Introducing ModeShape 3.0 (DRAFT)

ModeShape 3.0 is the first JCR content repository that is seriously fast, extremely scalable, and highly available. It's easy and lightweight enough to embed into small application, but also powerful enough to build a large peer-to-peer cluster capable of storing huge volumes of content and supporting many concurrent and active clients. It's fully open source with a business-friendly license and written in Java, so it can run almost anywhere.

This next generation of ModeShape has been redesigned as a thin JCR implementation on top of the superb Infinispan data grid platform. That means that ModeShape can be run in a local mode within a single process, or in a clustered fashion where all your content is either replicated or distributed across a large cluster of machines. Distributed mode is by far the most powerful, since each bit of content is stored in a fixed number of places in the cluster. This provides resiliency to server failures to ensure your content is always available, and is extremely scalable as the work to store content is constant even as more and more machines are added to the cluster. And ModeShape is fast, because all content is stored within the data grid's massive heap, where ModeShape can quickly access and update it. No longer does accessing content require talking to a large single database server!

ModeShape offers:

- **JCR 2.0 API** - ModeShape supports all of the required features of JSR–283 and most of the optional features. ModeShape supports all four JCR query languages (JCR-SQL2, JCR-QOM, JCR-SQL, and XPath) plus search-engine-like full-text search.
- **Large Repositories** - The amount of content you can store is a function of the size of the data grid and the number of copies stored. For example, if you have 100 blade servers each with 2GB of memory dedicated to the data grid and configured to store 4 copies of the data, that's 50GB of space for your content. Simply add more machines to increase the available memory, and the data grid automatically evolves to rebalance itself.
- **High availability** - Configure as a replicated or distributed grid to ensure multiple copies of the content survive hardware failures. Optionally back the cache with databases, file systems, cloud storage, or other stores to enable "warm starts", ensure data survives cluster-wide restarts, and overflow should your the content grow faster than the available memory of your cluster.
- **Scalability** - Discovering new machines and propagation of JCR events requires cluster-wide communication, but all other access patterns use peer-to-peer communication, where changes to content are sent only to those members that have a copy of the content. You can dynamically add or remove machines from the cluster, without incurring any downtime. In short, ModeShape scales very well and very easily.
- **Federation** - ModeShape pioneered the JCR repository that accesses data and content you already have. Federation is now more flexible, easier to configure, more efficient, and automatically leverages the distributed caching capabilities of the data grid. ModeShape comes with Infinispan Cache Loaders to access other JCR repositories, the file system and even SVN repositories, but it's pretty easy to write your own.
- **Sequencing** - Upload files and have ModeShape automatically extract the useful bits and derive useful structured content. ModeShape comes with several sequencers for common file types, but you can easily write your own.
- **Remote Access** - JCR client applications can embed a ModeShape instance that has a local cache but talks to a remote cluster, or they can embed a ModeShape instance that actively joins the cluster. Applications can also talk to a remote ModeShape instance using WebDAV or a RESTful API, and can even query remote repositories using the JDBC API and ModeShape's JDBC driver.
- **Transactions** - ModeShape now supports and can participate in JTA transactions. Optionally enable deadlock detection to automatically identify and rollback deadlocked transactions using a single repository.
- **Management** - Monitor and manage a ModeShape cluster, the repositories, sessions, queries, and other aspects using a plugin for RHQ, JBoss's enterprise management solution.

Other minor features include:

- **Flat hierarchies** - JCR repositories are often designed to be well-formed hierarchies that are not too flat, and use same-name siblings with caution. These constraints are often due to implementation details and are not an inherent limitation of the JCR API. As proof, ModeShape now has excellent support for efficiently handling hundreds of thousands of children under a single node, even when many of those nodes share similar names.
- **Large values** - JCR makes it very easy to store files within the repository, and JCR client applications shouldn't have to worry about storing duplicate files in multiple places. ModeShape automatically stores files, binary values, and large string values in a central area (keyed by the SHA1 secure hash of the content), so the same values are stored in the data grid only once.
- **Distributed queries** - ModeShape can now distribute and parallelize a query to all of the members of the cluster, allowing ModeShape to more quickly efficiently process the query and return the results. Lucene indexes are stored in the Infinispan cluster, as with Hibernate Search 4.
- **Computations** - Pass a Runnable around the repository's data grid to push complex data analyses to where the content is stored locally, using a map/reduce mechanism to efficiently process all content to produce (relatively) small amount of output.
- **Validation** - At any time, request an analysis to validate the entire content of the repository matches a set of node type definitions. This can be done using the node type definitions registered with the repository, or with a proposed set of node type definitions and modifications to identify whether any existing content would violate the proposed definitions. Validation is asynchronous and status and results are stored within the repository's "`/jcr:system`" area for later access.
- **Security** - ModeShape has a pluggable authentication and authorization model, with out-of-the-box support for JAAS integration, servlet authorization, and anonymous logins. Create your own custom providers by implementing simple interfaces.
- **JDBC API** - ModeShape provides a JDBC driver that allows client applications to query a local or remote repository using JCR-SQL2 queries. The driver also supports database metadata, enabling metadata-driven applications to discover tables and columns to dynamically build queries.
- **Deploy to JBoss AS7** - ModeShape has a kit for installing itself as a service within JBoss AS7. After installation, use the JBoss AS7 configuration tools (e.g., the command line utility, RESTful service, or web console) to manage and configuration ModeShape across your AS7 clsuter. This includes: deploying new repositories, starting and stoping repositories, changing repository configurations (while still running!), monitoring the health of repositories, and stopping and undeploying repositories.

Changes

ModeShape 3.0 behavior attempts to follow the JCR specification, but a few behaviors have changed in ModeShape 3.0. These changes are often dictated by parts of the specification that allow for variation and/or optional support:

- **Visibility of session changes** - Before a session persists its changes (using 'Session.save()' or committing the transaction), those transient changes are only visible to that session. However, once, those transient changes are persisted, the changes are immediately visible to all other active sessions. This behavior now matches other JCR implementations and will make it easier for client applications to switch JCR implementations or work with multiple JCR implementations. In ModeShape 2.x, a session would only see the changes persisted by other sessions when the session was saved or refreshed. (See MODE–951)
- **Change primary type** - ModeShape now supports changing the primary type of an existing node, assuming that the new primary type makes sense given existing properties and the parent's child node definitions.
- **Simplify reading of node types** - ModeShape now makes it easier to load custom node types from CND files, by extending the 'javax.jcr.nodetypes.NodeTypeManager' interface with additional methods to (re-)register node types found in a CND file.

Other changes are due to the major refactoring and simplification of ModeShape 3.0. Several of the extension APIs have changed, others moved to different packages (old packages remain for a few releases), and a few have been completely deprecated and replaced by more other APIs.

- **Configuration changes** - ModeShape 2.x used a single configuration file for multiple repositories and sources. ModeShape 3.0 uses a completely different approach: the engine itself requires no configuration, while each repository is defined by a separate configuration file (which often can be almost trivially small, since the defaults work very well most of the time). This new approach means that configuring a repository will be simpler and independent of the other repositories, while also making it possible to add, change, and remove one repository while the others are still being used. Plus, much of a repository's configuration can be changed while that repository is running and being used. ModeShape provides a utility to help migrate ModeShape 2.x installations to ModeShape 3.x installations, including the conversion of your 2.x configuration file to the corresponding 3.x configuration files, plus a utility to transfer all content from the 2.x sources into the 3.x installation.
- **Connectors** - The connector API and connector implementations provided in ModeShape 2.x are no longer supported. A few are no longer meaningful (e.g., the Infinispan or JBoss Cache connector), and most others have either been converted to Infinispan CacheLoader implementations or are superseded by existing Infinispan CacheLoaders (e.g., BerkleyDB, JDBC, file, and cloud cache loaders). If you've developed a custom connector and wish for help or guidance in adapting it to the new cache loader API, please contact us on our forum. Generally speaking, CacheLoader implementations are easier to implement.
- **Sequencers** - The old sequencer API still works in 3.0, but it has been deprecated and replaced by a new API that uses the JCR API and is significantly more flexible. All sequencers provided by ModeShape 2.x have been rewritten in 3.0 to use the new API. The old API will be removed in a future release. If you've developed a custom sequencer and wish for help or guidance in adapting to the new API, please contact us on our forum.
- **Text extractors** - The ModeShape 2.x API remains but has been deprecated, replaced by an identical interface in a different package. Custom text extractors based upon the 2.x API will continue to work in 3.0, but we urge you to change your extractor by simply changing a single import statement. The 2.x text extractor API will be removed in a future release.
- **MIME type detectors** - The ModeShape 2.x API remains but has been deprecated, replaced by an identical interface in a different package. Custom detectors based upon the 2.x API will continue to work in 3.0, but we urge you to change your extractor by simply changing a single import statement. The 2.x MIME type detector API will be removed in a future release.
- **JBoss AS 5 & 6** - ModeShape no longer provides kits for installing into a JBoss AS 5.x or 6.x installation. JBoss AS 7 is far superior to these older versions.

The ModeShape Engine

Like with ModeShape 2.x, running ModeShape is a matter of creating a single instance of the JcrEngine class. However, this class has changed significantly in 3.0 to add some very nice features.

1. The ModeShape engine requires no configuration: simply instantiate it and start it up.
2. Each repository is defined by its own configuration.
3. Add and remove repositories while the engine is running.
4. Start and stop repositories independently, while the engine is running.
5. Change the configuration of a repository, while it and the engine are running.
6. Shutting down the engine shuts down all repositories.

We'll cover each of these in more detail in the next few sections.

EmbeddedConfiguration

In ModeShape 2.x, you configured the engine and all repositories with a single configuration file. While it was convenient that this one file contained all configuration information, there were a number of significant disadvantages. For example, it was not possible to change the configuration without shutting down and restarting the engine (and all repositories). It also proved to be confusing for users, who didn't always understand how the various parts of the engine were shared/reused by one or more repositories.

When ModeShape 3.0 is embedded into an application, each repository has a separate configuration file and is treated completely independently than all other repositories within the engine. This means it is now possible to deploy a new repository to a running engine by simply providing the configuration for the new repository. A repository configuration is defined via a JSON file, and almost everything within the configuration has a default that makes sense in many/most situations[1].

For example, the following is a perfectly good repository configuration file:

```
{ "name" : "MyRepository" }
```

This repository will allow anonymous read/write access, will use the "MyRepository" Infinispan cache (which is configured separately), will bind itself into JNDI at "java:jcr/local/MyRepository", will allow clients to create and destroy workspaces, and will have no sequencers configured.

Of course, this can be changed by overriding the defaults in the configuration file. Here's another configuration file that has many more of the options specified:

```
{
  "name" : "My Repository",
  "jndiName" : "java:jcr/local/MyRepository",
  "largeValueSizeInBytes" : 10240,
  "monitoring" : { "enabled" : true },
  "storage" : { "cacheName" : "some-cache", "cacheConfiguration" : "infinispan.xml" },
  "workspaces" : { "predefined" : ["ws1","ws2"], "default" : "ws1", "allowCreation" : true },
  "security" : {
    "anonymous" : {
      "roles" : ["readonly","readwrite","admin"],
      "useOnFailedLogin" : false
    },
    "providers" : [
      { "classname" : "com.example.MyAuthenticationProvider" },
      { "type" : "JAAS", "policyName" : "modeshape-jcr" }
    ]
  },
  "query" : {
    "enabled" : true,
```

```
    "indexLocation" : "/some/path",
    "rebuildUponStartup" : "ifMissing",
    "extractors" : [ { "type" : "Tika" } ]
  },
  "sequencing" : {
    "removeDerivedContentWithOriginal" : true,
    "sequencers" : [
      {
        "name" : "ZIP Files",
        "type" : "ZipSequencer",
        "pathExpression" : "default:/files(//)(*.zip[*])/jcr:content[@jcr:data] => default:/sequenced/zip/$1",
        "description" : "Sequences ZIP files loaded under '/files' and extracting into '/sequenced/zip/$1'",
      },
      {
        "name" : "Delimited Text File Sequencer",
        "classname" : "org.modeshape.sequencer.text.DelimitedTextSequencer",
        "pathExpressions" : [
            "default:/files//(*.csv[*])/jcr:content[@jcr:data] => default:/sequenced/text/delimited/$1"
        ],
        "splitPattern" : ","
      }
    ]
  }
}
```

The repository configuration JSON files are validated with a JSON Schema, ensuring that the configuration files are valid. In fact, we'll even see how you can even validate a configuration before you use it.

Managing the ModeShape engine

Instantiating and starting a ModeShape engine to run embedded with your own application is now trivially easy:

```
JcrEngine engine = new JcrEngine();
engine.start();
```

An engine that is not started is extremely lightweight, and contains a single empty collection and a few null references. Even when started, however, the engine only contains a few components: an executor service for background operations (see below) and a scheduled garbage collection process that runs every 30 seconds.

Deploying and starting repositories

To deploy a repository, simply load the repository's configuration from a JSON file and use the engine to deploy it:

```
java.io.File configFile = ...
RepositoryConfiguration repoConfig = RepositoryConfiguration.read(configFile);
engine.deploy(repoConfig);
```

The `deploy(RepositoryConfiguration)` method will validate the configuration, and throw a `ConfigurationException` if there are any errors in the configuration. Of course, your code can validate the configuration and get all the warning and error messages:

```
Results validationResults = repoConfig.validate();
if ( validationResults.hasErrors()) {
    // handle this ...
}
```

Assuming that the `deploy(...)` method succeeds, then the repository is deployed but not yet running. A ModeShape repository will automatically start the first time it is used (via the `login(...)` methods), but your application may want to preemptively start the repositories so that logging in has no unnecessary delays[2].

```
Future<JcrRepository> starter = engine.startRepository(repo.getName());
JcrRepository repo = starter.get();   // optional
```

Note that the `startRepository(String)` method returns immediately but is actually asynchronously starting the repository. If your code does not need to wait for the repository to start, it can simply disregard the resulting `Future`. However, if you code does want to wait for the running `JcrRepository` instance, it can simply call the `get()` method, which blocks while the repository is being started and returns the running `JcrRepository` instance (or throws an exception if there is an error during startup).

Changing a repository's configuration

ModeShape 3.0 even allows you to change the configuration for a repository, even when that repository is running and being used. Although most changes take effect immediately, a few changes related to the Infinispan cache (where the content is stored) will take effect upon the next restart.

The `RepositoryConfiguration` object representing the configuration of a repository are immutable, so they cannot be changed directly. Instead, you basically record the changes you want to make (using the configuration's editor), and then apply those changes to the configuration. One added benefit to this is that should anyone else apply unrelated changes after you've started editing the (now out-of-date) configuration, your changes can easily be applied on top of others.

The basic outline for modifying a repository's configuration is as follows:

1. Get the current configuration for the repository that is to be changed
2. Get an editor from that configuration
3. Use the editor to capture and validate the changes you want to make

4. Update the repository's configuration with the changes from the editor

Here's some code that shows how this is done:

```
JcrEngine engine = ...
Repository deployed = engine.getRepository("repo");
RepositoryConfiguration deployedConfig = deployed.getConfiguration();

// Create an editor, which is actually manipulating a copy of the configuration document ...
Editor editor = deployedConfig.edit();

// Modify the copy of the configuration (we'll do something trivial here) ...
editor.setNumber(FieldName.LARGE_VALUE_SIZE_IN_BYTES,8096);

// Get our changes and validate them ...
Changes changes = editor.getChanges();
Results validationResults = deployedConfig.validate(changes);
if ( validationResults.hasErrors() ) {
    // you've done something wrong with your editor
} else {
    // Update the deployed repository's configuration with these changes ...
    Future<Boolean> future = engine.update("repo",changes);

    // Optionally block while the repository instance is changed to
    // reflect the new configuration ...
    JcrRepository updated = future.get();
}
```

Note that strictly speaking you don't need to validate your changes, because the `update(...)` method will always do that before it applies the changes. But it is recommended to make sure you're making the changes that you think you're making.

Also, Note that the `update(...)` method blocks while the changes to the configuration are validated and applied, but it then asynchronously changes the repository instance to reflect this configuration and returns a future to that asynchronous process. As with the other methods that return futures, your code can choose to block while the update is being performed, or can completely disregard the future.

Shutting down and undeploying repositories

At any time, repositories may be shutdown, preventing the creation of new sessions and terminating all existing sessions:

```
Future<Boolean> future = engine.shutdownRepository("MyRepository");
boolean success = future.get();   // optional
```

Again, note that the `shutdownRepository(String)` method initiates an asynchronous shutdown and immediately returns a `Future` to the shutdown activity. If your code does not need to wait for the shutdown to complete, it can simply disregard the future. However, if your code does want to wait for the shutdown to complete, it can simply call the `get()` method on the future, which blocks until the repository has been shutdown. Since this is a standard `java.util.concurrent.Future`, your code can also block for a maximum amount of time using the `get(long,TimeUnit)` method.

It is also possible to undeploy a repository to completely remove it from the engine.

```
Future<Boolean> future = engine.undeploy("MyRepository");
boolean success = future.get();   // optional
```

Again, this method performs the work asynchronously and returns a `Future` that can optionally be used to wait for the undeploy to complete. Generally, undeploying a repository that is already shutdown will complete very quickly. However, undeploying a repository that has not yet been shutdown will first shut it down before undeploying it.

Getting a JCR Repository instance

Applications need to get a hold of a JCR repository in order to use it, but we recommend minimizing your application's dependency on ModeShape-specific APIs. But there are a number of approaches recommended by the JCR specification:

Use JNDI

Perhaps the easiest is to simply look up the `javax.jcr.Repository` instance in JNDI. Recall that ModeShape will by default register each repository instance in JNDI, so as long as JNDI is available to your application (e.g., it is not within a standard Java SE environment) and you haven't explicitly turned off JNDI registration for the repository, you can use this approach:

```
Context context = ...
javax.jcr.Repository repo = context.lookup("java:jcr/local/MyRepository");
```

This approach is clean, but does require that the ModeShape engine is running and has the required libraries already deployed and started. Therefore, it works really well within web applications deployed to application servers (such as JBoss AS) where ModeShape is already deployed.

Use JCR's RepositoryFactory

The JCR 2.0 specification introduce a new mechanism to obtain a JCR repository instance using the Java ServiceLoader mechanism. Like the JNDI approach, this approach requires only standard APIs, and while his approach is more complex than just using JNDI directly, it does offers more flexibility: for example, with this approach the ModeShape engine does not already have to exist.

As shown below, it relies upon only the standard JCR 2.0 API and a few properties that are implementation-specific (which your application can read from a properties file if desired):

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

This code looks for all `RepositoryFactory` 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. Since the parameters are implementation-specific, only the correct implementation will understand the parameters.

ModeShape uses a single property named “`org.modeshape.jcr.URL`” with a value that contains one of the following:

1. the name in JNDI where the repository is bound, or
2. the name in JNDI where the engine is bound the name of the repository, or
3. a resolvable location of the repository configuration file, which will be deployed to a singleton ModeShape engine instance

The following table shows examples of “`org.modeshape.jcr.URL`” values:

ModeShape RepositoryFactory URLs

`org.modeshape.jcr.URL` value	**Description**
`jndi:name/in/jndi/MyRepository`	Use the Repository instance found in JNDI at `name/in/jndi`
`jndi:name/in/jndi?repositoryName=MyRepository`	Use the ModeShape engine instance found in JNDI at `name/in/jndi` and get the repository named “MyRepository”
`file://path/to/repositoryConfigFile.json`	Use a singleton ModeShape engine instance and deploy the repository (if not already deployed) described by the configuration file found at the given location on the file system
`http://www.example.com/path/to/repositoryConfigFile.json`	Use a singleton ModeShape engine instance and deploy the repository (if not already deployed) described by the configuration file found at the URL
`classpath:path/to/repositoryConfigFile.json`	Use a singleton ModeShape engine instance and deploy the repository (if not already deployed) described by the configuration file found at the given location on the classpath

Use the ModeShape Engine

This approach is one of the easiest, but we recommend using it *only* when the previously described approaches cannot be used. Assuming your application has a reference to the engine, then getting the `Repository` instance is a matter of a single method call:

```
javax.jcr.Repository repo = engine.getRepository("MyRepository");
```

Monitoring

ModeShape 3.0 includes a completely revamped mechanism for monitoring the state and health of the repositories. By default, each repository records the metric information, although this can be disabled in the repository configuration. (Note that it cannot be disabled in JBoss AS7.) Each running `JcrRepository` instance contains a `RepositoryStatistics` object with methods that expose the histories for a variety of metrics, including the number of:

1. active sessions
2. active queries
3. workspaces
4. listeners
5. opened non-scoped locks
6. opened session-scoped locks

plus several metrics that record durations of:

1. query execution time
2. session lifetime
3. sequencer execution time

The RepositoryStatistics interface provides a way to get the history for a particular metric during a specified window, where the statistics (the average value, minimum value, maximum value, variance, standard deviation, number of samples, and time interval of the statistics) are given for:

1. each ten 5-second intervals during the last minute (60 seconds); or
2. each minute during the last hour (60 minutes); or
3. each hour during the last day (24 hours); or
4. each day during the last week (7 days); or
5. each week during the last year (52 weeks)

The following code sample shows how to obtain the query execution time statistics for each hour during the last day:

```
JcrEngine engine = ...
JcrRepository repo = engine.getRepository("MyRepository");
RepositoryStatistics stats = repo.getRepositoryStatistics();
History hourly = stats.getHistory(DurationMetric.QUERY_EXECUTION_TIME,Window.PREVIOUS_24_HOURS);

// Extract the information from the history ...
DateTime startingTime = hourly.getStartingTime();
```

```
  DateTime endingTime = hourly.getEndingTime();
  // Get the statistics during each hour of the past 24 hours ...
  for ( Statistics stat : hourly.getStatistics() ) {
      int count = stat.getCount();
      long min = stat.getMinimum();
      long max = stat.getMaximum();
      double avg = stat.getMedian();
      double var = stat.getVariance();
      double std = stat.getStandardDeviation();
      // do something with this info
  }
```

It's also possible to get the longest duration activities for the duration-related metrics. The following example shows how to get information about the sessions with the longest lifetimes:

```
JcrEngine engine = ...
JcrRepository repo = engine.getRepository("MyRepository");
RepositoryStatistics stats = repo.getRepositoryStatistics();
DurationActivity[] lifetimes = stats.getLongestRunning(DurationMetric.SESSION_LIFETIME);
for (DurationActivity activity : lifetimes) {
    long duration = activity.getDuration(TimeUnit.SECONDS);
    String username = activity.getPayload();
    // do something with this info
}
```

Deploying to JBoss AS7

ModeShape 3.0 has excellent integration with JBoss AS7, and can easily be deployed as a service within the application server. Repositories can be added, configured, changed, started, stopped, clustered, monitored, and removed all through the standard JBoss AS7 mechanisms (e.g., command line utility, RESTful interface, and the web console). This approach replaces the use of the per-repository JSON configuration files, since the JBoss AS7 configuration tools allow centralized management and configuration of a cluster of machines.

(Show the steps for installing the ModeShape subsystem and configuring a repository.)

Using RHQ

If you're using RHQ for monitoring your JBoss AS7 cluster, you can install the ModeShape RHQ plugin and use RHQ to monitor the health and state of the repositories.

How ModeShape Uses Infinispan

One of the major changes in ModeShape 3.0 is the change to being a thin JCR implementation layer on top of the Infinispan data grid platform. Each node within a repository is represented as an Schematic document[3], which is essentially an in-memory object representation of JSON/BSON-style documents. (Technically, the first block of children are always stored in the node's document, but subsequent children may be stored in other blocks.)

JSON documents contains two types of structures: an unordered collection of name/value pairs, and an ordered list of values (e.g., arrays). Values can be strings, booleans, numbers, and even other JSON objects. Thus it's possible to create hierarchical data structures, such as this example JSON document:

```
{
  "product-id" : "1493AB38-43D",
  "name" : "RainBGone Small Umbrella",
  "description" : "Ultra compact umbrella",
  "colors" : ["black","red","blue","green","yellow"],
  "price" : 7.99,
  "priceUnit" : "dollars",
  "vendor" : {
    "name" : "RainBGone, Inc.",
    "vendor-id" : "39bce35884f3ad33"
  }
}
```

While JSON documents are written in textual format, the BSON specification defines a binary format and adds more value types, including URIs, timestamps (without timezone), nulls, binary data, regular expressions, and several other types.

Schematic documents are in-memory representations with the same structure as JSON/BSON that can be serialized to JSON or BSON file formats. They also support Infinispan's DeltaAware interface, so when changes are made to a document and Infinispan needs to send that new document to multiple processes in the cluster, Infinispan will obtain the delta and send only that information.

Before we talk about how these documents are structured, let's first talk about the keys that ModeShape uses to identify documents within Infinispan.

Document Keys

The document keys are comprised of three parts: the source key, the workspace key, and the node identifier. All document keys are treated as strings when using Infinispan.

The source key is the first 7 characters of the SHA–1 hash[4] of the name of the logical source where the document is to be persisted; thus an Infinispan Cache Loader can base whether it is responsible for persisting the document solely based upon this part of the document key.

The workspace key is the first 7 characters of the SHA–1 hash[4] of the workspace name. Thus it is always possible to determine from the key the workspace that "owns" a particular node. However, nodes from one workspace can reference nodes (with child references, or JCR reference and weak references) that are "owned" by other workspaces. For example, all of the nodes under the "/jcr:system" branch are "owned" by the (hidden) workspace named "system", yet the "/jcr:system" node is referenced as a child of the root node in each repository workspace.

The final part of the document key is the identifier for a particular node. The only requirements for this identifier is that it be unique within a particular workspace, and that the workspace key plus the identifier be unique within the whole repository. By default, ModeShape will generate a UUID for new identifiers, but other schemes can be used[5], especially when using custom cache loaders that interface with external systems that might have a system-specific scheme.

Documents Structure

There are a number of advantages to using Schematic documents:

- In-memory documents are comprised of Java objects representing the documents, arrays, and field values. Since Infinispan is basically a monstrous heap, it keeps the values in-memory across the cluster. If ModeShape instead stored JSON strings or BSON byte arrays as Infinispan values, then using the values would require parsing the documents before they could be used.
- Documents are nestable, allowing for a high degree of flexibility in encoding the information.
- If the document for a JCR node requires additional information, it can be added to that document without changing a schema or altering all other documents. This means that information (or metadata) used internally by ModeShape but not exposed through JCR (e.g., federation rules, segmentation of children, etc) can be included in the document without impacting the more typical property and children information.
- The document structure can evolve over time, on a node-by-node basis.
- Properties can be encoded in a nested document structure, where the property values easily encoded as native Schematic types. If no native type is appropriate, values can always be encoded in a structure using nested documents. For details, see [Prototype table].
- Multi-valued property can be stored as arrays of values, allowing ModeShape to easily maintain the cardinality of all properties.
- Schematic doesn't offer a true reference value type, but JCR reference values can easily be encoded as the string form of a document key.
- Additional metadata, such as the federation rules for a node or computed values, can be easily stored on the documents The documents can store additional information and metadata

Here's an example of a basic document for a node with just a few properties and child references:

```
{
  "parent" : [],
  "properties" : {
      "http://www.jcp.org/jcr/1.0" : {
          "primaryType" : { "$name" : "{http://www.modeshape.org/1.0}root" },
          "uuid" : "cafebabe-cafe-babe-cafe-babecafebabe"
      }
  },
  "children" : [
      {
          "key" : "28f35d2system-jcr:system",
          "name" : "{http://www.jcp.org/jcr/1.0}system"
      }
      {
          "key" : "28f35d27e7491747422530-15d7-4db3-80e9-7d8cc270629f",
          "name" : "childA"
      },
      {
          "key" : "28f35d27e749171aaee8f9-8aca-4b21-9c05-8ba870f5f7b2",
          "name" : "childB"
      }
  ]
}
```

The "parent" field contains an array of document keys for the parent nodes. Most of the time, a node only has a single parent. However, there are exceptions: the root node for each workspace has no parents, while other nodes (such as shareable nodes) may have multiple parents. So when a node with multiple parents is materialized into a workspace, only the parents in that workspace and the parents in the node's workspace are use, while the other parents are ignored. The first parent then becomes the "primary parent" (that is, the location where the node actually exists), while any other parents become "secondary parents" that still contain the node (e.g., a shareable node) as a child.

You might be able to infer that the JCR properties are stored the in the nested "properties" document, with each property stored by its local name within a nested document for the namespace. Many of the JCR property types (such as 'string', 'long', 'double', 'boolean', 'URI', and 'binary') can be represented using BSON native types, while other property types (such as 'name', 'path', 'reference', 'weak reference', 'date', 'decimal', etc.) do not have a corresponding BSON type and are represented with nested documents following certain patterns. The "primaryType" property value, for example, shows how a name value is stored. BSON arrays are used to store the values for all multi-valued properties, even if empty.

A node's list of children are stored within the "children" field, whose value is always an array of references to the child nodes, represented as a nested document with a "key" field containing the document key for the child node and a "name" field containing the expanded form of the namespaced name. Note that the same-name-sibling (SNS) index is never stored, as it will always be computed based upon the other referenced children.[6].

When the number of children is large, storing all of the child references within the node's document means that the child references take an much more significant portion of the document, increasing the overhead for materializing the node before the properties can be read. Therefore, ModeShape supports breaking the list of child references into multiple blocks of *n* child references, where the first block is stored within the node's document but subsequent blocks are stored in their own document.

Here is a portion of a node document that stores it's 100 children in two blocks of 50 references:

```
{
    "parent" : "28f35d27e74917cafebabe-cafe-babe-cafe-babecafebabe",
    "properties" : { ... }
```

```
    },
    "childrenInfo" : {
        "count" : 100,
        "blockSize" : 50,
        "nextBlock" : "ba986b40-6dcc-4580-b0b6-dacbc298ada2",
        "lastBlock" : "ba986b40-6dcc-4580-b0b6-dacbc298ada2"
    },
    "children" : [
        {
            "key" : "28f35d27e7491747422530-15d7-4db3-80e9-7d8cc270629f",
            "name" : "child1"
        },
        {
            "key" : "28f35d27e749171aaee8f9-8aca-4b21-9c05-8ba870f5f7b2",
            "name" : "child2"
        },
        ...
    ]
}
```

Note how the nested “childrenInfo” document contains the total child count, the size of the current block, the key to the next block, and the key for the last block (which is the same as the “nextBlock” value). The document containing the second block contains the 51–100th child references:

```
{
    "childrenInfo" : {
        "blockSize" : 50
    },
    "children" : [
        {
            "key" : "28f35d27e7491742256439-aadf-49f6-93f0-89d677c75cb3",
            "name" : "child51"
        },
        {
            "key" : "28f35d27e749172ef58009-28ab-4348-8445-b92b629e97e1",
            "name" : "child52"
        },
        ...
    ]
}
```

Of course, this pattern works for any number of blocks, and different sized blocks. In fact, ModeShape uses an asynchronous process to optimize the size of each block: blocks that are “too small” are merged with adjacent blocks, while blocks that are “too large” are split into multiple blocks. The algorithm is written such that any reads of the blocks will always produce consistent children. And future improvements could keep the initial block (within the node’s document) relatively small (e.g., 100) while using larger block sizes (e.g., 500 or 1000) for the remaining blocks. The algorithm could even adjust the ideal lock size based upon the total number of nodes.

Large values

With JCR, clients can store very large STRING and BINARY values in node properties. First of all, if large values are stored in the node’s document, then accessing the node requires materializing the entire node document, even when the large value is not accessed. Therefore, storing large values outside of the node document improves performance for most access patterns.

If the large values are stored in Infinispan with keys determined by the SHA–1 hash of the value, then each value needs to be stored only once, even if it is used in values for separate properties. These large values would also be store and workspace independent. ModeShape uses an asynchronous garbage collection process that uses Infinispan’s map/reduce framework and identifies and removes unused large values.

Property values

The following table lists the value types used for each of the JCR property types, where the `Document` and `Binary` classes are in the `org.modeshape.schematic.document` package:

Value type formats

JCR property type	Java type	Example
string	java.lang.String	`“This is the value.”`
boolean	java.lang.Boolean	`true`
long	java.lang.Long	`38443443294`
double	java.lang.Double	`3.14159d`
binary	org.i.s.d.Binary	<binary>
URI	java.net.URI	`“http://www.jcp.org/jcr/1.0”`
UUID	org.m.s.d.Document	`{ “$uuid” : “cafebabe-cafe-babe-cafe-babecafebabe” }`
decimal	org.m.s.d.Document	`{ “$dec” : “1.23456e7” }`
reference	org.m.s.d.Document	`{ “$ref” : “28f35d27e7491747422530–15d7–4db3–80e9–7d8cc270629f” }`
weak reference	org.i.s.d.Document	`{ “$wref” : “28f35d27e7491747422530–15d7–4db3–80e9–7d8cc270629f” }`

name	org.m.s.d.Document	`{ “$name” : “{http://www.jcp.org/jcr/1.0}system” }`
path	org.m.s.d.Document	`{ “$path” : [ “http://www.jcp.org/jcr/1.0}system”,“http://www.jcp.org/jcr/1.0}nodeTypes” ], “$relative” : false }`
date	org.m.s.d.Document	`{ “$date” : “1970–01–01T00:00:00.000Z” }`
decimal	org.m.s.d.Document	`{ “$dec” : “1.23456e7” }`
large value	org.m.s.d.Document	`{ “$sha1” : “33b8397fa71bf1b942bf90cc6f7067d5219589ea” }`

References

JCR REFERENCE properties are stored within the documents as regular properties. However, ModeShape also stores on the referenced object back-references to the referencing node. This is to enable quickly finding all of the nodes that contain a reference to a particular node. Here’s an example of what this looks like on a node that is referenced with a JCR REFERENCE by one node and with a JCR WEAKREFERENCE by another node:

```
{
  "parent" : [],
  "properties" : {
      "http://www.jcp.org/jcr/1.0" : {
          "primaryType" : { "$name" : "{http://www.modeshape.org/1.0}root" },
          "uuid" : "cafebabe-cafe-babe-cafe-babecafebabe"
      }
  },
  "referrers" : {
      "strong" : [ "28f35d27e749172ef58009-28ab-4348-8445-b92b3f9e97e3" ],
      "weak" : [ "28f35d27e7491742256439-aadf-49f6-93f0-89d677c769fa9" ]
  },
  "children" : [
      {
          "key" : "28f35d2system-jcr:system",
          "name" : "{http://www.jcp.org/jcr/1.0}system"
      }
      {
          "key" : "28f35d27e7491747422530-15d7-4db3-80e9-7d8cc270629f",
          "name" : "childA"
      },
      {
          "key" : "28f35d27e749171aaee8f9-8aca-4b21-9c05-8ba870f5f7b2",
          "name" : "childB"
      }
  ]
}
```

The strong referrers are used to enforce JCR’s reference integrity feature: a node can only be deleted from JCR if all of the strong referrers are also being deleted within the same transaction.

Workspace cache

Node documents stored within Infinispan are in-memory representations of the JSON documents. However, using these documents directly is not terribly straightforward, since they contain only data structure and no methods. Plus, although it is not typical, a single node can actually have more than one parent node and can appear in multiple workspaces. To maintain this contextual state and to make it easier to access the information within the documents, ModeShape has a workspace cache that contains an object representation of the nodes in the workspace that have been accessed by the sessions. As soon as one of the sessions changes a node, the cached object representation is discarded and the updated document is accessed via new cached object representation.

These cached node objects also provide an easy way to ensure that the information in one or more documents is changed correctly and consistently. For example, the cached node objects encapsulate the logic of getting and setting properties, finding or manipulating child nodes (in blocks if necessary), determining the indexes for children with the same names, maintenance of parent references, and maintenance of back-references to referring nodes.

Session state

While a ModeShape repository instance maintains a single cache for all the accessed nodes in each workspace, each JCR session maintains its own state of changed nodes. In fact, the session cache and workspace cache were both designed to enable the session cache to store only the **changes** to the nodes (e.g., that a property was added, or that a child node was moved before another sibling) yet efficiently access the properties and maintain the list of children (including same-name-sibling indexes). Because a session cache only maintains the changes made by a particular JCR session, it can see all of the changes to nodes in the workspace as soon as they are persisted (i.e., saved) by other sessions.

System content

ModeShape maintains a single repository-wide system branch for all namespaces, node types, version histories, and locks. This branch is exposed (e.g., federated) into each workspace under the “/jcr:system” node, but with ModeShape 3.0 this is done by making the “jcr:system” node be a child of each workspace’s root node. This information is cached by each workspace, but changed through a special hidden workspace called “system”. Thus, unlike ModeShape 2.x, changes to the system content is very efficiently propagated to each JCR session.

---

1. When ModeShape is installed into JBoss AS7, the configuration of the repositories is handled by the JBoss AS7 configuration system for standalone or domain configurations, and can be changed using AS7’s command line tool, RESTful interface, or web console. The repository configurations in AS7 is XML-based but structurally similar to the JSON configuration files. ↩
2. Web applications often create a JCR Session when handling each request, so any delay in the `login(...)` method will be apparent to the first users. Preemptively starting the repositories (e.g., when the application server starts up) is a clean way of minimizing the time required to login. ↩

3. Schematic is a thin add-on to Infinispan that provides the basic infrastructure needed to easily store and update JSON/BSON-style documents as values within an Infinispan Cache. Schematic also supports validating the structure of the documents using JSON Schema documents, and plans to offer automatic document indexing via Hibernate Search. ↩
4. Using the first 7 characters of the SHA–1 of the source and workspace names is a pragmatic shortcut. SHA–1 hashes are nicely distributed (especially in the leading characters), and the first 7 characters will for all intents and purposes be unique, even for highly similar names. ↩
5. The nodes under “/jcr:system/jcr:nodeTypes” that reflect the repository’s registered node type definitions use the expanded form of the node type definition names for node identifiers. This has a number of benefits, but perhaps the most important one is that it is far easier to update or modify the node type definitions. ↩
6. If ModeShape were to store the SNS index, then inserting or removing nodes would require that the SNS indexes for the other child references be re-evaluated and potentially modified. Plus, by not storing the SNS indexes and always computing them relative to the other children, especially when a session has transient (unsaved) changes to the children of a node. ↩