PathFactory pathFactory = context.getValueFactories().getPathFactory();
                                                    Path        metadataNode        =
pathFactory.createRelativePath(ImageMetadataLexicon.METADATA_NODE);

        // Place the image metadata into the output map ...
        output.setProperty(metadataNode, JcrLexicon.PRIMARY_TYPE, "image:metadata");
        // output.psetProperty(metadataNode, nameFactory.create(IMAGE_MIXINS), "");
        output.setProperty(metadataNode, JcrLexicon.MIMETYPE, metadata.getMimeType());
        // output.setProperty(metadataNode, nameFactory.create(IMAGE_ENCODING), "");
                output.setProperty(metadataNode,  ImageMetadataLexicon.FORMAT_NAME,
metadata.getFormatName());
        output.setProperty(metadataNode, ImageMetadataLexicon.WIDTH, metadata.getWidth());
      output.setProperty(metadataNode, ImageMetadataLexicon.HEIGHT, metadata.getHeight());
                output.setProperty(metadataNode,  ImageMetadataLexicon.BITS_PER_PIXEL,
metadata.getBitsPerPixel());
                 output.setProperty(metadataNode,   ImageMetadataLexicon.PROGRESSIVE,
metadata.isProgressive());
            output.setProperty(metadataNode,  ImageMetadataLexicon.NUMBER_OF_IMAGES,
metadata.getNumberOfImages());
            output.setProperty(metadataNode,  ImageMetadataLexicon.PHYSICAL_WIDTH_DPI,
metadata.getPhysicalWidthDpi());
          output.setProperty(metadataNode,  ImageMetadataLexicon.PHYSICAL_HEIGHT_DPI,
metadata.getPhysicalHeightDpi());
       output.setProperty(metadataNode, ImageMetadataLexicon.PHYSICAL_WIDTH_INCHES,
metadata.getPhysicalWidthInch());
      output.setProperty(metadataNode, ImageMetadataLexicon.PHYSICAL_HEIGHT_INCHES,
metadata.getPhysicalHeightInch());
    }
  }
}
```

where the _ImageMetadataLexicon_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/sequencer/image/ImageMetadataLexicon.html] class contains the _Name_ [http://

docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Name.html]
constants and is defined as:

```
/**
 * A lexicon of names used within the image sequencer.
 */
@Immutable
 public class ImageMetadataLexicon [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/sequencer/image/ImageMetadataLexicon.html] {

  public static class Namespace {
      public static final String URI = "http://www.modeshape.org/images/1.0";
      public static final String PREFIX = "image";
  }

  public static final Name METADATA_NODE = new BasicName(Namespace.URI, "metadata");
  public static final Name FORMAT_NAME = new BasicName(Namespace.URI, "formatName");
  public static final Name WIDTH = new BasicName(Namespace.URI, "width");
  public static final Name HEIGHT = new BasicName(Namespace.URI, "height");
  public static final Name BITS_PER_PIXEL = new BasicName(Namespace.URI, "bitsPerPixel");
  public static final Name PROGRESSIVE = new BasicName(Namespace.URI, "progressive");
      public static final Name NUMBER_OF_IMAGES = new BasicName(Namespace.URI,
"numberOfImages");
      public static final Name PHYSICAL_WIDTH_DPI = new BasicName(Namespace.URI,
"physicalWidthDpi");
      public static final Name PHYSICAL_HEIGHT_DPI = new BasicName(Namespace.URI,
"physicalHeightDpi");
      public static final Name PHYSICAL_WIDTH_INCHES = new BasicName(Namespace.URI,
"physicalWidthInches");
      public static final Name PHYSICAL_HEIGHT_INCHES = new BasicName(Namespace.URI,
"physicalHeightInches");

}
```

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.)

5.5.2. Testing custom sequencers

The sequencing framework was designed to make testing sequencers much easier. In particular, the *StreamSequencer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ sequencer/StreamSequencer.html] sequencer = new *ImageMetadataSequencer* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/image/ ImageMetadataSequencer.html]();
MockSequencerOutput [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ sequencer/MockSequencerOutput.html] output = new *MockSequencerOutput* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/sequencer/ MockSequencerOutput.html]();
MockSequencerContext [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/sequencer/MockSequencerContext.html] context = new *MockSequencerContext* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/sequencer/ MockSequencerContext.html]();
InputStream [http://java.sun.com/javase/6/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/2.6.0.Beta2/api/org/modeshape/repository/
sequencer/Sequencer.html] sequencer = new ImageMetadataSequencer [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/image/
ImageMetadataSequencer.html]();
MockSequencerOutput [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
sequencer/MockSequencerOutput.html]    output    =    new    MockSequencerOutput    [http://
docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/sequencer/
MockSequencerOutput.html]();
MockSequencerContext          [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
graph/sequencer/MockSequencerContext.html] context = new MockSequencerContext [http://
docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/sequencer/
MockSequencerContext.html]();
InputStream [http://java.sun.com/javase/6/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.

5.6. 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 II. ModeShape JCR

The ModeShape project provides an implementation of the *JCR 2.0 API* [http://www.jcp.org/en/jsr/detail?id=283], 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.

Configuration

Using ModeShape within your application is actually quite straightforward, and with JCR 2.0 it is possible for your application to do everything using only the JCR 2.0 API. Your application will first obtain a `javax.jcr.Repository` instance, and will use that object to create sessions through which your application will read, modify, search, or monitor content in the repository.

However, before you can use ModeShape, you need to configure it, and that's what this chapter covers.

6.1. Configuring ModeShape

There really are three options:

- **`Load from a file`** is conceptually the most straightforward and requires the least amount of Java code, but it does requires having a configuration file. This is easy, allows one to manage configurations in version control, enables your application to use only the standard JCR API, and will likely be the best approach for most applications. If you're not sure, use this approach.

- **`Programmatic configuration`** allows an application to define and edit a configuration using Java code. This is useful when you cannot pre-define your configuration, or when you want to start with a baseline configuration, make programmatic changes based upon some inputs or preferences, and then save the configuration to a file. However, this requires that you write your application directly against ModeShape-specific interfaces and class.

- **`Load from a configuration repository`** is an advanced technique that allows multiple *JcrEngine* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ jcr/JcrEngine.html] instances (usually in different processes perhaps on different machines) to easily access a (shared) configuration.

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

6.1.1. Configuration Files

By far the easiest approach to defining your ModeShape configuration is to use a configuration file. As mentioned above, you'll want to do this if your application uses the standard and implementation-independent *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/ jcr-2.0/javax/jcr/RepositoryFactory.html] mechanism to obtain the JCR *Repository* [http:// www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] reference.

Here is an example configuration file used in the repository example covered in the *Getting Started* [http://docs.jboss.org/modeshape/2.6.0.Beta2/manuals/gettingstarted/html/ index.html] document, though it has been slightly simplified for clarity):

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<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">
        <mode:option jcr:name="jaasLoginConfigName" 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 files. The values
          can be an absolute path to a classpath resource, an absolute file system
          path, a relative path on the file system (relative to where the process was
          started from), or a resolvable URL. If more than one node type definition
          file is needed, the files can be listed as a single comma-delimited string
          in the 'mode:resource' attribute of the 'jcr:nodeTypes' element, or listed
          individually using multiple mode:resource child elements (as shown below).
       -->
      <jcr:nodeTypes>
         <mode:resource>/org/example/my-node-types.cnd</mode:resource>
         <mode:resource>/org/example/additional-node-types.cnd</mode:resource>
      </jcr:nodeTypes>
    </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:predefinedWorkspaceNames>workspace2</mode:predefinedWorkspaceNames>
    <mode:predefinedWorkspaceNames>workspace3</mode:predefinedWorkspaceNames>
  </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>
```

6.1.2. Programmatic Configuration

Most likely you'll define your configuration *in a file*. But there are some situations where it's far easier - even necessary - to programmatically configure ModeShape. For example, you may not be able to predefine a configuration, because it needs parameters and information known only at runtime.

One obvious approach is to write code that takes this new information and generates a ModeShape configuration file. The challenge here is that a sizable amount of code may be required just to write out the XML file in the correct format.

Perhaps an easier approach is to use the ModeShape `JcrConfiguration` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] class to programmatically construct the configuration, and then have it write the configuration out to a file. You can even load a starting configuration, programmatically modify it, and write it out to a file. From there, your application can use the standard and implementation-independent JCR API

to find and use the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instances.

The `JcrConfiguration` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] class is used by ModeShape to read in the configuration files, but it was also 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. The next few sections describe how to configure the various parts of a ModeShape configuration.

6.1.2.1. Repository Sources

Each repository source definition must include the name of the *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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.

6.1.2.2. 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")     // can be called multiple times
    .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.

> **i** **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.

6.1.2.3. Sequencers

Each defined sequencer must specify the name of the *StreamSequencer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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.

6.1.2.4. MIME Type Detectors

Each defined MIME type detector must specify the name of the *MimeTypeDetector* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

6.1.2.5. Storing Configuration

Regardless of how the *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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, so it may be a bit hard to read. (Any good XML editor will be able to format it for readability.)

6.1.3. Loading from a Configuration Repository

So far, we've seen how to load a configuration from a file, how to programmatically define a configuration and write it out to a file. In this section, we'll see how ModeShape can load its configuration from another repository.

> **Note**
>
> This really is a very advanced way to define your configuration, so this is recommended only for those that are already very comfortable with ModeShape and its lower-level *graph API* and *connector API*.

The first step is to create and configure the *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html] instance that we'll use to access the repository where the configuration is stored. Then, create a *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance and load from this source:

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

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/2.6.0.Beta2/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 it is very useful if you ModeShape configuration file is embedded within another XML file.)

Once the `JcrConfiguration` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/ org/modeshape/jcr/JcrConfiguration.html] has been loaded from a *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ connector/RepositorySource.html], the `JcrConfiguration` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance can be used to modify the configuration and then save those changes back to the repository. This technique can be used to place a configuration into a repository (such as a database) for the first time:

```
RepositorySource configSource = ... // a RepositorySource to an empty source
JcrConfiguration config = new JcrConfiguration();

// Bind the configuration to the repository source (which is initially empty)...
configuration.loadFrom(configSource);

// Now load a configuration from a file (or construct one programmatically) ...
String pathToFile = ...
configuration.loadFrom(pathToFile);

// Now save the configuration into the source ...
configuration.save();
```

Now you can load this configuration in multiple processes, using the approach mentioned above.

6.2. JCR Repository options

ModeShape JCR repositories have a number of behaviors that can be controlled from within the configuration. These are known as *repository options*, and all have sensible defaults. However, they do allow you to better configure the JCR repository instances to best suit your needs.

As mentioned earlier, these options can be set programmatically or within the configuration file. When setting up the configuration programmatically, the actual enum literal values must be used, and all values are String literals:

```
JcrConfiguration config = ...
config.repository("repository A")
    .setOption(JcrRepository.Option.JAAS_LOGIN_CONFIG_NAME, "modeshape-jcr");
```

When using a configuration file, you set the option within the "`mode:options`" fragment under the "`mode:repository`" section. Each option fragment typically looks something like this:

```
<mode:option jcr:name="jaasLoginConfigName" mode:value="modeshape-jcr"/>
```

where the "`jcr:name`" XML attribute value contains the lower-camel-case form of the option literal, and the "`mode:value`" XML attribute value contains the repository option value. In the example above, the "jaasLoginConfigName" is the option name, and "modeshape-jcr" is the option value. An alternative representation is to set the name using the XML element name and set the primary type with an XML attribute. Thus, this fragment is equivalent to the previous listing:

```
<jaasLoginConfigName jcr:primaryType="mode:option" mode:value="modeshape-jcr"/>
```

The following table describes all of the current repository options.

Table 6.1. JCR Repository Options

Option	Description
jaasLoginConfigName	The JAAS *JAAS application configuration name* [http://java.sun.com/javase/6/docs/api/javax/security/auth/login/Configuration.html] that specifies which login module should be used to validate credentials. By default, "modeshape-jcr" is used. Set the option with an empty (zero-length) value to completely turn off JAAS authentication (see the *Built-In Providers*

Option	Description
	section for details). The enumeration literal is *Option.JAAS_LOGIN_CONFIG_NAME* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#JAAS_LOGIN_CONFIG_NAME]
systemSourceName	The name of the source (and optionally the workspace in the source) where the "/jcr:system" branch should be stored. The format is "name of workspace@name of source", or simply "name of source" if the default workspace is to be used. If this option is not used, a transient in-memory source will be used. Note that all leading and trailing whitespaces is removed for both the source name and workspace name. Thus, a value of "@" implies a zero-length workspace name and zero-length source name. Also, any use of the '@' character in source and workspace names must be escaped with a preceding backslash.

The enumeration literal is *Option.SYSTEM_SOURCE_NAME* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#SYSTEM_SOURCE_NAME] |
| anonymousUserRoles | A comma-delimited list of default roles provided for anonymous access. A null or empty value for this option means that anonymous access is disabled. The enumeration literal is *Option.ANONYMOUS_USER_ROLES* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#ANONYMOUS_USER_ROLES] |
| exposeWorksapceNamesInDescription | A boolean flag that indicates whether a complete list of workspace names should be exposed in the custom repository descriptor `"org.modeshape.jcr.api.Repository.REPOSITORY_WORKSPACES` If this option is set to true, then any code that can access the repository can retrieve a complete list of workspace names through the javax.jcr.Repository.getDescriptor(String) |

Option	Description
	method without logging in. The default value is 'true', meaning that the descriptor is populated. Since some ModeShape installations may consider the list of workspace names to be restricted information and limit the ability of some or all users to see a complete list of workspace names, this option can be set to "false" to disable this capability. If this option is set to "false", the `"org.modeshape.jcr.api.Repository.REPOSITORY_WORKSPACES` descriptor will not be set. The enumeration literal is *Option.EXPOSE_WORKSPACE_NAMES_IN_DESCRIPTOR* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#EXPOSE_WORKSPACE_NAMES_IN_
repositoryJndiLocation	A string property that when specified tells the `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] where to put the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] in JNDI. Assumes that you have write access to the JNDI tree. If no value set, then the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] will not be bound to JNDI. The enumeration literal is *Option.REPOSITORY_JNDI_LOCATION* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#REPOSITORY_JNDI_LOCATION]
queryExecutionEnabled	A boolean flag that specifies whether this repository is expected to execute searches and queries. If client applications will never perform searches or queries, then maintaining the query indexes is an unnecessary overhead, and can be disabled. Note that this is merely a hint, and that searches and queries might still work when

Option	Description
	this is set to 'false'. The default is 'true', meaning that clients can execute searches and queries. The enumeration literal is *Option.QUERY_EXECUTION_ENABLED* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#QUERY_EXECUTION_ENABLED]
queryIndexDirectory	The system may maintain a set of indexes that improve the performance of searching and querying the content. These size of these indexes depend upon the size of the content being stored, and thus may consume a significant amount of space. This option defines a location on the file system where this repository may (if needed) store indexes so they don't consume large amounts of memory.
	If specified, the value must be a valid path to a writable directory on the file system. If the path specifies a non-existant location, the repository may attempt to create the missing directories. The path may be absolute or relative to the location where this VM was started. If the specified location is not a readable and writable directory (or cannot be created as such), then this will generate an exception when the repository is created.
	The default value is null, meaning the search indexes may not be stored on the local file system and, if needed, will be stored within memory.
	The enumeration literal is *Option.QUERY_INDEX_DIRECTORY* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#QUERY_INDEX_DIRECTORY]
queryIndexesUpdatedSynchronously	An *advanced* boolean flag that specifies whether updates to the indexes (if used) should be made synchronously, meaning that a call to Session.save() will not return until

Option	Description
	the search indexes have been completely updated. The benefit of synchronous updates is that a search or query performed immediately after a save() will operate upon content that was just changed. The downside is that the save() operation will take longer. With asynchronous updates, however, the only work done during a save() invocation is that required to persist the changes in the underlying repository source, while changes to the search indexes are made in a different thread that may not run immediately. In this case, there may be an indeterminate lag before searching or querying after a save() will operate upon the changed content. The default is value '*false*', meaning the updates are performed asynchronously. The enumeration literal is *Option.QUERY_INDEXES_UPDATED_SYNCHRONOUSLY* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/jcr/ JcrRepository.Option.html#QUERY_INDEXES_UPDATED_SYNCI
queryIndexesRebuiltSynchronously	An *advanced* boolean flag that specifies whether the indexes should be rebuilt synchronously when the repository restarts. If this flag is set to '*true*', query indexes for each workspace in the repository will be rebuilt synchronously the first time that the repository is accessed (e.g., at the first login). If this flag is set to '*false*', the query indexes for each workspace in the repository will be rebuilt asynchronously. Rebuilding the indexes synchronously can cause very significant latency in the initial repository access if the repository contains a significant amount of content that must be reindexed. Updating the indexes asynchronously eliminates this latency, but repository queries may generate inconsistent results while the indexes are being updated.

Option	Description
	That is, query results may refer to content that is no longer in the repository or may fail to include appropriate results for nodes that had been added to the repository.
	The default is value '*true*', meaning the rebuilds are performed synchronously.
	The enumeration literal is *Option.QUERY_INDEXES_REBUILT_SYNCHRONOUSLY* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#QUERY_INDEXES_REBUILT_SYNCH
rebuildQueryIndexOnStartup	An *advanced* setting that specifies the strategy used to determine which query indexes need to be rebuilt when the repository restarts. ModeShape currently supports two strategies:
	• A value of "always" dictates that the query index for every workspace in the repository will be rebuilt each time that the repository restarts. This can sharply increase the startup time for the repository, particularly if the `queryIndexesRebuiltSynchronously` option is set to '*true*' (the default). However, this strategy ensures that any repository content that was modified outside of the repository (e.g., files in a *FileSystemSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/filesystem/FileSystemSource.html] that were directly modified on the file system) are properly indexed.
	• A value of "ifMissing" indicates that indexes should only be rebuilt if they do not currently exist or are obviously invalid. This strategy is always the most appropriate strategy for non-clustered repositories with repository sources that provide exclusive control over content

Option	Description
	(e.g., the *InfinispanSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/infinispan/InfinispanSource.html], the *JpaSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/store/jpa/JpaSource.html]) as it greatly reduces repository startup time for repositories with significant amounts of content. Note that repositories that do not configure the `queryIndexDirectory` option will always use an in-memory index. This type of index will not be persisted across repository restarts and will require ModeShape to rebuild the indexes each time the repository starts up even if the "ifMissing" strategy is specified. The "always" strategy is used by default and in cases where the option's value does not case-independently match the one of these two values. This was the only strategy available prior to ModeShape 2.5.0.Beta3. The enumeration literal is *Option.QUERY_INDEXES_REBUILT_SYNCHRONOUSLY* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#QUERY_INDEXES_REBUILT_SYNCH and the values are *RebuildQueryIndexOnStartupOption.ALWAYS* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.RebuildQueryIndexOnStartupOption.html#ALWAYS and *RebuildQueryIndexOnStartupOption.IF_MISSING* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.RebuildQueryIndexOnStartupOption.html#IF_MISSI
projectNodeTypes	An *advanced* boolean flag that defines whether or not the node types should be exposed as content under the "/jcr:system/jcr:nodeTypes" node. Value is either "true"

Option	Description
	or "false" (default). The enumeration literal is *Option.PROJECT_NODE_TYPES* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#PROJECT_NODE_TYPES]
readDepth	An *advanced* integer flag that specifies the depth of the subgraphs that should be loaded from the connectors during normal read operations. The default value is 1. The enumeration literal is *Option.READ_DEPTH* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#READ_DEPTH]
indexReadDepth	An *advanced* integer flag that specifies the depth of the subgraphs that should be loaded from the connectors during indexing operations. The default value is 4. The enumeration literal is *Option.INDEX_READ_DEPTH* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#INDEX_READ_DEPTH]
tablesIncludeColumnsForInheritedProperties	An *advanced* boolean flag that dictates whether the property definitions inherited from supertypes should be represented in the corresponding queryable table with columns. The JCR specification gives implementations some flexibility, so ModeShape allows this to be controlled. When this option is set to "false", then each table has only those columns representing the (single-valued) property definitions explicitly defined by the node type. When this option is set to "true" (the default), each table will contain columns for each of the (single-valued) property definitions explicitly defined on the node type and inherited by the node type from all of the supertypes. The enumeration literal is *Option.TABLES_INCLUDE_COLUMNS_FOR_INHERITED_PROP* [http://docs.jboss.org/modeshape/2.6.0.Beta2/

Option	Description
	api/org/modeshape/jcr/ JcrRepository.Option.html#TABLES_INCLUDE_COLUMNS_FOR_
performReferentialIntegrityChecks	An *advanced* boolean flag that specifies whether referential integrity checks should be performed upon Session.save(). If set to "true" (the default), referential integrity checks are performed to ensure that nodes referenced by other nodes cannot be removed. If the value is set to "false", then these referential integrity checks will not be performed when removing nodes.
	Many people generally discourage the use of REFERENCE properties because of the overhead and the need for referential integrity. These concerns are somewhat mitigated by the introduction in JCR 2.0 of the WEAKREFERENCE property type, which are excluded from referential integrity checks.
	This option is available for those cases where REFERENCE properties are *not* used within your content, and thus the referential integrity checks will never find violations. In these cases, you may disable these checks to slightly improve performance of delete operations.
	The enumeration literal is *Option.PERFORM_REFERENTIAL_INTEGRITY_CHECKS* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/jcr/ JcrRepository.Option.html#PERFORM_REFERENTIAL_INTEGRIT
versionHistoryStructure	An *advanced* flag that specifies the structure used to store version histories under the "/ jcr:system/jcr:versionStorage" branch. The JCR 2.0 specification does not predefine any particular structure, but ModeShape supports two types: • A value of "flat" dictates that all "nt:versionHistory" nodes are stored with a name matching the UUID of the

Option	Description
	versioned node and directly under the "`/jcr:system/jcr:versionStorage`" node. For example, given a "`mix:versionable`" node with the UUID `fae2b929-c5ef-4ce5-9fa1-514779ca0ae3`, the corresponding "`nt:versionHistory`" node will be at "`/jcr:system/jcr:versionStorage/fae2b929-c5ef-4ce5-9fa1-514779ca0ae3`". • A value of "hierarchical" dictates that all "`nt:versionHistory`" nodes are stored under a hierarchical structure created by the first 8 characters of the UUID string. For example, given a "`mix:versionable`" node with the UUID `fae2b929-c5ef-4ce5-9fa1-514779ca0ae3`, the corresponding "`nt:versionHistory`" node will be at "`/jcr:system/jcr:versionStorage/fa/e2/b9/29/c5ef-4ce5-9fa1-514779ca0ae3`. The "hierarchical" structure is used by default and in cases where the option's value does not case-independently match the one of these two values. The enumeration literal is _Option.VERSION_HISTORY_STRUCTURE_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#VERSION_HISTORY_STRUCTURE], and the values are _VersionHistoryOption.FLAT_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.VersionHistoryOption.html#FLAT] and _VersionHistoryOption.HIERARCHICAL_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.VersionHistoryOption.html#HIERARCHICAL]
removeDerivedContentWithOriginal	An _advanced_ boolean flag that dictates whether content derived from other content (e.g., that output by sequencers) should be

Option	Description
	automatically (re)moved when the content from which it was derived is (re)moved from the repository. For example, consider that a file is uploaded and sequenced, and that the content derived from the file is stored in the repository. When that file is (re)moved, this option dictates whether the derived content should also be (re)moved automatically. By default this option has a value of "true", ensuring that all derived content is deleted whenever the original content is deleted. A value of "false" will leave the derived content. The enumeration literal is *Option.REMOVE_DERIVED_CONTENT_WITH_ORIGINAL* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#REMOVE_DERIVED_CONTENT_WITI
useAnonymousAccessOnFailedLogin	A boolean flag that indicates whether any failed, non-anonymous login attempts will automatically cause the *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] to be created using the anonymous context. If anonymous logins are not enabled (with the anonymousUserRoles option), then the login will still fail. By default this option has a value of "false", ensuring that non-anonymous login attempts either succeed as the requested user or fail. The enumeration literal is *Option.USE_ANONYMOUS_ACCESS_ON_FAILED_LOGIN* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html#USE_ANONYMOUS_ACCESS_ON_F/
useSecurityContextCredentials	Older versions of ModeShape allowed client applications to pass in *Credentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] implementations that had a `getSecurityContext()` method that returned

Option	Description
	a *SecurityContext* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/ graph/SecurityContext.html] object, which ModeShape would then use for authorization. However, since ModeShape now provides support for customized authentication and authorization modules, this is no longer needed and has been deprecated. If, however, your applications were written to use this *SecurityContextCredentials* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/jcr/api/ SecurityContextCredentials.html] implementation, then you can enable this option to turn the old behavior back on. Note, however, that this option will be removed in the next major release. Value is either "true" or "false" (default). The enumeration literal is *Option.USE_SECURITY_CONTEXT_CREDENTIALS* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/jcr/ JcrRepository.Option.html#USE_SECURITY_CONTEXT_CREDEN

> **Warning**
>
> Setting the useAnonymousAccessOnFailedLogin option to "true" and setting the anonymousUserRoles to a valid value means that all login attempts will succeed, but named login attempts may actually succeed in an anonymous context. You can programattically determine which context is being used by checking the value of *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/ Session.html].getUserID().

6.3. Repository system content

Each JCR repository contains information about the system in the "/jcr:system" area of the repository content. All of this system content applies to the whole repository (e.g., namespaces, node types, locks, versions, etc.) and therefore every session for each workspace sees the exact same "/jcr:system" content.

ModeShape implements this behavior by storing all "/jcr:system" content in a separate workspace, and then using *federation* to project that content into each workspace. This ensures that all workspaces see the same content, without having to duplicate the "/jcr:system" content in each workspace and ensure those copies stay in sync. Federation is better than duplication.

By default, ModeShape creates this separate system workspace in a transient, in-memory store. This works great for some simplistic cases, but this doesn't work when using *clustering*, *versioning*, or dynamically registering namespaces or *adding or changing node types*. This is because these features all rely upon changing or adding content in the "`/jcr:system`" area. For example, version histories are stored under "`/jcr:system/jcr:versionStorage`", node types under "`/jcr:system/jcr:versionStorage`", and namespaces under "`/jcr:system/mode:namespaces`".

In these situations, it is necessary to persist the system content in a repository source, and if clustering is enabled this source needs to be accessible to all members of the cluster. Many times, the easiest approach is to simply define an extra workspace in your repository source where the system content can be stored. It's also possible to define a separate repository source with a separate workspace for each repository's system content. (Using a separate source is required when the repository is using a single repository source that can only store limited kinds of nodes, like the *file system connector* or *Subversion connector* that can only store `nt:file` and `nt:folder` nodes.)

You should always configure each ModeShape repository with a source for its system workspace by using the `SYSTEM_SOURCE_NAME` repository option with a value that defines the name of source and name of the workspace in that source where the system content should be stored, in the format:

workspaceName@sourceName

This specifies the system content should be stored in the workspace named "`workspaceName`" in the "`sourceName`" repository source.

The system content can be stored in any repository source capable of storing any content and, in the case of clustering, that is accessible across multiple processes. For most people, this will mean a relational database. Here is an abbreviated example of an XML configuration that defines a source for the system storage (in a MySQL database) and a repository that uses it:

```xml
<?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">
 <mode:repositories>
  <mode:repository jcr:name="car repository" mode:source="Cars">
   <mode:options jcr:primaryType="mode:options">
    <!-- Explicitly specify the "system" workspace in the "SystemStore" source. -->
    <systemSourceName jcr:primaryType="mode:option"
               mode:value="system@SystemStore"/>
    ...
   </mode:options>
   ...
  </mode:repository>
```

```
  ...
 </mode:repositories>
 <mode:sources jcr:primaryType="nt:unstructured">
  <!-- One source for the "/jcr:system" content ... -->
  <mode:source jcr:name="SystemStore"
          mode:classname="org.modeshape.connector.store.jpa.JpaSource"
          mode:description="The database store for our system content"
          mode:dialect="org.hibernate.dialect.MySQLDialect"
          mode:dataSourceJndiName="java:/MyDataSource"
          mode:defaultWorkspaceName="system"
          mode:autoGenerateSchema="validate"/>
  </mode:sources>
  <!-- An another source for the regular content ... -->
  <mode:source jcr:name="Cars"
          mode:classname="org.modeshape.connector.store.jpa.JpaSource"
          mode:description="The database store for our system content"
          mode:dialect="org.hibernate.dialect.MySQLDialect"
          mode:dataSourceJndiName="java:/MyDataSource"
          mode:defaultWorkspaceName="workspace1"
          mode:autoGenerateSchema="validate">
   <mode:predefinedWorkspaceNames>workspace1</mode:predefinedWorkspaceNames>
   <mode:predefinedWorkspaceNames>workspace2</mode:predefinedWorkspaceNames>
   <mode:predefinedWorkspaceNames>workspace3</mode:predefinedWorkspaceNames>
  </mode:sources>
  ...
 </mode:sources>
 ...
</configuration>
```

Of course, you can always use a separate workspace in your primary source, too:

```
<?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">
 <mode:repositories>
  <mode:repository jcr:name="car repository" mode:source="Cars">
   <mode:options jcr:primaryType="mode:options">
     <!-- Explicitly specify the "system" workspace in the "Cars" source. -->
     <systemSourceName jcr:primaryType="mode:option" mode:value="system@Cars"/>
     ...
   </mode:options>
   ...
  </mode:repository>
```

```
   ...
 </mode:repositories>
 <mode:sources jcr:primaryType="nt:unstructured">
  <!--
  Define one source for the regular content with a special workspace for the system content.
  -->
  <mode:source jcr:name="Cars"
          mode:classname="org.modeshape.connector.store.jpa.JpaSource"
          mode:description="The database store for our system content"
          mode:dialect="org.hibernate.dialect.MySQLDialect"
          mode:dataSourceJndiName="java:/MyDataSource"
          mode:defaultWorkspaceName="workspace1"
          mode:autoGenerateSchema="validate">
   <mode:predefinedWorkspaceNames>workspace1</mode:predefinedWorkspaceNames>
   <mode:predefinedWorkspaceNames>workspace2</mode:predefinedWorkspaceNames>
   <mode:predefinedWorkspaceNames>workspace3</mode:predefinedWorkspaceNames>
   <mode:predefinedWorkspaceNames>system</mode:predefinedWorkspaceNames>
  </mode:sources>
  ...
 </mode:sources>
 ...
</configuration>
```

6.4. Query index directory

ModeShape maintains a set of index files that are used to process *queries and searches*, using the Lucene search engine. By default, these indexes are kept in memory (primarily because it's easy to configure). But most production configurations should not store them in-memory but should instead store these index files on the local file system.

Each ModeShape repository can be configured where the indexes should be stored, using the "QUERY_INDEX_DIRECTORY" repository option (see *JcrRepository.Option* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.Option.html]) when using the programmatic API or the "queryIndexDirectory" repository option in a ModeShape configuration file. The value of this setting should be the absolute or relative path to the folder where the indexes should be stored. In this directory, ModeShape will store the index files for each workspace in a folder named similarly to the workspace. Note that ModeShape will dynamically create these workspace folders as required.

For example, here is part of a ModeShape configuration file that specifies these index files should be stored in the "`data/car_repository/indexes`" folder, relative to the folder where the JVM process was started:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<configuration xmlns:mode="http://www.modeshape.org/1.0"
          xmlns:jcr="http://www.jcp.org/jcr/1.0">
 <mode:repositories>
  <mode:repository jcr:name="car repository" mode:source="Cars">
    <mode:options jcr:primaryType="mode:options">
     <!-- Explicitly specify the directory where the index files should be stored. -->
     <queryIndexDirectory jcr:primaryType="mode:option"
                  mode:value="data/car_repository/indexes"/>
     ...
    </mode:options>
    ...
  </mode:repository>
  ...
 </mode:repositories>
 ...
</configuration>
```

6.5. Authentication and Authorization

ModeShape 2.6 introduced pluggable authentication and authorization modules. Several modules are included and configured out-of-the-box, but it is now possible to implement and configure customized authentication and authorization logic. This section describes how these modules work, what's there out-of-the-box, and how to implement and add your own modules.

The *AuthenticationProvider* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/AuthenticationProvider.html] interface defines a single method:

```java
public interface AuthenticationProvider {

 /**
  * Authenticate the user that is using the supplied credentials. If the supplied
  * credentials are authenticated, this method should construct an ExecutionContext
  * that reflects the authenticated environment, including the context's valid
  * SecurityContext that will be used for authorization throughout the Session.
  * <p>
  * Note that each provider is handed a map into which it can place name-value
  * pairs that will be used in the Session attributes of the Session that results
  * from this authentication attempt. ModeShape will ignore any attributes if
  * this provider does not authenticate the credentials.
  * </p>
  *
  * @param credentials the user's JCR credentials, which may be an
  *  AnonymousCredentials if authenticating as an anonymous user
```

```
    * @param repositoryName the name of the JCR repository; never null
    * @param workspaceName the name of the JCR workspace; never null
    * @param repositoryContext the execution context of the repository, which
    * may be wrapped by this method
    * @param sessionAttributes the map of name-value pairs that will be placed
    *  into the Session's attributes; never null
    * @return the execution context for the authenticated user, or null if
    * this provider could not authenticate the user
    */
  ExecutionContext authenticate( Credentials credentials,
                     String repositoryName,
                     String workspaceName,
                     ExecutionContext repositoryContext,
                     Map<String,Object> sessionAttributes );


}
```

When a client calls one of the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html]'s `login` methods, ModeShape calls the `authenticate` method on each of the *AuthenticationProvider* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/AuthenticationProvider.html] implementations registered with the Repository. As soon as one provider returns a non-null *ExecutionContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html], the caller is authenticated and ModeShape uses that ExecutionContext within the resulting *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html].

When the client uses the *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] and attempts to perform actions on the content, ModeShape uses the ExecutionContext's *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/SecurityContext.html] to determine whether the user has the necessary privileges. If the *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/SecurityContext.html] object implements the *AuthorizationProvider* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/AuthorizationProvider.html] interface, then ModeShape will call the `hasPermission(...)` method, passing in the ExecutionContext, the repository name, the name of the source used for the repository, the workspace name, the path of the node upon which the actions are being applied, and the array of actions (see *ModeShapePermissions* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/ModeShapePermissions.html] for the possible values):

```
public interface AuthorizationProvider {

  /**
```

```
 * Determine if the supplied execution context has permission for all of the
 * named actions in the named workspace. If not all actions are allowed, the
 * method returns false.
 *
 * @param context the context in which the subject is performing the
 *        actions on the supplied workspace
 * @param repositoryName the name of the repository containing the
 *        workspace content
 * @param repositorySourceName the name of the repository's source
 * @param workspaceName the name of the workspace in which the path exists
 * @param path the path on which the actions are occurring
 * @param actions the list of ModeShapePermissions actions to check
 * @return true if the subject has privilege to perform all of the named
 *         actions on the content at the supplied path in the
 *         given workspace within the repository, or false otherwise
 */
boolean hasPermission( ExecutionContext context,
                       String repositoryName,
                       String repositorySourceName,
                       String workspaceName,
                       Path path,
                       String... actions );
}
```

If the *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ SecurityContext.html] does not implement `AuthorizationProvider` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/AuthorizationProvider.html], then ModeShape uses role-based authorization by *mapping the actions into roles* and then for each role calling the *hasRole(...) method* on *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/graph/SecurityContext.html]. Only if all of these invocations returns `true` will the operation be allowed to continue.

6.5.1. Built-in Providers

ModeShape comes with several `AuthorizationProvider` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/AuthorizationProvider.html] implementations that are automatically configured with every Repository, depending upon other settings and options. These providers are as follows:

- `JaasProvider` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/ JaasProvider.html] uses *JAAS* for all authentication and role-based authorization. This provider authenticates clients that login to the *Repository* [http://www.day.com/maven/javax.jcr/ javadocs/jcr-2.0/javax/jcr/Repository.html] with a `SimpleCredentials` [http://www.day.com/ maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/SimpleCredentials.html] object, where the username

and password match that in the JAAS policy, or a `JaasCredentials` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/jcr/api/JaasCredentials.html] constructed with a specific and already-authenticated JAAS `LoginContext` [http://java.sun.com/javase/6/docs/ api/javax/security/auth/login/LoginContext.html]. This provider can be disabled by setting the *jaasLoginConfigName configuration option* to an empty (i.e., zero-length) value; otherwise, the option defines the name of the JAAS login configuration and will default to "modeshape-jcr" if not explicitly set.

- `ServletProvider` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/ security/ServletProvider.html] delegates all authentication and role-based authorization to the servlet framework. This provider authenticates clients that login to the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/ Repository.html] with a `ServletCredentials` [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/jcr/api/ServletCredentials.html] object, which can be constructed with the `HttpServletRequest` [http://java.sun.com/javaee/6/docs/api/javax/servlet/http/ HttpServletRequest.html]. Note this does require obtaining a session for each servlet request, which is actually how the JCR API was intended to be used within web applications.

- `AnonymousProvider` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/ security/AnonymousProvider.html] will allow clients without *Credentials* [http://www.day.com/ maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] to operate upon the repository, and will use role-based authorization based upon the roles defined by the *anonymousUserRoles configuration option*. This provider authenticates clients that provide an `AnonymousCredentials` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/jcr/api/AnonymousCredentials.html] to the *Repository* [http://www.day.com/ maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html]'s `login(...)` methods or use one of the `login(...)` methods that does not take a *Credentials* [http://www.day.com/maven/ javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] object.

> **ℹ Note**
>
> The `SecurityContextProvider` [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/jcr/security/SecurityContextProvider.html] is also configured only when the *useSecurityContextCredentials configuration option* is set to 'true'. This provider authenticates clients that pass a `SecurityContextCredentials` [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/jcr/api/SecurityContextCredentials.html] object, and delegates all authentication to the embedded SecurityContext. This deprecated approach not enabled by default, and will be removed in the next major release of ModeShape. It remains in place to enable applications that use this approach to upgrade to ModeShape 2.6 (or later) without breaking their authentication mechanism.

6.5.2. Custom Providers

It is possible to provide your own authentication and authorization logic by providing one (or more) classes that implements the `AuthorizationProvider` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/AuthorizationProvider.html] interface, specifying the names of these classes in the configuration (see below), and making the classes available on the correct classpath.

Implementing the `AuthorizationProvider` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/AuthorizationProvider.html] interface is pretty straightforward. Your class needs a no-arg constructor, and the `authenticate` method must simply authenticate the credentials for the named repository and workspace. If the credentials are not authenticated, simply return null. Otherwise, simply create an `ExecutionContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] instance (from the ExecutionContext supplied in the `repositoryContext` parameter) to contain an appropriate *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/SecurityContext.html] instance for the authenticated user. As mentioned above, the *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/SecurityContext.html] should also implement the `AuthorizationProvider` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/AuthorizationProvider.html] interface for non-role-based authorization.

For example, let's imagine that our JCR application has its own authentication and authorization system. We can integrate with that by create a new *Credentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] implementation called `MyAppCredentials` to encapsulate any information needed by the authentication/authorization system, which we'll assume is accessed by a singleton class `SecurityService`. We can then implement `AuthenticationProvider` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/AuthenticationProvider.html] as follows:

```java
public class MyAppAuthorizationProvider implements AuthorizationProvider {

  private String appName;

  /**
   * Any public JavaBean properties can be set in the configuration
   */
  public void setApplicationName( String appName ) {
    this.appName = appName;
  }

  /**
   * Authenticate the user that is using the supplied credentials. If the supplied
   * credentials are authenticated, this method should construct an ExecutionContext
   * that reflects the authenticated environment, including the context's valid
```

```java
 * SecurityContext that will be used for authorization throughout the Session.
 * <p>
 * Note that each provider is handed a map into which it can place name-value
 * pairs that will be used in the Session attributes of the Session that results
 * from this authentication attempt. ModeShape will ignore any attributes if
 * this provider does not authenticate the credentials.
 * </p>
 *
 * @param credentials the user's JCR credentials, which may be an
 *  AnonymousCredentials if authenticating as an anonymous user
 * @param repositoryName the name of the JCR repository; never null
 * @param workspaceName the name of the JCR workspace; never null
 * @param repositoryContext the execution context of the repository, which
 * may be wrapped by this method
 * @param sessionAttributes the map of name-value pairs that will be placed
 *  into the Session's attributes; never null
 * @return the execution context for the authenticated user, or null if
 * this provider could not authenticate the user
 */
public ExecutionContext authenticate( Credentials credentials,
                       String repositoryName,
                       String workspaceName,
                       ExecutionContext repositoryContext,
                       Map<String,Object> sessionAttributes );
    if ( credentials instanceof MyAppCredentials ) {
     // Try to authenticate ...
     MyAppCredentials appCreds = (MyAppCredentials)credentials;
     String user = appCreds.getUser();
     Object token = appCreds.getToken();
     AppCreds creds = SecurityService.login(appName,user,token);
     if ( creds != null ) {
       // We're in ...
       SecurityContext securityContext = new MyAppSecurityContext(creds);
       return repositoryContext.with(securityContext);
     }
    }
    return null;
 }
}
```

where the `MyAppSecurityContext` is as follows:

```java
public class MyAppSecurityContext
```

```java
        implements SecurityContext, AuthorizationProvider {
private final AppCreds creds;
public MyAppSecurityContext( AppCreds creds ) {
  this.creds = creds;
}

/**
 * {@inheritDoc SecurityContext#getUserName()}
 *
 * @see SecurityContext#getUserName()
 */
public final String getUserName() {
   return creds.getUser();
}

/**
 * {@inheritDoc SecurityContext#hasRole(String)}
 *
 * @see SecurityContext#hasRole(String)
 */
public final boolean hasRole( String roleName ) {
   // shouldn't be called since we've implemented AuthorizationProvider
   return false;
}

/**
 * {@inheritDoc}
 *
 * @see org.modeshape.graph.SecurityContext#logout()
 */
public void logout() {
   creds.logout();
}

/**
 * {@inheritDoc}
 *
 * @see org.modeshape.jcr.security.AuthorizationProvider.hasPermission
 */
public boolean hasPermission( ExecutionContext context,
              String repositoryName,
              String repositorySourceName,
              String workspaceName,
              Path path,
```

```
        String... actions ) {
   // This is imaginary and simplistic, but you'd implement any authorization logic here ...
   return this.creds.isAuthorized(repositoryName,workspaceName,path);
 }
}
```

Then we just need to configure the Repository to use this provider. In the ModeShape configuration files, there is an optional "`mode:authenticationProviders`" child element of "`mode:repository`", and within this fragment you can define zero or more authentication providers by specifying a name, the class, an optional description, and optionally any bean properties that should be called upon instantiation. (Note that the class will be instantiated only once per Repository instance). Here's an example configuration file:

```xml
<?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">
 <mode:repositories>
  <mode:repository jcr:name="MyApp Repository" mode:source="Store">
   ...
   <mode:authenticationProviders>
    <!-- Specify the providers in a manner similar to sequencer
         definitions are defined -->
    <mode:authenticationProvider jcr:name="CustomProviderA"
           mode:classname="org.example.MyAppAuthorizationProvider">
     <mode:description>My authentication provider</mode:description>
     <!-- Set JavaBean properties on provider if needed -->
     <mode:appName>MyAppName</mode:appName>
    </mode:authenticationProvider>
    ...
   </mode:authenticationProviders>
   ...
  </mode:repository>
  ...
 </mode:repositories>
 ...
</configuration>
```

6.6. Clustering

ModeShape 2.1 introduced the ability to have a cluster of *JcrEngine* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] instances distributed across multiple processes while behaving as though everything was happening in a single process. With

clusters, the workload can be distributed across multiple machines, increasing tolerance against failure while allowing ModeShape to scale out to handle more workload.

ModeShape clustering uses the powerful, flexible and mature *JGroups* [http://jgroups.org] library to handle all network communication within the cluster. JGroups provides a wealth of capabilities, including automatically detecting new engines in the cluster (called discovery), reliable multicast communication, and automatic determination of the master node in the cluster. JGroups has a flexible protocol stack, works across firewalls, WANs and LANs, and supports multiple transport protocols, failure detection, reliable unicast and multicast message transmission, and encryption.

By default, clustering is *not* enabled. This means that each `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] instance is self-contained and will not be aware of changes made in other `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] instances. This is perfect in many lightweight or embedded scenarios, because it does not introduce any overhead associated with network communication.

However, clustering ModeShape is very easy and requires only a few simple steps:

1. Enable clustering in the ModeShape configuration (more on this in a bit).

2. Include the `modeshape-clustering` module in your application, either by *JAR file* or *Maven dependency*.

3. Start (or deploy) multiple `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] instances using the same configuration. For embedded scenarios, this means simply instantiating multiple `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] instances in multiple processes. In other cases, this means deploying ModeShape to multiple servers (either using the *WebDAV* server, *REST server*, or *into JNDI* and using with your own applications).

Your JCR-based application doesn't need to change in any other ways. Any implementations registered in *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]s on any of the engines will be notified of all events, regardless of whether those events were due to changes in the local or remote engines.

It also doesn't matter how many *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instances are defined in the configuration and managed by each `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] instance: each engine in the cluster can manage multiple named repositories. ModeShape ensures that all *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]s for a named repository see the changes made to that repository, regardless of where those sessions are located in the cluster. Likewise, those same changes will not be visible to the sessions for any other named repository.

6.6.1. Enabling Clustering in ModeShape

A ModeShape configuration can have a `"clustering"` fragment that defines the name of the cluster and the JGroups configuration:

```
<mode:clustering          clusterName="modeshape-cluster"          configuration="jgroups-
modeshape.xml" />
```

The "`clusterName`" is a string that is a logical name of the cluster; all engines connecting to the same name form a cluster. Any messages multicast from one engine in the cluster will be received by all other members of the cluster. Again, the cluster name is *independent* of the repositories managed by th

The "`configuration`" value is a string that is one of:

* the absolute file system path to the file containing the JGroups XML configuration;

* the relative file system path to the file containing the JGroups XML configuration, relative to the current working directory of the Java process;

* the name of a resource on the classpath containing the JGroups XML configuration;

* the URL that can be resolved to the JGroups XML configuration; or

* the string representation of JGroups configuration, either in XML format or the older string format.

The format of this JGroups configuration will be described in the *next section*. If the "`configuration`" property is not given, ModeShape will use the default JGroups configuration (as defined by the specific JGroups version).

> **Note**
>
> Note that all engines in the cluster must have the same JGroups configuration. In fact, all engines in the cluster will almost always have exactly the same ModeShape configuration.

Here is an example of a "`clustering`" fragment defining a cluster named "modeshape-cluster" using the JGroups configuration defined in the "jgroups-modeshape.xml" file at the supplied URL:

```
<clustering clusterName="modeshape-cluster"
    configuration="file://some/path/jgroups-modeshape.xml" />
```

This next example uses the JGroups configuration defined in the "jgroups-modeshape.xml" resource file on the classpath (or as an absolute path on a *nix system):

```
<clustering clusterName="modeshape-cluster"
    configuration="/some/path/jgroups-modeshape.xml" />
```

Next is an example that specifies the JGroups configuration using the older string representation of the form:

```
<clustering clusterName="modeshape-cluster"
    configuration="PROTOCOL(param=value;param=value):PROTOCOL:PROTOCOL" />
```

Of course, the "configuration" property can be specified as a child element, too (line breaks added for readability):

```
<clustering clusterName="modeshape-cluster">
    <configuration>UDP(max_bundle_size="60000":max_bundle_timeout="30"):
            PING(timeout="2000"):...</configuration>
</clustering>
```

And finally an example that specifies the JGroups configuration using the newer XML representation (line breaks added for readability):

```
<clustering clusterName="modeshape-cluster">
    <configuration><![CDATA[<config><UDP max_bundle_size="60000"
        max_bundle_timeout="30".../><PING timeout="2000"/>...</config>]]>
    </configuration>
</clustering>
```

Note that the this example uses a child XML element for the "configuration", along with a CDATA section, so that the XML configuration can be nested within the ModeShape configuration.

> ⚠ **Warning**
>
> Remember to specify the *system workspace name* for each repository that is clustered.

6.6.2. JGroups configuration

The JGroups configuration defines a protocol stack that is used for messaging, starting with the bottom-most protocol and ending with the top-most protocol.

An example of the newer-style JGroups XML format is:

```
<config>
  <UDP
```

```xml
        mcast_addr="${jgroups.udp.mcast_addr:228.10.10.10}"
        mcast_port="${jgroups.udp.mcast_port:45588}"
        discard_incompatible_packets="true"
        max_bundle_size="60000"
        max_bundle_timeout="30"
        ip_ttl="${jgroups.udp.ip_ttl:2}"
        enable_bundling="true"
        thread_pool.enabled="true"
        thread_pool.min_threads="1"
        thread_pool.max_threads="25"
        thread_pool.keep_alive_time="5000"
        thread_pool.queue_enabled="false"
        thread_pool.queue_max_size="100"
        thread_pool.rejection_policy="Run"
        oob_thread_pool.enabled="true"
        oob_thread_pool.min_threads="1"
        oob_thread_pool.max_threads="8"
        oob_thread_pool.keep_alive_time="5000"
        oob_thread_pool.queue_enabled="false"
        oob_thread_pool.queue_max_size="100"
        oob_thread_pool.rejection_policy="Run"/>
<PING timeout="2000"
        num_initial_members="3"/>
<MERGE2 max_interval="30000"
        min_interval="10000"/>
<FD_SOCK/>
<FD timeout="10000" max_tries="5" />
<VERIFY_SUSPECT timeout="1500"  />
<BARRIER />
<pbcast.NAKACK
            use_mcast_xmit="false" gc_lag="0"
            retransmit_timeout="300,600,1200,2400,4800"
            discard_delivered_msgs="true"/>
<UNICAST timeout="300,600,1200,2400,3600"/>
<pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
            max_bytes="400000"/>
<VIEW_SYNC avg_send_interval="60000"   />
<pbcast.GMS print_local_addr="true" join_timeout="3000"
        view_bundling="true"/>
<FC max_credits="20000000"
            min_threshold="0.10"/>
<FRAG2 frag_size="60000"  />
<pbcast.STATE_TRANSFER  />
```

```
</config>
```

The older-style JGroups string format is of the form:

```
PROTOCOL(param1=value1:param2=value2):PROTOCOL:PROTOCOL
```

This format is generally harder to read and generally discouraged. Nevertheless, here's an example of the older string format defining the same stack as the previous XML example (line breaks have been added for readability):

```
UDP(
    mcast_addr="${jgroups.udp.mcast_addr:228.10.10.10}":
    mcast_port="${jgroups.udp.mcast_port:45588}":
    discard_incompatible_packets="true":
    max_bundle_size="60000":
    max_bundle_timeout="30":
    ip_ttl="${jgroups.udp.ip_ttl:2}":
    enable_bundling="true":
    thread_pool.enabled="true":
    thread_pool.min_threads="1":
    thread_pool.max_threads="25":
    thread_pool.keep_alive_time="5000":
    thread_pool.queue_enabled="false":
    thread_pool.queue_max_size="100":
    thread_pool.rejection_policy="Run":
    oob_thread_pool.enabled="true":
    oob_thread_pool.min_threads="1":
    oob_thread_pool.max_threads="8":
    oob_thread_pool.keep_alive_time="5000":
    oob_thread_pool.queue_enabled="false":
    oob_thread_pool.queue_max_size="100":
    oob_thread_pool.rejection_policy="Run"):
  PING(timeout="2000":
    num_initial_members="3"):
  MERGE2(max_interval="30000":
      min_interval="10000"):
  FD_SOCK:
  FD(timeout="10000":max_tries="5"):
  VERIFY_SUSPECT(timeout="1500"):
  BARRIER:
  pbcast.NAKACK(use_mcast_xmit="false":gc_lag="0":
          retransmit_timeout="300,600,1200,2400,4800":
```

```
            discard_delivered_msgs="true"):
  UNICAST(timeout="300,600,1200,2400,3600"):
  pbcast.STABLE(stability_delay="1000":desired_avg_gossip="50000":
          max_bytes="400000"):
  VIEW_SYNC(avg_send_interval="60000"):
  pbcast.GMS(print_local_addr="true":join_timeout="3000"
          view_bundling="true"):
  FC(max_credits="20000000":
    min_threshold="0.10"):
  FRAG2(frag_size="60000"):
  pbcast.STATE_TRANSFER
```

For more details on how to configure the JGroups stack, see the *JGroups Manual* [http://jgroups.org/ug.html].

> **Note**
>
> JGroups is also used in Infinispan, JBoss AS, and other open source projects, and many of the JGroups configurations will work with ModeShape deployed in those same environments. For example, *this blog post* [http://infinispan.blogspot.com/2010/05/infinispan-ec2-demo.html] describes how to configure JGroups with three autodiscovery options available on Amazon EC2.

6.7. Using ModeShape in Web Applications

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

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

Unfortunately, there's not single way to deploy ModeShape into any web or application server, since they all have slightly different deployment and configuration techniques. The remainder of this section will talk about how to deploy ModeShape to two popular open source servers.

6.7.1. Deploying ModeShape to JBoss AS

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

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

This JBoss AS distribution ZIP file contains several components:

- JAR files for the JCR 2.0 API and ModeShape's small extensions to the JCR API on the global classpath (that is, in the "`lib/`" directory). These APIs are available to all deployed applications, services and components. The JCR API contains the "`javax.jcr`" packages and has no other dependencies. ModeShape's extensions define interfaces in the "`org.modeshape.jcr.api`" packages; these extend a few of the standard JCR API interfaces and add several methods to make them more useful.

- The ModeShape Service, represented as an exploded JAR file in the "`deploy`" directory. This is where the *`JcrEngine`* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] is running, though any application (or other JBoss service) can access its JCR Repository instances using the standard *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] approach (covered in the *next chapter*) with JNDI URLs:

  jndi:jcr/local?repositoryName=repository

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

- A pair of JAAS properties files, located in the "`conf/props/`" directory, that come out of the box with an "admin" user (with password "admin") that has full read, write, and administration privileges, and a "guest" user (with password "guest") that has only read and write privileges. Simply edit these files to change users, passwords, and roles, or to configure JAAS differently.

- The ModeShape RESTful API, represented as an exploded WAR file in the "`deploy`" directory. This allows remote applications to interact with ModeShape to access and manipulate repository content using a RESTful API that uses JSON in the requests and responses. All ModeShape repositories can be accessed, and authentication is done using the ModeShape JAAS configuration.

- The ModeShape WebDAV API, represented as an exploded WAR file in the "`deploy`" directory. This web application allows external clients to access and manipulate the content in the ModeShape repositories using the standard WebDAV protocol. For example, you can mount a repository (or parts of it) as a network drive on most operating systems, and then

upload or download files and folders using standard OS operations and graphical tools. All ModeShape repositories can be accessed, and authentication is done using the ModeShape JAAS configuration.

- A plugin for the embedded JBoss AS console, represented as a WAR file in the "`deploy`" directory. This plugin also works with *RHQ* [http://support.rhq-project.org/display/RHQ/Home] administration, monitoring, alerting, operational control and configuration system. *(We plan to add more metrics and operations over the next few releases, as we gain more experience using the ModeShape RHQ plugin.)*

- A JDBC driver that allows applications also deployed on the same JBoss AS instance to query the repositories through JDBC. This driver is on the global classpath so it can be used in any deployed component. A single JDBC DataSource is also configured in the "`deploy/modeshape-services.jar/modeshape-jdbc-ds.xml`" file to use the single default in-memory repository available out of the box. Simply edit this file to add or change the DataSource definitions. The driver can also be used in a separate JVM to issue queries and access database metadata.

- A remote client JAR that can be used by Java applications to use JDBC or the RESTful API to remotely access a ModeShape repository deployed on JBoss AS. This JAR includes ModeShape's full JDBC driver.

Here are the contents of this file:

```
conf/
conf/props/
conf/props/modeshape-roles.properties
conf/props/modeshape-users.properties
lib/
lib/jcr-2.0.jar
lib/modeshape-jcr-api-2.6.0.Beta2.jar
lib/modeshape-jdbc-local-2.6.0.Beta2.jar
deploy/
deploy/modeshape-jboss-beans.xml
deploy/modeshape-services.jar/
deploy/modeshape-services.jar/META-INF/
deploy/modeshape-services.jar/aperture-1.1.0.Beta1.jar
deploy/modeshape-services.jar/joda-time-1.6.jar
deploy/modeshape-services.jar/lucene-analyzers-3.0.2.jar
deploy/modeshape-services.jar/lucene-core-3.0.2.jar
deploy/modeshape-services.jar/lucene-regex-3.0.2.jar
deploy/modeshape-services.jar/lucene-snowball-3.0.2.jar
deploy/modeshape-services.jar/lucene-misc-3.0.2.jar
deploy/modeshape-services.jar/poi-3.6.jar
deploy/modeshape-services.jar/poi-scratchpad-3.6.jar
deploy/modeshape-services.jar/managedConfigRepository.xml
```

deploy/modeshape-services.jar/rdf2go.api-4.6.2.jar

deploy/modeshape-services.jar/META-INF/jboss-beans.xml

deploy/modeshape-services.jar/modeshape-cnd-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-common-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-connector-filesystem-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-connector-infinispan-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-connector-jbosscache-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-connector-jcr-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-connector-jdbc-metadata-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-connector-store-jpa-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-connector-svn-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-graph-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-jbossas-service-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-jcr-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-jdbc-ds.xml

deploy/modeshape-services.jar/modeshape-mimetype-detector-aperture-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-repository-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-search-lucene-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-classfile-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-cnd-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-ddl-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-java-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-jbpm-jpdl-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-msoffice-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-teiid-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-text-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-xml-2.6.0.Beta2.jar

deploy/modeshape-services.jar/modeshape-sequencer-zip-2.6.0.Beta2.jar

deploy/modeshape-rest.war/

deploy/modeshape-rest.war/META-INF/

deploy/modeshape-rest.war/WEB-INF/

deploy/modeshape-rest.war/WEB-INF/lib/

deploy/modeshape-rest.war/META-INF/MANIFEST.MF

deploy/modeshape-rest.war/WEB-INF/jboss-web.xml

deploy/modeshape-rest.war/WEB-INF/lib/jaxrs-api-1.2.1.GA.jar

deploy/modeshape-rest.war/WEB-INF/lib/jettison-1.1.jar

deploy/modeshape-rest.war/WEB-INF/lib/modeshape-jcr-2.6.0.Beta2.jar

deploy/modeshape-rest.war/WEB-INF/lib/modeshape-web-jcr-2.6.0.Beta2.jar

deploy/modeshape-rest.war/WEB-INF/lib/modeshape-web-jcr-rest-2.6.0.Beta2.jar

deploy/modeshape-rest.war/WEB-INF/lib/resteasy-jaxb-provider-1.2.1.GA.jar

deploy/modeshape-rest.war/WEB-INF/lib/resteasy-jaxrs-1.2.1.GA.jar

deploy/modeshape-rest.war/WEB-INF/lib/resteasy-jettison-provider-1.2.1.GA.jar

deploy/modeshape-rest.war/WEB-INF/lib/scannotation-1.0.2.jar

deploy/modeshape-rest.war/WEB-INF/web.xml

```
deploy/modeshape-webdav.war/
deploy/modeshape-webdav.war/WEB-INF/
deploy/modeshape-webdav.war/WEB-INF/lib/
deploy/modeshape-webdav.war/WEB-INF/jboss-web.xml
deploy/modeshape-webdav.war/WEB-INF/lib/aperture-1.1.0.Beta1.jar
deploy/modeshape-webdav.war/WEB-INF/lib/modeshape-jcr-2.6.0.Beta2.jar
deploy/modeshape-webdav.war/WEB-INF/lib/modeshape-mimetype-detector-
aperture-2.6.0.Beta2.jar
deploy/modeshape-webdav.war/WEB-INF/lib/modeshape-web-jcr-2.6.0.Beta2.jar
deploy/modeshape-webdav.war/WEB-INF/lib/modeshape-web-jcr-webdav-2.6.0.Beta2.jar
deploy/modeshape-webdav.war/WEB-INF/lib/webdav-servlet-2.0.jar
deploy/modeshape-webdav.war/WEB-INF/web.xml
deploy/admin-console.war/
deploy/admin-console.war/plugins/
deploy/admin-console.war/plugins/modeshape-jbossas-console-2.6.0.Beta2.jar
```

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

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

Be sure to use the correct repository name.

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

6.7.2. Deploying ModeShape to Tomcat

Each kind of web server or application server is different, but all servlet containers do provide a way of configuring objects and placing them into JNDI. ModeShape provides a `JndiRepositoryFactory` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.html] class that implements and that can be used in the server's configuration. The `JndiRepositoryFactory` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.html] requires two properties:

- `configFile` is the path to the *configuration file* resource, which must be available on the classpath

- **`repositoryName`** is the name of a JCR repository that exists in the JCR configuration and that will be made available by this JNDI entry

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

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

Note that it is possible to have multiple `Resource` entries. The *`JndiRepositoryFactory`* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrRepository.html] ensures that only one *`JcrEngine`* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] is instantiated, but that a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance is registered for each entry.

Before the server can start, however, all of the ModeShape jars need to be placed on the classpath for the server. JAAS also needs to be configured, and this can be done using the application server's configuration or in your web application if you're using a simple servlet container. For more details, see the *Reference Guide* [http://docs.jboss.org/modeshape/2.6.0.Beta2/manuals/reference/html/index.html].

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

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

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

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

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

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

static {
  // Initialize IDTrust
  IDTrustConfiguration idtrustConfig = new IDTrustConfiguration();
  try {
    idtrustConfig.config("security/jaas.conf.xml");
  } catch (Exception ex) {
    throw new IllegalStateException(ex);
  }
}
%>
<%
Session sess = null;
try {
  InitialContext initCtx = new InitialContext();
  Context envCtx = (Context) initCtx.lookup("java:comp/env");
  Repository repo = (Repository) envCtx.lookup("jcr/local");
  sess = repo.login(new SimpleCredentials("readwrite", "readwrite".toCharArray()));

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

Since this uses a servlet container, there is no JAAS implementation configured, so note the loading of IDTrust to create the JAAS realm. (To make this work in Tomcat, the security folder that

contains the `jaas.conf.xml`, `users.properties`, and `roles.properties` needs to be moved into the `%CATALINA_HOME%` directory.)

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

6.8. Setting the Classpath

Before you deploy ModeShape into your application or its environment, you need to make sure that all of the ModeShape JARs are on the appropriate classpath. Two different scenarios are covered in this section: Maven-based, and using JARs with the traditional classpath.

6.8.1. Building against ModeShape via Maven

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

6.8.1.1. Using the JBoss Maven repository

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

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

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

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

6.8.1.2. Add dependency to ModeShape

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

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-jcr</artifactId>
  <version>2.4.0.Final</version>
</dependency>
```

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

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-clustering</artifactId>
  <version>2.4.0.Final</version>
</dependency>
```

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

```
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-cnd</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-ddl</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshapce</groupId>
  <artifactId>modeshape-sequencer-images</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-classfile</artifactId>
  <version>2.4.0.Final</version>
```

```xml
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-java</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-mp3</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-msoffice</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-xml</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-teiid</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-text</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-sequencer-zip</artifactId>
  <version>2.4.0.Final</version>
</dependency>
```

Here is the list of available connectors:

```xml
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-filesystem</artifactId>
  <version>2.4.0.Final</version>
```

```
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-infinispan</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-jcr</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-jbosscache</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-jdbc-metadata</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-store-jpa</artifactId>
  <version>2.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-connector-svn</artifactId>
  <version>2.4.0.Final</version>
</dependency>
```

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

6.8.2. Add dependencies for logging

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

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

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

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

6.8.3. Building against ModeShape via JARs

If your application doesn't use Maven, you'll need to obtain the ModeShape JARs and place them onto your application's classpath. ModeShape provides a *single download* [http://downloads.jboss.org/modeshape/2.6.0.Beta2/modeshape-2.6.0.Beta2-all-with-dependencies.jar] with all of the JARs for all ModeShape components and all dependencies. This file contains the following:

- **modeshape-jcr-2.6.0.Beta2-jar-with-dependencies.jar** contains all of the classes (except those under `javax.jcr`) necessary to run the core ModeShape JCR repository engine using the in-memory connector and the federating connector;

- one **modeshape-connector-<type>-2.6.0.Beta2-jar-with-dependencies.jar** for each type of connector, each containing all of the classes necessary for that connector, designed to be added to the classpath after the `modeshape-jcr-2.6.0.Beta2-jar-with-dependencies.jar` file;

- one **modeshape-sequencer-<type>-2.6.0.Beta2-jar-with-dependencies.jar** for each type of connector, each containing all of the classes necessary for that sequencer, designed to be added to the classpath after the `modeshape-jcr-2.6.0.Beta2-jar-with-dependencies.jar` file;

- **modeshape-mimetype-detector-aperture-2.6.0.Beta2-jar-with-dependencies.jar** containing all of the classes necessary for detecting the MIME type of files based upon their name and/or content, designed to be added to the classpath after the `modeshape-jcr-2.6.0.Beta2-jar-with-dependencies.jar` file;

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

logging and you must provide a logging implementation as well as the appropriate SLF4J binding JAR.

6.9. What's next

This chapter outlines how you configure ModeShape, how to deploy ModeShape into your application, and how to set up your application's environment with the required ModeShape JARs. The *next chapter* talks about how your application can use the JCR API to access ModeShape repositories.

Using the JCR API with ModeShape

The *Content Repository for Java Technology API 2.0* [http://www.jcp.org/en/jsr/detail?id=283] 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 and improved as JCR 2.0 under *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283]. Some of the improvements make it possible for your application to be written entirely against the JCR 2.0 API.

> **Note**
>
> In the interests of brevity, this chapter does not attempt to reproduce the JSR-283 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. So, for a detailed explanation of the JCR API and its many interfaces and methods, see the *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] specification.

Using ModeShape within your application is actually quite straightforward, and with JCR 2.0 it is possible for your application to do everything using only the JCR 2.0 API. Your application will first obtain a `javax.jcr.Repository` instance, and will use that object to create sessions through which your application will read, modify, search, or monitor content in the repository. JCR sessions are designed to be lightweight, so it is perfectly fine (and actually recommended) for your application to create many short-lived sessions while generally avoiding longer-lived sessions. In fact, `javax.jcr.Session` objects are not required to be thread-safe (and are not in ModeShape), so your application should avoid using a single Session instance in multiple threads.

7.1. What's new in JCR 2.0?

Before we get started talking about how to use ModeShape via the standard JCR 2.0 API, it's worth spending a little time talking about the changes in JCR 2.0 compared with JCR 1.0.

Although an application written against the JCR 1.0 API will for the most part work very well against a JCR 2.0 repository, there are a few improvements to the JCR 2.0 API that your application will likely want to leverage.

Let's look at some of the more important changes in the JCR 2.0 API. However, this is certainly not definitive nor a complete comparison, so please consult the *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] specification.

7.1.1. Connecting

JCR 1.0 did not specify a way for client applications to obtain the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance, though the JCR 1.0 specification did state this is typically done through JNDI. Consequently, JCR clients

either used the JNDI approach or were required to use implementation-specific code. Often, client applications abstracted this process to minimize their reliance upon implementation-specific interfaces.

While the JNDI approach still works, JCR 2.0 introduces a new mechanism that makes it possible to find a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/ Repository.html] instance using only the JCR API. Details of this are covered more in *later*, but suffice to say that ModeShape does support this new *RepositoryFactory* [http://www.day.com/ maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] approach.

How this affects your application: If your application used an implementation-specific approach to obtaining a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/ Repository.html] instance, you might consider changing it to use the *new RepositoryFactory [http:// www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] mechanism*.

7.1.2. Identifiers

JCR 1.0 used the notion of UUIDs on referenceable nodes - in other words those nodes with the "mix:referenceable" mixin. However, there were several disadvantages to this design. First, non-referenceable nodes had no such identifier in the JCR API, leading to difficulties in easily identifying nodes using an immutable and invariant identifier (unlike the path, which can change at any time) and requiring a fair amount of code to check whether a node is referenceable before its UUID could safely be obtained. Second and perhaps more importantly, only valid UUIDs could be used to identify nodes. This can cause difficulty when JCR is used as an API to another system that does not use UUIDs.

JCR 2.0 introduces the notion of an identifier on all nodes, and the format of this identifier is designed to be opaque to the client applications. This dramatically reduces the code to access a node's identifier down to a simple method call. And it makes it possible for an implementation to use any identifiers format. This is good for ModeShape federation, as connectors no longer need to force UUIDs for all nodes.

How this affects your application: The `Node.getUUID()` method is now deprecated, and instead your code should call `Node.getIdentifier()`, which works on any node. However, be aware that the resulting identifier is no longer required to be a valid UUID. ModeShape does support these methods and behavior.

7.1.3. Binary Values

JCR 1.0 has always supported storing binary values in properties, but clients could do little more than just stream the bytes for each value. JCR 2.0 introduces a *Binary* [http://www.day.com/ maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html] interface that defines a way to get the size of the binary value, an InputStream to the value, a method for random access to the value's bytes, and a way to dispose of the binary value when completed (allowing the implementation to better clean up memory and other resources).

How this affects your application: The way your existing JCR application accesses and sets binary values will still work, but the methods are now deprecated. Therefore, you will very likely want

to change to use the new *Binary* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/ Binary.html] interface. For example, code that previously accessed the input stream directly from the *Property* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/ Property.html]:

```
Property property = ...
InputStream stream = property.getInputStream();
try {
  // Read stream
} finally {
  stream.close();
}
```

can be minimally changed to first get the *Binary* [http://www.day.com/maven/javax.jcr/javadocs/ jcr-2.0/javax/jcr/Binary.html] value and then get the stream from this Binary value:

```
Property property = ...
InputStream stream = property.getBinary().getInputStream();
try {
  // Read stream
} finally {
  stream.close();
}
```

This second example is not using any deprecated methods, but does not actually dispose of the *Binary* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html] object. This actually works just fine in ModeShape, as closing the InputStream will automatically dispose of the *Binary* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Binary.html] object.

You may also consider whether your application may benefit from the new `Binary.getSize()` or `Binary.read(byte[],long)` methods.

7.1.4. Node Type Management

In JCR 1.0, client applications could discover node types, property definitions, and child node definitions, but the API did not provide a way for client applications to modify or create new node types. This has been rectified in the JCR 2.0 API, and is these methods are now supported by ModeShape.

Additionally, the JCR 2.0 specification formalized the *Compact Node Definition* grammar, and made a few minor improvements to the CND formats used in some JCR 1.0 implementations. Earlier ModeShape releases supported the older CND format, and ModeShape 2.6.0.Beta2 now supports the grammar as defined in the specification.

How this affects your application: Your application can now define its node types using the standard CND format and/or using the new programmatic mechanism. If you already used the older CND format, be aware of the few new options available when defining property definitions (e.g., searchable, queryable, etc.). Note that node type discovery is largely unchanged.

> **Note**
>
> Remember to specify the *system workspace name* for your repositories if dynamically adding or modifying node types. Otherwise, ModeShape will not persist your node type changes.

7.1.5. Queries

JCR 1.0 made it possible for applications to query the repository using XPath and JCR-SQL query languages. JCR 2.0 maintains the (mostly) similar Java interfaces for executing queries, but it deprecates the XPath and JCR-SQL query languages and introduces a new declarative language called "JCR-SQL2" that is a very good improvement over JCR-SQL. JCR 2.0 also introduces a new query object model (called "JCR-QOM") for defining queries using a programmatic API.

ModeShape supports all of these languages (*XPath*, *JCR-SQL*, *JCR-SQL2*, *JCR-QOM*), and also supports a *full-text query language* that is defined by the full-text search expression in the JCR-SQL2 language. Additionally, ModeShape extends most of these languages to support richer and more capable queries.

How this affects your application: Your application can continue to use XPath and JCR-SQL queries. However, your application may benefit from switching from JCR-SQL to JCR-SQL2 and its greater capabilities and expressive power. Leverage some of the ModeShape extensions to make your JCR-SQL2 queries even more powerful.

7.1.6. Workspace Management

Applications could not use the JCR 1.0 API to create or destroy workspaces, meaning such operations could only be done through a non-standard and implementation-specific API. The JCR 2.0 API now standardizes these operations, and although not all implementations are required to support them, ModeShape does support these (though not all connectors do support them).

How this affects your application: Your application can now create and remove workspaces using the standard JCR 2.0 API.

7.1.7. Observation

Applications could use the JCR 1.0 API to be notified of changes to the content, using the optional observation feature. However, the JCR 1.0 API required multiple events to be created when a subtree was moved or deleted. This requirement has been relaxed in JCR 2.0 and ModeShape now fully supports the optional observation feature.

How this affects your application: Your application can now use specification-compliant JCR 2.0 observation with ModeShape.

7.1.8. Locking

JCR 1.0 API had the notion of locking nodes, useful in situations that required synchronization around reading and modifying content. This optional API is simple and clean, and worked quite well. The JCR 2.0 API preserved all of the JCR 1.0 locking semantics, but added a few (optional) methods. ModeShape implements this optional locking feature.

How this affects your application: If your application is already using the JCR 1.0 locking feature, be aware that many of the locking-related methods on *Node* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Node.html] were deprecated in JCR 2.0 and moved to the new *LockManager* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/lock/LockManager.html] interface. However, locking semantics remain unchanged.

> **Note**
>
> Remember to specify the *system workspace name* for your repositories if clustering or if the lock information is to be persisted beyond the lifetime of the ModeShape engine.

7.1.9. Versioning

Versioning of nodes was defined as an optional feature of the JCR 1.0 API. The JCR 2.0 API expanded upon locking by defining a simple versioning model, introducing the *VersionManager* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/version/VersionManager.html] interface, and making some semantic changes as well. For example, restoring a version that contained a versioned child in its subgraph no longer automatically restores the versioned child. This behavior was ambiguous in the JCR 1.0 specification, and ModeShape 1.x performed the restore operation recursively down the graph. The JCR 2.0 specification more clearly requires a non-recursive restore. Therefore, ModeShape 2.6.0.Beta2 now supports the "full versioning" model.

How this affects your application: If your application is already using JCR 1.0 versioning feature, be aware that many of the version-related methods on *Node* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Node.html] were deprecated in JCR 2.0 and moved to the new *VersionManager* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/version/VersionManager.html] interface. Also, any reliance upon ModeShape's recursive restore operation must be changed, per the JCR 2.0 specification.

> ### Note
>
> Remember to specify the *system workspace name* for your repositories if using versioning. Otherwise, ModeShape will not persist your versioning information.

7.1.10. Importing and Exporting

Importing and exporting content is largely unchanged in JCR 2.0, with the exception of specific requirements on handling node identifiers.

How this affects your application: Exporting from a JCR 1.0 or 2.0 repository and importing into a JCR 2.0 repository should work as before. ModeShape does support importing and exporting.

7.1.11. Shareable Nodes

JCR 2.0 introduced the notion of *shareable nodes*, which allows a node that exists under one parent to be shared under multiple other nodes. These are similar to symbolic links in a *nix file system. For more details about how to create and use shareable nodes, please see the *JCR 2.0 specification* [http://www.jcp.org/en/jsr/detail?id=283].

How this affects your application: Your application can now use specification-compliant JCR 2.0 shareable nodes with ModeShape.

7.1.12. Orderable Child Nodes

Orderable child nodes was an optional feature in JCR 1.0, and has been carried over to JCR 2.0 unchanged. Node ordering has been supported by ModeShape since the initial release.

How this affects your application: No changes are required if your application relies upon node ordering.

7.1.13. Paths

As defined in JCR 1.0, paths only consisted of segments with node names. JCR 2.0 adds a new form of path called "identifier paths" that are of the form `'['` *identifier* `']'`, where *identifier* is an opaque identifier. (Note that the JCR 2.0 specification might appear to allow identifier segments and name segments to be used together, but Section 3.4.1.1 requires that an identifier segment must be the first and only segment in a path.)

How this affects your application: Any application written to JCR 1.0 paths will likely work as expected (this is certainly true when using ModeShape repositories). However, with JCR 2.0 it is now possible for your application to start making us of identifier paths. For example, PATH properties can now store identifier paths, and it is possible to resolve an identifier path to the actual node. And while the specification does not preclude an implementation returning an identifier path as the node's absolute path, ModeShape never does this and will always return the name-oriented path.

7.1.14. getItem(String)

The JCR 1.0 specification was slightly ambiguous in defining how the `getItem(String)` method behaved if the relative path could resolve to a node or a property. ModeShape always implemented this by first attempting to resolve to a node, and only if no such node could be found would it attempt to resolve to a property. The JCR 2.0 specification now explicitly specifies this behavior (see Section 3.4.2.2).

How this affects your application: Your application will need to change if it uses `getItem(String)` and expects relative paths to be resolved against properties before nodes, as this is clearly different from the JCR 2.0 specified behavior. Otherwise, your application needs no changes with respect to `getItem(String)`.

7.2. Obtaining a JCR Repository

Before your application can use a JCR repository, it has to find it. As mentioned *above*, the JCR 2.0 API defines a new *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] interface that can be used with the *Java Standard Edition Service Loader mechanism* [http://java.sun.com/javase/6/docs/api/java/util/ServiceLoader.html] to obtain a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance, all using the JCR API alone:

```
Map<String,String> parameters = ...
Repository repository = null;
for (RepositoryFactory factory : ServiceLoader.load(RepositoryFactory.class)) {
    repository = factory.getRepository(parameters);
    if (repository != null) break;
}
```

This code looks for all *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] implementations on the classpath (assuming those implementations properly defined the service provider within their JARs), and will ask each to create a repository given the supplied parameters. Thus, the parameters are specific to the implementation you want to use.

> ### Note
>
> With JCR 1.0, applications could only find a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance using implementation-specific code. This new JCR 2.0 approach is a bit more complicated, but should work with most JCR 2.0 implementations and does not require using any implementation classes. And your application can even load the

> parameters from a configuration resource, meaning nothing in your application
> depends on a particular JCR implementation.

ModeShape uses a single property named "`org.modeshape.jcr.URL`" with a value that is a URL that either resolves to a ModeShape configuration file. Pointing directly to a configuration file often works well in stand-alone applications or where the configuration is managed in a central system. JNDI works great for applications deployed to server platforms (e.g., an application server or servlet container) where multiple applications might want to use the same JCR repository (or same ModeShape engine). We'll see in the *next section* how to configure ModeShape's `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] explicitly and register it in JNDI.

So, here's the ServiceLoader example again, but with ModeShape-specific parameters:

```java
String configUrl = ... ; // URL that points to your configuration file
Map<String,String> parameters = Collections.singletonMap("org.modeshape.jcr.URL", configUrl);

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

Once you've gotten hold of a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance, you can use it to create *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]s, using code similar to:

```java
Credentials credentials = ...; // JCR credentials
String workspaceName = ...;  // Name of repository workspace
Session session = repository.login(credentials,workspaceName);
```

We'll talk about the various ways of creating sessions in a *later chapter*. First, let's look at the various kinds of URLs that you can use.

7.2.1. Configuration File URLs

The value of `configUrl` in the code snippets can be any URL that is resolvable on your system. For example:

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

In this example, the configuration file that specifies the repository setup will be loaded from the file path `relativePathToConfigFile` and the repository named `yourRepositoryName` will be returned. If ModeShape cannot find a file at the given path, it will try to load a configuration file as a resource through the classloader.

You might have noticed that this URL contains a query parameter (the "`?repositoryName=MyRepository`" part). ModeShape strips all query parameters when attempting to resolve `file:` URLs to the underlying file.

Here's another example of a file URL that uses an absolute path to the file:

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

Note the addition of the three forward slashes after the protocol portion of the URL (i.e., `file:`). These indicate the path is absolute.

Other URLs are possible, too. Here is a URL that points to a configuration file stored in a web-enabled service, such as a web server, WebDAV file share, or version control system:

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

Unlike with "`file:`" URLs, ModeShape does not strip the URL's query parameters when resolving to the configuration file, since most web servers ignore any query parameters not needed. This allows you to include additional query parameters in the URL if they're needed to retrieve the file from the server.

If your platform supports URLs with the "`classpath:`" scheme, you can point to a resource file on the classpath:

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

Not all environments have such support, however. Many application servers, including *JBoss AS* [http://jboss.org/jbossas] and *EAP* [http://www.jboss.com/products/platforms/application/], do include support by default. However, the Java Standard Edition (SE) does not come with a "`classpath:`" URL handler, though it is *easy to add* [http://stackoverflow.com/questions/861500/url-to-load-resources-from-the-classpath-in-java].

ModeShape does the same thing with all of these URLs: it looks to see whether it already has started a *JcrEngine* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/

JcrEngine.html] with a configuration file at the given URL. If so, it uses the value of the "`repositoryName`" query parameter and passes it to the `getRepository(String)` method. The result of this method call will be a *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] object that is then returned from the factory.

However, if the RepositoryFactory has not yet seen this URL, it will download the configuration file at the URL, load it using a new *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] object, and start a new *JcrEngine* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] instance. It then uses the "`repositoryName`" query parameter to obtain the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] as mentioned above.

7.2.2. Using JNDI URLs

The previous section showed how to use a URL to a configuration file to start a new ModeShape instance. However, ModeShape can be deployed and managed as a *central, shared service* in a variety of environments, including *JBoss AS* [http://jboss.org/jbossas] and *EAP* [http://www.jboss.com/products/platforms/application/]. Since a single ModeShape instance can manage multiple repositories, using a single shared instance will have a smaller footprint than multiple ModeShape instances each running a single repository. Plus, the central ModeShape instance can be configured, monitored, administered, and managed without requiring each application to perform these functions.

The easiest and most common way for applications to find and reuse this central, shared ModeShape service is to use JNDI. ModeShape's *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] implementation accepts "`jndi:`" URLs instead of the file-based URL described in the previous chapter. The format of these JNDI URLs is:

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

The RepositoryFactory will look for a ModeShape engine registered in JNDI at "`name/in/jndi`", and will ask that engine for the *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instance with the name "`MyRepository`". Note that when a JNDI URL is used, RepositoryFactory is will never create its own ModeShape engine instance: if none can be found in JNDI, the RepositoryFactory will simply return null.

Sometimes a JNDI implementation will require creating a new *InitialContext* [http://java.sun.com/javase/6/docs/api/javax/naming/InitialContext.html] instance with a hashtable of environment parameters. If this is the case for your environment, simply include those extra parameters in the Map passed into the `getRepository(Map)` method. ModeShape will forward these extra parameters into the *InitialContext* [http://java.sun.com/javase/6/docs/api/javax/naming/InitialContext.html] constructor it uses look up the JNDI reference.

7.2.3. Cleaning Up after JcrRepositoryFactory

If your application uses *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] with a ModeShape URL pointing to a configuration file, the *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] creates an embedded ModeShape engine (or several, if multiple configuration files are used) that maintains a serious of connections, thread pools, and other resources. In these cases, your application should shutdown ModeShape so that it can properly release all accumulated resources.

The *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] specification does not specify a standard way to shutdown engines or repositories created as a side effect of *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html], so ModeShape has an extension to the *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] API that provides this capability.

When you obtain your Repository instance using the ServiceLoader mechanism described earlier, keep a reference to the *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] that returns a non-null *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html]:

```
Map<String,String> parameters = ...
Repository repository = null;
RepositoryFactory factory = null;
for (RepositoryFactory aFactory : ServiceLoader.load(RepositoryFactory.class)) {
    repository = aFactory.getRepository(parameters);
    if (repository != null) {
        factory = aFactory;
        break;
    }
}
```

Save this reference where your application's shutdown code can access it, then when your application is terminating, check the type of the factory, cast to the ModeShape extension, and call the "shutdown()" method:

```
if ( factory instanceof org.modeshape.jcr.api.RepositoryFactory ) {
    ((org.modeshape.jcr.api.RepositoryFactory)factory).shutdown();
}
```

This call to shutdown(...) instructs each of the *JcrEngine* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] instances created by the factory to shutdown gracefully and return immediately (without waiting for any of them to complete the

shutdown process). If you'd rather block while the engines perform their shutdown, simply supply a timeout:

```
if ( factory instanceof org.modeshape.jcr.api.RepositoryFactory ) {
    ((org.modeshape.jcr.api.RepositoryFactory)factory).shutdown(30,TimeUnit.SECONDS);
}
```

This call will wait up to 30 seconds for each *JcrEngine* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] to shut down.

7.3. ModeShape's JcrEngine

Although the preferred mechanism to obtain a *Repository* [http://www.day.com/maven/ javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] object is through the *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] interface *described above*, there are times when an application wants or needs to have more control over an actual ModeShape engine, which encapsulates everything necessary to run one or more JCR repositories and managing the underlying repository sources, the pools of connections to the sources, the sequencers, the MIME type detector(s), and the *Repository* [http://www.day.com/ maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] implementations.

> **Note**
>
> If your application uses the *RepositoryFactory* [http://www.day.com/maven/ javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html], then you can proceed to the *next section*.

The first step to programmatically instantiating a ModeShape *JcrEngine* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] is to define a configuration file as described in the *previous chapter*. Then, load that configuration file and check for problems:

```
JcrConfiguration config = new JcrConfiguration();
configuration.loadFrom(file);
if ( !configuration.getProblems().isEmpty() ) {
    for ( Problem problem : configuration.getProblems() ) {
        // Report these problems!
    }
}
```

where the `file` parameter can actually be a *File* [http://java.sun.com/javase/6/docs/api/java/ io/File.html] instance, a *URL* [http://java.sun.com/javase/6/docs/api/java/net/URL.html] to the

file, an *InputStream* [http://java.sun.com/javase/6/docs/api/java/io/InputStream.html] containing the contents of the file, or a *String* [http://java.sun.com/javase/6/docs/api/java/lang/String.html] containing the path to the configuration file.

> **i** **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/2.6.0.Beta2/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 it is very useful if you ModeShape configuration file is embedded within another XML file.)

> **i** **Note**
>
> If your application is coding against the ModeShape classes, you may also consider *programmatically creating the configuration*. This is useful when you cannot predefine a configuration, but instead have to build one based upon some parameters known only at runtime. Of course, you can always create the configuration programmatically, write that configuration out to a file, and then load the configuration using the standard *RepositoryFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/RepositoryFactory.html] mechanism.

Once you have a valid `JcrConfiguration` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance with no errors, you can build and start the `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html]:

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

Obtaining a JCR *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] 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/2.6.0.Beta2/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* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] instances managed by the engine are marked as being shut down, and they will not be able to create new *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]s. However, any existing *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]s 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.

7.4. Creating JCR Sessions

Once you have obtained a reference to the JCR *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html], you can create a JCR session using one of its `login(...)` methods. The *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] specification provides four login methods, but the behavior of these methods depends on the kind of authentication system your application is using.

7.4.1. 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/javase/6/docs/api/java/security/AccessController.html]. This implies that this method will throw a *LoginException* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/LoginException.html] if it is not executed as a `PrivilegedAction` [http://java.sun.com/javase/6/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 [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] session =
 Subject.doAsPrivileged(subject,
   new PrivilegedExceptionAction<Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/
javax/jcr/Session.html]>() {
     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 [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] session =
 AccessController.doPrivileged(
   new PrivilegedExceptionAction<Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/
javax/jcr/Session.html]>() {
     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/javase/6/docs/api/java/lang/String.html] workspaceName = ...;
Session  [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]  session
     = (Session    [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html])
 Subject.doAsPrivileged(subject,
   new PrivilegedExceptionAction<Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/
javax/jcr/Session.html]>() {
     public Session run() throws Exception {
```

```
        return repository.login(workspaceName);
    }
}, AccessController.getContext());
```

The JCR API also allows supplying a JCR *Credentials* [http://www.day.com/maven/javax.jcr/
javadocs/jcr-2.0/javax/jcr/Credentials.html] object directly as part of the login process, although
ModeShape imposes some requirements on what types of *Credentials* [http://www.day.com/
maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] may be supplied. The simplest way
is to provide a JCR *SimpleCredentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/
javax/jcr/SimpleCredentials.html] 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/javase/6/docs/api/java/lang/String.html] userName = ...;
char[] password = ...;
Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]   session
  = repository.login(new *SimpleCredentials* [http://www.day.com/maven/javax.jcr/javadocs/
jcr-2.0/javax/jcr/SimpleCredentials.html](userName, password));

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

String [http://java.sun.com/javase/6/docs/api/java/lang/String.html] userName = ...;
char[] password = ...;
String [http://java.sun.com/javase/6/docs/api/java/lang/String.html] workspaceName = ...;
Credentials   [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html]
  credentials = new *SimpleCredentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/
javax/jcr/SimpleCredentials.html](userName, password);
Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] session =
 repository.login(credentials, workspaceName);

If you'd want to use a different JAAS realm that what ModeShape is configured to
use, you can use a *JaasCredentials* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/jcr/api/JaasCredentials.html] instance to pass the actual JAAS *LoginContext* [http://
java.sun.com/javase/6/docs/api/javax/security/auth/login/LoginContext.html] that should be used
for authentication and authorization:

```
LoginContext                              [http://java.sun.com/javase/6/docs/api/javax/security/auth/login/
LoginContext.html] loginContext = ...;
Credentials        [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html]
  credentials  =  new  JaasCredentials  [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/jcr/api/JaasCredentials.html](loginContext);
String [http://java.sun.com/javase/6/docs/api/java/lang/String.html] workspaceName = ...;
Session  [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]  session  =
 repository.login(credentials,workspaceName);
```

Note that even in this case, ModeShape will still use the same *roles* for authorization.

7.4.2. Using HTTP Servlet security

Servlet-based applications can make use of the servlet's existing authentication mechanism from *HttpServletRequest* [http://java.sun.com/javaee/6/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/6/docs/api/javax/servlet/http/
HttpServletRequest.html] request = ...;
Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] session =
  repository.login(new  ServletCredentials  [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/
org/modeshape/jcr/api/ServletCredentials.html](request));
```

The *ServletCredentials* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/api/ServletCredentials.html] is just a JCR *Credentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] implementation that is used by ModeShape's *ServletProvider* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/security/ServletProvider.html] to delegate the authorization requests to *HttpServletRequest* [http://java.sun.com/javaee/6/docs/api/javax/servlet/http/HttpServletRequest.html]'s "hasRole" method. The *ServletCredentials* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/api/ServletCredentials.html] class is in the small "modeshape-jcr-api" module, so feel free to use this class in your servlet-based applications.

7.4.3. 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.

Once ModeShape is configured properly, getting anonymous JCR sessions requires no authentication. The easiest way to do this is to use the JCR API methods that do not have *Credentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/ Credentials.html] parameters. For example, this gets an anonymous session to the default workspace:

```
Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] session =
 repository.login();
```

while the following gets an anonymous session to the workspace with the supplied name:

```
String [http://java.sun.com/javase/6/docs/api/java/lang/String.html] workspaceName = ...;
Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] session =
 repository.login(workspaceName);
```

Per the JCR API, these are equivalent to passing a null *Credentials* [http:// www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] reference to "login" methods, so you can choose that approach as well. ModeShape provides the `AnonymousCredentials` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/ api/AnonymousCredentials.html] implementation that can be used if your application expects a to use non-null Credentials object:

```
Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] session =
 repository.login(new AnonymousCredentials [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/
org/modeshape/jcr/api/AnonymousCredentials.html]());
```

or

String [http://java.sun.com/javase/6/docs/api/java/lang/String.html] workspaceName = ...;
Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] session =
 repository.login(new `AnonymousCredentials` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/
org/modeshape/jcr/api/AnonymousCredentials.html](),workspaceName);

If you supply any other *Credentials* [http://www.day.com/maven/javax.jcr/javadocs/
jcr-2.0/javax/jcr/Credentials.html] implementation to the "login" methods, ModeShape
will not treat it as an anonymous login and *will authenticate* using JAAS
or, if the credentials is a `SecurityContextCredentials` [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/jcr/api/SecurityContextCredentials.html] instance,
its *SecurityContext* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/
SecurityContext.html] instance. In other words, there's no way to turn off authentication, but you
can use anonymous sessions.

7.4.4. 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. Most of the steps are outlined in
the *previous chapter*, but when logging in your application needs to use a compatible *Credentials*
[http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] implementation,
similar to the examples shown above.

7.5. 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/browse/
MODE#selectedTab=com.atlassian.jira.plugin.system.project:roadmap-panel].*

7.5.1. Required features

ModeShape 2.6.0.Beta2 implements all of the JCR 2.0 required features:

- repository acquisition

- authentication

- reading/navigating

- query

- export

- node type discovery

- permissions and capability checking

ModeShape supports several query languages, including the *JCR-SQL2* and *JCR-QOM* query languages defined in *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283], and the *XPath* and *JCR-SQL* languages defined in *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170] but deprecated in *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283]. ModeShape also supports a *fulltext search* language that is defined by 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.

7.5.2. Optional features

ModeShape 2.6.0.Beta2 implements most of the JCR 2.0 optional features:

- writing

- import

- observation

- workspace management

- versioning

- locking

- node type management

- same-name siblings

- orderable child nodes

- shareable nodes

The remaining optional features (access control management, lifecycle management, retention and hold, and transactions) may be introduced in future versions.

7.5.3. TCK Compatibility features

The ModeShape project has not yet been certified to be fully-compliant with the JCR 2.0 specification, but does plan on attaining this certification in the very near future.

However, the ModeShape project also runs the JCR TCK unit tests from the reference implementation every night. These tests technically do not represent the official TCK, but are used within the TCK. Most of these unit tests are run in the `modeshape-jcr` module against the *in-memory repository* to ensure our JCR implementation behaves correctly, and the same tests

are run in the `modeshape-integration-tests` module against a variety of connectors to ensure they're implemented correctly. The `modeshape-jcr-tck` module runs *all* of these TCK unit tests, and currently there are only a handful of failures due to known issues (see the *JCR specification support* section for details).

ModeShape 2.6.0.Beta2 currently passes 1372 of the 1391 JCR TCK tests, where 17 of these 19 failures appear to be bugs in the TCK tests (see *JCR-2648* [https://issues.apache.org/jira/browse/JCR-2648], *JCR-2661* [https://issues.apache.org/jira/browse/JCR-2661], *JCR-2662* [https://issues.apache.org/jira/browse/JCR-2662], and *JCR-2663* [https://issues.apache.org/jira/browse/JCR-2663]). The remaining 2 failures are due to a known issue (see *MODE-760* [http://jira.jboss.org/browse/MODE-760]).

7.5.4. JCR Security

Although the *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] 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", "unlock_any", "create_workspace" and "delete_workspace"). The "register_type" and "register_namespace" permissions control 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). Finally, the "create_workspace" and "delete_workspace" permissions grant the user the ability to create workspaces and delete workspaces, respectively, using the corresponding methods on *Workspace* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/Workspace.html]. 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* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] passed into `Repository.login(...)` (or the *Subject* [http://java.sun.com/javase/6/docs/api/javax/security/auth/Subject.html] from the *AccessControlContext* [http://java.sun.com/javase/6/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 7.1. Role / Action Mapping

Action Name	readonly	readwrite	admin
read	Allows	Allows	Allows
add_node		Allows	Allows

Action Name	readonly	readwrite	admin
set_property		Allows	Allows
remove		Allows	Allows
register_namespace			Allows
register_type			Allows
unlock_any			Allows
create_workspace			Allows
delete_workspace			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 7.2. 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 in the future. A CONNECT role is already being used by the ModeShape REST Server to control whether users have access to the repository through that means.

7.5.5. Built-In Node Types

ModeShape supports all of the built-in node types described in the *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] specification. ModeShape also defines some custom node types in the mode namespace, but none of these node types (other than mode:resource) are intended to be used by developers integrating with ModeShape and may be changed or removed at any time.

7.5.6. Custom Node Type Registration

Although the *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283] 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, at runtime using the standard JCR API for managing node types, or at runtime by reading CND files or Jackrabbit XML files. These node type registration mechanisms are supported equally within ModeShape, although defining node types in standard CND files is recommended for portability.

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

7.5.6.1. Managing Node Types Using the JCR API

The JCR 2.0 API provides a mechanism for registering and unregistering node types. Registration is done by creating *NodeTypeTemplate* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/nodetype/NodeTypeTemplate.html] objects, *NodeDefinitionTemplate* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/nodetype/NodeDefinitionTemplate.html] objects (for child node definitions), and *PropertyDefinitionTemplate* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/nodetype/PropertyDefinitionTemplate.html] objects (for property definitions). Use the setter methods to set the various attributes, and then register the node type definition with the *NodeTypeManager* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/nodetype/NodeTypeManager.html]:

```
Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] session
 = ... ;
Workspace [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/Workspace.html] workspace = session.getWorkspace();
```

```
// Obtain the ModeShape-specific node type manager ...
NodeTypeManager     [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/nodetype/
NodeTypeManager.html] nodeTypeManager = workspace.getNodeTypeManager();

// Declare a mixin node type named "searchable" (with no namespace)
NodeTypeTemplate     [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/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://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/
nodetype/NodeDefinitionTemplate.html]                childNode                =
 nodeTypeManager.createNodeDefinitionTemplate();
childNode.setName("source");
childNode.setMandatory(true);
childNode.setRequiredPrimaryTypesNames(new String[] { "nt:file" });
childNode.setDefaultPrimaryTypeName("nt:file");
nodeType.getNodeDefinitionTemplates().add(childNode);

// Add a multi-valued STRING property named "keywords"
PropertyDefinitionTemplate          [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/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.

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, using the standard JCR 2.0 API:

```
String[] unusedNodeTypeNames = ...;
```

```
Session   [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]   session
 = ... ;
NodeTypeManager        [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/nodetype/
NodeTypeManager.html] nodeTypeManager = session.getWorkspace().getNodeTypeManager();
nodeTypeManager.unregisterNodeTypes(unusedNodeTypeNames);
```

This approach is often used to register custom node types within an application, when the application knows the node type definitions or retrieves these definitions from some persisted format (e.g., file, database, etc.). However, ModeShape provides some utilities if you want to programmatically register node types defined in certain file formats. We'll see in the next section how to use these.

7.5.6.2. Reading JCR CND files

Custom node types can be defined more succinctly through the CND file format defined by the JCR 2.0 specification. 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 mandatory
```

This definition could then be registered as part of the repository configuration (see the *previous chapter*). Or, you can also use a Session to programmatically register the node types in a CND file, but this requires ModeShape-specific class to read this file:

```
Session   [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]   session
 = ...
CndNodeTypeReader        [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/
CndNodeTypeReader.html]    reader   =   new   CndNodeTypeReader   [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/jcr/CndNodeTypeReader.html](session);
reader.read(cndFile); // from file, file system path, classpath resource, URL, etc.

if (!reader.getProblems().isEmpty()) {
    for  (Problem  [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/common/
collection/Problem.html] problem : nodeTypeSource.getProblems()) {
   // report or record problem
 }
} else {
  boolean allowUpdate = ...
```

```
    NodeTypeManager [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/nodetype/
NodeTypeManager.html] nodeTypeManager = session.getWorkspace().getNodeTypeManager();
  nodeTypeManager.registerNodeTypes(reader.getNodeTypeDefinitions(), allowUpdate);
}
```

The *CndNodeTypeReader* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/
CndNodeTypeReader.html] class provides a number of `read(...)` methods that accept
File [http://java.sun.com/javase/6/docs/api/java/io/File.html]s, paths to files on the file
system, the names of resources on the classpath, , and *InputStream* [http://java.sun.com/
javase/6/docs/api/java/io/InputStream.html]s. And *CndNodeTypeReader* [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/jcr/CndNodeTypeReader.html] will also register any
namespace mappings defined in the CND file but not yet registered in the session or
workspace. For details, see the JavaDoc for *CndNodeTypeReader* [http://docs.jboss.org/
modeshape/2.6.0.Beta2/api/org/modeshape/jcr/CndNodeTypeReader.html]. If you have multiple
CND files, you can either call `read(...)` multiple times before registering (as long as the CND
files don't contain duplicate node type definitions), or you can simply create and use a new reader
for each CND file. The choice is yours.

7.5.6.3. Reading Jackrabbit XML Node Type Files

ModeShape also provides a class that reads the node types defined in a Jackrabbit XML format.
This is useful if you've been using Jackrabbit, have defined your custom node types in the
Jackrabbit-specific format, but want to switch to ModeShape and don't want to have to manually
convert your node types in the standard CND format. This class is used almost identically
to the *CndNodeTypeReader* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/
jcr/CndNodeTypeReader.html] class described above:

```
Session [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]   session
 = ...
JackrabbitXmlNodeTypeReader                    [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/jcr/JackrabbitXmlNodeTypeReader.html]                    reader                    =
  new JackrabbitXmlNodeTypeReader [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/
modeshape/jcr/JackrabbitXmlNodeTypeReader.html](session);
reader.read(cndFile); // from file, file system path, classpath resource, URL, etc.

if (!reader.getProblems().isEmpty()) {
    for (Problem [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/common/
collection/Problem.html] problem : nodeTypeSource.getProblems()) {
  // report or record problem
 }
} else {
 boolean allowUpdate = ...
    NodeTypeManager [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/nodetype/
NodeTypeManager.html] nodeTypeManager = session.getWorkspace().getNodeTypeManager();
```

```
   nodeTypeManager.registerNodeTypes(reader.getNodeTypeDefinitions(), allowUpdate);
}
```

7.6. 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 ModeShape to query and search your JCR data.

Querying and Searching using JCR

The JCR API defines a way to query a repository for content that meets user-defined criteria. The JCR 2.0 API actually makes it possible for implementations to support multiple query languages, and the specification requires support for two languages: *JCR-SQL2* and *JCR-QOM*. *JCR 1.0* [http://www.jcp.org/en/jsr/detail?id=170] defined two other languages (*XPath* and *JCR-SQL*), though these languages were deprecated in *JCR 2.0* [http://www.jcp.org/en/jsr/detail?id=283].

At this time, ModeShape supports all of these query languages, plus one search-engine-like language called "*search*" that is actually just the *full-text search expression grammar* used in the second parameter of the CONTAINS(...) function of the JCR-SQL2 language.

ModeShape handles all of these languages in nearly the same manner, the only difference being whether the query is represented as a string or build programmatically using the javax.jcr.query.qom part of the JCR API.

1. A language-independent representation, called the query model, is constructed by parsing the string representation of the query (using a language-specific parser) or the JCR-QOM objects created by the client.

2. The language-independent query model is used to create a canonical (relational) query plan.

3. The canonical query plan is then validated to ensure that all identifiers in the query are resolvable.

4. The canonical query plan is then optimized using a flexible rule-based optimizer. Optimizations include (but are not limited to): replace view references; unify handling of aliases; convert right outer joins into left outer joins; choose algorithms for each join; raise and lower criteria; push projection of columns as low in the plan as possible; duplicate criteria across identity joins; rewrite identity joins involving only columns that form keys; remove parts of the plan that (based upon the criteria) will return no rows; determination of the low-level "access" queries that will be submitted to the connector layer.

5. The optimized query plan is then executed, whereby each access query is pushed down to the connector and the results are then processed and combined to produce the desired result set.
Note that only the parsing step is dependent upon the query language. This means that all of the query languages are processed using the same, unified engine.

The rest of this chapter describes how your applications can use queries to search your repositories, and outlines the specifics of each of the four query languages available in ModeShape.

8.1. JCR Query API

With ModeShape, all query operations can be performed using only the JCR API interfaces. The first step is to obtain the *QueryManager* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/query/QueryManager.html] from your *Session* [http://www.day.com/

maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] instance. The QueryManager interface defines methods for creating *Query* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/query/Query.html] objects, executing queries, storing queries (not results) as *Node* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Node.html]s in the repository, and reconstituting queries that were stored on *Node* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Node.html]s. Thus, querying a repository generally follows this pattern:

```java
// 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 chapter 6 of the *JCR 2.0 specification* [http://www.jcp.org/en/jsr/detail?id=283].

8.2. 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.

ModeShape offers a bit more functionality in the "`jcr:contains(...)`" clauses than required by the specification. In particular, the second parameter specifies the search expression, and for these ModeShape accepts *full-text search language* expressions, including wildcard support.

> **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.

8.2.1. 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 8.1. 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]

8.2.2. 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 8.2. Specifying type constraints

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/%'

XPath	JCR-SQL2
	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.

8.2.3. 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 8.3. 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
`//element(*,my:type) [@prop1=xs:boolean('true')]`	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'

XPath	JCR-SQL2

`//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.

8.2.4. 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 8.4. Specifying path constraints

XPath	JCR-SQL2
`/jcr:root/a[1]/b[2]`	SELECT * FROM [nt:base] WHERE PATH([nt:base]) = '/a[1]/b[2]'

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, although the XPath expressions are easier to understand and significantly shorter.

Also, path constraints in XPath do not need to specify wildcards for the same-name-sibling (SNS) indexes, as XPath should naturally find all nodes regardless of the SNS index, unless the SNS

index is explicitly specified. In other words, any path segment that does not have an explicit SNS index (or an SNS index of '[%]' or '[_]') will match *all* SNS index values. However, any segments in the path expression that have an explicit numeric SNS index will require an exact match. Thus this path constraint:

/a/b/c[2]/d[%]/%/e[_]

will effectively be converted into

/a[%]/b[%]/c[2]/d[%]/%/e[_]

This behavior is very different than how JCR-SQL and JCR-SQL2 path constraints are handled, since these languages interpret a lack of a SNS index as equating to '[1]'. To achieve the XPath-like matching, a query written in JCR-SQL or JCR-SQL2 would need to explicitly include '[%]' in each path segment where an SNS index literal is not already specified.

8.2.5. 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 may be used by default). For example, the following table shows several XPath queries and how they map to JCR-SQL2 queries.

Table 8.5. 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 jcr:score()`	SELECT * FROM [nt:base] AS nodeSet1 ORDER BY SCORE(nodeSet1)
`//element(*,my:type) order by jcr:score(my:type)`	SELECT * FROM [my:type] AS nodeSet1

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

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.

8.2.6. 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.

8.3. 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.

The JCR 2.0 specification defines how nodes in a repository are mapped onto relational tables queryable through a SQL-like language, including JCR-SQL and *JCR-SQL2*. Basically, each node type is mapped as a relational view with a single column for each of the node type's (residual and non-residual) property definitions. Conceptually, each node in the repository then appears as a record inside the view corresponding to the node type for which `"Node.isNodeType(nodeTypeName)"` would return true.

Since each node likely returns true from this method for multiple node type (e.g., the primary node type, the mixin types, and all supertypes of the primary and mixin node types), all nodes will likely appear as records in multiple views. And since each view only exposes those properties defined by (or inherited by) the corresponding node type, a full picture of a node will likely require joining the views for multiple node types. This special kind of join, where the nodes have the same identity on each side of the join, is referred to as an identity join, and is handled very efficiently by ModeShape.

ModeShape includes support for the JCR-SQL language, and adds several extensions to make it even more powerful and useful:

- 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 ..."`.

- 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.

- Support for the `IN` and `NOT IN` clauses to more easily and concisely supply multiple of discrete static operands. For example, `"WHERE ... prop1 IN (3,5,7,10,11,50) ..."`.

- Support for the `BETWEEN` clause to more easily and concisely supply a range of discrete operands. For example, `"WHERE ... prop1 BETWEEN 3 EXCLUSIVE AND 10 ..."`.

- Support for (non-correlated) subqueries in the `WHERE` clause, wherever a *static operand* can be used. Subqueries can even be used within another subquery. All subqueries must return a single column, and each row's single value will be treated as a literal value. If the subquery is used in a clause that expects a single value (e.g., in a comparison), only the subquery's first row will be used. If the subquery is used in a clause that allows multiple values (e.g., `IN (...)`), then all of the subquery's rows will be used. For example, this query `"WHERE ... prop1 IN ( SELECT my:prop2 FROM my:type2 WHERE my:prop3 < '1000' ) AND ..."` will use the results of the subquery as the literal values in the `IN` clause.

The grammar for the JCR-SQL query language is actually a superset of that defined by the *JCR 1.0 specification* [http://www.jcp.org/en/jsr/detail?id=170], and as such the complete grammar is included here.

> ### Note
>
> The grammar is presented using the same EBNF nomenclature as used in the JCR 1.0 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.

8.3.1. Queries

```
QueryCommand ::= Query | SetQuery

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

Query ::= Select From [Where] [OrderBy] [Limit]

Select ::= 'SELECT' ('*' | Proplist )

From ::= 'FROM' NtList

Where ::= 'WHERE' WhereExp

OrderBy ::= 'ORDER BY' propname [Order] {',' propname [Order]}

Order ::= 'DESC' | 'ASC'

Proplist ::= propname {',' propname}

NtList ::= ntname {',' ntname}

WhereExp ::= propname Op value |
        propname 'IS' ['NOT'] 'NULL' |
        like |
        contains |
        whereexp ('AND'|'OR') whereexp |
        'NOT' whereexp |
        '(' whereexp ')' |
```

```
          joinpropname '=' joinpropname |
          between |
          propname ['NOT'] 'IN' '(' value {',' value } ')'


Op ::= '='|'>'|'<'|'>='|'<='|'<>'


joinpropname ::= quotedjoinpropname | unquotedjoinpropname
quotedjoinpropname ::= ''' unquotedjoinpropname '''
unquotedjoinpropname ::= ntname '.jcr:path'


propname ::= quotedpropname | unquotedpropname
quotedpropname ::= ''' unquotedpropname '''
unquotedpropname ::= /* A property name, possible a pseudo-property: jcr:score or jcr:path */


ntname ::= quotedntname | unquotedntname
quotedntname ::= ''' unquotedntname '''
unquotedntname ::= /* A node type name */


value ::= literal | subquery


literal ::= ''' literalvalue ''' | literalvalue
literalvalue ::= /* A property value (in standard string form) */


subquery ::= '(' QueryCommand ')' | QueryCommand


like ::= propname 'LIKE' likepattern [ escape ]
likepattern ::= ''' likechar { likepattern } '''
likechar ::= char | '%' | '_'


escape ::= 'ESCAPE' ''' likechar '''


char ::= /* Any character valid within the string representation of a value
          except for the characters % and _ themselves. These must be escaped */


contains ::= 'CONTAINS(' scope ',' searchexp ')'
scope ::= unquotedpropname | '.'
searchexp ::= ''' exp '''
exp ::= ['-']term {whitespace ['OR'] whitespace ['-']term}
term ::= word | '"' word {whitespace word} '"'
word ::= /* A string containing no whitespace */
whitespace ::= /* A string of only whitespace*/


between ::= propname ['NOT'] 'BETWEEN' lowerBound ['EXCLUSIVE']
                    'AND' upperBound ['EXCLUSIVE']
```

```
lowerBound ::= value
upperBound ::= value


Limit ::= 'LIMIT' count [ 'OFFSET' offset ]
count ::= /* Positive integer value */
offset ::= /* Non-negative integer value */
```

8.4. 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 JCR-SQL language, providing among other things far richer specifications of joins and criteria.

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*.

- Support for (non-correlated) subqueries in the `WHERE` clause, wherever a *static operand* can be used. Subqueries can even be used within another subquery. All subqueries must return a single column, and each row's single value will be treated as a literal value. If the subquery is used in a clause that expects a single value (e.g., in a comparison), only the subquery's first row will be used. If the subquery is used in a clause that allows multiple values (e.g., `IN (...)`), then all of the subquery's rows will be used. For example, this query "`WHERE ... [my:type]. [prop1] IN ( SELECT [my:prop2] FROM [my:type2] WHERE [my:prop3] < '1000' ) AND ...`" will use the results of the subquery as the literal values in the `IN` clause.

- Support for several pseudo-columns ("`jcr:path`", "`jcr:score`", "`jcr:name`", "`mode:localName`", and "`mode:depth`") that can be used in the `SELECT`, *equijoin*, and `WHERE` clauses. These pseudo-columns make it possible to return location-related and score information within the *QueryResult* [http://www.day.com/maven/javax.jcr/javadocs/ jcr-2.0/javax/jcr/query/QueryResult.html]'s rows. They also make queries look more like SQL, and thus may be more friendly and easier to use in existing SQL-aware client applications. See the *detailed description* for more information.

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.

8.4.1. 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]

8.4.2. Sources

Source ::= Selector | Join

Selector ::= nodeTypeName ['AS' selectorName]

nodeTypeName ::= Name

8.4.3. 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

8.4.4. Equi-Join Conditions

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

selector1Name ::= selectorName
selector2Name ::= selectorName
property1Name ::= propertyName
property2Name ::= propertyName

8.4.5. Same-Node Join Conditions

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

selector2Path ::= Path

8.4.6. Child-Node Join Conditions

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

childSelectorName ::= selectorName

```
parentSelectorName ::= selectorName
```

8.4.7. Descendant-Node Join Conditions

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

8.4.8. Constraints

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

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

8.4.9. And Constraints

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

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

8.4.10. Or Constraints

Or ::= constraint1 'OR' constraint2

8.4.11. Not Constraints

Not ::= 'NOT' Constraint

8.4.12. Comparison Constraints

Comparison ::= DynamicOperand Operator StaticOperand

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

8.4.13. Between Constraints

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

lowerBound ::= StaticOperand
upperBound ::= StaticOperand

8.4.14. Property Existence Constraints

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

8.4.15. Set Constraints

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

8.4.16. 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 */
```

8.4.17. 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 */
```

8.4.18. 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 */
```

8.4.19. 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 */
```

8.4.20. 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) */

8.4.21. Static Operands

StaticOperand ::= Literal | BindVariableValue | Subquery

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 */

8.4.22. 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. */

8.4.23. Subqueries

Subquery ::= '(' QueryCommand ')' |
          QueryCommand

8.4.24. 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. */

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
```

8.4.25. Ordering

```
orderings ::= Ordering {',' Ordering}


Ordering ::= DynamicOperand [Order]


Order ::= 'ASC' | 'DESC'
```

8.4.26. 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
```

8.4.27. Limit and Offset

```
Limit ::= 'LIMIT' count [ 'OFFSET' offset ]
count ::= /* Positive integer value */
offset ::= /* Non-negative integer value */
```

8.4.28. Pseudo-columns

The design of the JCR-SQL2 query language makes fairly heavy use of functions, including `SCORE()`, `NAME()`, and `LOCALNAME()`. ModeShape adds several more useful functions, including `PATH()` and `DEPTH()`, that follow the same patterns.

However, there are several disadvantages of these functions. First, they make the JCR-SQL2 language less "SQL-like", since SQL-92 and -99 don't define these kinds of functions. (There are aggregate functions, like `COUNT`, `SUM`, etc., but they are not terribly analogous.) This means that applications that use SQL and SQL-like query languages are less likely to be able to build and issue JCR-SQL2 queries.

A second disadvantage of these functions is that JCR-SQL2 does not allow them to be used within the `SELECT` clause. As a result, the location-related and score information cannot be included as columns of values in the *QueryResult* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/query/QueryResult.html] rows. Instead, a client can *only* access this information by obtaining the *Node* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Node.html] object(s) for each row. Relying upon both the result set and additional Java objects makes it difficult to use.

For example, ModeShape's JDBC driver is designed to enable JDBC-aware applications to query repository content using JCR-SQL2 queries. The standard JDBC API cannot expose the *Node* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Node.html] objects, so the only way to return the path-related and score information is through additional columns in the result. While such columns could "magically" appear in the result set, doing this is not compatible with JDBC applications that dynamically build queries based upon database metadata. Such applications require the columns to be properly described in database metadata, and the columns need to be used within queries.

ModeShape attempts to solve these issues by directly supporting a number of "pseudo-columns" within JCR-SQL2 queries, wherever columns can be used. These "pseudo-columns" include:

- *jcr:score* is a column of type DOUBLE that represents the full-text search score of the node, which is a measure of the node's relevance to the full-text search expression. ModeShape does compute the scores for all queries, though the score for rows in queries that do not include a full-text search criteria may not be reliable.

- *jcr:path* is a column of type PATH that represents the normalized path of a node, including same-name siblings. This is the same as what would be returned by the getPath() method of *Node* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Node.html]. Examples of paths include "/jcr:system" and "/foo/bar[3]".

- *jcr:name* is a column of type NAME that represents the node name in its namespace-qualified form using namespace prefixes and excluding same-name-sibling indexes. Examples of node names include "jcr:system", "jcr:content", "ex:UserData", and "bar".

- *mode:localName* is a column of type STRING that represents the local name of the node, which excludes the namespace prefix and same-name-sibling index. As an example, the local name of the "jcr:system" node is "system", while the local name of the "ex:UserData[3]" node is "UserData".

- *mode:depth* is a column of type LONG that represents the depth of a node, which corresponds exactly to the number of path segments within the path. For example, the depth of the root node is 0, whereas the depth of the "/jcr:system/jcr:nodeTypes" node is 2.

All of these pseudo-columns can be used in the SELECT clause of any JCR-SQL2 query, and their use defines whether such columns appear in the result set. In fact, all of these pseudo-columns will be included when "SELECT *" clauses in JCR-SQL2 queries are expanded by the query engine. This means that every node type (even mixin node types that have no properties and are essentially markers) are represented by a queryable table.

Like any other column, all of these pseudo-columns can be also be used in the WHERE clause of any JCR-SQL2 query, even if they are not included in the SELECT clause. They can be used anywhere that a regular column can be used, including within *constraints* and *dynamic operands*. ModeShape will automatically rewrite queries that use pseudo-columns in the dynamic operands to use the corresponding function, such as SCORE(), PATH(), NAME(), LOCALNAME(), and DEPTH(). Additionally, any *property existence constraint* using these pseudo-columns will always evaluate to 'true' (and will thus be removed by the optimizer).

The jcr:path pseudo-column may also be used on both sides of an *equijoin constraint* clause. For example:

```
... selector1.[jcr:path] = selector2.[jcr:path] ...
```

Equijoins of this form will be automatically rewritten by the optimizer to the following form:

```
... ISSAMENODE(selector1,selector2) ...
```

As with regular columns, the pseudo-columns must be qualified with the selector name if the query contains more than one selector.

> **Note**
>
> Note that the `jcr:path` and `jcr:score` pseudo-columns are consistent with the pseudo-columns of the same names used in *JCR-SQL* query language. However, unlike in JCR-SQL, in JCR-SQL2 these columns are not automatically included in the results unless explicitly included in the `SELECT` clause or implicitly included via "`SELECT *`"

8.4.29. Example JCR-SQL2 queries

One of the simplest JCR-SQL2 queries finds all nodes in the current workspace of the repository:

```
SELECT * FROM [nt:base]
```

This query will return a result set containing a single "`jcr:primaryType`" column, since the `nt:base` defines only one single-valued property called "`jcr:primaryType`". (The `jcr:mixinTypes` property is multi-valued, and as such the JCR 2.0 specification does not require returning these in query results.)

Queries can explicitly specify the columns that are to be returned in the results. The following query is semantically equivalent to the previous query, and produces identical results:

```
SELECT [jcr:primaryType] FROM [nt:base]
```

The following query will return the same rows as in the previous queries, but the SELECT clause includes two pseudo-columns containing the values computed from the nodes' locations:

```
SELECT [jcr:primaryType], [jcr:path], [mode:depth] FROM [nt:base]
```

In JCR-SQL2, a table representing a particular node type will have a column for each of the node type's property definitions, including those inherited from supertypes. For example, the `nt:file` node type, its `nt:hierarchyNode` supertype, and the `mix:created` mixin type are defined using the CND notation as follows:

```
[mix:created] mixin
  - jcr:created (date) protected
```

```
 - jcr:createdBy (string) protected

[nt:hierarchyNode] > mix:created abstract

[nt:file] > nt:hierarchyNode
 + jcr:content (nt:base) primary mandatory
```

Therefore, the table representing the `nt:file` node type will have two three columns: the `jcr:created` and `jcr:createdBy` columns inherited from the `mix:created` mixin node type (via the `nt:hierarchyNode` node type), and the `jcr:primaryType` column inherited from the `nt:base` node type, which is the implicit supertype of the `nt:hierarchyNode`. Thus, this query:

```
 SELECT * FROM [nt:file]
```

is equivalent to this query:

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy],
     [jcr:path], [jcr:name], [jcr:score], [mode:localName], [mode:depth]
FROM [nt:file]
```

Here is an example query that selects some of the available columns from the `nt:file` table and uses a constraint to ensure the resulting file nodes have names that end in '.txt':

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path] FROM [nt:file]
WHERE LOCALNAME() LIKE '%.txt'
```

Of course, we could instead using `mode:localName` pseudo-column instead of the `LOCALNAME()` function. Such a query is equivalent to the previous query and will produce the exact same results:

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path] FROM [nt:file]
WHERE [mode:localName] LIKE '%.txt'
```

Although this query looks much more like SQL, the use of the '[' and ']' characters to quote the identifiers is not typical of a SQL dialect. ModeShape actually supports the using double-quote characters and square braces interchangeably around identifiers (although they must match

around any single identifier). Again, this next query, which looks remarkably like any SQL-92 or -99 dialect, is functionally identical to the previous two queries:

```
SELECT "jcr:primaryType", "jcr:created", "jcr:createdBy", "jcr:path" FROM "nt:file"
WHERE "mode:localName" LIKE '%.txt'
```

In JCR-SQL2, a node will appear as a row in each table that corresponds to the node types defined by that node's primary type or mixin types, or any supertypes of these node types. In other words, a node will appear in the table corresponding to each node type for which `Node.isNodeType(...)` returns true.

For example, consider a node that has a primary type of `nt:file` but has a mixin of `mix:referenceable`. This node will appear as a result in the `nt:file`, `mix:referenceable`, `nt:hierarchy`, `mix:created`, and `nt:base`. The table for `nt:file` contains all of the columns in the `nt:hierarchyNode`, `mix:referenceable`, and `nt:base`. However, the `nt:file` table does *not* contain the `jcr:uuid` column, since the `nt:file` node type does *not* extend `mix:referenceable`. Thus, to obtain the UUID for our node, we need to perform an *identity* join. The next query shows how this is done to return all properties for `nt:file` nodes that are also `mix:referenceable`:

```
SELECT file.*, ref.* FROM [nt:file] AS file JOIN [mix:referenceable] AS ref
JOIN ON ISSAMENODE(file,ref)
```

The select clause would be expanded to the following query:

```
SELECT file.[jcr:primaryType], file.[jcr:created], file.[jcr:createdBy], ref.[jcr:uuid]
    file.[jcr:path], file.[jcr:name], file.[jcr:score], file.[mode:localName], file.[mode:depth]
FROM [nt:file] AS file JOIN [mix:referenceable] AS ref
JOIN ON ISSAMENODE(file,ref)
```

Of course, would could return even more information and make the query look very SQL-like by using pseudo-columns:

```
SELECT file."jcr:primaryType", file."jcr:created", file."jcr:createdBy", ref."jcr:uuid",
    file."jcr:path", file."jcr:name", file."mode:localName", file."mode:depth", file."jcr:score"
FROM "nt:file" AS file JOIN "mix:referenceable" AS ref
```

```
JOIN ON file."jcr:path" = ref."jcr:path"
```

These are examples of two-way inner joins, but ModeShape supports joining multiple tables together in a single query. ModeShape also supports a variety of joins, including INNER JOIN (or just JOIN), LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN, and CROSS JOIN.

ModeShape supports several other query features beyond JCR-SQL2. One of these is support for UNION, INTERSECT and EXCEPT. Here is an example of a union:

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path] FROM [nt:file]
UNION
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path] FROM [nt:folder]
```

ModeShape also supports using (non-correlated) subqueries within the WHERE clause, wherever a *static operand* can be used. Subqueries can even be used within another subquery. All subqueries, though, must return a single column, and each row's single value will be treated as a literal value. If the subquery is used in a clause that expects a single value (e.g., in a comparison), only the subquery's first row will be used.

Subqueries in ModeShape are a powerful and easy way to use more complex criteria that is a function of the content in the repository, without having to resort to multiple queries (take the results of one query and dynamically generate the criteria of another query).

Here's an example of a query that finds all nt:file nodes in the repository whose paths are referenced in the vdb:originalFile property of the vdb:virtualDatabase nodes. (This query also uses bind variables in the subquery.)

```
SELECT [jcr:primaryType], [jcr:created], [jcr:createdBy], [jcr:path] FROM [nt:file]
WHERE PATH() IN (
   SELECT [vdb:originalFile] FROM [vdb:virtualDatabase]
   WHERE [vdb:version] <= $maxVersion AND CONTAINS([vdb:description],'xml OR xml maybe')
)
```

Without subqueries, this query would need to be broken into two separate queries: the first would find all of the paths referenced by the vdb:virtualDatabase nodes matching the version and description criteria, followed by one (or more) subsequent queries to find the nt:file nodes with the paths expressed as literal values (or bind variables).

The examples shown in this section hopefully begin to show the power and flexibility of JCR-SQL2 and the ModeShape extensions.

8.5. 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, such that a full-text search query supplied by the user, *full-text-query*, is equivalent to executing this JCR-SQL2:

```
SELECT * FROM [nt:base] WHERE CONTAINS([nt:base],'full-text-query')
```

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.

By default, all terms are assumed to be positive terms, in the sense that the occurrence of the term will increase the rank of any nodes containing the value. However, it is 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 ('-').

Each term may also contain wildcards to specify the pattern to be matched (or negated). ModeShape supports two different sets of wildcards:

- '*' matches zero or more characters, and '?' matches any single character; and

- '%' matches zero or more characters, and '_' matches any single character.

The former are wildcards that are more commonly used in various systems (including older JCR repository implementations), while the latter are the wildcards used in LIKE expressions in both

JCR-SQL and JCR-SQL2. Both families are supported for convenience, and you can also mix and match and combine the various wildcards, such as `ta**bl_` and `ta__ble%*`. (Of course, placing multiple '*' or '%' characters next to each other offers no real benefit, as it is equivalent to a single '*' or '%'.)

If you want to use these characters literally in a term and do not want them to be treated as wildcards, they must be escaped by prefixing them with a '\' character. For example, this full text search expression:

```
table\* 'customer invoice\?'
```

will would rank higher those nodes with properties containing 'table*' (including the asterisk) and those containing the phrase "customer invoice?" (including the question mark). To use a literal backslash character, simply escape it as well.

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.

8.5.1. Full-text Search Language

```
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* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/query/QueryResult.html] always contains the "jcr:path" and "jcr:score" columns.

8.6. JCR Query Object Model (JCR-QOM) API

JCR 2.0 introduces a new API for programmatically constructing a query. This API allows the client to construct the lower-level objects for each part of the query, and is a great fit for applications that would otherwise generate fairly complicated query expressions. Using this API is a matter of getting the *QueryObjectModelFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/query/qom/QueryObjectModelFactory.html] from the session's *QueryManager* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/query/QueryManager.html], and using the factory to create the various components, starting with the lowest-level components. Then, these lower-level components can be passed to other factory methods to create the higher-level components, and so on, until finally the `createQuery(...)` method is called to return the *QueryObjectModel* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/query/qom/QueryObjectModel.html].

Here is a simple example that shows how this is done for the simple query "`SELECT * FROM [nt:unstructured] AS unstructNodes`":

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

// Create a query object model factory ...
QueryObjectModelFactory factory = queryManager.getQOMFactory();

// Create the FROM clause: a selector for the [nt:unstructured] nodes ...
Selector source = factory.selector("nt:unstructured","unstructNodes");

// Create the SELECT clause (we want all columns defined on the node type) ...
Column[] columns = null;
```

```java
// Create the WHERE clause (we have none for this query) ...
Constraint constraint = null;

// Define the orderings (we have none for this query)...
Ordering[] orderings = null;

// Create the query ...
QueryObjectModel query = factory.createQuery(source,constraint,orderings,columns);

// Execute the query and get the results ...
// (This is the same as before.)
javax.jcr.QueryResult result = query.execute();
```

From this point on, processing the results is the same as when using the *JCR Query API*:

```java
// 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();
```

Of course, most queries will create the columns, orderings, and constraints using the *QueryObjectModelFactory* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/query/qom/QueryObjectModelFactory.html], whereas the example above just assumes all of the columns, no orderings, and no constraints.

Accessing ModeShape Remotely

ModeShape provides a pair of ways to connect from remote clients: a WebDAV interface and a RESTful interface. This chapter details the capabilities of both as well as the configuration required to use each.

> **Note**
>
> Although the WebDAV and REST servers are treated separately here, many of the configuration parameters are the same. This is because both share a fair amount of common code and have been designed to be able to be deployed simultaneously on the same server or even within the same web archive.

> **Note**
>
> The WebDAV and REST servers described here exist for easy use, though they may need to be customized and WAR files reassembled to fit your particular application server and configuration. ModeShape's *JBoss AS kit* is one such customization, with a number of additional components built specifically for the JBoss Application Server environment.

9.1. The ModeShape WebDAV Server

ModeShape provides a WebDAV server interface to its JCR implementation to ease integration with client applications. The WebDAV server maps some of the content nodes (by default, nodes with a primary type of `nt:file`) to WebDAV resources and the other nodes to WebDAV folders. This allows any WebDAV client to navigate through the content repository to store files in a given location, as well as to create or delete nodes in the repository. The remainder of this section describes how to configure and deploy the WebDAV server.

9.1.1. Configuring the ModeShape WebDAV Server

The ModeShape WebDAV 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 WebDAV server along with an explanation of its parts.

```xml
<?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.spi.RepositoryProvider interface.  It is required
  by the ModeShapeJcrDeployer that controls the lifecycle for the ModeShape WebDAV server.
-->
<context-param>
  <param-name>org.modeshape.web.jcr.REPOSITORY_PROVIDER</param-name>
  <param-value>org.modeshape.web.jcr.spi.FactoryRepositoryProvider</param-value>
</context-param>
```

As noted above, this parameter informs the `ModeShapeJcrDeployer` [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/ModeShapeJcrDeployer.html] of the specific repository provider in use. Unless you are using the ModeShape WebDAV server to connect to a different JCR implementation, this should never change. The ModeShape REST server also uses the `ModeShapeJcrDeployer` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ web/jcr/ModeShapeJcrDeployer.html] to get access to the JCR repository, so the two servlets can be deployed in the same WAR.

Next we configure the ModeShape `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/ org/modeshape/jcr/JcrEngine.html] itself.

```
<!--
  This parameter, specific to the FactoryRepositoryProvider 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.JCR_URL</param-name>
  <param-value>file:/configRepository.xml</param-value>
</context-param>
```

If you are not familiar with the file format for a *JcrEngine* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] configuration file, you can build one programatically with the *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] class and call `save(...)` instead of `build()` to output the configuration file that equates to the configuration.

The *ContentMapper* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/webdav/ContentMapper.html] implementation can also be configured, but this is optional.

```
<!--
    This parameter provides the fully-qualified name of a class that implements
    the o.m.w.jcr.webdav.ContentMapper interface.  If no value is provided for this
    parameter, o.m.w.jcr.webdav.DefaultContentMapper will be used.
-->
<context-param>
      <param-name>org.modeshape.web.jcr.webdav.CONTENT_MAPPER_CLASS_NAME</param-name>
    <param-value>org.modeshape.web.jcr.webdav.DefaultContentMapper</param-value>
</context-param>
```

This class is used to prepare WebDAV responses from content nodes. The *DefaultContentMapper* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/webdav/DefaultContentMapper.html] implementation creates nodes with type `nt:folder` and `nt:file` for WebDAV requests to create WebDAV folders and files, respectively. Users can provide their own implementation that maps WebDAV content to other node content or structures.

This is followed by some additional WebDAV configuration that controls the mapping between JCR node types and WebDAV files and resources. These parameters are all specific to the *DefaultContentMapper* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/webdav/DefaultContentMapper.html] implementation. You can omit this section entirely to use the default values or if a custom *ContentMapper* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/webdav/ContentMapper.html] is used.

```
<!--
    Nodes with any of the primary node types in this comma-delimited list will be treated by the
    WebDAV implementation as content nodes.  The value below is the default value for this
    parameter.  That is, if this init parameter is omitted, the value below will be used by default.
-->
<context-param>
      <param-name>org.modeshape.web.jcr.webdav.CONTENT_PRIMARY_TYPE_NAMES</param-name>
```

```xml
    <param-value>nt:resource, mode:resource</param-value>
</context-param>

<!--
   Nodes with any of the primary node types in this comma-delimited list will be treated by the
   WebDAV implementation as resource (file) nodes.  The value below is the default value for this
   parameter.  That is, if this init parameter is omitted, the value below will be used by default.
-->
<context-param>
     <param-name>org.modeshape.web.jcr.webdav.RESOURCE_PRIMARY_TYPE_NAMES</param-name>
   <param-value>nt:file</param-value>
</context-param>

<!--
  Each folder created through the WebDAV servlet will be created as a node with the primary node
   type below.  The value below is the default value for this parameter.  That is, if this init
   parameter is omitted, the value below will be used by default.
-->
<context-param>
     <param-name>org.modeshape.web.jcr.webdav.NEW_FOLDER_PRIMARY_TYPE_NAME</param-name>
   <param-value>nt:folder</param-value>
</context-param>

<!--
    Each resource (file created through the WebDAV servlet will be created as a node with the
 primary
   node type below.  The value below is the default value for this parameter.  That is, if this init
   parameter is omitted, the value below will be used by default.
-->
<context-param>
   <param-name>
     org.modeshape.web.jcr.webdav.NEW_RESOURCE_PRIMARY_TYPE_NAME
   </param-name>
   <param-value>nt:file</param-value>
</context-param>

<!--
   Content created through the WebDAV servlet will be created as a node with the primary node
   type below.  The value below is the default value for this parameter.  That is, if this init
   parameter is omitted, the value below will be used by default.
-->
<context-param>
```

```
    <param-name>
        org.modeshape.web.jcr.webdav.NEW_CONTENT_PRIMARY_TYPE_NAME
    </param-name>
    <param-value>nt:resource</param-value>
</context-param>
```

In general, this part of the web configuration file should not be modified.

Next, the _RequestResolver_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/web/jcr/webdav/RequestResolver.html] must be configured. The _RequestResolver_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/ webdav/RequestResolver.html] converts the incoming URI into a repository name, workspace name, and path within the repository. ModeShape provides several implementations:

- _MultiRepositoryRequestResolver_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/web/jcr/webdav/MultiRepositoryRequestResolver.html] - supports multiple repositories and workspaces, by using a URI format with repository name and workspace name as the first two levels of the URI. This was added in ModeShape 2.3.0.Final, and is now the resolver that is configured by default.

- _SingleRepositoryRequestResolver_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/web/jcr/webdav/SingleRepositoryRequestResolver.html] - maps URIs onto a single repository and workspace that are configured in the web.xml. This is useful if you want to limit which repository and workspace is exposed via WebDAV.

- _DefaultRequestResolver_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/web/jcr/webdav/DefaultRequestResolver.html] - maps URIs onto a single repository and workspace that are configured in the web.xml. This used to be the default resolver, and is identical to SingleRepositoryRequestResolver. However, it is now deprecated and will be removed in a future version.

If none of these fit your needs, it is easy to develop a custom implementation of this interface.

To specify the resolver, set the org.modeshape.web.jcr.webdav.REQUEST_RESOLVER_CLASS_NAME property to the name of the implementation class. For example, here is how the _MultiRepositoryRequestResolver_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/web/jcr/webdav/MultiRepositoryRequestResolver.html] class is specified:

```
<!--
    This optional parameter provides the name of the o.m.w.j.webdav.RequestResolver
    implementation class.  The provided value must be the name of a class that
    implements the RequestResolver interface and has a public, no-arg constructor.
    If no value is provided, o.m.w.j.webdav.MultiRepositoryRequestResolver will be used.
-->
```

```
<context-param>
    <param-name>org.modeshape.web.jcr.webdav.REQUEST_RESOLVER_CLASS_NAME</param-name>
    <param-value>org.modeshape.web.jcr.webdav.MultiRepositoryRequestResolver</param-value>
</context-param>
```

Alternatively, if the *SingleRepositoryRequestResolver* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/webdav/SingleRepositoryRequestResolver.html] class is to be used, then two additional properties must define the repository name and workspace name:

```
<!--
    This optional parameter provides the name of the o.m.w.j.webdav.RequestResolver
    implementation class.  The provided value must be the name of a class that
    implements the RequestResolver interface and has a public, no-arg constructor.
-->
<context-param>
    <param-name>org.modeshape.web.jcr.webdav.REQUEST_RESOLVER_CLASS_NAME</param-name>
    <param-value>org.modeshape.web.jcr.webdav.SingleRepositoryRequestResolver</param-value>
</context-param>

<!--
    This parameter is required if (and only if) the SingleRequestResolver is used.
    It provides the name of the JCR repository that will be accessed.  An exception
    will be thrown if no value is provided for this parameter.
-->
<context-param>
    <param-name>

 org.modeshape.web.jcr.webdav.SINGLE_REPOSITORY_RESOLVER_REPOSITORY_NAME
    </param-name>
    <param-value>repository</param-value>
</context-param>

<!--
    This parameter is required if (and only if) the SingleRequestResolver is used.
    It provides the name of the JCR workspace that will be accessed.  An exception
    will be thrown if no value is provided for this parameter.
-->
```

```xml
<context-param>
  <param-name>

 org.modeshape.web.jcr.webdav.SINGLE_REPOSITORY_RESOLVER_WORKSPACE_NAME
  </param-name>
  <param-value>default</param-value>
</context-param>
```

ModeShape also provides the older *DefaultRequestResolver* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/webdav/DefaultRequestResolver.html] class is to be used, which is now deprecated. Please switch use the *SingleRepositoryRequestResolver* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/web/jcr/webdav/SingleRepositoryRequestResolver.html] or *MultiRepositoryRequestResolver* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/web/jcr/webdav/MultiRepositoryRequestResolver.html] classes. This class is provided for backward compatibility.

Once the *RequestResolver* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ web/jcr/webdav/RequestResolver.html] has been specified, then more brief boilerplate ensues defines additional configuration information:

```xml
<!-- Required parameter for ModeShape WebDAV - should not be modified -->
<listener>
  <listener-class>org.modeshape.web.jcr.ModeShapeJcrDeployer</listener-class>
</listener>

<!-- Required WebDAV servlet - should not be modified -->
<servlet>
  <servlet-name>WebDAV</servlet-name>
  <servlet-class>org.modeshape.web.jcr.webdav.ModeShapeWebdavServlet</servlet-class>

  <!--
      The webdav library requires this parameter to be present, but does not use it.
  -->
  <init-param>
    <param-name>rootpath</param-name>
    <param-value>.</param-value>
  </init-param>
</servlet>

<!-- Required parameter for ModeShape WebDAV - should not be modified -->
<servlet-mapping>
```

```xml
    <servlet-name>WebDAV</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Finally, security must be configured for the WebDAV server.

```xml
  <!--
          The ModeShape WebDAV implementation leverages the HTTP credentials to for
  authentication
      and authorization within the JCR repository.  Unless the repository provides for anonymous
      access, 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 WebDAV</display-name>
    <web-resource-collection>
      <web-resource-name>WebDAV</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 WebDAV 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.

9.1.2. Deploying the ModeShape WebDAV Server

Deploying the ModeShape WebDAV 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 WebDAV 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>2.0</version>
  <relativePath>../..</relativePath>
 </parent>
 <artifactId>modeshape-web-jcr-webdav-war</artifactId>
 <packaging>war</packaging>
 <name>ModeShape JCR WebDAV Servlet</name>
 <description>ModeShape servlet that provides WebDAV access to JCR items</description>
 <url>http://www.modeshape.org</url>
 <dependencies>
  <dependency>
   <groupId>org.modeshape</groupId>
   <artifactId>modeshape-web-jcr-webdav</artifactId>
   <version>${project.version}</version>
  </dependency>

  <dependency>
   <groupId>org.slf4j</groupId>
```

```
   <artifactId>slf4j-log4j12</artifactId>
   <version>1.5.8</version>
   <scope>runtime</scope>
  </dependency>
 </dependencies>
</project>
```

If you use this approach, make sure that web configuration file is in the `/src/main/webapp/WEB-INF` directory.

Of course, the JBoss WebDAV Server WAR can still be built if you are not using Maven. Simply construct a WAR with the following contents:

```
+ /WEB-INF
 + /classes
 | + configRepository.xml
 | + log4j.properties (Optional)
 + /lib
 | + aperture-1.1.0.Beta1.jar
 | + hamcrest-core-1.1.jar
 | + jakarta-regexp-1.4.jar
 | + jcr-2.0.jar
 | + joda-time-1.6.jar
 | + junit-dep-4.4.jar
 | + lucene-analyzers-3.0.2.jar
 | + lucene-core-3.0.2.jar
 | + lucene-regex-3.0.2.jar
 | + lucene-snowball-3.0.2.jar
 | + lucene-misc-3.0.2.jar
 | + modeshape-cnd-2.6.0.Beta2.jar
 | + modeshape-common-2.6.0.Beta2.jar
 | + modeshape-graph-2.6.0.Beta2.jar
 | + modeshape-jcr-2.6.0.Beta2.jar
 | + modeshape-jcr-api-2.6.0.Beta2.jar
 | + modeshape-mimetype-detector-aperture-2.6.0.Beta2.jar
 | + modeshape-repository-2.6.0.Beta2.jar
 | + modeshape-search-lucene-2.6.0.Beta2.jar
 | + modeshape-web-jcr-2.6.0.Beta2.jar
 | + modeshape-web-jcr-webdav-2.6.0.Beta2.jar
 | + rdf2go.api-4.6.2.jar
 | + slf4j-api-1.5.11.jar
```

```
| + slf4j-log4j12-1.5.8.jar
| + stax-api-1.0-2.jar
| + webdav-servlet-2.0.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.

> **Note**
>
> Your servlet container may already provide a logging system, and you may need to remove the "slf4j-log4j12-1.5.8.jar" and replace with the appropriate *SLF4J binding* [http://www.slf4j.org/manual.html] jar. Or, if your servlet container already uses SLF4J globally, you may want to remove all of the "slf4j*.jar" files.

This WAR can be deployed into your servlet container.

9.2. The ModeShape REST Server

ModeShape 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 section describes how to configure and deploy the REST server.

9.2.1. 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.

> **Note**
>
> The *JBoss AS kit* by default will deploy the RESTful service at the "/modeshape-rest" location, which is more descriptive and better fits with the other deployed applications and services. To use these examples against this RESTful service, simply replace "/resources" with "/modeshape-rest" in each of the URLs.

Currently, only JSON-encoded responses are provided.

Table 9.1. 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	GET, POST, PUT, DELETE
/resources/{repositoryName}/{workspaceName}/query	Executes the query in the request body	POST

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" }
   "metadata" : {
     "jcr.specification.name" : "Content Repository for Java Technology API",
       "jcr.specification.version" : "2.0",
```

```
        "jcr.repository.name" : "ModeShape JCR Repository",
        "jcr.repository.vendor.url" : "http://www.modeshape.org",
        "jcr.repository.version" : "2.6.0.FINAL",
        "option.versioning.supported" : "true",

        ... etc. ...

      }
    }
   }
  }
}
```

The actual response wouldn't be pretty-printed like the example, but the format would be the same. The name of the repository ("repository" URL-encoded) is mapped to a repository object that contains a name (the redundant "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.

Also, the "metadata" section is included only in responses from RESTful services starting with the version 2.5.0.Final release, and contains the *JCR descriptors* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html#getDescriptor(java.lang.String)] keys and values, where each value will either be a string or, if there are multiple values for the descriptor, an array of strings. Note not all the descriptors are shown in the above example.

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* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] 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",
    "query":"/resources/modeshape%3arepository/default/query"
  },
 }
 }
}
```

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 and the "query" resource builds the full URI needed to execute queries.

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 JSON-encoded response from the request will contain the node that you just posted, including any autocreated properties and child nodes.

If you do not need this information, add `mode:includeNode=false` as a query parameter to your URL.

POST http://www.example.com/resources/modeshape%3arepository/default/items/newNode? mode:includeNode=false

```
{
 "properties": {
  "jcr:primaryType": "nt:unstructured",
  "jcr:mixinTypes": "mix:referenceable",
  "someProperty": "foo"
 },
 "children": {
  "newChildNode": {
   "properties": {"jcr:primaryType": "nt:unstructured"}
  }
 }
}
```

This will instruct the REST server to only return the path of the newly-created node in the response.

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/some/existing/ node/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/some/existing/
node

{
 "someProperty": "foobar",
 "someOtherProperty": "newValue"
}
```

The JSON request can even contain a properties container:

```
PUT  http://www.example.com/resources/modeshape%3arepository/default/items/some/existing/
node

{
 "properties": {
  "someProperty": "foobar",
  "someOtherProperty": "newValue"
 }
}
```

A subgraph can be updated all at once using a PUT against a URI of the top node in the subgraph. Note that in this case, very node in the subgraph must be provided in the JSON request (any node not in the request will be removed). This method will attempt to set all of the properties to the new value(s) as specified in the JSON request, plus any descendant node in the JSON request that doesn't reflect an existing node will be created while any existing node not reflected in the JSON request will be removed. (Any specifications of "jcr:primaryType" are ignored if the node already exists.) In other words, the request only needs to contain the properties that are changed. Of course, if a node is being added, all of its properties need to be included in the request.

Here is an example:

```
PUT  http://www.example.com/resources/modeshape%3arepository/default/items/some/existing/
node
```

```
{
 "properties": {
  "jcr:primaryType": "nt:unstructured",
  "jcr:mixinTypes": "mix:referenceable",
  "someProperty": "foo"
 },
 "children": {
  "childNode": {
   "properties": {"jcr:primaryType": "nt:unstructured"}
  }
 }
}
```

This will update the existing node at "/some/existing/node" with the specified properties, and ensure that it contains one child node named "childNode". Note that the body of this request is identical in structure to that of the POST requests.

Queries can be executed through the REST interface by POSTing to the query URI with the query statement in the body of the request. The query language *must* be specified by setting the appropriate MIME type.

Table 9.2. Query Content Types for the ModeShape REST Server

Query Language	Content Type
XPath	application/jcr+xpath
JCR-SQL	application/jcr+sql
JCR-SQL2	application/jcr+sql2
Full Text Search	application/jcr+search

If no content type is specified or the content type for the request is not one of the content types listed above, the request will generate a response code of 400 (BAD REQUEST).

All queries for a given workspace are posted to the same URI and the request body is not JSON-encoded.

POST http://www.example.com/resources/modeshape%3arepository/default/query

/a/b/c/d[@foo='bar']

Assuming that the request above was POSTed with a content type of `application/jcr+xpath`, a response would be generated that consisted of a JSON object that contained a property named

"rows". The "rows" property would contain an array of rows with each element being a JSON object that represented one row in the query result set.

```
{
 "types": {
  "someProperty": "STRING",
  "someOtherProperty": "BOOLEAN",
  "jcr:path": "STRING",
  "jcr:score": "DECIMAL"
 },
 "rows": {
  {
   "someProperty": "foobar",
   "someOtherProperty": "true",
   "jcr:path" : "/a/b/c/d",
   "jcr:score" : 0.9327
  },
  {
   "someProperty": "localValue",
   "someOtherProperty": "false",
   "jcr:path" : "/a/b/c/d[2]",
   "jcr:score" : 0.8143
  }
 }
}
```

If ModeShape is used as the underlying JCR implementation, the JSON object in the response will also contain a "types" property. The value of the "types" property is a JSON object that maps column names to their JCR type.

9.2.1.1. 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).

9.2.2. 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
<?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*.

```xml
<!--
   This parameter provides the fully-qualified name of a class that implements
   the o.m.web.jcr.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.REPOSITORY_PROVIDER</param-name>
  <param-value>org.modeshape.web.jcr.spi.FactoryRepositoryProvider</param-value>
</context-param>
```

As noted above, this parameter informs the `ModeShapeJcrDeployer` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/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/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] itself.

```
<!--
  This parameter, specific to the FactoryRepositoryProvider 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.JCR_URL</param-name>
  <param-value>file:/configRepository.xml</param-value>
</context-param>
```

If you are not familiar with the file format for a `JcrEngine` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrEngine.html] configuration file, you can build one programatically with the `JcrConfiguration` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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>
```

```xml
    <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.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.

```xml
 <!--
   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>
```

```xml
    <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.

9.2.3. Deploying the ModeShape REST Server

Just as with the ModeShape WebDAV 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.

```xml
<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>2.0</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>2.0</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>1.2.1.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
 | + 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
 | + jcl-over-slf4j-1.5.8.jar
 | + jcr-2.0.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-2.6.0.Beta2.jar
 | + modeshape-common-2.6.0.Beta2.jar
 | + modeshape-graph-2.6.0.Beta2.jar
 | + modeshape-jcr-2.6.0.Beta2.jar
 | + modeshape-jcr-api-2.6.0.Beta2.jar
 | + modeshape-repository-2.6.0.Beta2.jar
 | + modeshape-search-lucene-2.6.0.Beta2.jar
 | + modeshape-web-jcr-2.6.0.Beta2.jar
 | + modeshape-web-jcr-rest-2.6.0.Beta2.jar
 | + resteasy-jaxb-provider-1.2.1.GA.jar
 | + resteasy-jaxrs-1.2.1.GA.jar
 | + resteasy-jettison-provider-1.2.1.GA.jar
 | + scannotation-1.0.2.jar
 | + sjsxp-1.0.1.jar
 | + slf4j-api-1.5.11.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.

> **Note**
>
> Your servlet container may already provide a logging system, and you may need to remove the "slf4j-log4j12-1.5.8.jar" and replace with the appropriate *SLF4J binding* [http://www.slf4j.org/manual.html] jar. Or, if your servlet container already uses SLF4J globally, you may want to remove all of the "slf4j*.jar" files.

This WAR can be deployed into your servlet container.

9.2.4. 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.

9.3. Repository Providers

Both the ModeShape REST server and the ModeShape WebDAV 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/2.6.0.Beta2/api/org/modeshape/web/jcr/spi/RepositoryProvider.html] interface that connects to the other repository.

The *RepositoryProvider* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/spi/RepositoryProvider.html] only has a few methods that must be implemented. When the *ModeShapeJcrDeployer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/ModeShapeJcrDeployer.html] starts up, it will dynamically load the *RepositoryProvider* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/web/jcr/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/2.6.0.Beta2/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 repository.

```
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_ [http://www.day.com/maven/javax.jcr/ javadocs/jcr-2.0/javax/jcr/Session.html] for the user from the current request in a given repository and workspace. The provider can use the _HttpServletRequest_ [http://java.sun.com/javaee/6/ 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.

9.4. Summary

This chapter has described two ways to access a ModeShape JCR repository remotely through HTTP-based protocols. In the *next chapter*, the different repository connectors will be described so that you can start to use ModeShape to store new data, connect to existing data through JCR, or both.

Part III. 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/2.6.0.Beta2/api/org/ modeshape/graph/connector/inmemory/InMemoryRepositorySource.html] class provides a number of JavaBean properties that control its behavior:

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 is an empty string (or null) and implies that this source does not define a specific duration for caching information provided by this repository source.

defaultWorkspaceName

> Optional property that defines the name for the workspace that will be used in cases when clients do not explicitly specify the workspace name. If not specified, "default" will be used.

jndiName

> Optional property that, if used, specifies the name in JNDI where an *InMemoryRepository* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/connector/inmemory/InMemoryRepository.html] instance can be found. This is an advanced property that is infrequently used.

name

> Required property that specifies the name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/ RepositoryConnection.html] by name.

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 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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/ jcr/JcrConfiguration.html] instance with a repository source that uses the

InMemoryRepositorySource [http://docs.jboss.org/modeshape/2.6.0.Beta2/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("predefinedWorkspaceNames", new String[] { "staging", "dev"})
    .setProperty("defaultWorkspaceName", workspaceName);
```

Another way to configure the in-memory connector is to create *JcrConfiguration* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/ graph/connector/inmemory/InMemoryRepositorySource.html] class. For example a file named configRepository.xml can be created with these contents:

```xml
<?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:predefinedWorkspaceNames>staging</mode:predefinedWorkspaceNames>
        <mode:predefinedWorkspaceNames>dev</mode:predefinedWorkspaceNames>
      </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");
```

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. *If the directory for a workspace does not exist, this connector will attempt to create the directory (and any missing parent directories).*

By default, this connector is not capable of storing extra properties other than those defined on the `nt:file`, `nt:folder` and `nt:resource` node types. This is because such properties cannot be represented natively on the file system. When the connector is asked to store such properties, the default behavior is to log warnings and then to ignore these extra properties. Obviously this is probably not sufficient for production (unless only the standard properties are to be used). To explicitly turn on this behavior, set the "extraPropertiesBehavior" to "log".

However, the connector can be configured differently. If the "extraPropertiesBehavior" is set to "ignore", then these extra properties will simply be silently ignored and lost: none will be stored, none will be loaded, and no warnings will be logged. If the "extraPropertiesBehavior" is set to "error", the connector will throw an exception if any extra properties are used.

Perhaps the best setting for general use, however, is to set the "extraPropertiesBehavior" to "store". In this mode, any extra properties are written to files on the file system that are adjacent to the actual file or folder. For example, given a "nt:folder" node that represents the "folder1" directory, all extra properties will be stored in a text file named "folder1.modeshape" in the same parent directory as the "folder1" directory. Similarly, given a "nt:file" node that represents the "file1" file on the file system, all extra properties will be stored in a text file named "file1.modeshape" located next to the "file1" file. Note that the "nt:resource" node for our "nt:file" node also is stored in the same location, so we can't use the "file1.modeshape" file (it's already used for the "nt:file" node), so the connector uses the "file1.content.modeshape" file instead.

> **Note**
>
> The "store" behavior may result in the creation of many "*.modeshape" files, and because of this the "store" behavior is not the default.

The *FileSystemSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/filesystem/FileSystemSource.html] class provides a number of JavaBean properties that control its behavior:

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".

customPropertiesFactory

Optional property that specifies the *CustomPropertiesFactory* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/connector/filesystem/ CustomPropertiesFactory] implementation that should be used to augment the default properties available on each node. This property can be set either from an object that implements the *CustomPropertiesFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/connector/filesystem/CustomPropertiesFactory] interface or from the name of a class with a public, no-argument constructor that implements the *CustomPropertiesFactory* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/connector/filesystem/CustomPropertiesFactory] interface. In the latter case, a the named class will be instantiated and used as the custom properties factory implementation.

This is really intended for cases where the "extraPropertiesBehavior" is not sufficient. Most often, however, the "extraPropertiesBehavior" setting will be sufficient and should be used instead of "customPropertiesFactory".

defaultWorkspaceName

Optional property that defines the name for the workspace that will be used in cases when clients do not explicitly specify the workspace name. If not specified, "default" will be used.

extraPropertiesBehavior

Optional setting that specifies how to handle the extra properties on "nt:file", "nt:folder", and "nt:resource" nodes that cannot be represented on the native files themselves. Set this to "log" if warnings are to be sent to the log (the default), or "error" if setting such properties should cause an error, or "store" if they should be stored in ancillary files next to the files and folders, or "ignore" if they should be silently ignored. The "log" value will be used by default or an invalid value is specified.

This setting will be ignored if a "customPropertiesFactory" class name is specified.

exclusionPattern

Optional property that specifies a regular expression that is used to determine which files and folders in the underlying file system are exposed through this connector. Files and folders with a name that matches the provided regular expression will *not* be exposed by this source. Setting this property to null has the effect of removing the exclusion pattern.

This may be combined with an "inclusionPattern", in which a file or folder will be exposed by this connector only when it satisfies the "inclusionPattern" and does not satisfy the "exclusionPattern". Also, the "inclusionPattern" and "exclusionPattern" cannot be used with "filenameFilter", since the latter will always override the patterns.

inclusionPattern

Optional property that specifies a regular expression that is used to determine which files and folders in the underlying file system are exposed through this connector. Files and folders with

a name that matches the provided regular expression will be exposed by this source. Setting this property to `null` has the effect of removing the inclusion pattern.

This may be combined with an "exclusionPattern", in which a file or folder will be exposed by this connector only when it satisfies the "inclusionPattern" and does not satisfy the "exclusionPattern". Also, the "inclusionPattern" and "exclusionPattern" cannot be used with "filenameFilter", since the latter will always override the patterns.

filenameFilter

Optional property that specifies the name of the *FilenameFilter* [http://java.sun.com/ javase/6/docs/api/java/io/FilenameFilter.html] implementation class that used to determine which files and folders in the underlying file system are exposed through this connector. Only files and folders that the filter accepts will be accessible through this source. The *FilenameFilter* [http://java.sun.com/javase/6/docs/api/java/io/ FilenameFilter.html] implementation class must have a public, no-argument constructor. Use this when the "inclusionPattern" and "exclusionPattern" values would be too complicated or are not able to represent the logic.

Note that the "filenameFilter", "exclusionPattern", and "inclusionPattern" properties are somewhat mutually exclusive. If a "filenameFilter" is specified, then "exclusionPattern" and "inclusionPattern" are both ignored. Setting this property to an empty value (or null) has the effect of clearing the filter.

name

Required property that specifies the name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/ RepositoryConnection.html] by name.

nodeCachePolicy

Optional property that, if used, defines the cache policy to use for caching nodes within the connector.

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.

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.

retryLimit

Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.

temporaryStoragePath

> Optional property that specifies a location for the file system connector's temporary storage. When writing file content, this connector first writes the content to a file in the temporary storage area. After that write succeeds in full, the temporary file is moved to its final location in the workspace. This extra step is taken so that an error or failure while writing the file does not cause corruption in the existing target file.
>
> This path must be set to a path on the same file system specified by the `workspaceRootPath` property. Otherwise, the temporary storage area will be located on a different file system than where the file will ultimately be written, so the rename operation cannot be used and a separate file copy must occur, increasing the risk of data loss if a network failure or hardware problem occurs.

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 as a relative path from the current working directory of this virtual machine (as defined by `new File(".")`.
>
> As an example, if `workspaceRootPath` is set to a non-null value, then for a workspace named `"default/foo"` the source will use `new File(workspaceRootPath, "default/foo")` as the source directory for the workspace content. If `workspaceRootPath` is not set (or set to an empty string or null value), then the source will use `new File(".", "default/foo")` as the source directory for the workspace content.

One way to configure the file system connector is to create *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the *FileSystemSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/connector/filesystem/FileSystemSource.html] class. For example a file named configRepository.xml can be created with these contents:

```xml
<?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:creatingWorkspacesAllowed="false"
      mode:rootNodeUuid="fd129c12-81a8-42ed-aa4b-820dba49e6f0"
      mode:updatesAllowed="true" >
      <mode:predefinedWorkspaceNames>staging</mode:predefinedWorkspaceNames>
      <mode:predefinedWorkspaceNames>dev</mode:predefinedWorkspaceNames>
      <!--
      If desired, specify a cache policy that caches items in memory for 5 minutes (300 s).
      This fragment can be left out if the connector should not cache any content.
      -->
      <mode:cachePolicy jcr:name="nodeCachePolicy"
       mode:classname="org.modeshape.graph.connector.base.cache.InMemoryNodeCache$PathCachePolicy"
```

```
        mode:timeToLive="300" />
    </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/2.6.0.Beta2/api/org/modeshape/connector/ store/jpa/JpaSource.html] class provides a number of JavaBean properties that control its behavior:

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/javase/6/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/javase/6/docs/api/javax/persistence/ EntityManagerFactory.html] is closed.
>
> - `"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 the default value because it is the least intrusive and safest option, since it will verify the database's schema matches what the connector expects.
>
> - `"disable"` - Does nothing and assumes that the database is already properly configured. This should be the setting used in production, as it is a best-practice that DB administrators explicitly configure/upgrade production database schemas (using scripts).

cacheConcurrencyStrategy
> Optional property that specifies the cache concurrency strategy to use. When Hibernate, ModeShape's default JPA provider, is used, this value should be one of `"read-only"`, `"read-write"` (the default), `"nonstrict-read-write"`, or `"transactional"`.

cacheProviderClassName

Optional property that specifies the class name of the cache provider. The default value of an empty string (or null) indicates that no caching should occur. Valid values for this property are JPA implementation-dependent. When using Hibernate, (ModeShape's default JPA provider), this value is used to set the "`hibernate.cache.provider_class`" property.

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 optional 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

The dialect of the database, which must match one of the *Hibernate dialect names* [http://docs.jboss.org/hibernate/core/3.5/reference/en-US/html_single/#configuration-optional-dialects], and must correspond to the type of driver being used. If not provided, the dialect will be auto-discovered by Hibernate. Because Hibernate does a good job of auto-determining the dialect, it is recommended that you set this only if auto-discovery fails for your database.

However, it is recommended that MySQL users always set this value, as Hibernate's auto-discovery of the dialect does not work for many MySQL installations. For example, the most common MySQL installation is MySQL 5.x with InnoDB, which requires a value of "`org.hibernate.dialect.MySQLInnoDBDialect`" for the "dialect" property.

dataSourceJndiName

The JNDI name of the JDBC DataSource instance that should be used. If not specified, the other driver properties must be set.

driverClassloaderName

Optional property that defines the name of the ModeShape 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, or if the driver is on the application's classpath.

driverClassName

The name of the JDBC driver class. This is not required if the DataSource is found in JNDI, but is required otherwise.

idleTimeInSecondsBeforeTestingConnections

Optional property that specifies 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).

isolationLevel

Optional property that, if used, denotes which of the `java.sql.Connection#TRANSACTION_*` constants should be used to control the transaction isolation level. Valid values are: "`TRANSACTION_READ_COMMITTED`", "`TRANSACTION_READ_UNCOMMITTED`", "`TRANSACTION_REPEATABLE_READ`", "`TRANSACTION_SERIALIZABLE`", and "`TRANSACTION_NONE`". When this property is not used, the default isolation level is set to whichever isolation level was previously set on the connection. Note that not all JDBC drivers support all isolation levels.

largeValueSizeInBytes

An advanced optional 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 a space and performance optimization that stores each unique large value only once. The default value is "1024" bytes, or 1 kilobyte.

maximumConnectionsInPool

Optional property that specifies the maximum number of connections that may be in the connection pool. The default is "5".

maximumConnectionIdleTimeInSeconds

Optional property that specifies 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

Optional property that specifies the maximum number of statements that should be cached. Statement caching can be disabled by setting to "0". The default is "100".

minimumConnectionsInPool

Optional property that specifies 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 value is "`Simple`", which is also the default value.

name

Required property that specifies the name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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

Optional property that defines 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 or used 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.

retryLimit

Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

schemaName

Optional property that, if set, specifies the name of the schema in which this repository source will read and write data. If no schema name is specified, then data will be read from the default schema associated with the database connection.

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 or used 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 or used if the DataSource is found in JNDI.

One way to configure the JPA connector is to create *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the *JpaSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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("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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/connector/store/jpa/JpaSource.html] class. For example a file named configRepository.xml can be created with these contents:

```xml
<?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.connector.store.jpa.JpaSource"
          mode:description="The database store for our content"
          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 a zip in utils/modeshape-jpa-ddl-gen/target/distribution when the Maven assembly profile -Passembly is run. Unzip the contents and run the ddl-gen script with the following syntax:

```
ddl-gen.sh(.bat) -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.

> **Note**
>
> It is strongly recommended that production users of ModeShape utilize this tool to generate the DDL for production and test databases. After this tool is used, the `autoGenerateSchema` property on the *JpaSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/store/jpa/JpaSource.html] should be set to "disable". This will prevent the schema from being dropped and recreated (or needlessly re-validated) each time that a ModeShape instance starts.

12.1. 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.

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.

- 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.

In the Simple model, subgraph queries are used to efficiently process a number of different requests, including *ReadBranchRequest* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/request/ReadBranchRequest.html] and *DeleteBranchRequest* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/request/ReadBranchRequest.html] need to know the complete subgraph.

> ### Warning
>
> Most DBMS systems have built-in sizes for LOB columns (although many allow DB admins to control the size), and thus do not require any special consideration. However, Apache Derby and IBM DB2 require explicit sizes on LOB columns. Currently, the ModeShape database schema has two such columns: the `MODE_SIMPLE_NODE.DATA` and `MODE_LARGE_VALUES.DATA` columns. The sizes of these columns are sufficiently large (1MB and 1GB, respectively), but attempts to store larger values than these sizes will fail.
>
> Therefore, **when using IBM DB2 and Apache Derby**, determine the appropriate size of these columns for your environment. For production systems, ModeShape recommends using the DDL generation utility (provided with ModeShape, see above) to generate the DDL for your particular DBMS, and its very easy to adjust that file to specify alternative sizes for the two columns. Alternatively, database administrators can alter the two tables by increasing the size of these columns.
>
> Other databases do not seem to be affected by this issue.

JCR Connector

This connector enables ModeShape to access and store content in another separate JCR repository instance. With it, ModeShape can integrate with other JCR implementations and even federate multiple JCR repositories into a single unified repository.

> **i** **Note**
>
> This connector is currently a technical preview, and we're seeking feedback and assistance in identifying bugs and specifying the required functionality.

The connector is designed to find the external JCR Repository instance in JNDI, though the location in JNDI can be easily configured with the "repositoryJndiName" property.

The connector also has several ways to authenticate and obtain a *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]. First of all, if a fixed *Credentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] are set via the properties, then the *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] will always be obtained using these credentials. If a username and password are set via the properties (and no *Credentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] is set), then the connector will create a `SimpleCredentials` [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/SimpleCredentials.html] and use it to obtain a *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]. In all other cases, the connector will use the `login` methods on *Repository* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Repository.html] that do not require a *Credentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html].

This means that when JAAS is used by both the ModeShape repository using the JCR connector and a ModeShape repository accessed by the connector, the connector will obtain a *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html] to the underlying repository using the same `Subject` [http://java.sun.com/javase/6/docs/api/javax/security/auth/Subject.html] used in the incoming request.

The connector automatically handles any differences in namespaces between the underlying JCR repository and the ModeShape system. However, at this time it is a requirement that node types used by the content in the underlying JCR repository must also be registered into the ModeShape repository (or repositories) using the connector.

The `JcrRepositorySource` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/jcr/JcrRepositorySource.html] class provides a number of JavaBean properties that control its behavior:

defaultCachePolicy
   Optional property that, if used, defines the default cache policy for this repository source. When not used, this source will not define a specific duration for caching information.

repositoryJndiName

Property that defines where in JNDI the connector can find the `javax.jcr.Repository` instance.

username

Optional property that defines the username that should be used when logging into the Repository to obtain a *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]. When used, the connector creates a *SimpleCredentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/SimpleCredentials.html] instance. Should not be used if the "credentials" properties is to be used.

password

Optional property that defines the password that should be used when logging into the Repository to obtain a *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]. When used, the connector creates a *SimpleCredentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/SimpleCredentials.html] instance. Should not be used if the "credentials" properties is to be used.

credentials

Optional property that, if used, defines *Credentials* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Credentials.html] instance that should be used when logging into the Repository to obtain a *Session* [http://www.day.com/maven/javax.jcr/javadocs/jcr-2.0/javax/jcr/Session.html]. Should be used only if the "username" and "password" properties are not set.

name

Required property that specifies the name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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 JCR connector is to create *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the *JcrRepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/jcr/JcrRepositorySource.html] class. For example:

```
JcrConfiguration config = ...
config.repositorySource("Repository Source")
```

```
.usingClass(FileSystemSource.class)
.setDescription("The repository for our content")
.setProperty("repositoryJndiName", "java:/myRepository")
.setProperty("username", "jsmith")
.setProperty("password", "secret");
```

Another way to configure the JCR connector is to create *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance and load an XML configuration file that contains a repository source that uses the *JcrRepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/jcr/JcrRepositorySource.html] class. For example, here's a file named configRepository.xml that represents the same configuration as above:

```xml
<?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 'Repository Source' repository is a JCR source.
    -->
    <mode:source jcr:name="Repository Source"
      mode:classname="org.modeshape.connector.jcr.JcrRepositorySource"
      mode:description="The repository for our content"
      mode:repositoryJndiName="java:/myRepository"
      mode:username="jsmith"
      mode:password="secret">
      <!--
      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$InMemoryCachePolicy"
        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");
```

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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/ modeshape/graph/connector/federation/FederatedRepositorySource.html] is typically configured with the name of another *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/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.
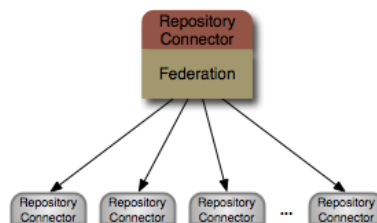
Figure 14.1. Federating multiple sources using the Federated Repository Connector

14.1. 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/2.6.0.Beta2/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.

14.2. 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.

14.3. Processing flow

This connector executes *Request* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/request/processor/RequestProcessor.html] implementations to do Step 1 and 3 (called `ForkRequestProcessor` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/federation/ForkRequestProcessor.html] and `JoinRequestProcessor` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/javase/6/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/request/Request.html]s that include the *Path* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html] within the request's `Location` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/ graph/request/Request.html] objects include the *Path* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/property/Path.html].

14.4. 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/2.6.0.Beta2/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/2.6.0.Beta2/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/ browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:summary-panel]. Be sure to include as much detail as possible about the scenario, the problem condition, and the desired behavior.

14.5. Configuration

The federated repository uses other *RepositorySource* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositorySource.html]s that are to be federated and a *RepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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>
```

> **ℹ️ 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.

14.6. 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/2.6.0.Beta2/api/org/
modeshape/graph/connector/federation/FederatedRepositorySource.html] class does have a few
JavaBean properties:

name

    Required property that specifies the name of the repository source, which is used by
    the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/

repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/connector/svn/SvnRepositorySource.html] class provides a number of JavaBean properties that control its behavior:

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, if used, specifies the name of the workspace to use when no workspace name is specified in an operation. If not specified, "`trunk`" is used.

> Each workspace name is treated as a path relative to the SVN repository being exposed. For example, given a repository root URL of "http://acme.com/repo/", a workspace name of "`trunk`" will map to "http://acme.com/repo/trunk".

name

> Required property that specifies the name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositoryConnection.html] by name.

nodeCachePolicy

> Optional property that, if used, defines the cache policy to use for caching nodes within the connector.

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. 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/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.

repositoryRootURL

Required property that should be set with the URL to the Subversion repository.

username

The username that should be used to establish a connection to the repository. This is not required if the URL represents an anonymous SVN repository address.

One way to configure the Subversion connector is to create *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the *SvnRepositorySource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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","tags" });
```

Another way to configure the Subversion connector is to create *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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.connector.svn.SVNRepositorySource"
        mode:description="The ModeShape SVN repository (anonymous access)"
        mode:repositoryRootUrl="http://anonsvn.jboss.org/repos/modeshape"
        mode:defaultWorkspaceName="trunk"
        mode:defaultWorkspaceName="default" >
   <mode:predefinedWorkspaceNames>tags</mode:predefinedWorkspaceNames>
   <mode:predefinedWorkspaceNames>trunk</mode:predefinedWorkspaceNames>
    <!--
    If desired, specify a cache policy that caches items in memory for 5 minutes (300 s).
    This fragment can be left out if the connector should not cache any content.
    -->
    <mode:cachePolicy **jcr:name**="nodeCachePolicy"
     mode:classname="org.modeshape.graph.connector.base.cache.InMemoryNodeCache
$PathCachePolicy"
        mode:timeToLive="300" />
  </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/2.6.0.Beta2/api/org/modeshape/connector/jbosscache/JBossCacheSource.html] class provides a number of JavaBean properties that control its behavior:

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 defines a cache, 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 is an empty string (or null) and implies that this source does not define a specific duration for caching information provided by this repository source.

defaultWorkspaceName

Optional property that defines the name for the workspace that will be used in cases when clients do not explicitly specify the workspace name. If not specified, "`default`" will be used.

name

Required property that specifies the name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

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.

retryLimit

> Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.

updatesAllowed

> Optional property that 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/2.6.0.Beta2/api/org/modeshape/ jcr/JcrConfiguration.html] instance with a repository source that uses the *`JBossCacheSource`* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/ connector/jbosscache/JBossCacheSource.html] class. For example a file named configRepository.xml can be created with these contents:

```xml
<?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</mode:predefinedWorkspaceNames>
      <mode:predefinedWorkspaceNames>dev</mode:predefinedWorkspaceNames>
    </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:

```java
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.

There are two connectors that can be used. The `InfinispanSource` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/infinispan/InfinispanSource.html] class, which was introduced in ModeShape 1.x, stores the content in an Infinispan data grid that is running, at least in part, within the same process. The `RemoteInfinispanSource` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/infinispan/RemoteInfinispanSource.html] class, which was introduced in ModeShape 2.3.0.Final, is capable of storing content in a remote Infinispan data grid (where Infinispan is running in remote processes). Because these two connectors communicate with Infinispan in different ways, they are configured differently.

The `InfinispanSource` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/infinispan/InfinispanSource.html] class provides a number of JavaBean properties that control its behavior:

cacheContainerJndiName

> 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.
>
> *Note that the "`cacheManagerJndiName`" property is checked first as a pointer to the Infinispan `CacheManager`. If the JNDI name points to a CacheManager, the "`cacheConfigurationName`" property will not be considered.*

cacheConfigurationName

> Optional property that, if used, specifies the name of the configuration resource or file that is supplied to the cache manager when creating a new Infinispan DefaultCacheManager instance. The configuration name is first treated as a resource name and will be attempted to be loaded from the *ClassLoader* [http://java.sun.com/javase/6/docs/api/java/lang/ClassLoader.html]. If that is unsuccessful, the configuration name is assumed to be a file name and will be loaded from the file system. This initialization happens the first time that the source is used.
>
> *Note that the "`cacheManagerJndiName`" property is checked first as a pointer to the Infinispan `CacheManager`. If the JNDI name points to a CacheManager, the "`cacheConfigurationName`" property will not be considered.*

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 is an empty string (or null) and implies that this source does not define a specific duration for caching information provided by this repository source.

defaultWorkspaceName

    Optional property that defines the name for the workspace that will be used in cases when clients do not explicitly specify the workspace name. If not specified, `"default"` will be used.

name

    Required property that specifies the name of the repository source, which is used by the `RepositoryService` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ repository/RepositoryService.html] when obtaining a `RepositoryConnection` [http:// docs.jboss.org/modeshape/2.6.0.Beta2/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.

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 pre-defined UUID constant is used. A custom value need only be supplied for Infinispan sources created prior to ModeShape 2.0, or if a specific UUID is desired or needed.

retryLimit

    Optional property that, if used, defines the number of times that any single operation on a `RepositoryConnection` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.

updatesAllowed

    Optional property that 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".

The `RemoteInfinispanSource` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/connector/infinispan/RemoteInfinispanSource.html] class provides a number of JavaBean properties that control its behavior:

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 is an empty string (or null) and implies that this source does not define a specific duration for caching information provided by this repository source.

defaultWorkspaceName

Optional property that defines the name for the workspace that will be used in cases when clients do not explicitly specify the workspace name. If not specified, `"default"` will be used.

name

Required property that specifies the name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

remoteInfinispanServerList

Optional property that defines the list of Infinispan HotRod servers. The list must be in the appropriate format of `"host:port[;host:port...]"` that would be used when defining an Infinispan `RemoteCacheManager` instance. If the value is missing, `"localhost:11311"` is assumed.

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 pre-defined UUID constant is used. A custom value need only be supplied for Infinispan sources created prior to ModeShape 2.0, or if a specific UUID is desired or needed.

retryLimit

Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositoryConnection.html] to this source should be retried following a communication failure. The default value is '0'.

updatesAllowed

Optional property that 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/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the *InfinispanSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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("predefinedWorkspaceNames", new String[] { "staging", "dev"});
```

Another way to configure the Infinispan connector is to create *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/connector/infinispan/InfinispanSource.html] class. For example a file named configRepository.xml can be created with these contents:

```xml
<?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.connector.infinispan.InfinispanSource"
          mode:description="The repository for our content"
          mode:defaultworkspaceName="prod">
    <mode:predefinedWorkspaceNames>staging</mode:predefinedWorkspaceNames>
    <mode:predefinedWorkspaceNames>dev</mode:predefinedWorkspaceNames>
    </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");
```

17.1. Considerations for Distributed Sources

The `InfinispanSource` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/infinispan/InfinispanSource.html] can be used to provide access to an Infinispan cluster, but be sure to use the DIST_SYNC cache mode. Using other modes will likely lead to data inconsistency. The `RemoteInfinispanSource` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/infinispan/RemoteInfinispanSource.html] can be used to connect to an Infinispan HotRod cache server.

Additionally, some operating systems (e.g., OS X) require you to set either the `java.net.preferIPv4Stack` or the `java.net.preferIPv6Addresses` system property to `true`. These properties are used by JGroups, the communications library that underlies Infinispan, to help determine which address type to use.

> **Note**
>
> The `rootNodeUuid` property *must* be set to the same value for all Infinispan sources in the cluster.

17.1. Considerations for using the HotRod Server

The Infinispan HotRod Server must be made aware of the ModeShape classes that will be stored in it, since it will be running in a separate JVM. The following jar files should be added to the classpath of the HotRod Server:

- modeshape-common

- modeshape-graph

- modeshape-connector-infinispan

- joda-time

Only these libraries contain the classes that are used to store content within Infinispan.

Disk Connector

This connector stores content in a ModeShape-specific file format on disk. Although this may seem similar in concept to the *File System Connector*, this connector actually serves a much different purpose. While the File System Connector is designed to expose existing files and folders on the disk and allow ModeShape users to create content that can be read directly by other applications, the Disk Connector is designed for efficiency and stores content in a serialized representation that is not readily accessible to other applications. Conversely, the Disk Connector supports referenceable nodes and can efficiently access nodes by UUID, unlike the File System Connector.

The `DiskSource` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/disk/DiskSource.html] class provides a number of JavaBean properties that control its behavior:

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 the workspace that will be used by default if none is specified.

largeValuePath

    Optional, advanced property that, if specified, specifies the path to the large value area. *This path is relative to the value of the* `repositoryRootPath` *property.* The default value for this property is `"largeValues"` and it only needs to be changed if there will be a workspace named `"largeValues"`.

largeValueSizeInBytes

    Optional property that, if specified, sets the threshold for large values. Binary property values that exceed this size will be copied into the large value area for this repository, where they can be shared between nodes and lazily loaded to improve node retrieval time. The default value is "8192".

lockFileUsed

    An advanced property that, if set to "true", indicates that repository read and write locks should be synchronized with file lock options on a file in the on-disk storage. This causes a performance penalty, but allows disk sources in different JVMS (e.g., clustered disk sources) to coordinate their locks *as long as all cluster members share the same disk*. This approach uses Java NIO file locking and is subject to the limitations of the Java NIO file locking for the current JVM implementation.

    The default value is "false", but this should always be set to "true" when used in a clustered environment.

name

> Required property that specifies the name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/connector/RepositoryConnection.html] by name.

nodeCachePolicy

> Optional property that, if used, defines the cache policy to use for caching nodes within the connector.

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.

repositoryRootPath

> Optional property that specifies a path on the local file system to the root of all workspaces. The connector will use this as the root for a file and folder structure for storing content. The default value is "/tmp", so setting this property to a more logical value is strongly recommended.

retryLimit

> Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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

> Optional property that 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 "true".

One way to configure the file system connector is to create *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance with a repository source that uses the *DiskSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/disk/DiskSource.html] class. For example:

```
JcrConfiguration config = ...
config.repositorySource("Disk Store")
    .usingClass(DiskSource.class)
    .setDescription("The repository for our content")
    .setProperty("repositoryRootPath", "/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/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html] instance
and load an XML configuration file that contains a repository source that uses
the *DiskSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/
disk/DiskSource.html] class. For example a file named configRepository.xml can be created with
these contents:

```xml
<?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 'Disk Store' repository is a disk source with a three predefined workspaces
    ("prod", "staging", and "dev").
    -->
    <mode:source jcr:name="Disk Store"
      mode:classname="org.modeshape.connector.disk.DiskSource"
      mode:description="The repository for our content"
      mode:repositoryRootPath="/home/content/someApp"
      mode:defaultWorkspaceName="prod"
      mode:creatingWorkspacesAllowed="false"
      mode:rootNodeUuid="fd129c12-81a8-42ed-aa4b-820dba49e6f0"
      mode:updatesAllowed="true" >
      <mode:predefinedWorkspaceNames>staging</mode:predefinedWorkspaceNames>
      <mode:predefinedWorkspaceNames>dev</mode:predefinedWorkspaceNames>
      <!--
      If desired, specify a cache policy that caches items in memory for 5 minutes (300 s).
      This fragment can be left out if the connector should not cache any content.
      -->
```

```
        <mode:cachePolicy jcr:name="nodeCachePolicy"
         mode:classname="org.modeshape.graph.connector.base.cache.InMemoryNodeCache
$MapCachePolicy"
          mode:timeToLive="300" />
      </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");
```

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/javase/6/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.

> **i** **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/javase/6/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/2.6.0.Beta2/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/javase/6/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/2.6.0.Beta2/manuals/gettingstarted/html/index.html] Guide for detailed examples of how to import custom JCR node types.

The *JdbcMetadataSource* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/meta/jdbc/JdbcMetadataSource.html] class provides a number of JavaBean properties that control its behavior:

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

> Optional property that defines 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

> Optional property that defines 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

> Optional property that defines the name of the ModeShape 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, or if the driver is on the application's classpath.

driverClassName

> The name of the JDBC driver class. This is not required if the DataSource is found in JNDI, but is required otherwise.

idleTimeInSecondsBeforeTestingConnections

> Optional property that defines 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).

maximumConnectionsInPool

> Optional property that defines the maximum number of connections that may be in the connection pool. The default is "5".

maximumConnectionIdleTimeInSeconds

> Optional property that defines 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

Optional property that defines the maximum number of statements that should be cached. The default value is "100", but statement caching can be disabled by setting to "0".

metadataCollectorClassName

Advanced optional property that defines the name of a custom class to use for metadata collection, which is typically needed for JDBC drivers that don't properly support the standard *DatabaseMetaData* [http://java.sun.com/javase/6/docs/api/java/sql/DatabaseMetaData.html] methods. The specified class must implement the *MetadataCollector* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/meta/jdbc/MetadataCollector] interface and must have a public no-argument constructor. If an empty string (or null) value is specified for this property, a default *MetadataCollector* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/connector/meta/jdbc/MetadataCollector] implementation will be used that relies on the driver's *DatabaseMetaData* [http://java.sun.com/javase/6/docs/api/java/sql/DatabaseMetaData.html].

minimumConnectionsInPool

Optional property that defines the minimum number of connections that will be kept in the connection pool. The default is "0".

name

Required property that specifies the name of the repository source, which is used by the *RepositoryService* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/repository/RepositoryService.html] when obtaining a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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.

nodeCachePolicy

Optional property that, if used, defines the cache policy to use for caching nodes within the connector.

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.

retryLimit

Optional property that, if used, defines the number of times that any single operation on a *RepositoryConnection* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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 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/2.6.0.Beta2/api/org/modeshape/ jcr/JcrConfiguration.html] instance with a repository source that uses the `JdbcMetadataSource` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/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">
```

```xml
<!--
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.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 (300 s).
    This fragment can be left out if the connector should not cache any content.
    -->
    <mode:cachePolicy jcr:name="nodeCachePolicy"
     mode:classname="org.modeshape.graph.connector.base.cache.InMemoryNodeCache$PathCachePolicy"
        mode:timeToLive="300" />
  </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:

```java
JcrConfiguration config = new JcrConfiguration().loadFrom("/configRepository.xml");
```

Part IV. 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`.

20.1. Example

This sequencer generates a graph structure that corresponds to what can be found in the /jcr:system/jcr:nodeTypes subtree. As an example, the CND file below:

```
<mode = "http://www.modeshape.org/1.0">

// My CND type
[mode:example] mixin
- mode:name (string) multiple copy
+ mode:child (mode:example) = mode:example version
```

The resulting graph structure (listed in the JCR document view) contains the node type information from the CND file above. Note that comments are not sequenced.

```
<mode:example jcr:primaryType="nt:nodeType"
        jcr:mixinTypes="mode:derived"
        mode:derivedAt="2011-05-13T13:12:03.925Z"
        mode:derivedFrom="/files/docForReferenceGuide.xml"
        jcr:nodeTypeName="mode:example"
        jcr:supertypes="nt:base"
        jcr:isAbstract="false"
        jcr:isMixin="true"
        jcr:isQueryable="true"
        jcr:hasOrderableChildNodes="false">
  <nt:propertyDefinition jcr:name="mode:name"
              jcr:autoCreated="false"
              jcr:mandatory="false"
              jcr:isFullTextSearchable="true"
              jcr:isQueryOrderable="true"
```

```
                    jcr:onParentVersion="copy"
                    jcr:protected="false"
                    jcr:requiredType="STRING"
                    jcr:availableQueryOperators="= > >= < <= <> LIKE"
                    jcr:multiple="true" />
  <nt:childNodeDefinition jcr:name="mode:child"
                    jcr:autoCreated="false"
                    jcr:mandatory="false"
                    jcr:onParentVersion="VERSION"
                    jcr:protected="false"
                    jcr:requiredPrimaryTypes="mode:example"
                    jcr:defaultPrimaryType="mode:example"
                    jcr:sameNameSiblings="false" />
</mode:example>
```

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.

21.1. Example

For this XML document:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.4//EN" "http://www.oasis-open.org/docbook/xml/4.4/docbookx.dtd" [
<!ENTITY % RH-ENTITIES SYSTEM "Common_Config/rh-entities.ent">
<!ENTITY versionNumber "0.1">
<!ENTITY copyrightYear "2008">
<!ENTITY copyrightHolder "Red Hat Middleware, LLC.">]>
<?target content ?>
<?target2 other stuff ?>
<Cars xmlns:jcr="http://www.jcp.org/jcr/1.0">
   <!-- This is a comment -->
   <Hybrid>
      <car jcr:name="Toyota Prius"/>
   </Hybrid>
   <Sports>
   </Sports>
</Cars>
```

The sequencer will generate this content (listed in document view) if the sequencer `outputtingTo` property generates an output path ending in "myxml":

```
 xml jcr:primaryType=nt:unstructured
 <myxml jcr:primaryType="modexml:document"
      jcr:mixinTypes="mode:derived"
      mode:derivedAt="2011-05-13T13:12:03.925Z"
      mode:derivedFrom="/files/docForReferenceGuide.xml"
      modedtd:name="book"
      modedtd:publicId="-//OASIS//DTD DocBook XML V4.4//EN"
      modedtd:systemId="http://www.oasis-open.org/docbook/xml/4.4/docbookx.dtd">
 <modedtd:entity jcr:primaryType="modedtd:entity"
         modedtd:name="%RH-ENTITIES"
```

```
          modedtd:systemId="Common_Config/rh-entities.ent" />
 <modedtd:entity[2] jcr:primaryType="modedtd:entity"
          modedtd:name="versionNumber"
          modedtd:value="0.1" />
 <modedtd:entity[3] jcr:primaryType="modedtd:entity"
          modedtd:name="copyrightYear"
          modedtd:value="2008" />
 <modedtd:entity[4] jcr:primaryType="modedtd:entity"
          modedtd:name="copyrightHolder"
          modedtd:value="Red Hat Middleware, LLC." />
 <modexml:processingInstruction jcr:primaryType="modexml:processingInstruction"
              modexml:processingInstructionContent="content"
              modexml:target="target" />
 <modexml:processingInstruction[2] jcr:primaryType="modexml:processingInstruction"
                modexml:processingInstructionContent="other stuff"
                modexml:target="target2" />
 <Cars jcr:primaryType="modexml:element">
  <modexml:comment jcr:primaryType="modexml:comment"
          modexml:commentContent="This is a comment" />
  <Hybrid jcr:primaryType="modexml:element">
   <car jcr:primaryType="modexml:element" />
  </Hybrid>
  <Sports jcr:primaryType="modexml:element" />
 </Cars>
</myxml>
```

The CND used by this sequencer is provided below. Note that the XML sequencer will parse CDATA into its own node in the sequenced output even though the example above does not explicitly demonstrate this.

```
<modexml='http://www.modeshape.org/xml/1.0'>
<modedtd='http://www.modeshape.org/dtd/1.0'>

[modexml:document] > nt:unstructured, mix:mimeType
 - modexml:cDataContent (string)

[modexml:comment] > nt:unstructured
 - modexml:commentContent (string)

[modexml:element] > nt:unstructured

[modexml:elementContent] > nt:unstructured
```

- modexml:elementContent (string)

[modexml:cData] > nt:unstructured
 - modexml:cDataContent (string)

[modexml:processingInstruction] > nt:unstructured
 - modexml:processingInstruction (string)
 - modexml:target (string)

[modedtd:entity] > nt:unstructured
 - modexml:name (string)
 - modexml:value (string)
 - modexml:publicId (string)
 - modexml:systemId (string)

```
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");
```

XML Schema Document (XSD) Sequencer

The XSD sequencer included in ModeShape can parse XML Schema Documents that adhere to the W3C's XML Schema *Part 1* [http://www.w3.org/TR/xmlschema-1/] and *Part 2* [http://www.w3.org/TR/xmlschema-2/] specifications, and output a representation of the XSD's attribute declarations, element declarations, simple type definitions, complex type definitions, import statements, include statements, attribute group declarations, annotations, other components, and even attributes with a non-schema namespace. This derived information is intended to accurately reflect the structure and semantics of the XSD files while also making it possible for ModeShape users to easily navigate, query and search over this derived information. This sequencer captures the namespace and names of all referenced components, and will resolve references to components appearing within the same files.

The design of this sequencer and it's output structure have been influenced by the SOA Repository Artifact Model and Protocol (S-RAMP) draft specification, which is currently under development as an OASIS Technology Committee. S-RAMP defines a model for a variety of file types, including WSDL and XSD. This sequencer's output was designed to mirror that model, and thus some of the properties and node types used are defined within the "sramp" namespace.

The XML Schema specification is powerful, flexible, rich, and complicated. This means that many XML Schema Documents themselves are complicated. But it also means that there is a lot of variation in XSDs, and consequently there is a lot of variation in the output structure that this sequencer derives from XSD files.

22.1. Example

So before we get too far, let's look at an example XML Schema Document taken from the *XML Schema Primer* [http://www.w3.org/TR/xmlschema-0/]:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:annotation>
  <xsd:documentation xml:lang="en">
   Purchase order schema for Example.com.
   Copyright 2000 Example.com. All rights reserved.
  </xsd:documentation>
 </xsd:annotation>

 <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
```

```xsd
<xsd:element name="comment" type="xsd:string"/>

<xsd:complexType name="PurchaseOrderType">
 <xsd:sequence>
  <xsd:element name="shipTo" type="USAddress"/>
  <xsd:element name="billTo" type="USAddress"/>
  <xsd:element ref="comment" minOccurs="0"/>
  <xsd:element name="items"  type="Items"/>
 </xsd:sequence>
 <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:complexType name="USAddress">
 <xsd:sequence>
  <xsd:element name="name"   type="xsd:string"/>
  <xsd:element name="street" type="xsd:string"/>
  <xsd:element name="city"   type="xsd:string"/>
  <xsd:element name="state"  type="xsd:string"/>
  <xsd:element name="zip"    type="xsd:decimal"/>
 </xsd:sequence>
 <xsd:attribute name="country" type="xsd:NMTOKEN"
          fixed="US"/>
</xsd:complexType>

<xsd:complexType name="Items">
 <xsd:sequence>
  <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="productName" type="xsd:string"/>
     <xsd:element name="quantity">
      <xsd:simpleType>
       <xsd:restriction base="xsd:positiveInteger">
        <xsd:maxExclusive value="100"/>
       </xsd:restriction>
      </xsd:simpleType>
     </xsd:element>
     <xsd:element name="USPrice"  type="xsd:decimal"/>
     <xsd:element ref="comment"   minOccurs="0"/>
     <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="partNum" type="SKU" use="required"/>
   </xsd:complexType>
  </xsd:element>
```

```
    </xsd:sequence>
  </xsd:complexType>

  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
   <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
   </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

This schema defines the structure of several XML elements used to represent purchase orders, and describes an XML document such as the following:

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild<!comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
```

```
        <USPrice>39.98</USPrice>
        <shipDate>1999-05-21</shipDate>
      </item>
    </items>
</purchaseOrder>
```

The XSD sequencer will derive the following content from the above XSD:

```
po.xsd jcr:primaryType=xs:schemaDocument jcr:mixinTypes=[mode:derived]
  - jcr:uuid=ca46f972-6875-481d-b9e1-cfb64ae76f74
  - mode:derivedAt=2011-05-18T18:34:08.922Z
  - mode:derivedFrom=/files/po.xsd
  - sramp:contentEncoding="UTF-8"
  - sramp:contentSize=2353
  - sramp:contentType="application/xsd"
  - sramp:description="Purchase order schema for Example.com.
  Copyright 2000 Example.com. All rights reserved."
  purchaseOrder jcr:primaryType=xs:elementDeclaration
    - jcr:uuid=eff3bcfb-42d1-4d55-805b-5133279e15eb
    - xs:abstract=false
    - xs:form="qualified"
    - xs:ncName="purchaseOrder"
    - xs:nillable=false
    - xs:type=5088dc05-ad30-4d7d-8d24-3edc548a777f
    - xs:typeName="PurchaseOrderType"
  comment jcr:primaryType=xs:elementDeclaration
    - jcr:uuid=2daaa747-01f1-41f3-b5c2-ec218d8a7290
    - xs:abstract=false
    - xs:form="qualified"
    - xs:ncName="comment"
    - xs:nillable=false
    - xs:typeName="string"
    - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
  PurchaseOrderType jcr:primaryType=xs:complexTypeDefinition
    - jcr:uuid=5088dc05-ad30-4d7d-8d24-3edc548a777f
    - xs:abstract=false
    - xs:baseTypeName="anyType"
    - xs:baseTypeNamespace="http://www.w3.org/2001/XMLSchema"
    - xs:method="restriction"
    - xs:mixed=false
    - xs:ncName="PurchaseOrderType"
    xs:sequence jcr:primaryType=xs:sequence
```

```
              - jcr:uuid=1b87d92d-4d59-44ac-859f-2a51c3a48eb2
              - xs:maxOccurs=1
              - xs:minOccurs=1
          shipTo jcr:primaryType=xs:elementDeclaration
              - jcr:uuid=994ba18b-c389-4635-8ce3-27d3a81cf97d
              - xs:abstract=false
              - xs:form="qualified"
              - xs:maxOccurs=1
              - xs:minOccurs=1
              - xs:ncName="shipTo"
              - xs:nillable=false
              - xs:type=dd683707-83bb-4893-aa6e-f3ce81237e76
              - xs:typeName="USAddress"
          billTo jcr:primaryType=xs:elementDeclaration
              - jcr:uuid=e260c1aa-5a5a-4db5-a962-b02576359ee7
              - xs:abstract=false
              - xs:form="qualified"
              - xs:maxOccurs=1
              - xs:minOccurs=1
              - xs:ncName="billTo"
              - xs:nillable=false
              - xs:type=dd683707-83bb-4893-aa6e-f3ce81237e76
              - xs:typeName="USAddress"
          comment jcr:primaryType=xs:elementDeclaration
              - jcr:uuid=a7796d20-0e7b-4833-96b6-16e0ac6676ca
              - xs:abstract=false
              - xs:form="qualified"
              - xs:maxOccurs=1
              - xs:minOccurs=0
              - xs:nillable=false
              - xs:ref=2daaa747-01f1-41f3-b5c2-ec218d8a7290
              - xs:refName="comment"
          items jcr:primaryType=xs:elementDeclaration
              - jcr:uuid=02ab83d1-ea1a-4a7b-b66d-a1974f13ca63
              - xs:abstract=false
              - xs:form="qualified"
              - xs:maxOccurs=1
              - xs:minOccurs=1
              - xs:ncName="items"
              - xs:nillable=false
              - xs:type=7543bf0f-1753-4813-9a31-f2bbed34fd11
              - xs:typeName="Items"
        orderDate jcr:primaryType=xs:attributeDeclaration
            - jcr:uuid=8b23e048-c683-4d6d-8835-faf81df6912d
```

```
      - xs:ncName="orderDate"
      - xs:typeName="date"
      - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
      - xs:use="optional"
  USAddress jcr:primaryType=xs:complexTypeDefinition
    - jcr:uuid=dd683707-83bb-4893-aa6e-f3ce81237e76
    - xs:abstract=false
    - xs:baseTypeName="anyType"
    - xs:baseTypeNamespace="http://www.w3.org/2001/XMLSchema"
    - xs:method="restriction"
    - xs:mixed=false
    - xs:ncName="USAddress"
    xs:sequence jcr:primaryType=xs:sequence
      - jcr:uuid=82411c47-7f1a-4b11-9778-acc310c9e51c
      - xs:maxOccurs=1
      - xs:minOccurs=1
      name jcr:primaryType=xs:elementDeclaration
        - jcr:uuid=40dcb6fc-386c-4d3a-841b-dab478348d74
        - xs:abstract=false
        - xs:form="qualified"
        - xs:maxOccurs=1
        - xs:minOccurs=1
        - xs:ncName="name"
        - xs:nillable=false
        - xs:typeName="string"
        - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
      street jcr:primaryType=xs:elementDeclaration
        - jcr:uuid=a3ff1a2d-38e7-442a-a46b-141fa1ac4442
        - xs:abstract=false
        - xs:form="qualified"
        - xs:maxOccurs=1
        - xs:minOccurs=1
        - xs:ncName="street"
        - xs:nillable=false
        - xs:typeName="string"
        - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
      city jcr:primaryType=xs:elementDeclaration
        - jcr:uuid=30d4215f-cd44-4857-9589-3df127e42cf3
        - xs:abstract=false
        - xs:form="qualified"
        - xs:maxOccurs=1
        - xs:minOccurs=1
        - xs:ncName="city"
        - xs:nillable=false
```

```
        - xs:typeName="string"
        - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
      state jcr:primaryType=xs:elementDeclaration
        - jcr:uuid=061a58d9-94fd-4dca-84e2-6ced7fe523fe
        - xs:abstract=false
        - xs:form="qualified"
        - xs:maxOccurs=1
        - xs:minOccurs=1
        - xs:ncName="state"
        - xs:nillable=false
        - xs:typeName="string"
        - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
      zip jcr:primaryType=xs:elementDeclaration
        - jcr:uuid=100dc3cc-b59f-4835-b14e-243b9e7a2ecf
        - xs:abstract=false
        - xs:form="qualified"
        - xs:maxOccurs=1
        - xs:minOccurs=1
        - xs:ncName="zip"
        - xs:nillable=false
        - xs:typeName="decimal"
        - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
    country jcr:primaryType=xs:attributeDeclaration
      - jcr:uuid=f323219f-bea0-4d6f-9ad5-f51cf8409f13
      - xs:ncName="country"
      - xs:typeName="NMTOKEN"
      - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
      - xs:use="optional"
  Items jcr:primaryType=xs:complexTypeDefinition
    - jcr:uuid=7543bf0f-1753-4813-9a31-f2bbed34fd11
    - xs:abstract=false
    - xs:baseTypeName="anyType"
    - xs:baseTypeNamespace="http://www.w3.org/2001/XMLSchema"
    - xs:method="restriction"
    - xs:mixed=false
    - xs:ncName="Items"
    xs:sequence jcr:primaryType=xs:sequence
      - jcr:uuid=d907da56-f370-40e3-b06e-e3a5ae957f4d
      - xs:maxOccurs=1
      - xs:minOccurs=1
      item jcr:primaryType=xs:elementDeclaration
        - jcr:uuid=87cc1352-2f90-49f4-9f36-3db7b9ffcf26
        - xs:abstract=false
        - xs:form="qualified"
```

```
        - xs:minOccurs=0
        - xs:ncName="item"
        - xs:nillable=false
  SKU jcr:primaryType=xs:simpleTypeDefinition
    - jcr:uuid=4127108d-a699-461e-8210-3bb40c923318
    - xs:baseTypeName="string"
    - xs:baseTypeNamespace=http://www.w3.org/2001/XMLSchema
    - xs:ncName="SKU"
    - xs:pattern="\d{3}-[A-Z]{2}"
```

The first thing to note is that the sequencer produces a node of type `xs:schemaDocument` that includes the `mode:derived` information (e.g., the time of sequencing and the path to the file from which this information was derived), information about the XSD itself, plus an `sramp:description` property containing the documentation content from any annotations directly under the `schema` element in the XSD.

Secondly, there is a node for each top-level element declaration, namely "`purchaseOrder`" and "`comment`", with properties capturing the element's name, namespace (not shown since there is no target namespace for the schema), and XSD type name, namespace and reference. The "`comment`" element declaration has a base type of "`xs:string`", whereas the "`purchaseOrder`" element declaration has a type of "`PurchaseOrderType`" (defined later in the XSD and in the derived content). Each node is "`mix:referenceable`" and has a `jcr:uuid` property, allowing the "`purchaseOrder`" element declaration to have a "`xs:type`" REFERENCE property pointing to the "`PurchaseOrderType`" complex type definition node.

There are also nodes representing each of the global complex type definitions, including "`PurchaseOrderType`", "`USAddress`", "`Items`", and "`SKU`". Each of these nodes has properties representing the complex type's features (such as `abstract`, `mixed`, `name`, etc.), as well as child nodes that represent the definition of the complex type's content (e.g., sequence, choice, all, simple content, complex content, etc.).

This example shows some of the structure that this sequencer derives from the XML Schema Documents. Our goal for this sequencer was to output content that reflected as accurately as possible the structure of the XML Schema Documents while also making the content easy to navigate, search and query.

22.2. Node Types

The XSD sequencer follows JCR best-practices by defining all nodes to have a primary type that allows any single or multi-valued property, meaning it's possible and valid for any node to have any property (with single or multiple values). In fact, this feature is used when XSD files contain attributes with non-schema namespaces, which are then mapped onto properties with the attributes name and possibly-empty namespace. However, it is still useful to capture the metadata about what that node represents, and so the sequencer use explicit node type definitions and mixins for this.

The compact node definitions for the "xs" namespace are as follows:

```
<jcr='http://www.jcp.org/jcr/1.0'>
<nt='http://www.jcp.org/jcr/nt/1.0'>
<mix='http://www.jcp.org/jcr/mix/1.0'>
<sramp = "http://s-ramp.org/xmlns/2010/s-ramp">
<xs = "http://www.w3.org/2001/XMLSchema">


//-----------------------------------------------------------------------------
// N O D E T Y P E S
//-----------------------------------------------------------------------------


[xs:component] > sramp:derivedArtifactType abstract
- xs:id (string)
- * (undefined) multiple
- * (undefined)


[xs:namespaced] mixin
- xs:namespace (uri) mandatory


[xs:located] mixin
- xs:schemaLocation (string)


[xs:import] > xs:component, xs:located, xs:namespaced


[xs:include] > xs:component, xs:located


[xs:redefine] > xs:component, xs:located


[xs:named] > xs:namespaced mixin
- xs:ncName (string) mandatory


[xs:typeDefinition] > xs:component


// A mixin representing a reference to an 'xs:typeDefinition'
[xs:typed] mixin
- xs:typeName (string)
- xs:typeNamespace (uri)
- xs:type (weakreference) < 'xs:typeDefinition'


// Attribute wildcard
[xs:anyAttribute] > xs:component
- xs:minOccurs (long) < '[0,)'
```

- xs:maxOccurs (long) < '[0,)'
- xs:namespace (uri) multiple
- xs:processContents (string) = 'strict' < 'lax', 'strict', 'skip'

//
// The 'group', 'all', 'sequence' and 'choice' components
//
[xs:modelGroup] > xs:component abstract
- xs:minOccurs (long) < '[0,)'
- xs:maxOccurs (long) < '[0,)'
- xs:refName (string)
- xs:refNamespace (uri)
- xs:ref (weakReference) < 'xs:modelGroup'
+ * (xs:elementDeclaration)

[xs:group] > xs:modelGroup
+ 'xs:anyAttribute' (xs:anyAttribute)

[xs:all] > xs:modelGroup

[xs:sequence] > xs:modelGroup
+ 'xs:sequence' (xs:sequence)
+ 'xs:choice' (xs:choice)
+ 'xs:all' (xs:all)
+ 'xs:anyAttribute' (xs:anyAttribute)

[xs:choice] > xs:modelGroup
+ 'xs:sequence' (xs:sequence)
+ 'xs:choice' (xs:choice)
+ 'xs:all' (xs:all)
+ 'xs:anyAttribute' (xs:anyAttribute)

//
// The 'simpleContent' and 'complexContent' components
//
[xs:complexContent] > xs:component
- xs:method (string) < 'restriction', 'extension'
+ * (xs:attributeDeclaration)
+ * (xs:attributeGroup)
+ * (xs:group)
+ 'xs:anyAttribute' (xs:anyAttribute)
+ 'xs:sequence' (xs:sequence)
+ 'xs:choice' (xs:choice)
+ 'xs:all' (xs:all)

```
[xs:simpleContent]  > xs:component
- xs:method (string) < 'restriction', 'extension'
- xs:minValueExclusive (*)
- xs:minValueInclusive (*)
- xs:maxValueExclusive (*)
- xs:maxValueInclusive (*)
- xs:totalDigits (long) < '[0,]'
- xs:fractionDigits (long) < '[0,]'
- xs:length (long)
- xs:maxLength (long) < '[0,]'
- xs:minLength (long) < '[0,]'
- xs:enumeratedValues (string) multiple
- xs:whitespace (string) < 'preserve','collapse','replace'
- xs:pattern (string)
+ * (xs:attributeDeclaration) sns
+ * (xs:attributeGroup) sns
+ * (xs:simpleTypeDefinition) sns
+ 'xs:anyAttribute' (xs:anyAttribute)


//
// Attribute Groups
//
[xs:attributeGroup] > xs:component
- xs:ncName (string)
- xs:namespace (uri)
- xs:refName (string)
- xs:refNamespace (uri)
- xs:ref (weakReference) < 'xs:attributeGroup'
+ * (xs:attributeDeclaration) sns
+ * (xs:attributeGroup) sns
+ 'xs:anyAttribute' (xs:anyAttribute)


//
// Complex and simple type definitions
//
[xs:complexTypeDefinition] > xs:typeDefinition, xs:named
- xs:abstract (boolean) = 'false'
- xs:mixed (boolean) = 'false'
- xs:block (string) multiple < 'restriction', 'extension', 'all'
- xs:final (string) multiple < 'restriction', 'extension', 'all'
+ * (xs:attributeDeclaration) sns
+ * (xs:attributeGroup) sns
+ * (xs:complexContent) sns
```

+ * (xs:simpleContent) sns

+ * (xs:group) sns

+ 'xs:anyAttribute' (xs:anyAttribute)

+ 'xs:sequence' (xs:sequence)

+ 'xs:choice' (xs:choice)

+ 'xs:all' (xs:all)

[xs:simpleTypeDefinition] > xs:typeDefinition, xs:named

- xs:baseTypeName (string)

- xs:baseTypeNamespace (uri)

- xs:baseType (weakreference) < 'xs:typeDefinition'

- xs:final (string) multiple < 'restriction', 'list', 'union', 'all'

//

// Attribute declaration

//

[xs:attributeDeclaration] > xs:component, xs:named, xs:typed

- xs:length (long)

- xs:maxLength (long)

- xs:minLength (long)

- xs:enumeratedValues (string) multiple

- xs:whitespace (string) < 'preserve','collapse','replace'

- xs:maxValueExclusive (*)

- xs:minValueExclusive (*)

- xs:maxValueInclusive (*)

- xs:minValueInclusive (*)

- xs:totalDigits (long)

- xs:fractionDigits (long)

- xs:pattern (string)

- xs:use (string)

//

// Identity constraint definition

//

[xs:selector] > xs:component

- xs:xpath (string) mandatory

[xs:field] > xs:component

- xs:xpath (string) mandatory

[xs:identityConstraintDefinition] > xs:component abstract

- xs:ncName (string) mandatory

+ 'selector' (xs:selector)

+ 'field' (xs:field) sns

[xs:unique] > xs:identityConstraintDefinition

[xs:key] > xs:identityConstraintDefinition

[xs:keyref] > xs:identityConstraintDefinition
- xs:refer (string) mandatory

//
// Element declaration
//
[xs:elementDeclaration] > xs:component, xs:named, xs:typed
- xs:abstract (boolean) = 'false'
- xs:nillable (boolean) = 'false'
- xs:final (string) multiple < 'all', 'extension', 'restriction'
- xs:block (string) multiple < 'all', 'extension', 'restriction', 'substitution'
- xs:default (string)
- xs:fixed (string)
- xs:form (string) < 'qualified', 'unqualified'
- xs:minOccurs (long) < '[0,)'
- xs:maxOccurs (long) < '[0,)'
- xs:refName (string)
- xs:refNamespace (uri)
- xs:ref (weakReference) < 'xs:elementDeclaration'
- xs:substitutionGroupName (string)
- xs:substitutionGroup (weakReference) < 'xs:elementDeclaration'
+ * (xs:typeDefinition)
+ * (xs:identityConstraintDefinition)

//
// XML Schema Document
//
[xs:schemaDocument] > sramp:xmlDocument
- xs:id (string)
- xs:targetNamespace (uri)
- xs:version (string)
- xs:attributeFormDefault (string) = 'unqualified' < 'qualified', 'unqualified'
- xs:elementFormDefault (string) = 'unqualified' < 'qualified', 'unqualified'
- xs:finalDefault (string) multiple < 'all', 'extension', 'restriction', 'list', 'union'
- xs:blockDefault (string) multiple < 'all', 'extension', 'restriction', 'substitution'
- xs:importedXsds (weakreference) multiple < 'xs:xsdDocument'
- xs:includedXsds (weakreference) multiple < 'xs:xsdDocument'
- xs:redefinedXsds (weakreference) multiple < 'xs:xsdDocument'
- * (undefined) multiple

```
- * (undefined)
+ * (xs:import) sns
+ * (xs:include) sns
+ * (xs:redefine) sns
// Technically need 'sns' because the attributes, elements, simple types, complex types, attribute
 groups,
// and groups don't share same name scopes
+ * (xs:attributeDeclaration) sns
+ * (xs:elementDeclaration) sns
+ * (xs:attributeGroup) sns
+ * (xs:group) sns
+ * (xs:simpleTypeDefinition) sns
+ * (xs:complexTypeDefinition) sns
```

These types use some of the node types and mixins defined in the "sramp" namespace:

```
<jcr='http://www.jcp.org/jcr/1.0'>
<nt='http://www.jcp.org/jcr/nt/1.0'>
<mix='http://www.jcp.org/jcr/mix/1.0'>
<sramp = "http://s-ramp.org/xmlns/2010/s-ramp">


//----------------------------------------------------------------------------
// N O D E T Y P E S
//----------------------------------------------------------------------------


// ------------------------------------------------------
// S-RAMP Core Model Artifacts
// ------------------------------------------------------

[sramp:baseArtifactType] > mix:created, mix:lastModified, mix:referenceable, mix:versionable
 abstract mixin
- sramp:classifiedBy (reference) multiple < 'owl:class'
- sramp:description (string)
- * (string)
- * (string) multiple

[sramp:documentArtifactType] > sramp:baseArtifactType abstract mixin
- sramp:contentType (string)
- sramp:contentSize (long)

[sramp:xmlDocument] > sramp:documentArtifactType mixin
- sramp:contentEncoding (string) mandatory
```

```
[sramp:document] > sramp:documentArtifactType mixin

[sramp:derivedArtifactType] > sramp:baseArtifactType abstract mixin
- sramp:relatedDocuments (reference) < 'sramp:documentArtifactType'

[sramp:userDefinedArtifactType] > sramp:baseArtifactType mixin
- sramp:userType (string) mandatory

[sramp:storedQuery] > nt:query
- sramp:propertyList (string) multiple

[sramp:relatedTo] mixin
- * (weakreference) multiple
```

22.3. Configuration

To use this sequencer, simply include the appropriate version of the Maven artifact with a `"org.modeshape"` group ID and `"modeshape-sequencer-xsd"` artifact ID. Or, if you're using JAR files and manually setting up the classpath for your application, use the `"modeshape-sequencer-xsd-2.6.0.Beta2-jar-with-dependencies.jar"` file. Then, define a sequencing configuration in the ModeShape configuration, using something similar to:

```xml
<configuration xmlns:mode="http://www.modeshape.org/1.0"
        xmlns:jcr="http://www.jcp.org/jcr/1.0">
  <mode:sequencers>
      ...
    <mode:sequencer jcr:name="XSD Sequencer"
            mode:classname="org.modeshape.sequencer.xsd.XsdSequencer">
      <mode:description>Sequences XML Schema Documents (e.g., *.xsd) loaded
          into the repository under '/files', extracting the XSD attribute
          and element declarations, simple and complex type definitions,
          attribute and element groups, annotations, imports, includes and
          any other component of XSDs.</mode:description>
      <!-- Note this path expression captures the path below '/files' but
          excludes the filename, and places the sequenced content under the
          same relative path below '/sequenced/xsd'. For example, if an XSD
          file is uploaded to '/files/my/favorites/Customers.xsd', then the
          sequenced output will be placed at the
          '/sequenced/xsd/my/favorites/Customer.xsd' node, which will have
          a primary type of 'xs:schemaDocument' and will contain under it
          the components within the schema document.  Of course, the path
```

```
        expression can be modified as needed. -->
      <mode:pathExpression>/files(//)(*.xsd[*])/jcr:content[@jcr:data]
              => /sequenced/xsd/$1 </mode:pathExpression>
    </mode:sequencer>
    ...
  </mode:sequencers>
  ...
</configuration>
```

or using the *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html]:

```
JcrConfiguration config = ...

config.sequencer("XSD Sequencer")
    .usingClass(XsdSequencer.class)
    .setDescription("Sequences XML Schema documents")
    .sequencingFrom("/files(//)(*.xsd[*])/jcr:content[@jcr:data]")
    .andOutputtingTo("/sequenced/xsd/$1");
```

Web Service Definition Language (WSDL) 1.1 Sequencer

The XSD sequencer included in ModeShape can parse WSDL files that adhere to the *W3C's Web Service Definition Language (WSDL) 1.1* [http://www.w3.org/TR/wsdl] specification, and output a representation of the WSDL file's messages, port types, bindings, services, types (including embedded XML Schemas), documentation, and extension elements (including HTTP, SOAP and MIME bindings). This derived information is intended to mirror the structure and semantics of the actual WSDL files while also making it possible for ModeShape users to easily navigate, query and search over this derived information. This sequencer captures the namespace and names of all referenced components, and will resolve references to components appearing within the same file.

The design of this sequencer and it's output structure have been influenced by the SOA Repository Artifact Model and Protocol (S-RAMP) draft specification, which is currently under development as an OASIS Technology Committee. S-RAMP defines a model for a variety of file types, including WSDL and XSD. This sequencer's output was designed to mirror that model, and thus some of the properties and node types used are defined within the "sramp" namespace. However, the structure derived by the ModeShape WSDL sequencer is a superset of that defined by S-RAMP.

The WSDL specification allows for a fair amount of variation in WSDL files, and consequently this variation is reflected in the derived output structure.

23.1. Example

Let's look at an example WSDL file from the *WSDL 1.1 specification* [http://www.w3.org/TR/wsdl#_wsdl]:

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml version="1.0"?>
<definitions name="StockQuote"
        targetNamespace="http://example.com/stockquote.wsdl"
        xmlns:tns="http://example.com/stockquote.wsdl"
        xmlns:xsd1="http://example.com/stockquote.xsd"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
        xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
```

```
        <all>
          <element name="tickerSymbol" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="TradePrice">
      <complexType>
        <all>
          <element name="price" type="float"/>
        </all>
      </complexType>
    </element>
  </schema>
</types>

<message name="GetLastTradePriceInput">
  <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
  <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>

<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document"
         transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetLastTradePrice">
    <soap:operation
        soapAction="http://example.com/GetLastTradePrice"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

```
<service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteBinding">
        <soap:address location="http://example.com/stockquote"/>
    </port>
</service>

</definitions>
```

This WSDL definition includes an embedded XML Schema that defines the structure of two XML elements used in the web service messages, and it defines a 'StockQuotePortType' port type with input and output messages, a SOAP binding, and a SOAP service. The WSDL sequencer will derive from this file the following content:

```
stockQuote.wsdl jcr:primaryType=wsdl:wsdlDocument jcr:mixinTypes=[mode:derived]
  - jcr:uuid=d69d9fac-c5b5-42fc-ae70-0947d5986744
  - mode:derivedAt=2011-05-24T20:22:23.404Z
  - mode:derivedFrom=/files/stockQuote.wsdl
  - sramp:contentSize=2210
  - sramp:contentType="application/wsdl"
  wsdl:schema jcr:primaryType=xs:schemaDocument
   - jcr:uuid=8e0b8a17-11d2-4611-bc83-ef067526329c
   - sramp:contentSize=623
   - sramp:contentType="application/xsd"
   - targetNamespace="http://example.com/stockquote.xsd"
   - xmlns:xmlns="http://www.w3.org/2001/XMLSchema"
   TradePriceRequest jcr:primaryType=xs:elementDeclaration
    - jcr:uuid=8407370d-9c6a-43ad-84ee-53f480524432
    - xs:abstract=false
    - xs:form="qualified"
    - xs:namespace=http://example.com/stockquote.xsd
    - xs:ncName="TradePriceRequest"
    - xs:nillable=false
    - xs:typeNamespace=http://example.com/stockquote.xsd
    xs:complexType jcr:primaryType=xs:complexTypeDefinition
     - jcr:uuid=5afc2fe3-e6c3-4cc2-8667-e5a9faf8963d
     - xs:abstract=false
     - xs:baseTypeName="anyType"
     - xs:baseTypeNamespace="http://www.w3.org/2001/XMLSchema"
     - xs:method="restriction"
     - xs:mixed=false
```

      - xs:namespace=http://example.com/stockquote.xsd
     xs:all jcr:primaryType=xs:all
       - jcr:uuid=e491f657-c20a-43e7-99b7-e5f76778c11e
       - xs:maxOccurs=1
       - xs:minOccurs=1
     tickerSymbol jcr:primaryType=xs:elementDeclaration
       - jcr:uuid=22c26a7f-e9fa-4c44-a346-7df3cf436c7a
       - id="string"
       - name="string"
       - xs:abstract=false
       - xs:form="qualified"
       - xs:maxOccurs=1
       - xs:minOccurs=1
       - xs:ncName="tickerSymbol"
       - xs:nillable=false
       - xs:typeName="string"
       - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
   TradePrice jcr:primaryType=xs:elementDeclaration
    - jcr:uuid=5667cfcc-d87e-4ef3-811c-4e64dc27f263
    - xs:abstract=false
    - xs:form="qualified"
    - xs:namespace=http://example.com/stockquote.xsd
    - xs:ncName="TradePrice"
    - xs:nillable=false
    - xs:typeNamespace=http://example.com/stockquote.xsd
   xs:complexType jcr:primaryType=xs:complexTypeDefinition
    - jcr:uuid=b2eb5936-4a12-4d2f-854c-ca4b251c6a74
    - xs:abstract=false
    - xs:baseTypeName="anyType"
    - xs:baseTypeNamespace="http://www.w3.org/2001/XMLSchema"
    - xs:method="restriction"
    - xs:mixed=false
    - xs:namespace=http://example.com/stockquote.xsd
   xs:all jcr:primaryType=xs:all jcr:uuid=57d8f62f-71b1-44c7-8807-a1faac3582a4
    - xs:maxOccurs=1
    - xs:minOccurs=1
   price jcr:primaryType=xs:elementDeclaration
    - jcr:uuid=049a905c-1c1d-4122-aa2f-7d2fe7d45bef
    - id="float"
    - name="float"
    - xs:abstract=false
    - xs:form="qualified"
    - xs:maxOccurs=1
    - xs:minOccurs=1

```
            - xs:ncName="price"
            - xs:nillable=false
            - xs:typeName="float"
            - xs:typeNamespace=http://www.w3.org/2001/XMLSchema
  wsdl:messages jcr:primaryType=wsdl:messages
    - jcr:uuid=3ae584b3-2807-4022-b1fb-c7d39d0cfc48
    GetLastTradePriceInput jcr:primaryType=wsdl:message
      - jcr:uuid=6eac84de-e7e3-4e12-ac5e-d5a8dfe11c7f
      - wsdl:namespace=http://example.com/stockquote.wsdl
      - wsdl:ncName="GetLastTradePriceInput"
      body jcr:primaryType=wsdl:part
        - jcr:uuid=28d5bc74-f21c-49c2-9850-a9992cbbf88e
        - wsdl:elementName="TradePriceRequest"
        - wsdl:elementNamespace=http://example.com/stockquote.xsd
        - wsdl:ncName="body"
    GetLastTradePriceOutput jcr:primaryType=wsdl:message
      - jcr:uuid=1be232c8-898b-49ce-90c7-8e31b20f991f
      - wsdl:namespace=http://example.com/stockquote.wsdl
      - wsdl:ncName="GetLastTradePriceOutput"
      body jcr:primaryType=wsdl:part
        - jcr:uuid=06feaf78-f1ce-4f6c-a8e3-65eda3d600da
        - wsdl:elementName="TradePrice"
        - wsdl:elementNamespace=http://example.com/stockquote.xsd
        - wsdl:ncName="body"
  wsdl:portTypes jcr:primaryType=wsdl:portTypes
    - jcr:uuid=44afcb97-9b19-4dd0-98ca-191ca14495b2
    StockQuotePortType jcr:primaryType=wsdl:portType
      - jcr:uuid=3e81f0fd-7759-445a-b540-1253605ce0fd
      - wsdl:namespace=http://example.com/stockquote.wsdl
      - wsdl:ncName="StockQuotePortType"
      GetLastTradePrice jcr:primaryType=wsdl:operation
        - jcr:uuid=bd5d2f23-5454-4de2-9962-93c30b1be6d9
        - wsdl:ncName="GetLastTradePrice"
        wsdl:input jcr:primaryType=wsdl:operationInput
          - jcr:uuid=fba4398b-84c8-4ebe-8eb8-f83ce867329b
          - wsdl:message=6eac84de-e7e3-4e12-ac5e-d5a8dfe11c7f
          - wsdl:messageName="GetLastTradePriceInput"
          - wsdl:messageNamespace="http://example.com/stockquote.wsdl"
          - wsdl:ncName="GetLastTradePriceRequest"
        wsdl:output jcr:primaryType=wsdl:operationOutput
          - jcr:uuid=aa7a2ef8-883e-4598-a822-15283c0b63d4
          - wsdl:message=1be232c8-898b-49ce-90c7-8e31b20f991f
          - wsdl:messageName="GetLastTradePriceOutput"
          - wsdl:messageNamespace="http://example.com/stockquote.wsdl"
```

```
        - wsdl:ncName="GetLastTradePriceResponse"
  wsdl:bindings jcr:primaryType=wsdl:bindings
    - jcr:uuid=f736166e-cf40-45ec-b4a4-23243e241205
    StockQuoteBinding jcr:primaryType=wsdl:binding
      - jcr:uuid=b224c1f5-d223-483b-ab43-479ceef3e015
      - wsdl:namespace=http://example.com/stockquote.wsdl
      - wsdl:ncName="StockQuoteBinding"
    StockQuoteSoapBinding jcr:primaryType=wsdl:binding
      - jcr:uuid=cd65da16-bc97-479c-bb27-c9766ee5c946
      - wsdl:namespace=http://example.com/stockquote.wsdl
      - wsdl:ncName="StockQuoteSoapBinding"
      - wsdl:type=3e81f0fd-7759-445a-b540-1253605ce0fd
      - wsdl:typeName="StockQuotePortType"
      - wsdl:typeNamespace="http://example.com/stockquote.wsdl"
      GetLastTradePrice jcr:primaryType=wsdl:bindingOperation
        - jcr:uuid=949919a7-23c4-4994-853a-5a14b1fd04ed
        - wsdl:ncName="GetLastTradePrice"
        wsdl:input jcr:primaryType=wsdl:bindingOperationInput
          - jcr:uuid=76069b3a-c73e-4c23-be3d-b0ee6f874e7a
          - wsdl:input="fba4398b-84c8-4ebe-8eb8-f83ce867329b"
          - wsdl:inputName="GetLastTradePriceRequest"
          - wsdl:ncName="GetLastTradePriceRequest"
          wsdl:soapBody jcr:primaryType=wsdl:soapBody
            - jcr:uuid=22bd5f19-5450-4720-ab23-e4d97c8adee5
            - wsdl:use="literal"
        wsdl:output jcr:primaryType=wsdl:bindingOperationOutput
          - jcr:uuid=03b70411-d992-41db-ade1-de70ddd7822a
          - wsdl:ncName="GetLastTradePriceResponse"
          - wsdl:output="aa7a2ef8-883e-4598-a822-15283c0b63d4"
          - wsdl:outputName="GetLastTradePriceResponse"
          wsdl:soapBody jcr:primaryType=wsdl:soapBody
            - jcr:uuid=5d9d8127-8617-4947-b142-6d31e0b84c03
            - wsdl:use="literal"
        wsdl:soapOperation jcr:primaryType=wsdl:soapOperation
          - jcr:uuid=52ce3adf-b018-4148-a679-64822b870908
          - wsdl:soapAction=http://example.com/GetLastTradePrice
      wsdl:soapBinding jcr:primaryType=wsdl:soapBinding
        - jcr:uuid=659102a6-206e-4ebc-8d51-9b21e5dcc431
        - wsdl:style="document"
        - wsdl:transport=http://schemas.xmlsoap.org/soap/http
  wsdl:services jcr:primaryType=wsdl:services
    - jcr:uuid=3dbd2a54-9d2d-4223-98a2-8362369e8f0d
    StockQuoteService jcr:primaryType=wsdl:service
      - jcr:uuid=72420bcb-dd3f-4a5e-ba13-811af5a98bd5
```

```
    - sramp:description="My first service"
    - wsdl:namespace=http://example.com/stockquote.wsdl
    - wsdl:ncName="StockQuoteService"
    StockQuotePort jcr:primaryType=wsdl:port
      - jcr:uuid=24779c9f-ebe6-4030-b9cd-3f0e623b94fa
      - wsdl:binding=b224c1f5-d223-483b-ab43-479ceef3e015
      - wsdl:ncName="StockQuotePort"
    wsdl:soapAddress jcr:primaryType=wsdl:soapAddress
      - jcr:uuid=d015a2ee-fbae-4b28-bda8-16a8295d8e02
      - wsdl:soapLocation=http://example.com/stockquote
```

The first thing to note is that the sequencer produces a node of type `wsdl:wsdlDocument` that includes the `mode:derived` information (e.g., the time of sequencing and the path to the file from which this information was derived), and information about the WSDL file itself. If the WSDL file contained `documentation` elements directly under the root element, the content of those elements would have been placed inside an `sramp:description` property.

Secondly, the WSDL file contains an embedded XML Schema document, and this XSD was sequenced also. See the *XML Schema sequencer* documentation for the structure of the XML Schema documents. Any references to the XSD components in the embedded schema(s) will be captured as REFERENCE properties as well as properties containing the local name and namespace of the components.

Thirdly, there are several "container" nodes underneath the top-level `wsdl:wsdlDocument` node, and are named `wsdl:messages`, `wsdl:portTypes`, `wsdl:bindings`, and `wsdl:services`. These container nodes serve to separate out the various kinds of definitions, since per the WSDL 1.1 specification the name scope of each kind of component is distinct from the other kinds.

Within the `wsdl:messages` container node are all of the messages. In this case, there are two: the "`GetLastTradePriceInput`" input message and "`GetLastTradePriceOutput`" output message for the "`GetLastTracePrice`" operation defined a bit later in the structure. Note how these messages contain the name, namespace URI, and REFERENCE to the corresponding `element` node in the embedded schema content. (If the element reference could not be resolved, REFERENCE property would not be set.)

Within the `wsdl:portTypes` container node are all of the port types. In this example, there is just one: the "`StockQuotePortType`" that contains a single "`GetLastTradePrice`" operation. Here, the operation's input and output reference the corresponding message nodes vi the name, namespace URI, and REFERENCE property. Again, the REFERENCE property would not be set if the input and/or output use a message that is not in this WSDL file.

Within the `wsdl:bindings` container node are all of the bindings defined in the WSDL. In this example, there is just a single binding that uses SOAP extensions, which describe all of the SOAP-specific information for the port type. The sequencer also supports HTTP and MIME extensions. And node how the input, output and faults of each binding operation reference (using

the name, namespace URI, and REFERENCE properties) the corresponding input, output and fault (respectively) in the correct port type.

Finally, within the `wsdl:services` container node are all of the services defined in the WSDL. In this example, there is just a single SOAP service that references the "`StockQuotePortType`" port type.

This example shows the basic structure this sequencer derives from WSDL 1.1 files. Not only does this structure mirror that of the actual WSDL file, but it makes this structure easy to navigate, search and query, especially when it includes the names and namespace URIs of the referenced components (and setting REFERENCE properties to the referenced component where possible).

23.2. Node Types

The WSDL 1.1 sequencer follows JCR best-practices by defining all nodes to have a primary type that allows any single or multi-valued property, meaning it's possible and valid for any node to have any property (with single or multiple values). This sequencer doesn't add any such properties or nodes, but you are free to annotate the structure as needed.

The compact node definitions for the "wsdl" namespace are as follows:

```
<jcr='http://www.jcp.org/jcr/1.0'>
<nt='http://www.jcp.org/jcr/nt/1.0'>
<mix='http://www.jcp.org/jcr/mix/1.0'>
<sramp = "http://s-ramp.org/xmlns/2010/s-ramp">
<xs = "http://www.w3.org/2001/XMLSchema">
<wsdl = "http://schemas.xmlsoap.org/wsdl/">


//-----------------------------------------------------------------------------
// N O D E T Y P E S
//-----------------------------------------------------------------------------

[wsdl:wsdlExtension] > sramp:derivedArtifactType
- wsdl:ncName (string)
- wsdl:namespace (uri) mandatory

[wsdl:wsdlDerivedArtifactType] > sramp:derivedArtifactType abstract
- wsdl:namespace (uri) mandatory
+ * (wsdl:wsdlExtension)

[wsdl:namedWsdlDerivedArtifactType] > wsdl:wsdlDerivedArtifactType
- wsdl:ncName (string) mandatory

/*
```

```
   * Messages and parts
   */
[wsdl:part] > wsdl:namedWsdlDerivedArtifactType
- wsdl:element (reference) < 'xs:elementDeclaration'
- wsdl:elementName (string)
- wsdl:elementNamespace (uri)
- wsdl:type (reference) < 'xs:simpleTypeDefinition'
- wsdl:typeName (string)
- wsdl:typeNamespace (uri)

[wsdl:message] > wsdl:namedWsdlDerivedArtifactType
+ * (wsdl:part) = wsdl:part multiple

/*
 * Port types, operations, inputs, outputs, and faults
 */
[wsdl:operationInput] > wsdl:namedWsdlDerivedArtifactType
- wsdl:message (reference) mandatory < 'wsdl:message'

[wsdl:operationOutput] > wsdl:namedWsdlDerivedArtifactType
- wsdl:message (reference) mandatory < 'wsdl:message'

[wsdl:fault] > wsdl:namedWsdlDerivedArtifactType
- wsdl:message (reference) mandatory < 'wsdl:message'

[wsdl:operation] > wsdl:namedWsdlDerivedArtifactType
- wsdl:parameterOrder (string) multiple
+ wsdl:input (wsdl:operationInput) = wsdl:operationInput
+ wsdl:output (wsdl:operationOutput) = wsdl:operationOutput
+ wsdl:fault (wsdl:fault) = wsdl:fault sns

[wsdl:portType] > wsdl:namedWsdlDerivedArtifactType
+ * (wsdl:operation) sns

/*
 * Bindings, binding operations, inputs, outputs
 */
[wsdl:bindingOperationOutput] > wsdl:namedWsdlDerivedArtifactType
- wsdl:input (reference) < 'wsdl:operationInput'
- wsdl:inputName (string)

[wsdl:bindingOperationInput] > wsdl:namedWsdlDerivedArtifactType
- wsdl:output (reference) < 'wsdl:operationOutput'
- wsdl:outputName (string)
```

[wsdl:bindingOperationFault] > wsdl:namedWsdlDerivedArtifactType

[wsdl:bindingOperation] > wsdl:namedWsdlDerivedArtifactType
+ wsdl:input (wsdl:bindingOperationInput) = wsdl:bindingOperationInput
+ wsdl:output (wsdl:bindingOperationOutput) = wsdl:bindingOperationOutput
+ wsdl:fault (wsdl:bindingOperationFault) = wsdl:bindingOperationFault sns

[wsdl:binding] > wsdl:namedWsdlDerivedArtifactType
- wsdl:type (reference) < 'wsdl:portType'
+ * (wsdl:bindingOperation) sns

/*
 * Ports and services
 */
[wsdl:port] > wsdl:namedWsdlDerivedArtifactType
- wsdl:binding (reference) < 'wsdl:binding'
- wsdl:bindingName (string)
- wsdl:bindingNamespace (uri)

[wsdl:service] > wsdl:namedWsdlDerivedArtifactType
+ * (wsdl:port) sns

/*
 * Types, schemas, and schema references
 */
[wsdl:referencedXsd] > sramp:derivedArtifactType abstract
- xs:id (string)
- xs:schemaLocation (string)
- * (undefined) multiple
- * (undefined)

[wsdl:importedXsd] > wsdl:referencedXsd
- xs:namespace (uri) mandatory

[wsdl:includedXsd] > wsdl:referencedXsd

[wsdl:redefinedXsd] > wsdl:referencedXsd

/*
 * The containers for the different kinds of components within WSDL documents.
 * Strictly speaking, the containers should not allow SNS, but these components'
 * names in WSDL are QNames, and we're only using the local part for the node name.
 * Therefore, two components might have the same local part but different namespaces.

```
 * (This is probably not a common occurance.)
 */
[wsdl:container] > sramp:derivedArtifactType abstract
- * (string)
- * (string) multiple


[wsdl:messages] > wsdl:container
+ * (wsdl:message) = wsdl:message sns


[wsdl:portTypes] > wsdl:container
+ * (wsdl:portType) = wsdl:portType sns


[wsdl:bindings] > wsdl:container
+ * (wsdl:binding) = wsdl:binding sns


[wsdl:services] > wsdl:container
+ * (wsdl:service) = wsdl:service sns


/*
 * WSDL documents
 */
[wsdl:wsdlDocument] > sramp:xmlDocument
- wsdl:importedXsds (weakreference) multiple < 'xs:schemaDocument'
- wsdl:includedXsds (weakreference) multiple < 'xs:schemaDocument'
- wsdl:redefinedXsds (weakreference) multiple < 'xs:schemaDocument'
- wsdl:importedWsdls (weakreference) multiple < 'wsdl:wsdlDocument'
+ wsdl:schema (xs:schemaDocument) = xs:schemaDocument sns
+ wsdl:importedXsd (wsdl:importedXsd) sns
+ wsdl:includedXsd (wsdl:includedXsd) sns
+ wsdl:redefinedXsd (wsdl:redefinedXsd) sns
+ wsdl:messages (wsdl:messages) = wsdl:messages
+ wsdl:portTypes (wsdl:portTypes) = wsdl:portTypes
+ wsdl:bindings (wsdl:bindings) = wsdl:bindings
+ wsdl:services (wsdl:services) = wsdl:services


// -------------------------------------------------------
// HTTPWSDL Model
// -------------------------------------------------------


[wsdl:httpExtension] > wsdl:wsdlExtension


[wsdl:httpAddress] > wsdl:httpExtension
- wsdl:location (uri) mandatory
```

[wsdl:httpBinding] > wsdl:httpExtension
- wsdl:verb (string) mandatory

[wsdl:httpOperation] > wsdl:httpExtension
- wsdl:location (uri) mandatory

[wsdl:httpUrlEncoded] > wsdl:httpExtension

[wsdl:httpUrlReplacement] > wsdl:httpExtension

// ---
// SOAPWSDL Model
// ---

[wsdl:soapExtension] > wsdl:wsdlExtension

[wsdl:soapAddress] > wsdl:soapExtension
- wsdl:soapLocation (uri) mandatory

[wsdl:soapBinding] > wsdl:soapExtension
- wsdl:style (string)
- wsdl:transport (uri)

[wsdl:soapOperation] > wsdl:soapExtension
- wsdl:style (string)
- wsdl:soapAction (uri)

[wsdl:soapBody] > wsdl:soapExtension
- wsdl:encodingStyle (uri) multiple
- wsdl:parts (string)
- wsdl:use (string) < 'literal','encoded'

[wsdl:soapFault] > wsdl:soapExtension
- wsdl:encodingStyle (uri) multiple
- wsdl:use (string) < 'literal','encoded'

[wsdl:soapHeader] > wsdl:soapExtension
- wsdl:message (string)
- wsdl:part (string)
- wsdl:encodingStyle (uri) multiple
- wsdl:use (string) < 'literal','encoded'
+ * (wsdl:soapHeaderFault) = wsdl:soapHeaderFault

[wsdl:soapHeaderFault] > wsdl:soapExtension

```
- wsdl:encodingStyle (uri) multiple
- wsdl:use (string) < 'literal','encoded'


// -------------------------------------------------------
// SOAPMIME Model
// -------------------------------------------------------


[wsdl:mimeExtension] > wsdl:wsdlExtension


[wsdl:mimeMultipartRelated] > wsdl:mimeExtension
+ wsdl:mimePart (wsdl:mimePart) sns


[wsdl:mimePart] > wsdl:mimeExtension
+ * (wsdl:mimeExtension) sns


[wsdl:mimeContent] > wsdl:mimeExtension
- wsdl:mimeType (string)
- wsdl:mimePart (string)


[wsdl:mimeXml] > wsdl:mimeExtension
- wsdl:mimePart (string)
```

These types use some of the node types and mixins defined in the "sramp" namespace:

```
<jcr='http://www.jcp.org/jcr/1.0'>
<nt='http://www.jcp.org/jcr/nt/1.0'>
<mix='http://www.jcp.org/jcr/mix/1.0'>
<sramp = "http://s-ramp.org/xmlns/2010/s-ramp">


//-------------------------------------------------------------------------
// N O D E T Y P E S
//-------------------------------------------------------------------------


// -------------------------------------------------------
// S-RAMP Core Model Artifacts
// -------------------------------------------------------


[sramp:baseArtifactType] > mix:created, mix:lastModified, mix:referenceable, mix:versionable
 abstract mixin
- sramp:classifiedBy (reference) multiple < 'owl:class'
- sramp:description (string)
- * (string)
```

- * (string) multiple

[sramp:documentArtifactType] > sramp:baseArtifactType abstract mixin
- sramp:contentType (string)
- sramp:contentSize (long)

[sramp:xmlDocument] > sramp:documentArtifactType mixin
- sramp:contentEncoding (string) mandatory

[sramp:document] > sramp:documentArtifactType mixin

[sramp:derivedArtifactType] > sramp:baseArtifactType abstract mixin
- sramp:relatedDocuments (reference) < 'sramp:documentArtifactType'

[sramp:userDefinedArtifactType] > sramp:baseArtifactType mixin
- sramp:userType (string) mandatory

[sramp:storedQuery] > nt:query
- sramp:propertyList (string) multiple

[sramp:relatedTo] mixin
- * (weakreference) multiple

23.3. Configuration

To use this sequencer, simply include the appropriate version of the Maven artifact with a "`org.modeshape`" group ID and "`modeshape-sequencer-wsdl`" artifact ID. Or, if you're using JAR files and manually setting up the classpath for your application, use the "`modeshape-sequencer-wsdl-2.6.0.Beta2-jar-with-dependencies.jar`" file. Then, define a sequencing configuration in the ModeShape configuration, using something similar to:

```
<configuration xmlns:mode="http://www.modeshape.org/1.0"
        xmlns:jcr="http://www.jcp.org/jcr/1.0">
  <mode:sequencers>
      ...
    <mode:sequencer jcr:name="WSDL Sequencer"
            mode:classname="org.modeshape.sequencer.wsdl.WsdlSequencer">
      <mode:description>Sequences WSDL 1.1 files (e.g., *.wsdl) loaded into the
          repository under '/files', extracting the WSDL components as well as
          any XSD components contained within an embedded XSD.
      </mode:description>
      <!-- Note this path expression captures the path below '/files' (in the
```

```
                    'store' source and 'default' workspace only), including the filename,
                    and places the sequenced content under the same relative path below
                    '/sequenced/wsdl'. For example, if a WSDL file is uploaded to
                    '/files/my/favorites/Customers.wsdl', then the sequenced output will
                    be placed at the '/sequenced/wsdl/my/favorites/Customer.wsdl' node,
                    which will have a primary type of 'wsdl:wsdlDocument' and will
                    contain under it the nodes representing the WSDL components. Of
                    course, the path expression can be modified as needed; for example,
                    to exclude the filename extension, or to exclude the relative path.
                    -->
            <mode:pathExpression>/files(//)*.wsdl[*]/jcr:content[@jcr:data]
                        => /sequenced/wsdl/$1 </mode:pathExpression>
        </mode:sequencer>
        ...
    </mode:sequencers>
    ...
</configuration>
```

or using the *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html]:

```
JcrConfiguration config = ...

config.sequencer("WSDL Sequencer")
    .usingClass(WsdlSequencer.class)
    .setDescription("Sequences WSDL 1.1 files")
    .sequencingFrom("/files(//)(*.wsdl[*])/jcr:content[@jcr:data]")
    .andOutputtingTo("/sequenced/wsdl/$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.

24.1. Example

This sequencer generates a graph structure that maps to the files and folders in the ZIP file. An example (listed in the JCR document view) from sequencing a ZIP file written into /a/foo and containing one file, /x/y/z.txt is provided below:

```xml
<foo jcr:primaryType="zip:file"
    jcr:mixinTypes="mode:derived"
    mode:derivedAt="2011-05-13T13:12:03.925Z"
    mode:derivedFrom="/files/docForReferenceGuide.xml" >
  <x jcr:primaryType="nt:folder"
    jcr:created="2011-05-12T20:07Z"
    jcr:createdBy="currentJcrUser">
    <y jcr:primaryType="nt:folder"
      jcr:created="2011-05-12T20:09Z"
      jcr:createdBy="currentJcrUser">
      <z.txt jcr:primaryType="nt:file">
        <jcr:content jcr:primaryType="nt:resource"
                jcr:data="This is the file content"
                jcr:lastModified="2011-05-12T20:12Z"
                jcr:lastModifiedBy="currentJcrUser"
                jcr:mimeType="text/plain" />
      </z.txt>
    </y>
  </x>
</foo>
```

The CND for the zip:file node type is listed below.

```
[zip:file] > nt:folder, mix:mimeType
```

To use this sequencer, simply include the `modeshape-sequencer-zip` JAR in your application and configure the *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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]")
    .andOuputtingTo("/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.

25.1. Example

This sequencer generates a simple graph structure containing a variety of metadata from the Office document. The example below provides example output (in the JCR document view) from a Word document sequenced into /document.

```xml
<document jcr:primaryType="msoffice:metadata"
    jcr:mixinTypes="mode:derived"
    mode:derivedAt="2011-05-13T13:12:03.925Z"
    mode:derivedFrom="/files/docForReferenceGuide.xml"
    msoffice:title="My Word Document"
    msoffice:subject="My Subject"
    msoffice:author="James Joyce"
    msoffice:keywords="essay english term paper"
    msoffice:comment="This is my English 101 term paper"
    msoffice:template="term_paper.dot"
    msoffice:last_saved_by="jjoyce"
    msoffice:revision="42"
    msoffice:total_editing_time="1023"
    msoffice:last_printed="2011-05-12T14:33Z"
    msoffice:created="2011-05-10T20:07Z"
    msoffice:saved="2011-05-12T14:32Z"
    msoffice:pages="14"
    msoffice:words="3025"
    msoffice:characters="12420"
    msoffice:creating_application="MSWORD.EXE"
    msoffice:thumbnail="..." />
```

As indicated in the CND below, sequencing Excel spreadsheets also populates the `msoffice:full_content` property with all text in the document and the `msoffice:sheets` multi-valued string property with one value for each worksheet name. Sequencing PowerPoint presentations adds a child node for each slide containing the title (`msoffice:title`), slide text (`msoffice:text`), and thumbnail image (`msoffice:thumbnail`) for each slide.

```
[msoffice:metadata] > nt:unstructured, mix:mimeType
  - msoffice:title (string)
  - msoffice:subject (string)
  - msoffice:author (string)
  - msoffice:keywords (string)
  - msoffice:comment (string)
  - msoffice:template (string)
  - msoffice:last_saved_by (string)
  - msoffice:revision (string)
  - msoffice:total_editing_time (long)
  - msoffice:last_printed (date)
  - msoffice:created (date)
  - msoffice:saved (date)
  - msoffice:pages (long)
  - msoffice:words (long)
  - msoffice:characters (long)
  - msoffice:creating_application (string)
  - msoffice:thumbnail (binary)

// PowerPoint specific data
  + msoffice:slide (msoffice:pptslide) sns

// Excel specific data
  - msoffice:full_content (string)
  - msoffice:sheet_name (string) multiple

[msoffice:pptslide]
  - msoffice:title (string)
  - msoffice:text (string)
  - msoffice:thumbnail (binary)
```

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/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/api/org/ modeshape/sequencer/java/SourceFileRecorder.html] implementation to use to map the extracted metadata to an output location:

Table 26.1. *JavaMetadataSequencer* **[http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/java/ JavaMetadataSequencer.html] properties**

Property	Description
sourceFileRecorder	Optional property that, if set, provides an instance of the *SourceFileRecorder* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/ 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 class file recorder (called *ClassSourceFileRecorder* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/java/ClassSourceFileRecorder.html]) is used when these properties are not set, and creates a subgraph rooted at the output location that takes the following form:

```
<nt:unstructured jcr:name="packageName1"
          jcr:mixinTypes = "mode:derived"
          mode:derivedAt="2011-05-13T13:12:03.925Z"
          mode:derivedFrom="/files/org/modeshape/Foo.java">
  ...
  <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>
```

This is the *same structure* that is produced by the *Java class file sequencer*, meaning that by default the same structure will be produced when sequencing Java source or class files.

The compact node definitions for the class:* types is provided below:

```
[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

> **Note**
>
> This sequencer defaulted to using a different recorder implementation in ModeShape 1.x, but this earlier structure did not match that produced by the `ClassFileSequencer` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/classfile/ClassFileSequencer.html] and a different default recorder is used in ModeShape 2.0 (or later). The sequencer can be configured to use the original structure

by using the `OriginalFormatSourceFileRecorder` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/java/OriginalFormatSourceFileRecorder.html] class.

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/2.6.0.Beta2/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/2.6.0.Beta2/ api/org/modeshape/sequencer/classfile/ClassFileRecorder.html] interface. A default implementation (*DefaultClassFileRecorder* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/ 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/2.6.0.Beta2/api/org/modeshape/ sequencer/classfile/ClassFileRecorder.html] implementation to use to map the extracted metadata to an output location:

Table 27.1. *ClassFileSequencer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/sequencer/classfile/ClassFileSequencer.html] properties

Property	Description
classFileRecorder	Optional property that, if set, provides an instance of the *ClassFileRecorder* [http:// docs.jboss.org/modeshape/2.6.0.Beta2/ 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/2.6.0.Beta2/ 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"
        jcr:mixinTypes = mode:derived
        mode:derivedAt="2011-05-13T13:12:03.925Z"
        mode:derivedFrom="/files/org/modeshape/Foo.class">
  ...
  <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/2.6.0.Beta2/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]")
```

```
        .andOuputtingTo("/classes");
```

Image Sequencer

The _ImageMetadataSequencer_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/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:mixinTypes** - "mode:derived"

  - **mode:derivedAt** - the date that at which content was sequenced to produce this record

  - **mode:derivedFrom** - the repository path to the content that was sequenced

  - **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/2.6.0.Beta2/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.

29.1. Example

This sequencer generates a node with the name `mp3:metadata` below the sequencing target. That is if the `ouputtingTo` property from the sequencer configuration is /mp3s/LivinOnAPrayer, the sequencer will output a node to /mp3s/LivinOnAPrayer/mp3:metadata.

```
<mp3:metadata jcr:primaryType="mp3:metadata"
        jcr:mixinTypes="mode:derived"
        mode:derivedAt="2011-05-13T13:12:03.925Z"
        mode:derivedFrom="/files/LOP.mp3"
        mp3:title="Livin' on a Prayer"
        mp3:author="Bon Jovi"
        mp3:album="Slippery When Wet"
        mp3:year="1986"
        mp3:comment="Rock 'n' roll!" />
```

The CND used by this sequencer is provided below.

```
[mp3:metadata] > nt:unstructured, mix:mimeType
  - mp3:title (string)
  - mp3:author (string)
  - mp3:album (string)
  - mp3:year (long)
  - mp3:comment (string)
```

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/2.6.0.Beta2/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/sequencer/classfile/DefaultClassFileRecorder.html] may change in future versions.

30.1. 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"
          jcr:mixinTypes = "mode:derived"
          mode:derivedAt="2011-05-13T13:12:03.925Z"
          mode:derivedFrom="/files/foo.sql"
          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/2.6.0.Beta2/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/2.6.0.Beta2/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/javase/6/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/2.6.0.Beta2/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 31.1. *AbstractTextSequencer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/text/AbstractTextSequencer.html] 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/2.6.0.Beta2/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 jcr:mixinTypes = mode:derived,
        mode:derivedAt="2011-05-13T13:12:03.925Z",
        mode:derivedFrom="/files/foo.dat">
   + 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>)
```

31.1. Delimited Text Sequencer

The *DelimitedTextSequencer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/sequencer/text/DelimitedTextSequencer] class provides an additional JavaBean property to override the default regular expression pattern:

Table 31.2. *DelimitedTextSequencer* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/text/ DelimitedTextSequencer] properties

Property	Description
splitPattern	Optional property that, if set, sets the regular expression pattern that is used to split each

Property	Description
	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/2.6.0.Beta2/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");
```

31.2. Fixed Width Text Sequencer

The *FixedWidthTextSequencer* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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 31.3. *FixedWidthTextSequencer* **[http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/text/FixedWidthTextSequencer] 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/2.6.0.Beta2/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");
```

Teiid Relational Model Sequencer

Teiid Designer [http://www.jboss.org/teiiddesigner], is a visual tool that enables rapid, model-driven definition, integration, management and testing of data services without programming using the Teiid runtime engine. It is capable of modeling several different kinds of data structures, but the most common and widely-used are relational models that describe a relational database schema, including the catalogs/schemas, tables, views, columns, primary keys, foreign keys, indexes, procedures, procedure results, procedure results, and logical relationships. Teiid Designer can reverse-engineer a relational model from a JDBC relational database or DDL file. It can also define "virtual" models that are transformations of other models (where the transformations are defined in terms of SQL select, insert, update, and delete statements). These models can then be packaged into a *virtual database*, which can be deployed to a Teiid runtime engine.

Teiid [http://www.jboss.org/teiid] is a high-performance database virtualization engine that allows JDBC and ODBC client applications access the virtual database as if it were a real database, using relational, XML, XQuery and procedural queries. Teiid dynamically (and in real-time) figures out how to answer the queries and operations issued by clients by efficiently accessing and manipulating the data inside the underlying data sources. Teiid's sophisticated engine is able to plan and optimize these operations, even when multiple heterogeneous relational and non-relational data sources must be accessed to obtain the required information.

The Teiid relational model sequencer parses the model files produced by the Teiid Designer, and extracts the structured relational data model described by the XMI file. This means that when these models are uploaded into a ModeShape repository, the sequencer writes to the repository all this relational metadata, where it can be queried and accessed by JCR, RESTful, and even JDBC clients.

The `ModelSequencer` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/teiid/ModelSequencer.html] has a single JavaBean properties for changing behavior:

Table 32.1. `ClassFileSequencer` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/classfile/ClassFileSequencer.html] properties

Property	Description
useXmiUuidsAsJcrUuids	Optional property that, if set to 'true', reuses the model's "`xmi:uuid`" values as the generated nodes' "`jcr:uuid`" identifiers. In such cases, a model may only appear in the repository once, so even those a model might be uploaded into the repository multiple times, each time the model is sequenced the newly generated output will overwrite any output from previous sequencing operations.

Property	Description
	If set the 'false', a model (or different versions of a model) can be uploaded into the repository multiple times, where they all can be accessed at any time. The default value for this property is 'false'; changing it may have drastic ramifications and is suggested only for advanced users.

32.1. UUIDs

As mentioned above, the Teiid model sequencer can operate in two modes. The behavior you choose will dramatically change what you can do with the sequenced relational models.

The first mode reuses the "`xmi:uuid`" identifiers on each object in the model as the "`jcr:uuid`" node identifiers. In this mode, the sequencer represents each model reference as a JCR WEAKREFERENCE, making it very easy to navigate and query relationships. However, there is one major disadvantage of this approach: each time a model is uploaded into the repository, the sequencer will override any output generated by earlier sequencing operations upon that file (or other versions of it). Thus, the sequenced representation of an uploaded model can ever appear only once within the repository, even though different versions of that model might exist in the repository at different locations. This may be desirable in some situations, but for most situations it is simply not acceptable.

In the second mode of operation (which is the default mode), there is no correlation between the model's "`xmi:uuid`" and "`jcr:uuid`" node identifiers. Various versions of a given model can be uploaded into the repository at multiple locations, yet each model's relational schema will exist in the repository. The downside of this approach is that references are no longer simply WEAKREFERENCE properties. Instead, each single-valued reference will be represented as a series of four properties:

- `{referenceName}Href` - stores the href literal value from the XMI file; this is always set

- `{referenceName}XmiUuid` - stores the XMI UUID to the referenced node; this is set only if the href had an embedded UUID (hrefs to data types and XSD components don't use UUIDs)

- `{referenceName}Name` - stores the name of the resolved node, though this may not be set if the object being referenced is in another model

- `{referenceName}` - stores the JCR weak reference to the resolve node, though this may not be set if the object being referenced is in another model

where "*{referenceName}*" is the name of the model reference. Multi-value references are also represented as a series of four properties, but with a slightly different naming pattern:

- `{singularReferenceName}Href` - stores the href literal values from the XMI file; this is always set

- `{singularReferenceName}XmiUuid` - stores the XMI UUID to the referenced nodes; this is set only if the hrefs have an embedded UUID (hrefs to data types and XSD components don't use UUIDs)

- `{singularReferenceName}Name` - stores the name of the resolved nodes, though this may not be set if the object being referenced is in another model

- `{pluralReferenceName}` - stores the JCR weak reference to the resolve nodes, though this may not be set if the object being referenced is in another model

Here, "*{singularReferenceName}*" is the singular form of the model reference name, and "*{pluralReferenceName}*" is the plural form of the model reference name. For example, for a reference named "columns", the "*{singularReferenceName}*" value would be "column" and the plural form is "columns". If the reference name is "properties", the singular form is "property" and the plural form is "properties". (ModeShape uses a novel algorithm to determine the singular and plural forms of many English words.)

References to model objects within the same model are easily resolved upon sequencing, and so we set all of the properties (regardless of the mode). However, references to objects in other models cannot be resolved at sequencing time.

> **Note**
>
> The *Teiid VDB sequencer* behavior is unrelated to this mode, since it always sequences models with new "`jcr:uuid`" identifiers that are unrelated to the "`xmi:uuid`" values. In this manner, each sequencing of a VDB will produce the relational model representation for each model in the VDB (with all valid references resolved between all models), independent of any generated output from the Teiid model sequencer.

32.2. Node Types

The model sequencer follows JCR best-practices by defining all nodes to have a primary type of "`nt:unstructured`" (or a node type that extends "`nt:unstructured`"), meaning it's possible and valid for any node to have any property (with single or multiple values). However, it is still useful to capture the metadata about what that node represents, and so the sequencer use mixins for this. For example, there is a "`xmi:referenceable`" mixin with a single "`xmi:uuid`" property (patterned after the built-in "`mix:referenceable`" mixin). Since all model objects have mmuuids, all nodes produced by this sequencer will have this mixin.

The rest of this section covers the various (and many!) node types defined for and used by this sequencer. Note that these are non-normative definitions of the node types; see the CND files in the "`modeshape-sequencer-teiid`" JAR file (or source) for the official definitions.

32.2.1. XMI Namespace

The compact node definitions for the "xmi" namespace are as follows:

```
<nt  = "http://www.jcp.org/jcr/nt/1.0">
<xmi = "http://www.omg.org/XMI">


//-------------------------------------------------------------------------------
// N O D E T Y P E S
//-------------------------------------------------------------------------------


[xmi:referenceable] mixin
 - xmi:uuid (string) mandatory


[xmi:model] > nt:unstructured, xmi:referenceable orderable
 - xmi:version (double) = '2.0'
```

32.2.2. Core Namespace

The compact node definitions for the "mmcore" namespace are as follows:

```
<nt  = "http://www.jcp.org/jcr/nt/1.0">
<xmi = "http://www.omg.org/XMI">
<mmcore = "http://www.metamatrix.com/metamodels/Core">
<mode = "http://www.modeshape.org/1.0">

[mmcore:model] > xmi:referenceable, mode:hashed mixin
  - mmcore:modelType (string) = 'UNKNOWN' < 'PHYSICAL','VIRTUAL','TYPE','VDB_ARCHIVE',
                        'UNKNOWN','FUNCTION','CONFIGURATION','METAMODEL',
                        'EXTENSION','LOGICAL','MATERIALIZATION'
  - mmcore:primaryMetamodelUri (string)
  - mmcore:description (string)
  - mmcore:nameInSource (string)
  - mmcore:maxSetSize (long) = '100'
  - mmcore:visible (boolean) = 'true'
  - mmcore:supportsDistinct (boolean) = 'true'
  - mmcore:supportsJoin (boolean) = 'true'
  - mmcore:supportsOrderBy (boolean) = 'true'
  - mmcore:supportsOuterJoin (boolean) = 'true'
  - mmcore:supportsWhereAll (boolean) = 'true'
  - mmcore:supportsDistinct (boolean) = 'true'
  - mmcore:producerName (string)
  - mmcore:producerVersion (string)
  - mmcore:originalFile (string)
```

```
  - mmcore:sha1 (string)


[mmcore:import] > nt:unstructured, xmi:referenceable orderable
  - mmcore:modelType (string) = 'UNKNOWN' < 'PHYSICAL','VIRTUAL','TYPE','VDB_ARCHIVE',
                                'UNKNOWN','FUNCTION','CONFIGURATION','METAMODEL',
                                'EXTENSION','LOGICAL','MATERIALIZATION'
  - mmcore:primaryMetamodelUri (string)
  - mmcore:path (string)
  - mmcore:name (string)
  - mmcore:modelLocation (string)


[mmcore:annotated] mixin
  - mmcore:description (string)
  - mmcore:keywords (string) multiple


[mmcore:tags] mixin
  - * (undefined) multiple
  - * (undefined)
```

32.2.3. Relational Namespace

The compact node definitions for the "relational" namespace are as follows:

```
<nt  = "http://www.jcp.org/jcr/nt/1.0">
<relational='http://www.metamatrix.com/metamodels/Relational'>
<xmi = "http://www.omg.org/XMI">


//----------------------------------------------------------------------------
// N O D E T Y P E S
//----------------------------------------------------------------------------


[relational:relationalEntity] > xmi:referenceable abstract mixin
 - relational:nameInSource (string)


[relational:relationship] > nt:unstructured, relational:relationalEntity abstract


// ------------------------------------------
// Columns and Column Sets
// ------------------------------------------


[relational:column] > nt:unstructured, relational:relationalEntity
 - relational:nativeType (string)
```

- relational:type (weakreference)
- relational:typeHref (string)
- relational:typeXmiUuid (string)
- relational:typeName (string)
- relational:length (long)
- relational:fixedLength (boolean)
- relational:precision (long)
- relational:scale (long)
-   relational:nullable   (string)   =   'NULLABLE'   <   'NO_NULLS',   'NULLABLE', 'NULLABLE_UNKNOWN'
- relational:autoIncremented (boolean) = 'false'
- relational:defaultValue (string)
- relational:minimumValue (string)
- relational:maximumValue (string)
- relational:format (string)
- relational:characterSetName (string)
- relational:collationName (string)
- relational:selectable (boolean) = 'true'
- relational:updateable (boolean) = 'true'
- relational:caseSensitive (boolean) = 'true'
- relational:searchability (string) = 'SEARCHABLE' < 'SEARCHABLE',
                         'ALL_EXCEPT_LIKE', 'LIKE_ONLY', 'UNSEARCHABLE'
- relational:currency (boolean) = 'false'
- relational:radix (long) = '10'
- relational:signed (boolean) = 'true'
- relational:distinctValueCount (long) = '-1'
- relational:nullValueCount (long) = '-1'
- relational:uniqueKeys (weakreference) multiple
- relational:uniqueKeyHrefs (string) multiple
- relational:uniqueKeyXmiUuids (string) multiple
- relational:uniqueKeyNames (string) multiple
- relational:indexes (weakreference) multiple
- relational:indexHrefs (string) multiple
- relational:indexXmiUuids (string) multiple
- relational:indexNames (string) multiple
- relational:foreignKeys (weakreference) multiple
- relational:foreignKeyHrefs (string) multiple
- relational:foreignKeyXmiUuids (string) multiple
- relational:foreignKeyNames (string) multiple
- relational:accessPatterns (weakreference) multiple
- relational:accessPatternHrefs (string) multiple
- relational:accessPatternXmiUuids (string) multiple
- relational:accessPatternNames (string) multiple

```
[relational:columnSet] > nt:unstructured, relational:relationalEntity abstract orderable
 + * (relational:column) = relational:column copy


// -------------------------------------------
// Constraints
// -------------------------------------------


[relational:uniqueKey] > nt:unstructured, relational:relationalEntity abstract
 - relational:columns (weakreference) multiple
 - relational:columnXmiUuids (string) multiple
 - relational:columnNames (string) multiple
 - relational:foreignKeys (weakreference) multiple
 - relational:foreignKeyHrefs (string) multiple
 - relational:foreignKeyXmiUuids (string) multiple
 - relational:foreignKeyNames (string) multiple

[relational:uniqueConstraint] > relational:uniqueKey


[relational:primaryKey] > relational:uniqueKey


[relational:foreignKey] > relational:relationship
 - relational:foreignKeyMultiplicity (string) = 'ZERO_TO_MANY' < 'ONE', 'MANY',
                              'ZERO_TO_ONE', 'ZERO_TO_MANY', 'UNSPECIFIED'
 - relational:primaryKeyMultiplicity (string) = 'ONE' < 'ONE', 'MANY', 'ZERO_TO_ONE',
                              'ZERO_TO_MANY', 'UNSPECIFIED'
 - relational:columns (weakreference) multiple
 - relational:columnXmiUuids (string) multiple
 - relational:columnNames (string) multiple
 - relational:uniqueKeys (weakreference) multiple
 - relational:uniqueKeyHrefs (string) multiple
 - relational:uniqueKeyXmiUuids (string) multiple
 - relational:uniqueKeyNames (string) multiple

[relational:index] > nt:unstructured, relational:relationalEntity
 - relational:filterCondition (string)
 - relational:nullable (boolean) = 'true'
 - relational:autoUpdate (boolean)
 - relational:unique (boolean)
 - relational:columns (weakreference) multiple
 - relational:columnXmiUuids (string) multiple
 - relational:columnNames (string) multiple

[relational:accessPattern] > nt:unstructured, relational:relationalEntity orderable
 - relational:columns (UNDEFINED) multiple
```

```
// -------------------------------------------
// Tables and Views
// -------------------------------------------


[relational:table] > relational:columnSet abstract orderable
 - relational:system (boolean) = 'false'
 - relational:cardinality (long)
 - relational:supportsUpdate (boolean) = 'true'
 - relational:materialized (boolean) = 'false'
 - relational:logicalRelationships (weakreference) multiple
 - relational:logicalRelationshipHrefs (string) multiple
 - relational:logicalRelationshipXmiUuids (string) multiple
 - relational:logicalRelationshipNames (string) multiple
 + * (relational:primaryKey) = relational:primaryKey copy
 + * (relational:foreignKey) = relational:foreignKey copy
 + * (relational:accessPattern) = relational:accessPattern copy sns

[relational:baseTable] > relational:table orderable

[relational:view] > relational:table orderable


// -------------------------------------------
// Procedures
// -------------------------------------------


[relational:procedureParameter] > nt:unstructured, relational:relationalEntity
 - relational:direction (string) < 'IN', 'OUT', 'INOUT', 'RETURN', 'UNKNOWN'
 - relational:defaultValue (string)
 - relational:nativeType (string)
 - relational:type (weakreference)
 - relational:typeXmiUuid (string)
 - relational:typeName (string)
 - relational:length (long)
 - relational:precision (long)
 - relational:scale (long)
    -    relational:nullable    (string)    =    'NULLABLE'    <    'NO_NULLS',    'NULLABLE',
 'NULLABLE_UNKNOWN'
 - relational:radix (long) = '10'

[relational:procedureResult] > relational:columnSet orderable

[relational;procedure] > nt:unstructured, relational:relationalEntity orderable
```

```
 - relational:function (boolean)
 - relational:updateCount (string) < 'AUTO', 'ZERO', 'ONE', 'MULTIPLE'
 + * (relational:procedureParameter) = relational:procedureParameter copy sns
 + * (relational:procedureResult) = relational:procedureResult copy



// -----------------------------------------
// Logical Relationships
// -----------------------------------------

[relational:logicalRelationshipEnd] > nt:unstructured, relational:relationalEntity
  -  relational:multiplicity  (string)  <  'ONE',  'MANY',  'ZERO_TO_ONE',  'ZERO_TO_MANY',
 'UNSPECIFIED'
 - relational:table (weakreference)
 - relational:tableHref (string)
 - relational:tableXmiUuid (string)
 - relational:tableName (string)


[relational:logicalRelationship] > relational:relationship orderable
 + * (relational:logicalRelationshipEnd) = relational:logicalRelationshipEnd copy sns


// -----------------------------------------
// Catalogs and Schemas
// -----------------------------------------

[relational:schema] > nt:unstructured, relational:relationalEntity orderable
 + * (relational:table) = relational:baseTable copy
 + * (relational:procedure) = relational:procedure copy sns
 + * (relational:index) = relational:index copy
 + * (relational:logicalRelationship) = relational:logicalRelationship copy

[relational:catalog] > nt:unstructured, relational:relationalEntity orderable
 + * (relational:schema) = relational:schema copy
 + * (relational:table) = relational:baseTable copy
 + * (relational:procedure) = relational:procedure copy sns
 + * (relational:index) = relational:index copy
 + * (relational:logicalRelationship) = relational:logicalRelationship copy
```

32.2.4. JDBC Source Namespace

The compact node definitions for the "jdbcs" namespace are as follows:

```
<nt  = "http://www.jcp.org/jcr/nt/1.0">
<xmi = "http://www.omg.org/XMI">
<jdbcs = "http://www.metamatrix.com/metamodels/JDBC">


//-----------------------------------------------------------------------------
// N O D E T Y P E S
//-----------------------------------------------------------------------------


[jdbcs:source] > nt:unstructured, xmi:referenceable
 - jdbcs:name (string)
 - jdbcs:driverName (string)
 - jdbcs:driverClass (string)
 - jdbcs:username (string)
 - jdbcs:url (string)

[jdbcs:imported] > nt:unstructured, xmi:referenceable
 - jdbcs:createCatalogsInModel (boolean) = 'true'
 - jdbcs:createSchemasInModel (boolean) = 'true'
 - jdbcs:convertCaseInModel (string) < 'NONE', 'TO_UPPERCASE', 'TO_LOWERCASE'
 - jdbcs:generateSourceNamesInModel (string) = 'UNQUALIFIED' < 'NONE', 'UNQUALIFIED',
'FULLY_QUALIFIED'
 - jdbcs:includedCatalogPaths (string) multiple
 - jdbcs:includedSchemaPaths (string) multiple
 - jdbcs:excludedObjectPaths (string) multiple
 - jdbcs:includeForeignKeys (boolean) = 'true'
 - jdbcs:includeIndexes (boolean) = 'true'
 - jdbcs:includeProcedures (boolean) = 'false'
 - jdbcs:includeApproximateIndexes (boolean) = 'true'
 - jdbcs:includeUniqueIndexes (boolean) = 'false'
 - jdbcs:includedTableTypes (string) multiple
```

32.2.5. Transformation Namespace

The compact node definitions for the "transformation" namespace are as follows:

```
<transform='http://www.metamatrix.com/metamodels/Transformation'>


//-----------------------------------------------------------------------------
// N O D E T Y P E S
//-----------------------------------------------------------------------------

[transform:transformed] mixin
```

```
 - transform:transformedFrom (weakreference)
 - transform:transformedFromHrefs (string)
 - transform:transformedFromXmiUuids (string)
 - transform:transformedFromNames (string)

[transform:withSql] mixin
 - transform:selectSql (string)
 - transform:insertSql (string)
 - transform:updateSql (string)
 - transform:deleteSql (string)
 - transform:insertAllowed (boolean) = 'true'
 - transform:updateAllowed (boolean) = 'true'
 - transform:deleteAllowed (boolean) = 'true'
 - transform:outputLocked (boolean) = 'false'
 - transform:insertSqlDefault (boolean) = 'true'
 - transform:updateSqlDefault (boolean) = 'true'
 - transform:deleteSqlDefault (boolean) = 'true'
```

32.3. Default values

Teiid Designer does not persist default values in the XMI files. The sequencer knows these default values, and includes them in the sequenced output so that they can be accessed and queried.

32.4. Annotations

Rather than creating a separate "Annotation" object like what exist in the XMI models, the annotation's description and keywords are simply recorded as a "`mmcore:description`" and "`mmcore:keywords`" properties on the node created for the target of the annotation. This is really nice, because if a description is placed on a relational column object in a model, then that description appears as a property directly on the corresponding "`relational:column`" node. Note that when any annotation properties are placed on a node, the "`mmcore:annotated`" mixin is added to that node.

32.5. Tags

Tags are also stored on "Annotation" objects, and each tag consist of a key-value pair. The sequencer does two things depending upon what the key looks like. When the key is a simple string without a ':', then a property is created on the annotation's target object using this string as the property name and the tag's value as the property's value. More recently Teiid Designer has started to use tags with keys of the form "`namespace:name`", where "`namespace`" is really informal and can theoretically be any string value. While this format is the same as JCR property names, treating them as namespaced JCR property names would require there be a namespace URI registered with the prefix matching the "namespace" value.

The sequencer tries to parse the tag key as a property name, and if it works then the tag is added as a property just as mentioned earlier. However, if the namespace does not exist, then the sequencer splits the key into the two parts, where the first is used to identify a child node and the second is used as a property name.

For example, a tag on the "ID" column object under the "MyTable" base table:

```
foo="bar"
```

will be stored as a property "`foo`" with value "`bar`" on the "`MyTable/ID`" node. However, the

```
connection:driver-class="oracle.jdbc.OracleDriver"
```

tag on the same object would be stored as the "`driver-class`" property (with value "`oracle.jdbc.OracleDriver`") on the "`MyTable/ID/connection`" object.

32.6. Transformation

The transformation information, like with annotations, is projected onto the nodes representing the model objects that are the "output" of the transformation, where the objects that are "inputs" to the transformation are recorded as a (potentially multi-valued) property on the "output" object, and the "`transform:transformed`" mixin is added to the output node. In other words, virtual base tables, columns, procedures, etc., are marked as "transform:transformed" and have an "input" property pointing to the node(s) that are the inputs for the transformation. The SQL statements, supports flags, and defaults flags are also added as properties on the output virtual base table and procedures, and the "`transform:withSql`" mixin that defines these properties is added to that output node.

32.7. Configuration

To use this sequencer, simply include the `modeshape-sequencer-teiid-2.6.0.Beta2.jar` file in your application and define a sequencing configuration in the ModeShape configuration it using something similar to:

```
<configuration xmlns:mode="http://www.modeshape.org/1.0" xmlns:jcr="http://www.jcp.org/jcr/1.0">
   <mode:sequencers>
      ...
                                    <mode:sequencer jcr:name="Teiid Model
 Sequencer" mode:classname="org.modeshape.sequencer.teiid.ModelSequencer">
```

```
<mode:description>Sequences Teiid relational models (e.g., *.xmi) loaded into the repository under '/
files', extracting the structure defined in the models.</mode:description>
        <!-- Note this path expression captures the path below '/files' but excludes the filename,
 and places the sequenced
            content under the same relative path below '/sequenced/teiid/models'. For example,
 if an XMI model is uploaded
          to '/files/my/favorites/CustomerDetails.xmi', then the sequenced output will be placed at
            the '/sequenced/teiid/models/CustomerDetails' node, which will have a primary type
 of 'xmi:model' and will
            contain under it the nodes representing the catalogs, schemas, tables, views, columns,
 etc. Of course, the
              path expression can be modified as needed; for example, to include the filename
 of the XMI model in the
            sequenced output path. -->
            <mode:pathExpression>/files(//)(*.xmi[*])/jcr:content[@jcr:data] => /sequenced/teiid/
models$1 </mode:pathExpression>
    </mode:sequencer>
    ...
  </mode:sequencers>
  ...
</configuration>
```

or using the `JcrConfiguration` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html]:

```
JcrConfiguration config = ...

config.sequencer("Teiid Model Sequencer")
    .usingClass(ModelSequencer.class)
    .setDescription("Sequences Teiid relational models")
    .sequencingFrom("/files(//)(*.xmi[*])/jcr:content[@jcr:data]")
    .andOutputtingTo("/sequenced/teiid/models$1");
```

32.8. Example

Here is a representation of the nodes output by the sequencing of an example virtual relational model:

```
PartsVirtual jcr:primaryType="xmi:model"
```

- jcr:mixinTypes=["mmcore:model","mix:referenceable","xmi:referenceable", "mode:derived"]
- mode:derivedAt="2011-05-13T13:12:03.925Z"
- mode:derivedFrom="/files/foo.xmi"
- jcr:uuid="d1a1b82f-055b-4db2-a3e7-a9668f3a70b6"
- mmcore:maxSetSize="100"
- mmcore:modelType="VIRTUAL"
- mmcore:originalFile="/model/parts/PartsVirtual.xmi"
- mmcore:primaryMetamodelUri="http://www.metamatrix.com/metamodels/Relational"
- mmcore:producerName="Teiid Designer"
- mmcore:producerVersion="6.0"
- mode:sha1="84a77940f9140a358861d12d4bbb4160afadc08c"
- mmcore:supportsDistinct="true"
- mmcore:supportsJoin="true"
- mmcore:supportsOrderBy="true"
- mmcore:supportsOuterJoin="true"
- mmcore:supportsWhereAll="true"
- xmi:uuid="fb52cb80-128a-1eec-8518-c32201e76066"
- xmi:version="2.0"
- mmcore:visible="true"
PartSupplier_SourceB jcr:primaryType="mmcore:import"
  - jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]
  - jcr:uuid="c3a98bf2-7dbf-4c46-8baa-bf32e389cddd"
  - mmcore:modelType="PHYSICAL"
  - mmcore:primaryMetamodelUri="http://www.metamatrix.com/metamodels/Relational"
  - xmi:uuid="mmuuid:980de782-b1e5-1f55-853c-ed5dfdd1bb78"
PartsSupplier_SourceA jcr:primaryType="mmcore:import"
  - jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]
  - jcr:uuid="55385418-01c9-4d5c-9f79-91b8e10c6946"
  - mmcore:modelType="PHYSICAL"
  - mmcore:primaryMetamodelUri="http://www.metamatrix.com/metamodels/Relational"
  - xmi:uuid="mmuuid:980de784-b1e5-1f55-853c-ed5dfdd1bb78"
XMLSchema jcr:primaryType="mmcore:import"
  - jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]
  - jcr:uuid="8b5c2268-0770-405b-a4d8-12a868cc27a4"
  - mmcore:modelType="PHYSICAL"
  - mmcore:primaryMetamodelUri="http://www.eclipse.org/xsd/2002/XSD"
  - xmi:uuid="mmuuid:a6591280-bf1d-1f2c-9911-b53abd16b14e"
SupplierInfo jcr:primaryType="relational:baseTable"
  - jcr:mixinTypes=["transform:transformed","mix:referenceable","xmi:referenceable"]
  - jcr:uuid="37bf368e-0618-4f2f-b4c2-2ab4c0729502"
  - transform:deleteAllowed="true"
  - transform:deleteSqlDefault="true"
    - transform:inputHrefs="PartsSupplier_SourceA.xmi#mmuuid/bc400080-1284-1eec-8518-c32201e76066"

- transform:inputXmiUuids="bc400080-1284-1eec-8518-c32201e76066"
- transform:insertAllowed="true"
- transform:insertSqlDefault="true"
- relational:materialized="false"
   - transform:selectSql="SELECT PartSupplier_Oracle.SUPPLIER_PARTS.SUPPLIER_ID,

PartsSupplier_SQLServer.SUPPLIER.SUPPLIER_STATE
   PartSupplier_Oracle.SUPPLIER_PARTS,   PartsSupplier_SQLServer.SUPPLIER   WHERE
                                 PartSupplier_Oracle.SUPPLIER_PARTS.SUPPLIER_ID
PartsSupplier_SQLServer.SUPPLIER.SUPPLIER_ID"
   - relational:supportsUpdate="true"
   - relational:system="false"
   - transform:updateAllowed="true"
   - transform:updateSqlDefault="true"
   - xmi:uuid="2473dbc0-128c-1eec-8518-c32201e76066"
   SUPPLIER_ID jcr:primaryType="relational:column"
    - jcr:mixinTypes=["transform:transformed","mix:referenceable","xmi:referenceable"]
   - jcr:uuid="5f62a519-7948-4c9d-95df-131b489cec8e"
    - relational:autoIncremented="false"
    - relational:caseSensitive="true"
    - relational:currency="false"
    - relational:distinctValueCount="-1"
      - transform:inputHrefs="PartSupplier_SourceB.xmi#mmuuid/55e12d01-1275-1eec-8518-c32201e76066"
    - transform:inputXmiUuids="55e12d01-1275-1eec-8518-c32201e76066"
    - relational:length="10"
    - relational:nativeType="VARCHAR2"
    - relational:nullValueCount="-1"
    - relational:nullable="NULLABLE"
    - relational:radix="10"
    - relational:searchability="SEARCHABLE"
    - relational:selectable="true"
    - relational:signed="true"
    - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
    - relational:typeName="string"
    - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
    - relational:updateable="true"
    - xmi:uuid="143ff680-1291-1eec-8518-c32201e76066"
   PART_ID jcr:primaryType="relational:column"

```
    - jcr:mixinTypes=["transform:transformed","mix:referenceable","xmi:referenceable"]
  - jcr:uuid="bcce191f-acfd-48b9-8be8-ea04c0d37283"
  - relational:autoIncremented="false"
  - relational:caseSensitive="true"
  - relational:currency="false"
  - relational:distinctValueCount="-1"
  - relational:fixedLength="true"
      - transform:inputHrefs="PartSupplier_SourceB.xmi#mmuuid/54ed0902-1275-1eec-8518-
c32201e76066"
  - transform:inputXmiUuids="54ed0902-1275-1eec-8518-c32201e76066"
  - relational:length="4"
  - relational:nativeType="CHAR"
  - relational:nullValueCount="-1"
  - relational:nullable="NULLABLE"
  - relational:radix="10"
  - relational:searchability="SEARCHABLE"
  - relational:selectable="true"
  - relational:signed="true"
  - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
  - relational:typeName="string"
  - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
  - relational:updateable="true"
  - xmi:uuid="1d9b97c0-1291-1eec-8518-c32201e76066"
QUANTITY jcr:primaryType="relational:column"
  - jcr:mixinTypes=["transform:transformed","mix:referenceable","xmi:referenceable"]
 - jcr:uuid="126d6138-ce5e-40e3-92d9-48a239453dbb"
  - relational:autoIncremented="false"
  - relational:caseSensitive="true"
  - relational:currency="false"
  - relational:distinctValueCount="-1"
  - relational:fixedLength="true"
      - transform:inputHrefs="PartSupplier_SourceB.xmi#mmuuid/55e12d02-1275-1eec-8518-
c32201e76066"
  - transform:inputXmiUuids="55e12d02-1275-1eec-8518-c32201e76066"
  - relational:nativeType="NUMBER"
  - relational:nullValueCount="-1"
  - relational:nullable="NULLABLE"
  - relational:precision="3"
  - relational:radix="10"
  - relational:searchability="SEARCHABLE"
  - relational:selectable="true"
  - relational:signed="true"
  - relational:typeHref="http://www.w3.org/2001/XMLSchema#short"
  - relational:typeName="short"
```

- relational:typeXmiUuid="5bbcf140-b9ae-1e21-b812-969c8fc8b016"
- relational:updateable="true"
- xmi:uuid="250ef100-1291-1eec-8518-c32201e76066"
SHIPPER_ID jcr:primaryType="relational:column"
- jcr:mixinTypes=["transform:transformed","mix:referenceable","xmi:referenceable"]
- jcr:uuid="d9856363-6950-40ea-9c9a-44c4af43ec38"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- relational:fixedLength="true"
    - transform:inputHrefs="PartSupplier_SourceB.xmi#mmuuid/54ed0903-1275-1eec-8518-c32201e76066"
- transform:inputXmiUuids="54ed0903-1275-1eec-8518-c32201e76066"
- relational:nativeType="NUMBER"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:precision="2"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#short"
- relational:typeName="short"
- relational:typeXmiUuid="5bbcf140-b9ae-1e21-b812-969c8fc8b016"
- relational:updateable="true"
- xmi:uuid="2b8e2640-1291-1eec-8518-c32201e76066"
SUPPLIER_NAME jcr:primaryType="relational:column"
- jcr:mixinTypes=["transform:transformed","mix:referenceable","xmi:referenceable"]
- jcr:uuid="d0b9d5cc-f95a-4e97-a3f9-59571f58e206"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
    - transform:inputHrefs="PartsSupplier_SourceA.xmi#mmuuid/bc400084-1284-1eec-8518-c32201e76066"
- transform:inputXmiUuids="bc400084-1284-1eec-8518-c32201e76066"
- relational:length="30"
- relational:nativeType="varchar"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"

- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="34da8540-1291-1eec-8518-c32201e76066"
  SUPPLIER_STATUS jcr:primaryType="relational:column"
- jcr:mixinTypes=["transform:transformed","mix:referenceable","xmi:referenceable"]
- jcr:uuid="06253965-9f6f-4d6e-8219-2eb70a2745ed"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- relational:fixedLength="true"
  - transform:inputHrefs="PartsSupplier_SourceA.xmi#mmuuid/bc400083-1284-1eec-8518-c32201e76066"
  - transform:inputXmiUuids="bc400083-1284-1eec-8518-c32201e76066"
- relational:nativeType="numeric"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:precision="2"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#short"
- relational:typeName="short"
- relational:typeXmiUuid="5bbcf140-b9ae-1e21-b812-969c8fc8b016"
- relational:updateable="true"
- xmi:uuid="3c4dde80-1291-1eec-8518-c32201e76066"
  SUPPLIER_CITY jcr:primaryType="relational:column"
- jcr:mixinTypes=["transform:transformed","mix:referenceable","xmi:referenceable"]
- jcr:uuid="a9cfd1fd-1a99-4b7d-83dc-3dbeb86c7f0a"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
  - transform:inputHrefs="PartsSupplier_SourceA.xmi#mmuuid/bc400081-1284-1eec-8518-c32201e76066"
  - transform:inputXmiUuids="bc400081-1284-1eec-8518-c32201e76066"
- relational:length="30"
- relational:nativeType="varchar"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"

- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="43c137c0-1291-1eec-8518-c32201e76066"
SUPPLIER_STATE jcr:primaryType="relational:column"
- jcr:mixinTypes=["transform:transformed","mix:referenceable","xmi:referenceable"]
- jcr:uuid="8e040c5d-acf8-407f-a090-4bc1feac45cc"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="PartsSupplier_SourceA.xmi#mmuuid/bc400082-1284-1eec-8518-c32201e76066"
- transform:inputXmiUuids="bc400082-1284-1eec-8518-c32201e76066"
- relational:length="2"
- relational:nativeType="varchar"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="4a4faf40-1291-1eec-8518-c32201e76066"

Teiid VDB Sequencer

Teiid Designer [http://www.jboss.org/teiiddesigner], is a visual tool that enables rapid, model-driven definition, integration, management and testing of data services without programming using the Teiid runtime engine. It is capable of modeling several different kinds of data structures, but the most common and widely-used are relational models that describe a relational database schema, including the catalogs/schemas, tables, views, columns, primary keys, foreign keys, indexes, procedures, procedure results, procedure results, and logical relationships. Teiid Designer can reverse-engineer a relational model from a JDBC relational database or DDL file. It can also define "virtual" models that are transformations of other models (where the transformations are defined in terms of SQL select, insert, update, and delete statements). These models can then be packaged into a *virtual database*, which can be deployed to a Teiid runtime engine.

Teiid [http://www.jboss.org/teiid] is a high-performance database virtualization engine that allows JDBC and ODBC client applications access the virtual database as if it were a real database, using relational, XML, XQuery and procedural queries. Teiid dynamically (and in real-time) figures out how to answer the queries and operations issued by clients by efficiently accessing and manipulating the data inside the underlying data sources. Teiid's sophisticated engine is able to plan and optimize these operations, even when multiple heterogeneous relational and non-relational data sources must be accessed to obtain the required information.

The Teiid VDB sequencer parses the VDB archive files produced by the Teiid Designer, and extracts the structured relational data model described by each of the contained XMI files. This means that when VDB files are uploaded into a ModeShape repository, the sequencer writes to the repository all this virtual database and relational metadata contained in the VDB, where it can be queried and accessed by JCR, RESTful, and even JDBC clients.

The `VdbSequencer` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/sequencer/teiid/VdbSequencer.html] has no properties for changing behavior.

33.1. UUIDs and References

A Teiid virtual database file is entirely self-contained: it contains all of the models required for the VDB. No model can contain references to objects outside of these models, so the entire VDB archive is consistent and complete. When the sequencer extracts the relational information from these models, it automatically resolves all references. Also, the resulting content is independent of any the content from all other previous sequencing operations, including that of the *Teiid Model Sequencer*.

33.2. Node Types

The VDB sequencer follows JCR best-practices by defining all nodes to have a primary type of "`nt:unstructured`" (or a node type that extends "`nt:unstructured`"), meaning it's possible and valid for any node to have any property (with single or multiple values). However, it is still useful to capture the metadata about what that node represents, and so the sequencer use mixins for this.

The VDB sequencer reuses all of the *model node types* from the *Teiid Model Sequencer*, plus several new node types that are used for the VDB-specific metadata, as described below. Note that these are non-normative definitions of the node types; see the CND files in the "`modeshape-sequencer-teiid`" JAR file (or source) for the official definitions.

33.2.1. VDB Namespace

The compact node definitions for the "vdb" namespace are as follows:

```
<nt  = "http://www.jcp.org/jcr/nt/1.0">
<xmi = "http://www.omg.org/XMI">
<vdb = "http://www.metamatrix.com/metamodels/VirtualDatabase">
<mmcore = "http://www.metamatrix.com/metamodels/Core">


//---------------------------------------------------------------------------
// N O D E T Y P E S
//---------------------------------------------------------------------------

[vdb:virtualDatabase] > nt:unstructured
 - vdb:description (string)
 - vdb:version (long) = '1'
 - vdb:preview (boolean) = 'false'
 - vdb:originalFile (string)
 - mmcore:sha1 (string)

[vdb:model] > xmi:model, mmcore:model
 - vdb:visible (boolean) = 'true'
 - vdb:checksum (long)
 - vdb:builtIn (boolean) = 'false'
 - vdb:pathInVdb (string)
 - vdb:sourceTranslator (string)
 - vdb:sourceJndiName (string)
 - vdb:sourceName (string)
 + vdb:markers (vdb:markers) = vdb:markers copy

[vdb:markers] > nt:unstructured
 + vdb:marker (vdb:marker) = vdb:marker copy sns

[vdb:marker] > nt:unstructured
 - vdb:severity (string) = 'WARNING' < 'WARNING','ERROR','INFO'
 - vdb:path (string)
 - vdb:message (string)
```

33.3. Configuration

To use this sequencer, simply include the `modeshape-sequencer-teiid-2.6.0.Beta2.jar` file in your application and define a sequencing configuration in the ModeShape configuration, using something similar to:

```xml
<configuration xmlns:mode="http://www.modeshape.org/1.0" xmlns:jcr="http://www.jcp.org/jcr/1.0">
   <mode:sequencers>
      ...
            <mode:sequencer jcr:name="Teiid VDB Sequencer" mode:classname="org.modeshape.sequencer.teiid.VdbSequencer">
<mode:description>SequencesTeiidVirtualDatabases(e.g.,*.vdb)loadedintotherepositoryunder'/files', extracting the VDB metadata and the structure defined in the VDB's relational models.</mode:description>
         <!-- Note this path expression captures the path below '/files' but excludes the filename, and places the sequenced
               content under the same relative path below '/sequenced/teiid/models'. For example, if a VDB file is uploaded
               to '/files/my/favorites/Customers.vdb', then the sequenced output will be placed at
                the '/sequenced/teiid/models/Customer' node, which will have a primary type of 'vdb:virtualDatabase' and will
                contain under it the nodes representing the models (which will each contain the nodes representing that
             model's catalogs, schemas, tables, views, columns, etc.).  Of course, the path expression
                can be modified as needed; for example, to include the filename of the XMI model in the sequenced output path. -->
            <mode:pathExpression>/files(//)(*.vdb[*])/jcr:content[@jcr:data] => /sequenced/teiid/vdbs$1 </mode:pathExpression>
      </mode:sequencer>
      ...
   </mode:sequencers>
   ...
</configuration>
```

or using the *JcrConfiguration* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/jcr/JcrConfiguration.html]:

```
JcrConfiguration config = ...
```

```
config.sequencer("Teiid VDB Sequencer")
    .usingClass(VdbSequencer.class)
    .setDescription("Sequences Teiid VDBs")
    .sequencingFrom("/files(//)(*.vdb[*])/jcr:content[@jcr:data]")
    .andOutputtingTo("/sequenced/teiid/vdbs$1");
```

33.4. Example

Here is a representation of the nodes output by the sequencing of an example "`qe.2.vdb`" virtual database:

```
qe jcr:primaryType="vdb:virtualDatabase"
  - jcr:mixinTypes=["mix:referenceable", "mode:derived"]
  - jcr:uuid="1d110326-f8e9-4f5e-becd-2f3e4d63296e"
  - mode:derivedAt="2011-05-13T13:12:03.925Z"
  - mode:derivedFrom="/files/foo.vdb"
  - vdb:description="This VDB is for testing Recursive XML documents and Text Sources"
  - vdb:originalFile="/vdb/qe.vdb"
  - vdb:preview="false"
  - mode:sha1="4cec9166f20a8d3772a1cfddb493329e35c3adb7"
  - vdb:version="2"
                                     text                     jcr:primaryType="vdb:model"

 jcr:uuid="5cffd0ee-2edd-44af-8a8d-46459d849afe"
    - vdb:builtIn="true"
    - vdb:checksum="958072371"
    - mmcore:maxSetSize="100"
    - mmcore:modelType="PHYSICAL"
    - mmcore:originalFile="/vdb/qe.vdb"
    - vdb:pathInVdb="QuickText/text.xmi"
    - mmcore:primaryMetamodelUri="http://www.metamatrix.com/metamodels/Relational"
    - mmcore:producerName="Teiid Designer"
    - mmcore:producerVersion="7.0.0.v20100807-1026-H168-M1"
    - mode:sha1="893accdcb0745f8061626b4ab60079daeb3eb74f"
    - vdb:sourceJndiName="empdata-file"
    - vdb:sourceName="text"
    - vdb:sourceTranslator="file"
    - mmcore:supportsDistinct="true"
    - mmcore:supportsJoin="true"
    - mmcore:supportsOrderBy="true"
    - mmcore:supportsOuterJoin="true"
```

- mmcore:supportsWhereAll="true"
- xmi:uuid="ba1f1ca6-b9a7-44f8-9d89-8d9ba9f801ba"
- xmi:version="2.0"
- mmcore:visible="true"
- vdb:visible="true"
vdb:markers jcr:primaryType="vdb:markers"
  vdb:marker jcr:primaryType="vdb:marker"
    - vdb:message="Missing or invalid Length on column with a string/character datatype (See validation Preferences)"
    - vdb:path="getTextFiles/NewProcedureResult/filePath"
    - vdb:severity="WARNING"
                                        XMLSchema   jcr:primaryType="mmcore:import"
  jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="1787cc24-d545-437c-a7ef-e18569eec9c3"
    - mmcore:modelType="TYPE"
    - mmcore:primaryMetamodelUri="http://www.eclipse.org/xsd/2002/XSD"
    - xmi:uuid="mmuuid:5a23faba-871a-490e-9799-efdffea80b6b"
                            SimpleDatatypes-instance   jcr:primaryType="mmcore:import"

 jcr:uuid="4e11258b-06e2-4d39-8a10-7e6b8e02dc37"
    - mmcore:modelType="TYPE"
    - mmcore:primaryMetamodelUri="http://www.eclipse.org/xsd/2002/XSD"
    - xmi:uuid="mmuuid:b09c455c-1c5a-4de4-8373-e823482ce517"
                                getTextFiles   jcr:primaryType="relational:procedure"
      jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="137968e0-e375-43c3-b25a-03953ff975ff"
    - xmi:uuid="bf60b5cb-fd8c-474a-9f4c-68eb42ca40f2"
                          pathAndPattern   jcr:primaryType="relational:procedureParameter"

 jcr:uuid="2a2de9a3-561d-4d14-82cf-8155b965d2bb"
    - relational:nullable="NULLABLE"
    - relational:radix="10"
    - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
    - relational:typeName="string"
    - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
    - xmi:uuid="f44bb026-8bdf-413b-b705-65dcd40bf437"
                          NewProcedureResult   jcr:primaryType="relational:procedureResult"
      jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="b5d7be35-9a73-477e-ab20-5a4b9248da9f"
    - xmi:uuid="eb2f5c65-bede-4dd2-8c85-441c240ebca1"
                                        file   jcr:primaryType="relational:column"

 jcr:uuid="2ebe184b-25ce-4cb4-89f4-0e6289112c68"
      - relational:autoIncremented="false"

- relational:caseSensitive="true"

- relational:currency="false"

- relational:distinctValueCount="-1"

- relational:nullValueCount="-1"

- relational:nullable="NULLABLE"

- relational:radix="10"

- relational:searchability="SEARCHABLE"

- relational:selectable="true"

- relational:signed="true"

- relational:typeHref="http://www.metamatrix.com/metamodels/SimpleDatatypes-instance#clob"

- relational:typeName="clob"

- relational:typeXmiUuid="559646c0-4941-1ece-b22b-f49159d22ad3"

- relational:updateable="true"

- xmi:uuid="092a2a85-7ec6-40da-9437-afd0812eccbb"

filePath   jcr:primaryType="relational:column" jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="ec43aa0d-20df-49ff-8b4f-ed95961aa9a5"

- relational:autoIncremented="false"

- relational:caseSensitive="true"

- relational:currency="false"

- relational:distinctValueCount="-1"

- relational:nullValueCount="-1"

- relational:nullable="NULLABLE"

- relational:radix="10"

- relational:searchability="SEARCHABLE"

- relational:selectable="true"

- relational:signed="true"

- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"

- relational:typeName="string"

- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"

- relational:updateable="true"

- xmi:uuid="232d5fd7-e5a6-49a7-bd5f-e5d6b7e753a3"

Employees   jcr:primaryType="vdb:model" jcr:mixinTypes=["mmcore:model","mix:referenceable","xmi:referenceable"] jcr:uuid="88ca643b-ebab-43a1-902c-462f3ea17fd8"

- vdb:builtIn="true"

- vdb:checksum="1269937912"

- mmcore:maxSetSize="100"

- mmcore:modelType="VIRTUAL"

- mmcore:originalFile="/vdb/qe.vdb"

- vdb:pathInVdb="QuickEmployees/Employees.xmi"

- mmcore:primaryMetamodelUri="http://www.metamatrix.com/metamodels/Relational"

- mmcore:producerName="Teiid Designer"

- mmcore:producerVersion="7.0.0.v20100807-1026-H168-M1"
- mode:sha1="a63c108098232739aad1d6ab4cf0d3cc1911aa12"
- mmcore:supportsDistinct="true"
- mmcore:supportsJoin="true"
- mmcore:supportsOrderBy="true"
- mmcore:supportsOuterJoin="true"
- mmcore:supportsWhereAll="true"
- xmi:uuid="9c034c0d-10c7-4fa5-beae-ff602bfcf88e"
- xmi:version="2.0"
- mmcore:visible="true"
- vdb:visible="true"
vdb:markers jcr:primaryType="vdb:markers"
  vdb:marker jcr:primaryType="vdb:marker"
    - vdb:message="Possible cross-join: Group/s '[f, emp]' are not joined either directly or transitively to other groups through a join criteria. Check all queries in the transformation."
  - vdb:path="EmpTable"
  - vdb:severity="WARNING"

text   jcr:primaryType="mmcore:import" jcr:mixinTypes=["mix:referenceable","xmi:referenceable"] jcr:uuid="ee4b288a-47b0-4c81-98e5-ddf01f8a4cda"
  - mmcore:modelType="PHYSICAL"
  - mmcore:primaryMetamodelUri="http://www.metamatrix.com/metamodels/Relational"
  - xmi:uuid="mmuuid:46ba6b40-bb81-43ba-996e-6f3ebaffea3b"

SimpleDatatypes-instance   jcr:primaryType="mmcore:import" jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="98b1dbae-5ad6-4439-adb0-64d6e5d0a42f"
  - mmcore:modelType="TYPE"
  - mmcore:primaryMetamodelUri="http://www.eclipse.org/xsd/2002/XSD"
  - xmi:uuid="mmuuid:36a2080b-7243-445c-a153-79a19d42f558"

XMLSchema   jcr:primaryType="mmcore:import" jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="71eba18c-195e-47ec-b925-415f981bcd45"
  - mmcore:modelType="TYPE"
  - mmcore:primaryMetamodelUri="http://www.eclipse.org/xsd/2002/XSD"
  - xmi:uuid="mmuuid:ea4a1ff7-fa32-4348-b5a2-192c554b70a4"

EmpTable   jcr:primaryType="relational:baseTable"

jcr:uuid="6209d827-62eb-4909-8e66-edbf615a42db"
  - transform:deleteAllowed="true"
  - transform:deleteSqlDefault="true"
    - transform:inputHrefs="../QuickText/text.xmi#mmuuid/bf60b5cb-fd8c-474a-9f4c-68eb42ca40f2"
  - transform:inputNames="getTextFiles"
  - transform:inputXmiUuids="bf60b5cb-fd8c-474a-9f4c-68eb42ca40f2"

   - transform:inputs="137968e0-e375-43c3-b25a-03953ff975ff"
   - transform:insertAllowed="true"
   - transform:insertSqlDefault="true"
   - relational:materialized="false"
     - transform:selectSql="SELECT * FROM (EXEC text.getTextFiles('EmpData.txt')) AS f, TEXTTABLE(F.file COLUMNS lastName string, firstName string, middleName string, empId biginteger, department string, annualSalary double, title string, homePhone string, mgrId biginteger, street string, city string, state string, ZipCode string HEADER 3) AS emp"
   - relational:supportsUpdate="true"
   - relational:system="false"
   - transform:updateAllowed="true"
   - transform:updateSqlDefault="true"
   - xmi:uuid="6179a495-7b7e-4e12-9da3-998e4f709de4"
                                                  file   jcr:primaryType="relational:column"

jcr:uuid="de7902d1-9782-4137-a002-85681e45c0c6"
   - relational:autoIncremented="false"
   - relational:caseSensitive="true"
   - relational:currency="false"
   - relational:distinctValueCount="-1"
           - transform:inputHrefs="../QuickText/text.xmi#mmuuid/092a2a85-7ec6-40da-9437-afd0812eccbb"
   - transform:inputNames="file"
   - transform:inputXmiUuids="092a2a85-7ec6-40da-9437-afd0812eccbb"
   - transform:inputs="2ebe184b-25ce-4cb4-89f4-0e6289112c68"
   - relational:nullValueCount="-1"
   - relational:nullable="NULLABLE"
   - relational:radix="10"
   - relational:searchability="SEARCHABLE"
   - relational:selectable="true"
   - relational:signed="true"
           - relational:typeHref="http://www.metamatrix.com/metamodels/SimpleDatatypes-instance#clob"
   - relational:typeName="clob"
   - relational:typeXmiUuid="559646c0-4941-1ece-b22b-f49159d22ad3"
   - relational:updateable="true"
   - xmi:uuid="5ca79549-8edc-4972-9d05-cb3066d41676"
                                          filePath   jcr:primaryType="relational:column"

jcr:uuid="4b19a0f6-d65b-4b5b-845f-f30027947f6c"
   - relational:autoIncremented="false"
   - relational:caseSensitive="true"
   - relational:currency="false"
   - relational:distinctValueCount="-1"

- transform:inputHrefs="../QuickText/text.xmi#mmuuid/232d5fd7-e5a6-49a7-bd5f-e5d6b7e753a3"
    - transform:inputNames="filePath"
    - transform:inputXmiUuids="232d5fd7-e5a6-49a7-bd5f-e5d6b7e753a3"
    - transform:inputs="ec43aa0d-20df-49ff-8b4f-ed95961aa9a5"
    - relational:length="10"
    - relational:nullValueCount="-1"
    - relational:nullable="NULLABLE"
    - relational:radix="10"
    - relational:searchability="SEARCHABLE"
    - relational:selectable="true"
    - relational:signed="true"
    - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
    - relational:typeName="string"
    - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
    - relational:updateable="true"
    - xmi:uuid="fea43d8f-94e4-41f3-9743-3286f8c28590"

lastName   jcr:primaryType="relational:column"

jcr:uuid="6672adb4-1ded-4289-989d-3b707fc7384b"
    - relational:autoIncremented="false"
    - relational:caseSensitive="true"
    - relational:currency="false"
    - relational:distinctValueCount="-1"
    - relational:length="10"
    - relational:nullValueCount="-1"
    - relational:nullable="NULLABLE"
    - relational:radix="10"
    - relational:searchability="SEARCHABLE"
    - relational:selectable="true"
    - relational:signed="true"
    - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
    - relational:typeName="string"
    - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
    - relational:updateable="true"
    - xmi:uuid="f0b80cce-dd11-44b7-ab2d-4e382befd701"

firstName   jcr:primaryType="relational:column"
    jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="13986096-1e2f-483b-b603-3e7098bc0897"
    - relational:autoIncremented="false"
    - relational:caseSensitive="true"
    - relational:currency="false"
    - relational:distinctValueCount="-1"
    - relational:length="10"

- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="aae0eea7-fb09-4b46-9a41-8815bf5331db"

middleName   jcr:primaryType="relational:column"
jcr:mixinTypes=["mix:referenceable","xmi:referenceable"] jcr:uuid="53ba886e-e6b7-4771-8a70-bf0e6c9cad63"

- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="74333281-f3f8-4907-8ac1-4c819dfc76a8"

empId   jcr:primaryType="relational:column"
jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="80fcf4ac-d51d-4d24-b84f-3db7dbbcfa2b"

- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"

- relational:typeHref="http://www.metamatrix.com/metamodels/SimpleDatatypes-instance#biginteger"
    - relational:typeName="biginteger"
    - relational:typeXmiUuid="822b9a40-a066-1e26-9b08-d6079ebe1f0d"
    - relational:updateable="true"
    - xmi:uuid="5e42fcfc-fe7a-476d-8b55-8a5ce0cd7050"

department   jcr:primaryType="relational:column"
jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="8c805831-d1f9-4070-9e4b-a95234e6a7d7"
    - relational:autoIncremented="false"
    - relational:caseSensitive="true"
    - relational:currency="false"
    - relational:distinctValueCount="-1"
    - relational:length="10"
    - relational:nullValueCount="-1"
    - relational:nullable="NULLABLE"
    - relational:radix="10"
    - relational:searchability="SEARCHABLE"
    - relational:selectable="true"
    - relational:signed="true"
    - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
    - relational:typeName="string"
    - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
    - relational:updateable="true"
    - xmi:uuid="36ea6df0-ddc0-4311-be2e-f4a6cbe2b580"

annualSalary   jcr:primaryType="relational:column"

jcr:uuid="d9ee87bd-3567-4446-80f8-17bead52dd4b"
    - relational:autoIncremented="false"
    - relational:caseSensitive="true"
    - relational:currency="false"
    - relational:distinctValueCount="-1"
    - relational:nullValueCount="-1"
    - relational:nullable="NULLABLE"
    - relational:radix="10"
    - relational:searchability="SEARCHABLE"
    - relational:selectable="true"
    - relational:signed="true"
    - relational:typeHref="http://www.w3.org/2001/XMLSchema#double"
    - relational:typeName="double"
    - relational:typeXmiUuid="1f18b140-c4a3-1e24-9b01-c8207cd53eb7"
    - relational:updateable="true"
    - xmi:uuid="79c7b080-c9de-42c9-b252-a449d44e5d34"

title   jcr:primaryType="relational:column"

jcr:uuid="794a1e06-6160-4255-8f7a-23f30e5e9af5"
     - relational:autoIncremented="false"
     - relational:caseSensitive="true"
     - relational:currency="false"
     - relational:distinctValueCount="-1"
     - relational:length="10"
     - relational:nullValueCount="-1"
     - relational:nullable="NULLABLE"
     - relational:radix="10"
     - relational:searchability="SEARCHABLE"
     - relational:selectable="true"
     - relational:signed="true"
     - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
     - relational:typeName="string"
     - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
     - relational:updateable="true"
     - xmi:uuid="001ac238-21c6-45f3-8959-3fa0c7bea6c6"
                                        homePhone   jcr:primaryType="relational:column"

jcr:uuid="24edebac-093d-4aaf-8063-2b4e48c9f08d"
     - relational:autoIncremented="false"
     - relational:caseSensitive="true"
     - relational:currency="false"
     - relational:distinctValueCount="-1"
     - relational:length="10"
     - relational:nullValueCount="-1"
     - relational:nullable="NULLABLE"
     - relational:radix="10"
     - relational:searchability="SEARCHABLE"
     - relational:selectable="true"
     - relational:signed="true"
     - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
     - relational:typeName="string"
     - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
     - relational:updateable="true"
     - xmi:uuid="432c3937-e7ad-40de-9cb4-deb9d52511b2"
                                           mgrId   jcr:primaryType="relational:column"
     jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="c61173a4-27c8-41c2-
bebf-05e24fa82f94"
     - relational:autoIncremented="false"
     - relational:caseSensitive="true"
     - relational:currency="false"

- relational:distinctValueCount="-1"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
    - relational:typeHref="http://www.metamatrix.com/metamodels/SimpleDatatypes-instance#biginteger"
- relational:typeName="biginteger"
- relational:typeXmiUuid="822b9a40-a066-1e26-9b08-d6079ebe1f0d"
- relational:updateable="true"
- xmi:uuid="9c7b26dc-bbf6-4b83-9f03-438ad6a0b3f0"
                                        street   jcr:primaryType="relational:column"
    jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="ae505cbf-13bd-4c87-b020-526dece5c8b9"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="1181dfe5-0d2b-4331-b10b-5d6409dd6cbe"
                                        city   jcr:primaryType="relational:column"
    jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="7f32a1c1-622f-40ff-8005-ab42bb02a857"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"

- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="60792162-1659-416b-a6da-b78119429247"

      state   jcr:primaryType="relational:column" jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="41067115-320e-44e3-a70a-8a71e85fa8d8"

- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="67ed3d16-7fd6-43bb-b16a-61579a49db91"

      ZipCode   jcr:primaryType="relational:column" jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="93c47676-fec3-46fa-aa19-777be6136de2"

- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"

- xmi:uuid="2c2267b6-bddf-4d42-aef8-7d24e7527b65"
                                        EmpV   jcr:primaryType="vdb:model"

jcr:uuid="c9722b47-03ad-4cdd-81d1-d75e639517a1"
  - vdb:builtIn="true"
  - vdb:checksum="2273245105"
  - mmcore:maxSetSize="100"
  - mmcore:modelType="VIRTUAL"
  - mmcore:originalFile="/vdb/qe.vdb"
  - vdb:pathInVdb="QuickEmployees/EmpV.xmi"
  - mmcore:primaryMetamodelUri="http://www.metamatrix.com/metamodels/Relational"
  - mmcore:producerName="Teiid Designer"
  - mmcore:producerVersion="7.0.0.v20100807-1026-H168-M1"
  - mode:sha1="502cc1e3dbec4c5cd880662473e8dc2a668d5e78"
  - mmcore:supportsDistinct="true"
  - mmcore:supportsJoin="true"
  - mmcore:supportsOrderBy="true"
  - mmcore:supportsOuterJoin="true"
  - mmcore:supportsWhereAll="true"
  - xmi:uuid="e17f3917-d880-4bad-9a19-7d0f8f3d2135"
  - xmi:version="2.0"
  - mmcore:visible="true"
  - vdb:visible="true"
  vdb:markers jcr:primaryType="vdb:markers"
   vdb:marker jcr:primaryType="vdb:marker"
      - vdb:message="Missing or invalid Precision on column with a numeric datatype (See validation Preferences)"
    - vdb:path="EmpTable/empId"
    - vdb:severity="WARNING"
                                        XMLSchema   jcr:primaryType="mmcore:import"

jcr:uuid="54a5401e-3bab-4918-81cb-4a278d0263c4"
   - mmcore:modelType="TYPE"
   - mmcore:primaryMetamodelUri="http://www.eclipse.org/xsd/2002/XSD"
   - xmi:uuid="mmuuid:deb854d2-af4d-4158-9846-4ac17f207291"
                          SimpleDatatypes-instance   jcr:primaryType="mmcore:import"
      jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="699f153e-7301-4ef4-bffa-7522475f8c0a"
   - mmcore:modelType="TYPE"
   - mmcore:primaryMetamodelUri="http://www.eclipse.org/xsd/2002/XSD"
   - xmi:uuid="mmuuid:6471e823-eeee-46e8-8d7d-fb00b336cfe7"
                                        Employees   jcr:primaryType="mmcore:import"
  jcr:mixinTypes=["mix:referenceable","xmi:referenceable"]   jcr:uuid="5f672add-38cd-469f-a180-ac75306298b5"

- mmcore:modelType="VIRTUAL"
- mmcore:primaryMetamodelUri="http://www.metamatrix.com/metamodels/Relational"
- xmi:uuid="mmuuid:5806eb6e-fc70-4ad7-b7ff-13f14ec00ca2"
                                    EmpTable   jcr:primaryType="relational:baseTable"

jcr:uuid="0568b4b9-44c9-4798-9bee-662094015d67"
    - transform:deleteAllowed="true"
    - transform:deleteSqlDefault="true"
    - transform:inputHrefs="Employees.xmi#mmuuid/6179a495-7b7e-4e12-9da3-998e4f709de4"
    - transform:inputNames="EmpTable"
    - transform:inputXmiUuids="6179a495-7b7e-4e12-9da3-998e4f709de4"
    - transform:inputs="6209d827-62eb-4909-8e66-edbf615a42db"
    - transform:insertAllowed="true"
    - transform:insertSqlDefault="true"
    - relational:materialized="false"
                        -      transform:selectSql="SELECT   "Employees.EmpTable.lastName",
                "Employees.EmpTable.firstName",   "Employees.EmpTable.middleNam
                    "Employees.EmpTable.empId",   "Employees.EmpTable.depa
                    "Employees.EmpTable.annualSalary",   "Employees.EmpTa
                    "Employees.EmpTable.homePhone",   "Employees.EmpTable
    "Employees.EmpTable.street",   "Employees.EmpTable.city",   "Employees.EmpTable.state",
"Employees.EmpTable.ZipCode" FROM "Employees.EmpTable""
    - relational:supportsUpdate="true"
    - relational:system="false"
    - transform:updateAllowed="true"
    - transform:updateSqlDefault="true"
    - xmi:uuid="92cbc96b-f080-42d6-85dc-95cd07edd682"
                                        lastName   jcr:primaryType="relational:column"

jcr:uuid="cf26def7-de5a-4a1a-8276-f48d988439e4"
    - relational:autoIncremented="false"
    - relational:caseSensitive="true"
    - relational:currency="false"
    - relational:distinctValueCount="-1"
   - transform:inputHrefs="Employees.xmi#mmuuid/f0b80cce-dd11-44b7-ab2d-4e382befd701"
    - transform:inputNames="lastName"
    - transform:inputXmiUuids="f0b80cce-dd11-44b7-ab2d-4e382befd701"
    - transform:inputs="6672adb4-1ded-4289-989d-3b707fc7384b"
    - relational:length="10"
    - relational:nullValueCount="-1"
    - relational:nullable="NULLABLE"
    - relational:radix="10"
    - relational:searchability="SEARCHABLE"
    - relational:selectable="true"

- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="a4c30553-7f10-445b-971b-c54cee534639"

firstName   jcr:primaryType="relational:column"

jcr:uuid="5b932651-6de2-4e89-917f-0d515a5270b0"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="Employees.xmi#mmuuid/aae0eea7-fb09-4b46-9a41-8815bf5331db"
- transform:inputNames="firstName"
- transform:inputXmiUuids="aae0eea7-fb09-4b46-9a41-8815bf5331db"
- transform:inputs="13986096-1e2f-483b-b603-3e7098bc0897"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="a48c7515-d271-45ed-8920-22cf8c9d01bb"

middleName   jcr:primaryType="relational:column"

jcr:uuid="ac7511b4-4b3d-4d42-886e-26ea5b669b5b"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="Employees.xmi#mmuuid/74333281-f3f8-4907-8ac1-4c819dfc76a8"
- transform:inputNames="middleName"
- transform:inputXmiUuids="74333281-f3f8-4907-8ac1-4c819dfc76a8"
- transform:inputs="53ba886e-e6b7-4771-8a70-bf0e6c9cad63"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"

- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="1d792d5e-ae70-4855-b59f-3eb7dceeb5a3"

            empId   jcr:primaryType="relational:column"

jcr:uuid="d1a61bb1-f94d-40c9-bee7-f91d12b26d80"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="Employees.xmi#mmuuid/5e42fcfc-fe7a-476d-8b55-8a5ce0cd7050"
- transform:inputNames="empId"
- transform:inputXmiUuids="5e42fcfc-fe7a-476d-8b55-8a5ce0cd7050"
- transform:inputs="80fcf4ac-d51d-4d24-b84f-3db7dbbcfa2b"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
    - relational:typeHref="http://www.metamatrix.com/metamodels/SimpleDatatypes-instance#biginteger"
- relational:typeName="biginteger"
- relational:typeXmiUuid="822b9a40-a066-1e26-9b08-d6079ebe1f0d"
- relational:updateable="true"
- xmi:uuid="d9cc45f7-c9de-44f9-b22e-3674b1a7d33c"

          department   jcr:primaryType="relational:column"

jcr:uuid="46c5910c-0cb0-481c-bbad-577e52ac9c96"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="Employees.xmi#mmuuid/36ea6df0-ddc0-4311-be2e-f4a6cbe2b580"
- transform:inputNames="department"
- transform:inputXmiUuids="36ea6df0-ddc0-4311-be2e-f4a6cbe2b580"
- transform:inputs="8c805831-d1f9-4070-9e4b-a95234e6a7d7"
- relational:length="10"
- relational:nullValueCount="-1"

- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="19932ef1-4794-496d-a98b-027971cb5599"

annualSalary   jcr:primaryType="relational:column"

jcr:uuid="61cb7c5a-c7f8-420d-b84a-0a4cabceed81"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="Employees.xmi#mmuuid/79c7b080-c9de-42c9-b252-a449d44e5d34"
- transform:inputNames="annualSalary"
- transform:inputXmiUuids="79c7b080-c9de-42c9-b252-a449d44e5d34"
- transform:inputs="d9ee87bd-3567-4446-80f8-17bead52dd4b"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#double"
- relational:typeName="double"
- relational:typeXmiUuid="1f18b140-c4a3-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="925999e2-15a5-4728-a76e-e0c9ae235d80"

title   jcr:primaryType="relational:column"

jcr:uuid="e2db3e5e-d2c9-462c-9581-24c700792f0c"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="Employees.xmi#mmuuid/001ac238-21c6-45f3-8959-3fa0c7bea6c6"
- transform:inputNames="title"
- transform:inputXmiUuids="001ac238-21c6-45f3-8959-3fa0c7bea6c6"
- transform:inputs="794a1e06-6160-4255-8f7a-23f30e5e9af5"
- relational:length="10"

- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="36cf25e4-5164-4c9f-81a9-572d0fc11e8b"

                         homePhone   jcr:primaryType="relational:column"

jcr:uuid="6ef5a18f-db08-44e6-beb0-d7e6842f3ca5"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
        - transform:inputHrefs="Employees.xmi#mmuuid/432c3937-e7ad-40de-9cb4-deb9d52511b2"
- transform:inputNames="homePhone"
- transform:inputXmiUuids="432c3937-e7ad-40de-9cb4-deb9d52511b2"
- transform:inputs="24edebac-093d-4aaf-8063-2b4e48c9f08d"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="8a782b60-0296-4e10-85b1-e0ec03b34d00"

                         mgrId   jcr:primaryType="relational:column"

jcr:uuid="11a41f81-5a32-434a-8cd9-04bbbdc4b00c"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="Employees.xmi#mmuuid/9c7b26dc-bbf6-4b83-9f03-438ad6a0b3f0"
- transform:inputNames="mgrId"

- transform:inputXmiUuids="9c7b26dc-bbf6-4b83-9f03-438ad6a0b3f0"
- transform:inputs="c61173a4-27c8-41c2-bebf-05e24fa82f94"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
        - relational:typeHref="http://www.metamatrix.com/metamodels/SimpleDatatypes-instance#biginteger"
- relational:typeName="biginteger"
- relational:typeXmiUuid="822b9a40-a066-1e26-9b08-d6079ebe1f0d"
- relational:updateable="true"
- xmi:uuid="4f0439b3-8899-44f9-99a6-30971c4a563f"

                                              street   jcr:primaryType="relational:column"

jcr:uuid="8b7dea23-1a6b-4a76-9fbf-5b4770b8cac9"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="Employees.xmi#mmuuid/1181dfe5-0d2b-4331-b10b-5d6409dd6cbe"
- transform:inputNames="street"
- transform:inputXmiUuids="1181dfe5-0d2b-4331-b10b-5d6409dd6cbe"
- transform:inputs="ae505cbf-13bd-4c87-b020-526dece5c8b9"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="c68dfbeb-bc26-4932-ae0b-d354ed000a4e"

                                              city   jcr:primaryType="relational:column"

jcr:uuid="a1c94897-33ae-4d9e-9dd5-7c9a09e7ebf2"
- relational:autoIncremented="false"
- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"

- transform:inputHrefs="Employees.xmi#mmuuid/60792162-1659-416b-a6da-b78119429247"
    - transform:inputNames="city"
    - transform:inputXmiUuids="60792162-1659-416b-a6da-b78119429247"
    - transform:inputs="7f32a1c1-622f-40ff-8005-ab42bb02a857"
    - relational:length="10"
    - relational:nullValueCount="-1"
    - relational:nullable="NULLABLE"
    - relational:radix="10"
    - relational:searchability="SEARCHABLE"
    - relational:selectable="true"
    - relational:signed="true"
    - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
    - relational:typeName="string"
    - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
    - relational:updateable="true"
    - xmi:uuid="b27874ae-4c7b-4545-84f1-5d95c6a70b3a"
                                         state   jcr:primaryType="relational:column"

jcr:uuid="d5fded76-1ddc-4a87-af76-a566bb5919dc"
    - relational:autoIncremented="false"
    - relational:caseSensitive="true"
    - relational:currency="false"
    - relational:distinctValueCount="-1"
  - transform:inputHrefs="Employees.xmi#mmuuid/67ed3d16-7fd6-43bb-b16a-61579a49db91"
    - transform:inputNames="state"
    - transform:inputXmiUuids="67ed3d16-7fd6-43bb-b16a-61579a49db91"
    - transform:inputs="41067115-320e-44e3-a70a-8a71e85fa8d8"
    - relational:length="10"
    - relational:nullValueCount="-1"
    - relational:nullable="NULLABLE"
    - relational:radix="10"
    - relational:searchability="SEARCHABLE"
    - relational:selectable="true"
    - relational:signed="true"
    - relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
    - relational:typeName="string"
    - relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
    - relational:updateable="true"
    - xmi:uuid="248a034b-7331-46ee-a4a4-5db5176ce1bc"
                                  ZipCode   jcr:primaryType="relational:column"

jcr:uuid="a73b2a7c-9b19-4697-ae83-f7be5cca7778"
    - relational:autoIncremented="false"

- relational:caseSensitive="true"
- relational:currency="false"
- relational:distinctValueCount="-1"
- transform:inputHrefs="Employees.xmi#mmuuid/2c2267b6-bddf-4d42-aef8-7d24e7527b65"
- transform:inputNames="ZipCode"
- transform:inputXmiUuids="2c2267b6-bddf-4d42-aef8-7d24e7527b65"
- transform:inputs="93c47676-fec3-46fa-aa19-777be6136de2"
- relational:length="10"
- relational:nullValueCount="-1"
- relational:nullable="NULLABLE"
- relational:radix="10"
- relational:searchability="SEARCHABLE"
- relational:selectable="true"
- relational:signed="true"
- relational:typeHref="http://www.w3.org/2001/XMLSchema#string"
- relational:typeName="string"
- relational:typeXmiUuid="bf6c34c0-c442-1e24-9b01-c8207cd53eb7"
- relational:updateable="true"
- xmi:uuid="836656b4-30b1-4c57-a64b-f810763a4a0c"

Part V. 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 configuring ModeShape to use them.

Aperture MIME type detector

The *ApertureMimeTypeDetector* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/mimetype/aperture/ApertureMimeTypeDetector.html] class is an implementation of *MimeTypeDetector* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] subcontext with it:

MimeTypeDetector [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] myDetector = new ApertureMimeTypeDetector();
ExecutionContext [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/ExecutionContext.html] contextWithMyDetector = context.with(myDetector);

Writing custom detectors

Creating a custom detector involves the following steps:

- Create a Maven 3 project for your detector;

- Implement the *MimeTypeDetector* [http://docs.jboss.org/modeshape/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/repository/mimetype/MimeTypeDetectorConfig.html] to the `MimeType` [http://docs.jboss.org/modeshape/2.6.0.Beta2/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 3 project that you can use to compile your code and build the JARs. Maven 3 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 3, 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 Git repository: *http://github.com/ModeShape/modeshape//tree/modeshape-2.6.0.Beta2/extensions/modeshape-mimetype-detector-aperture/*

You can create your Maven project any way you'd like. For examples, see the *Maven 3 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:

```xml
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-common</artifactId>
  <version>2.5.0.Beta-1</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>2.5.0.Beta-1</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.8.4</version>
</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:

```xml
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8</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>1.8.4</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
```

```
  <version>1.2.16</version>
  <scope>test</scope>
</dependency>
```

After you've created the project, simply implement the *MimeTypeDetector* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/mimetype/MimeTypeDetector.html] interface. And testing should be quite straightforward, as MIME type detectors don't require any other components. In your tests, simply instantiate your *MimeTypeDetector* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/mimetype/MimeTypeDetector.html] implementation, supply various combinations of names and/or *InputStream* [http://java.sun.com/javase/6/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/2.6.0.Beta2/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/2.6.0.Beta2/api/org/modeshape/repository/ mimetype/MimeType.html] class using the `addDetector(MimeTypeDetectorConfig [http:// docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/repository/mimetype/ MimeTypeDetectorConfig.html] config)` method. Then, just use the `MimeType` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/repository/ mimetype/MimeType.html] class.

Part VI. Text Extractor Library

The ModeShape project provides a number of *text extractors* out-of-the-box. These are ready to be used by simply including them in the classpath and configuring ModeShape to use them.

Teiid text extractor

The *TeiidVdbTextExtractor* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/extractor/teiid/TeiidVdbTextExtractor.html] class is an implementation of *TextExtractor* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/text/TextExtractor.html] that extracts from Teiid virtual database (i.e., ".vdb") files the virtual database's logical name, description, and version, plus the logical name, description, source name, source translator name, and JNDI name for each of the virtual database's models.

This sequencer is not enabled by default, but it's very easy to add this text extractor to the ModeShape configuration. To do so in a configuration file, simply add the following fragment under the "<mode:textExtractors>" element (which should be immediately under the "<configuration>" root element):

```
<mode:textExtractor jcr:name="VDB Text Extractors">
  <mode:description>Extract text from Teiid VDB files</mode:description>
  <mode:classname>org.modeshape.extractor.teiid.TeiidVdbTextExtractor</mode:classname>
</mode:textExtractor>
```

Then, make sure the `modeshape-sequencer-teiid.jar` file on the classpath or, if you're using Maven, your application has a dependency on the `org.modeshape:modeshape-sequencer-teiid` library.

Tika text extractor

The _TikaTextExtractor_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/extractor/tika/TikaTextExtractor.html] class is an implementation of _TextExtractor_ [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/graph/text/TextExtractor.html] that uses the _Tika_ [http://tika.apache.org/] toolkit from Apache to parse and extract text from a variety of file types, including Microsoft Office, PDF, HTML, plain text, XML, and others.

This sequencer is not enabled by default, but it's very easy to add this text extractor to the ModeShape configuration. To do so in a configuration file, simply add the following fragment under the "<mode:textExtractors>" element (which should be immediately under the "<configuration>" root element):

```
<mode:textExtractor jcr:name="Tika Text Extractors">
 <mode:description>Text extractors using Tika parsers</mode:description>
 <mode:classname>org.modeshape.extractor.tika.TikaTextExtractor</mode:classname>
 <!--
 A comma- or whitespace-delimited list of MIME types that are to be excluded.
 The following are excluded by default, but the default is completely overridden
 when this property is set. In other words, if you explicitly exclude any MIME types,
 be sure to list all of the MIME types you want to exclude. Exclusions always
 have a higher precedence than inclusions.
 -->
 <mode:excludedMimeTypes>
    application/x-archive,application/x-bzip,application/x-bzip2,
    application/x-cpio,application/x-gtar,application/x-gzip,
    application/x-ta,application/zip,application/vnd.teiid.vdb
 </mode:excludedMimeTypes>
 <!--
 A comma- or whitespace-delimited list of MIME types that are to be included.
 If this is used, then the extractor will include only those MIME types found
 in this list for which there is an available parser (unless the MIME type
 is also excluded). Including explicit MIME types is often easier if text is
 to be extracted for are only a few MIME types.
 -->
 <mode:includedMimeTypes>
    application/msword,application/vnd.oasis.opendocument.text
 </mode:includedMimeTypes>
</mode:textExtractor>
```

Note that because Tika can process many different MIME types, you can easily specify which MIME types should be included or excluded. It is considered a best practice to specifically include

all of the MIME types from which text should be extracted. One reason is that text extraction can be an expensive operation, so you may want to limit it to a specific set of file types. Second, explicitly listing out all of the MIME types is much easier to see and understand. And third, Tika supports a few MIME types without extra libraries, but generally it requires additional dependencies for each type of file, and you probably want to depend on only those libraries that you actually need.

After changing the configuration, be sure to include the necessary libraries. If your application is using Maven, you will need the following dependency:

```xml
<dependency>
   <groupId>org.modeshape</groupId>
   <artifactId>modeshape-extractor-tika</artifactId>
   <version>2.5.0.Beta-1</version>
</dependency>
```

plus the following dependencies based upon the file types:

Table 37.1. Tika third-party dependencies that must be manually included

Dependency	Description of files

``xml
<dependency>
 <groupId>org.apache.commons</groupId>
 <artifactId>commons-compress</artifactId>
 <version>1.1</version>
</dependency>
``
	Compressed archive formats, such as 'ar', 'cpio', 'tar', 'zip', 'gzip' and 'bzip2'.

``xml
<dependency>
 <groupId>asm</groupId>
 <artifactId>asm</artifactId>
 <version>3.1</version>
</dependency>
``
	Used for parsing Java files.

``xml
<dependency>
 <groupId>com.drewnoakes</groupId>
 <artifactId>metadata-extractor</artifactId>
 <version>2.4.0-beta-1</version>
``
	Exif and other image metadata.

Dependency	Description of files
`</dependency>`	

`<dependency>`
 `<groupId>de.l3s.boilerpipe</groupId>`
 `<artifactId>boilerpipe</artifactId>`
 `<version>1.1.0</version>`
`</dependency>`
	Boilerpipe [http://code.google.com/p/boilerpipe/] HTML templates

`<dependency>`
 `<groupId>rome</groupId>`
 `<artifactId>rome</artifactId>`
 `<version>0.9</version>`
`</dependency>`
	RSS and Atom feeds using the *Rome* [http://java.net/projects/rome/] library.

`<dependency>`
 `<groupId>edu.ucar</groupId>`
 `<artifactId>netcdf</artifactId>`
 `<version>4.2-min</version>`
`</dependency>`
`<dependency>`
 `<groupId>commons-httpclient</groupId>`
 `<artifactId>commons-httpclient</artifactId>`
 `<version>3.1</version>`
`</dependency>`
	NetCDF [http://www.unidata.ucar.edu/software/netcdf/] and *HDF* [http://en.wikipedia.org/wiki/Hierarchical_Data_Format] file formats, which are used within the scientific data community but generally not elsewhere.

`<dependency>`
 `<groupId>org.apache.james</groupId>`
 `<artifactId>apache-mime4j</artifactId>`
 `<version>0.6</version>`
`</dependency>`
	Raw email messages and mbox files typically used within a file-based email system.

The following dependencies are automatically included by the Tika text extractor module, but if any are not needed in your application or project may be explicitly excluded without problems.

Table 37.2. Tika third-party dependencies (included by default)

Dependency	Description of files

``xml
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi</artifactId>
  <version>${poi.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi-scratchpad</artifactId>
  <version>${poi.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi-ooxml</artifactId>
  <version>${poi.version}</version>
  <exclusions>
    <exclusion>
      <groupId>stax</groupId>
      <artifactId>stax-api</artifactId>
    </exclusion>
    <exclusion>
      <groupId>xml-apis</groupId>
      <artifactId>xml-apis</artifactId>
    </exclusion>
  </exclusions>
</dependency>
``
	Microsoft Office and Open Office file formats

``xml
<dependency>
  <groupId>org.apache.geronimo.specs</groupId>
  <artifactId>geronimo-stax-api_1.0_spec</artifactId>
  <version>1.0.1</version>
</dependency>
``
	XML files

``xml
<dependency>
  <groupId>org.ccil.cowan.tagsoup</groupId>
``
	HTML files

Dependency	Description of files
<artifactId>tagsoup</artifactId> <version>1.2</version> </dependency>	
| ```
<dependency>
  <groupId>org.apache.pdfbox</groupId>
  <artifactId>pdfbox</artifactId>
  <version>1.4.0</version>
</dependency>
<!-- TIKA-370: PDFBox declares the Bouncy
 Castle dependencies
    as optional, but we prefer to have them
 always to avoid
    problems with encrypted PDFs. -->
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcmail-jdk15</artifactId>
  <version>1.45</version>
</dependency>
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcprov-jdk15</artifactId>
  <version>1.45</version>
</dependency>
``` | PDF files |

If you're not using Maven, the be sure to put onto your classpath all of the JAR files from the Maven modules listed above.

Writing custom text extractors

Creating a custom text extractor involves the following steps:

- Create a Maven 3 project for your detector;

- Implement the *TextExtractor* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/graph/text/TextExtractor.html] interface with your own implementation, and create unit tests to verify the functionality and expected behavior; and

- Deploy the JAR file with your implementation (as well as any dependencies), and make them available to ModeShape in your application via ModeShape's configuration as described *earlier*.

It's that simple.

The first step is to create the Maven 3 project that you can use to compile your code and build the JARs. Maven 3 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 3, 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-extractor-tika** 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://github.com/ModeShape/ modeshape//tree/modeshape-2.6.0.Beta2/extensions/modeshape-extractor-tika/*

You can create your Maven project any way you'd like. For examples, see the *Maven 3 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>
```

```xml
  <version>2.5.0.Beta-1</version>
</dependency>
<dependency>
  <groupId>org.modeshape</groupId>
  <artifactId>modeshape-graph</artifactId>
  <version>2.5.0.Beta-1</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.8.4</version>
</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:

```xml
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8</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>1.8.4</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.16</version>
  <scope>test</scope>
</dependency>
```

After you've created the project, simply implement the *TextExtractor* [http://docs.jboss.org/ modeshape/2.6.0.Beta2/api/org/modeshape/graph/text/TextExtractor.html] interface. As mentioned in the JavaDoc, the "supportsMimeType" method will be called by ModeShape first, and only if your implementation returns true for a given MIME type will the "extractFrom" method be called. The supplied `TextExtractorContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/ org/modeshape/graph/text/TextExtractorContext.html] object provides information about the text being processed, while the *TextExtractorOutput* [http://docs.jboss.org/modeshape/2.6.0.Beta2/ api/org/modeshape/graph/text/TextExtractorOutput.html] is a simple interface that your extractor uses to record one or more strings containing the extracted text.

Testing should be quite straightforward as text extractors can simply be instantiated and called by your test methods, which can simply instantiate the `TextExtractorContext` [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/modeshape/ graph/text/TextExtractorContext.html] class (with the correct information) and either mock or implement the *TextExtractorOutput* [http://docs.jboss.org/modeshape/2.6.0.Beta2/api/org/ modeshape/graph/text/TextExtractorOutput.html] interface. Again, see the test cases in the *Tika text extractor module* [http://github.com/ModeShape/modeshape//tree/modeshape-2.6.0.Beta2/ extensions/modeshape-extractor-tika/] for ideas.

Part VII. Administration and Monitoring

The ModeShape project provides an RHQ plugin for use in the Administrative Console available in JBossAS. The console is accessible at http://{host}:{port}/admin-console.

ModeShape Administration Console

The ModeShape Administration Console is a web based administrative and monitoring tool for ModeShape. ModeShape's Administration Console is built using the RHQ library and adds an additional plugin into the Embedded JOPR application already available in the JBoss AS.

Figure 39.1. Administration Console

39.1. What can be managed and/or monitored?

- The ModeShape Engine

- Repositories

- The Sequencing Service

- Sequencers

- Connectors

39.2. Configuration

While you cannot directly configure ModeShape through the Administration Console, the configuration properties are displayed for several ModeShape components including:

- Repositories

- Sequencers

- Connectors

39.3. Metrics

Table 39.1. Repository

# of Active Sessions	The number of JCR sessions that are currently active.

Table 39.2. Sequencing Service

Number Of Nodes Sequenced	The number of nodes sequenced.
Number Of Nodes Skipped	The number of nodes that were skipped because no sequencers applied.

Table 39.3. Connectors

Total Connections in Use	The number of connections in use.

39.4. Control (Operations)

Table 39.4. ModeShape Engine

Start/Restart	Start the engine or restart if it's already running.
Shutdown	Stop the engine.

Table 39.5. Sequencing Service

Number Of Nodes Sequenced	The number of nodes sequenced.
Number Of Nodes Skipped	The number of nodes that were skipped because no sequencers applied.

Table 39.6. Connectors

Ping	Ping the connector to test availability.

Looking to the future

ModeShape 2.6.0.Beta2 provides a very capable JCR implementation with powerful and unique features not matched by other implementations. Version 2.6.0.Beta2 improves the JDBC driver, the JBoss AS deployment, and the RHQ monitoring plugin, along with numerous other fixes and improvements. Of course, the release also comes with quite a few *bug fixes and improvements* [http://docs.jboss.org/modeshape/2.6.0.Beta2/release.html].

ModeShape now implements all of the required JCR 2.0 features:

- repository acquisition

- authentication

- reading/navigating

- query

- export

- node type discovery

- permissions and capability checking

and implements most of the optional JCR 2.0 features:

- writing

- import

- observation

- workspace management

- versioning

- locking

- node type management

- same-name siblings

- orderable child nodes

ModeShape supports the following query languages:

- *JCR-SQL2* defined in *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283]

- *JCR-QOM* defined in *JSR-283* [http://www.jcp.org/en/jsr/detail?id=283]

- *XPath* defined in *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170]

- *JCR-SQL* defined in *JSR-170* [http://www.jcp.org/en/jsr/detail?id=170]

- *Full-text search* that is similar to a search engine

At this point, ModeShape passes virtually all of the JCR Technology Compatibility Kit (TCK) tests, except for a few known issues with ModeShape as well as bugs in the TCK tests. As soon as these are fixed, we'll start the process of attaining JCR 2.0 certification for ModeShape.

What's next for ModeShape? The focus for the next release will be to focus on optimization and performance improvements. Other items on our longer term *roadmap* [http://jira.jboss.org/browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:roadmap-panel] include a web user interface, Seam integration, and integration with even more kinds of information systems and repositories.

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/browse/MODE#selectedTab=com.atlassian.jira.plugin.system.project:summary-panel].

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/2.6.0.Beta2/manuals/gettingstarted/html/index.html] guide, which has examples that you can build and run to see ModeShape in action.