

JBoss DNA

1

Getting Started Guide

ISBN:

Publication date:

JBoss DNA: Getting Started Guide

by Randall Hauch

Legal Notice

1801 Varsity Drive
Raleigh, NC27606-2072USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588Research Triangle Park, NC27709USA

Copyright © 2008 by Red Hat, Inc. This copyrighted material is made available to anyone wishing to use, modify, copy, or redistribute it subject to the terms and conditions of the GNU [Lesser General Public License](http://www.gnu.org/licenses/lgpl-2.1.html) [http://www.gnu.org/licenses/lgpl-2.1.html], as published by the Free Software Foundation.

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

What this book covers	vii
1. Introduction	1
2. Understanding JBoss DNA	3
1. Overview	3
2. Architecture	3
3. Sequencing content	6
4. Federating content	8
4.1. Connecting to information sources	9
4.2. Building the unified graph	10
4.3. Searching and querying	10
4.4. Updating content	11
4.5. Observing changes	11
3. Running the example application	13
1. Downloading and compiling	14
2. Running the example	16
3. Summarizing what we just did	22
4. Using JBoss DNA	25
1. Configuring the Sequencing Service	25
2. Configuring the Observation Service	28
3. Shutting down JBoss DNA services	29
4. Reviewing the example application	30
5. Summarizing what we just did	36
5. Creating custom sequencers	37
1. Creating the Maven 2 project	37
2. Implementing the StreamSequencer interface	40
3. Testing custom sequencers	44
4. Deploying custom sequencers	46
6. Looking to the future	47

What this book covers

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

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

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

If you have any questions or comments, please feel free to contact JBoss DNA's [user mailing list](mailto:dna-users@jboss.org) [mailto:dna-users@jboss.org] or use the [user forums](http://www.jboss.com/index.html?module=bb&op=viewforum&f=272) [http://www.jboss.com/index.html?module=bb&op=viewforum&f=272] . If you'd like to get involved on the project, join the [mailing lists](http://www.jboss.org/dna/lists.html) [http://www.jboss.org/dna/lists.html] , [download the code](http://www.jboss.org/dna/subversion.html) [http://www.jboss.org/dna/subversion.html] and get it building, and visit our [JIRA issue management system](http://jira.jboss.org/jira/browse/DNA) [http://jira.jboss.org/jira/browse/DNA] . If there's something in particular you're interested in, talk with the community - there may be others interested in the same thing.

Introduction

There are a lot of 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 offer better choices.

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.

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

Object-relational mapping libraries automate the creation of this mapping layer and result in far less code that is much more maintainable with performance that is often as good as (if not better than) handwritten JDBC code. The new [Java Persistence API \(JPA\)](http://java.sun.com/developer/technicalArticles/J2EE/jpa/) [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) [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.

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

The *JBoss DNA project* is building the tools and services that surround content repositories. Nearly all of these capabilities are to be hidden below the JCR API and involve automated processing of the information in the repository. Thus, JBoss DNA can add value to existing repository implementations. For example, JCR repositories offer the ability to upload files into the repository and have the file content indexed for search purposes. JBoss DNA also defines a library for "sequencing" content - to extract meaningful information from that content and store it in the repository, where it can then be searched, accessed, and analyzed using the JCR API.

JBoss DNA is building other features as well. One goal of JBoss DNA is to create federated repositories that dynamically merge the information from multiple databases, services, applications, and other JCR repositories. Another is to create customized views based upon the type of data and the role of the user that is accessing the data. And yet another is to create a REST-ful API to allow the JCR content to be accessed easily by other applications written in other languages.

The [next chapter](#) in this book goes into more detail about JBoss DNA and its architecture, the different components, what's available now, and what's coming in future releases. [Chapter 3](#) then provides instructions for downloading and running the sequencer examples for the current release. [Chapter 4](#) walks through how to use JBoss DNA in your applications, while [Chapter 5](#) goes over how to create custom sequencers. Finally, [Chapter 6](#) wraps things up with a discussion about the future of JBoss DNA.

Understanding JBoss DNA

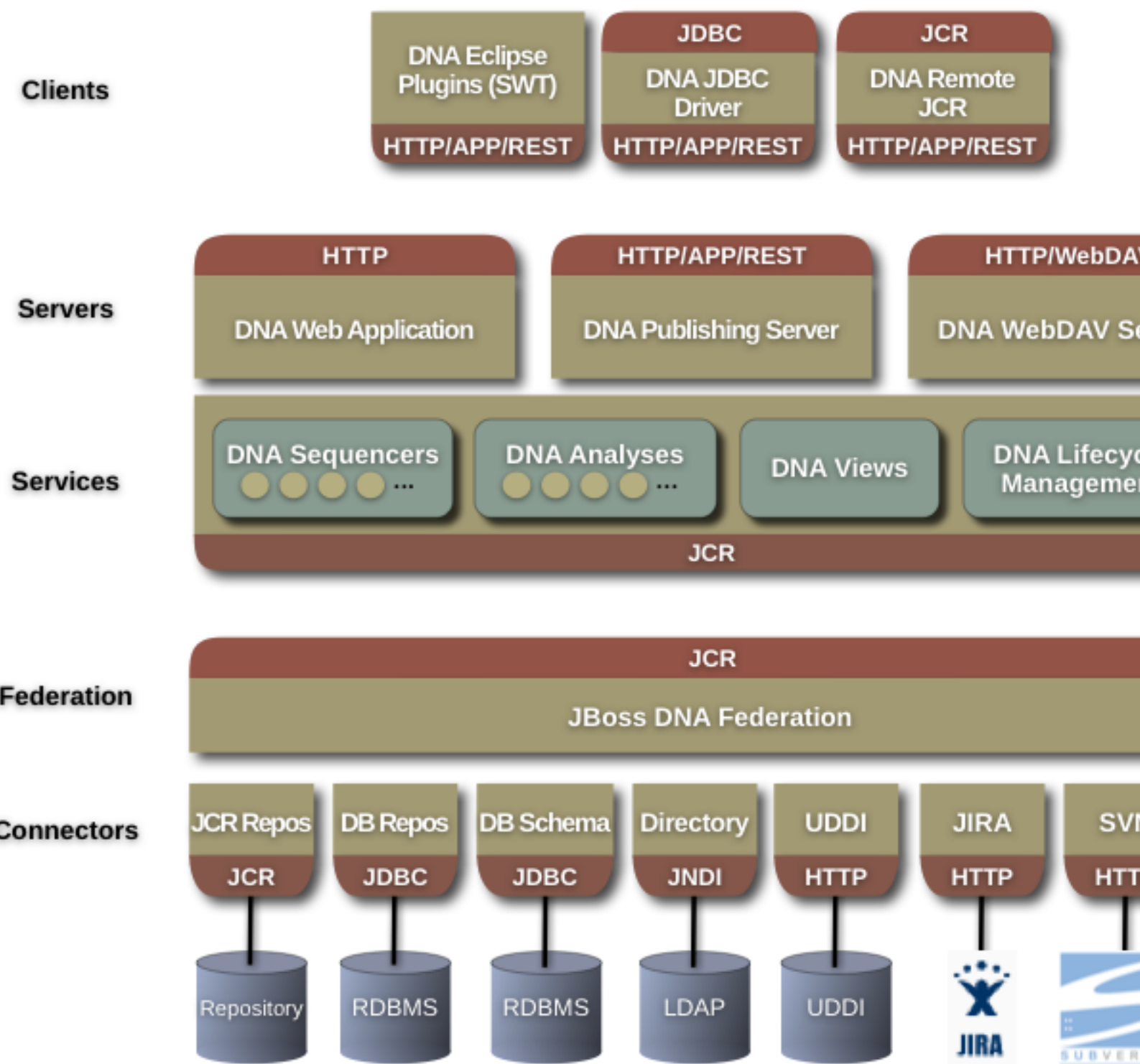
1. Overview

JBoss DNA is a repository and set of tools that make it easy to capture, version, analyze, and understand the fundamental building blocks of information. As models, service and process definitions, schemas, source code, and other artifacts are added to the repository, JBoss DNA "sequences" the makeup of these components and extracts their structure and interdependencies. The JBoss DNA web application allows end users to access, visualize, and edit this information in the terminology and structure they are familiar with. Such domain-specific solutions can be easily created with little or no programming.

JBoss DNA supports the Java Content Repository (JCR) standard and is able to provide a single integrated view of multiple repositories, external databases, services, and applications, ensuring that JBoss DNA has access to the latest and most reliable master data. For instance, DNA could provide in a single view valuable insight into the business processes and process-level services impacted by a change to in an intermediary web server operation defined via WSDL. Similarly, a user could quickly view and navigate the dependencies between the data source models and transformation information stored within a content repository, the code base stored within a version control system, and the database schemas used by an application.

2. Architecture

The architecture for JBoss DNA consists of several major components that will be built on top of standard APIs, including JCR, JDBC, JNDI and HTTP. The goal is to allow these components to be assembled as needed and add value on top of other DNA components or third-party systems that support these standard APIs.



As shown in the diagram above, the major components are (starting at the top):

- **DNA Eclipse Plugins** enable Eclipse users to access the contents of a JBoss DNA repository.
- **DNA JDBC Driver** provides a driver implementation, allowing JDBC-aware applications to connect to and use a JBoss DNA repository.

- **DNA Remote JCR** is a client-side component for accessing remote JCR repositories.
- **DNA Web Application** is used by end users and domain experts to visualize, search, edit, change and tag the repository content. The web application uses views to define how different types of information are to be presented and edited in domain-specific ways. The goal is that this web application is easily customized and branded for inclusion into other solutions and application systems. The DNA Web Application operates upon any JCR-compliant repository, although it does rely upon the DNA analysis and templating services.
- **DNA Publishing Server** allows content to be downloaded, uploaded, and edited using the Atom Publishing Protocol. With the DNA Publishing Server, the content of the repository can easily be created, read, edited, and deleted using the standard HTTP operations of POST, GET, PUT, and DELETE (respectively). More and more tools are being created that support working with Atom Publishing servers. The DNA Publishing Server operates upon any JCR-compliant repository.
- **DNA WebDAV Server** allows clients such as Microsoft Windows and Apple OS X to connect to, read, and edit the content in the repository using the WebDAV standard. Since WebDAV is an extension of HTTP, web browsers are able to read (but not modify) the content served by a WebDAV compliant server. The DNA WebDAV Server operates upon any JCR-compliant repository.
- **DNA Sequencers** are pluggable components that make it possible for content to be uploaded to the repository and automatically processed to extract meaningful structure and place that structure in the repository. Once this information is in the repository, it can be viewed, edited, analyzed, searched, and related to other content. DNA defines a Java interface that sequencers must implement. DNA sequencers operate upon any JCR-compliant repository.
- **DNA Analyses** are pluggable components that analyze content and the relationships between content to generate reports or to answer queries. DNA will include some standard analyzers, like dependency analysis and similarity analysis, that are commonly needed by many different solutions. DNA analyzers operate upon any JCR-compliant repository.
- **DNA Views** are definitions of how types of information are to be presented in a user interface to allow for creation, reading, editing, and deletion of information. DNA view definitions consist of data stored in a JCR repository, and as such views can be easily added, changed or removed entirely by using the DNA Web Application, requiring no programming.
- **DNA Federation** is an implementation of the JCR API that builds the content within the repository by accessing and integrating information from multiple sources. DNA Federation allows the integration of external systems, like other JCR repositories, databases, applications, and services.

- **DNA Connectors** are used to communicate with these external sources of information. In the federation engine, each source is able to contribute node structure and node properties to any part of the federated graph, although typically many connectors will contribute most of their information to isolated subgraphs. The result is that integration from a wide range of systems can be integrated and accessed through the DNA Web Application, DNA Publishing Server, and DNA WebDAV Server. Connectors also may optionally participate in distributed transactions by exposing an XAResource.
- **DNA Maven** is a classloader library compatible with Maven 2 project dependencies. This allows the creation of Java ClassLoader instances using Maven 2 style paths, and all dependencies are transitively managed and included.

Continue reading the rest of this chapter for more detail about the [sequencing framework](#) available in this release, or the [federation engine](#) and [connectors](#) that will be the focus of the next release. Or, skip to the [examples](#) to see how to start using JBoss DNA 0.1 today.

3. Sequencing content

The current JBoss DNA release contains a sequencing framework that is designed to sequence data (typically files) stored in a JCR repository to automatically extract meaningful and useful information. This additional information is then saved back into the repository, where it can be accessed and used.

In other words, you can just upload various kinds of files into a JCR repository, and DNA automatically processes those files to extract meaningful structured information. For example, load DDL files into the repository, and let sequencers extract the structure and metadata for the database schema. Load Hibernate configuration files into the repository, and let sequencers extract the schema and mapping information. Load Java source into the repository, and let sequencers extract the class structure, JavaDoc, and annotations. Load a PNG, JPEG, or other image into the repository, and let sequencers extract the metadata from the image and save it in the repository. The same with XSDs, WSDL, WS policies, UML, MetaMatrix models, etc.

JBoss DNA sequencers sit on top of existing JCR repositories (including federated repositories) - they basically extract more useful information from what's already stored in the repository. And they use the existing JCR versioning system. Each sequencer typically processes a single kind of file format or a single kind of content.

The following sequencers are included in JBoss DNA:

- **Image sequencer** - A sequencer that processes the binary content of an image file, extracts the metadata for the image, and then writes that image metadata to the repository. It gets the file format, image resolution, number of bits per

pixel (and optionally number of images), comments and physical resolution from JPEG, GIF, BMP, PCX, PNG, IFF, RAS, PBM, PGM, PPM, and PSD files. (This sequencer may be improved in the future to also extract EXIF metadata from JPEG files; see [DNA-26](http://jira.jboss.org/jira/browse/DNA-26) [http://jira.jboss.org/jira/browse/DNA-26] .)

- **MP3 sequencer** - A sequencer that processes the contents of an MP3 audio file, extracts the metadata for the file, and then writes that image metadata to the repository. It gets the title, author, album, year, and comment. (This sequencer may be improved in the future to also extract other ID3 metadata from other audio file formats; see [DNA-26](http://jira.jboss.org/jira/browse/DNA-66) [http://jira.jboss.org/jira/browse/DNA-66] .)

As the community develops additional sequencers, they will also be included in JBoss DNA. Some of those that have been identified as being useful include:

- **XML Schema Document (XSD) Sequencer** - Process XSD files and extract the various elements, attributes, complex types, simple types, groups, and other information. (See [DNA-32](http://jira.jboss.org/jira/browse/DNA-32) [http://jira.jboss.org/jira/browse/DNA-32])
- **Web Service Definition Language (WSDL) Sequencer** - Process WSDL files and extract the services, bindings, ports, operations, parameters, and other information. (See [DNA-33](http://jira.jboss.org/jira/browse/DNA-33) [http://jira.jboss.org/jira/browse/DNA-33])
- **Hibernate File Sequencer** - Process Hibernate configuration (cfg.xml) and mapping (hbm.xml) files to extract the configuration and mapping information. (See [DNA-61](http://jira.jboss.org/jira/browse/DNA-61) [http://jira.jboss.org/jira/browse/DNA-61])
- **XML Metadata Interchange (XMI) Sequencer** - Process XMI documents that contain UML models or models using another metamodel, extracting the model structure into the repository. (See [DNA-31](http://jira.jboss.org/jira/browse/DNA-31) [http://jira.jboss.org/jira/browse/DNA-31])
- **ZIP Archive Sequencer** - Process ZIP archive files to extract (explode) the contents into the repository. (See [DNA-63](http://jira.jboss.org/jira/browse/DNA-63) [http://jira.jboss.org/jira/browse/DNA-63])
- **Java Archive (JAR) Sequencer** - Process JAR files to extract (explode) the contents into the classes and file resources. (See [DNA-64](http://jira.jboss.org/jira/browse/DNA-64) [http://jira.jboss.org/jira/browse/DNA-64])
- **Java Class File Sequencer** - Process Java class files (bytecode) to extract the class structure (including annotations) into the repository. (See [DNA-62](http://jira.jboss.org/jira/browse/DNA-62) [http://jira.jboss.org/jira/browse/DNA-62])
- **Java Source File Sequencer** - Process Java source files to extract the class structure (including annotations) into the repository. (See [DNA-51](http://jira.jboss.org/jira/browse/DNA-51) [http://jira.jboss.org/jira/browse/DNA-51])

- **PDF Sequencer** - Process PDF files to extract the document metadata, including table of contents. (See [DNA-50](#) [<http://jira.jboss.org/jira/browse/DNA-50>])
- **Maven 2 POM Sequencer** - Process Maven 2 Project Object Model (POM) files to extract the project information, dependencies, plugins, and other content. (See [DNA-24](#) [<http://jira.jboss.org/jira/browse/DNA-24>])
- **Data Definition Language (DDL) Sequencer** - Process various dialects of DDL, including that from Oracle, SQL Server, MySQL, PostgreSQL, and others. May need to be split up into a different sequencer for each dialect. (See [DNA-26](#) [<http://jira.jboss.org/jira/browse/DNA-26>])
- **MP3 and MP4 Sequencer** - Process MP3 and MP4 audio files to extract the name of the song, artist, album, track number, and other metadata. (See [DNA-30](#) [<http://jira.jboss.org/jira/browse/DNA-30>])

The [examples](#) in this book go into more detail about how sequencers are managed and used, and [Chapter 5](#) goes into detail about how to write custom sequencers.

4. Federating content

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 move the information into a JCR repository? Most likely there are existing applications that rely upon that information being where it is. If we were to move it, then all those applications would break. Or they'd have to be changed to use JCR. If the information is being used, the most practical thing is to leave it where it is.

Then why not just copy the information into a JCR repository? Actually, there are times when it's perfectly reasonable to make a copy of the data. Perhaps the system managing the existing information cannot handle the additional load of more clients. Or, perhaps the information doesn't change, or it does change and we want snapshots that don't change. But more likely, the data *does* change. So if applications are to use the most current information and we make copies of the data, we have to keep the copies synchronized with the master. That's generally a lot of work.

The JBoss DNA federation engine lets us leave the information where it is, yet lets client applications use the JCR API to access all the information without caring where the information really exists. If the underlying information changes, client applications using JCR observation will be notified of the changes. If a JBoss DNA federated repository is configured to allow updates, client applications can change the information in the repository and JBoss DNA will propagate those changes down to the original source.

4.1. Connecting to information sources

The JBoss DNA federation engine will use connectors to interact with different information sources to get at the content in those systems. Some ideas for connectors include:

- **JCR Repository Connector** - Connect to and interact with other JCR repositories.
- **File System Connector** - Expose the files and directories on a file system through JCR.
- **Maven 2 Repository Connector** - Access and expose the contents of a Maven 2 repository (either on the local file system or via HTTP) through JCR.
- **JDBC Metadata Connector** - Connect to relational databases via JDBC and expose their schema as content in a repository.
- **UDDI Connector** - Interact with UDDI registries to integrate their content into a repository.
- **SVN Connector** - Interact with Subversion software configuration management (SCM) repositories to expose the managed resources through JCR. Consider using the [SVNKit](http://svnkit.com/) [http://svnkit.com/] (dual license) library for an API into Subversion.
- **CVS Connector** - Interact with CVS software configuration management (SCM) repositories to expose the managed resources through JCR.
- **JDBC Storage Connector** - Store and access information in a relational database. Also useful for persisting information in the federated repository not stored elsewhere.
- **Distributed Database Connector** - Store and access information in a [Hypertable](http://www.hypertable.org/) [http://www.hypertable.org/] or [HBase](http://hadoop.apache.org/hbase/) [http://hadoop.apache.org/hbase/] distributed databases. Also useful for persisting information in the federated repository not stored elsewhere.

If the connectors allow the information they contribute to be updated, they must provide an `XAResource` implementation that can be used with a Java Transaction Service. Connectors that provide read-only access need not provide an implementation.

Also, connectors talk to *sources* of information, and it's quite likely that the same connector is used to talk to different sources. Each source contains the configuration details (e.g., connection information, location, properties, options, etc.) for working with that particular source, as well as a reference to the connector that should be used to establish connections to the source. And of course, sources can be added or removed without having to stop and restart the federated repository.

4.2. Building the unified graph

The federation engine works by effectively building up a single graph by querying each source and merging or unifying the responses. This information is cached, which improves performance, reduces the number of (potentially expensive) remote calls, reduces the load on the sources, and helps mitigate problems with source availability. As clients interact with the repository, this cache is consulted first. When the requested portion of the graph (or "subgraph") is contained completely in the cache, it is returned immediately. However, if any part of the requested subgraph is not in the cache, each source is consulted for their contributions to that subgraph, and any results are cached.

This basic flow makes it possible for the federated repository to build up a local cache of the integrated graph (or at least the portions that are used by clients). In fact, the federated repository caches information in a manner that is similar to that of the Domain Name System (DNS). As sources are consulted for their contributions, the source also specifies whether it is the authoritative source for this information (some sources that are themselves federated may not be the information's authority), whether the information may be modified, the time-to-live (TTL) value (the time after which the cached information should be refreshed), and the expiration time (the time after which the cached information is no longer valid). In effect, the source has complete control over how the information it contributes is cached and used.

The federated repository also needs to incorporate *negative caching*, which is storage of the knowledge that something does not exist. Sources can be configured to contribute information only below certain paths (e.g., `/A/B/C`), and the federation engine can take advantage of this by never consulting that source for contributions to information on other paths. However, below that path, any negative responses must also be cached (with appropriate TTL and expiry parameters) to prevent the exclusion of that source (in case the source has information to contribute at a later time) or the frequent checking with the source.

4.3. Searching and querying

The JBoss DNA federated repository will also support queries against the integrated and unified graph. In some situations the query can be determined to apply to a single source, but in most situations the query must be planned (and possibly rewritten) such that it can be pushed down to all the appropriate sources. Also, the cached results must be consulted prior to returning the query results, as the results from one source might have contributions from another source.



Note

It is hoped that the MetaMatrix query engine can be used for this purpose after it is open-sourced. This engine implements

sophisticated query planning and optimization techniques for working efficiently with multiple sources.

Searching the whole federated repository is also important. This allows users to simply supply a handful of search terms, and to get results that are ranked based upon how close each result is to the search terms. (Searching is very different from querying, which involves specifying the exact semantics of what is to be searched and how the information is to be compared.) JBoss DNA will incorporate a search engine (e.g., likely to be Lucene) and will populate the engine's indexes using the federated content and the cached information. Notifications of changing information will be reflected in the indexes, but some sources may want to explicitly allow or disallow periodic crawling of their content.

4.4. Updating content

The JBoss DNA federated repositories also make it possible for client applications to make changes to the unified graph within the context of distributed transactions. According to the JCR API, client applications use the Java Transaction API (JTA) to control the boundaries of their transactions. Meanwhile, the federated repository uses a *distributed transaction service* [<http://www.jboss.org/jbosstm/>] to coordinate the XA resources provided by the connectors.

It is quite possible that clients add properties to nodes in the unified graph, and that this information cannot be handled by the same underlying source that contributed to the node. In this case, the federated repository can be configured with a fallback source that will be used to store this "extra" information.

It is a goal that non-XA sources (i.e., sources that use connectors without XA resources) can participate in distributed transactions through the use of *compensating transactions*. Because the JBoss DNA federation engine implements the JCR observation system, it is capable of recording all of the changes made to the distributed graph (and those changes sent to each updatable source). Therefore, if a non-XA source is involved in a distributed transaction that must be rolled back, any changes made to non-XA sources can be undone. (Of course, this does not make the underlying source transactional: non-transactional sources still may expose the interim changes to other clients.)

4.5. Observing changes

The JCR API supports observing a repository to receive notifications of additions, changes and deletions of nodes and properties. The JBoss DNA federated repository will support this API through two primary means.

When the changes are made through the federated repository, the JBoss DNA federation engine is well aware of the set of changes that have been (or are being) made to the unified graph. These events are directly propagated to listeners.

Sources have the ability to publish events, making it possible for the JBoss DNA federation engine and clients that have registered listeners to be notified of changes in the information managed by that source. These events are first processed by the federation engine and possibly altered based upon contributions from other sources. (The federation engine also uses these events to update or purge information in the cache, which may add to the event set.) The resulting (and possibly altered) event set is then sent to all client listeners.

Running the example application

This chapter provides instructions for downloading and running a sample application that demonstrates how JBoss DNA works with a JCR repository to automatically sequence changing content to extract useful information. So read on to get the simple application running, and then in the [next chapter](#) we'll dive into the source code for the example and show how to use JBoss DNA in your own applications.

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



Note

To use Maven with JBoss DNA, you'll need to have *JDK 5 or 6* [http://java.sun.com/javase/downloads/index_jdk5.jsp] and Maven 2.0.7 (or higher).

Maven can be downloaded from <http://maven.apache.org/>, and is installed by unzipping the `maven-2.0.7-bin.zip` file to a convenient location on your local disk. Simply add `$MAVEN_HOME/bin` to your path and add the following profile to your `~/.m2/settings.xml` file:

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

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

This profile informs Maven of the two JBoss repositories (snapshots and releases) that contain all of the JARs for JBoss DNA and all dependent libraries.

1. Downloading and compiling

The next step is to [download](#)

[<http://www.jboss.org/file-access/default/members/dna/downloads/0.1/jboss-dna-0.1-gettingstarted-examples.zip>] the example for this Getting Started guide, and extract the contents to a convenient location on your local disk. You'll find the example contains the following files, which are organized according to the standard Maven directory structure:

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

There are essentially two Maven projects: a `sequencers` project and a parent project. All of the source for the example is located in the `sequencers` subdirectory. And you may have noticed that none of the JBoss DNA libraries are there. This is where Maven comes in. The two `pom.xml` files tell Maven everything it needs to know about what libraries are required and how to build the example.

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

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

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

If you've successfully built the examples, there will be a `examples/sequencers/target/dna-example-sequencers-basic.dir/` directory that contains the following:

- `run.sh` is the *nix shell script that will run the example.
- `log4j.properties` is the Log4J configuration file.
- `jackrabbitConfig.xml` is the Jackrabbit configuration file, which is set up to use a transient in-memory repository.
- `jackrabbitNodeTypes.cnd` defines the additional JCR node types used by this example.
- `sample1.mp3` is a sample MP3 audio file you'll use later to upload into the repository.
- `caution.gif`, `caution.png`, and `caution.jpg` are images that you'll use later and upload into the repository.
- `lib` subdirectory contains the JARs for all of the JBoss DNA artifacts as well as those for other libraries required by JBoss DNA and the example.



Note

JBoss DNA 0.1 and the examples are currently tested with *Apache Jackrabbit* [<http://jackrabbit.apache.org/>] version 1.3.3. This version is stable and used by a number of other projects and applications. However, you should be able to use a newer version of Jackrabbit, as long as that version uses the same JCR API. For example, version 1.4.2 was released on March 26, 2008 and should be compatible.

Just remember, if the version of Jackrabbit you want to use for these examples is not in the Maven repository, you'll have to either add it or add it locally. For more information, see the *Maven documentation* [<http://maven.apache.org/>].

2. Running the example

This example consists of a client application that sets up an in-memory JCR repository and that allows a user to upload files into that repository. The client also sets up the DNA services with two sequencers so that if any of the uploaded files are PNG, JPEG, GIF, BMP or other images, DNA will automatically extract the image's metadata (e.g., image format, physical size, pixel density, etc.) and store that in the

repository. Alternatively, if the uploaded file is an MP3 audio file, DNA will extract some of the ID3 metadata (e.g., the author, title, album, year and comment) and store that in the repository.

To run the client application, go to the

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

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

-----
Menu:

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

d) Display statistics

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

Figure 3.1. Example Client

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

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

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

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

```
d) Display statistics
```

```
? ) Show this menu
```

```
q) Quit
```

```
>u
```

```
Please enter the file to upload:
```

```
caution.png
```

```
Please enter the repository path where the file should be placed [/a/b/caution.png]
```

```
/a/b/c/caution.png
```

```
>_
```

Figure 3.2. Uploading an image using the Example Client

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

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

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

```
file}                                @jcr:mimeType = {mime type of the  
                                     @jcr:lastModified = {now}
```

Other kinds of files are treated in a similar way.

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

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

```
-----  
Menu:  
  
u) Upload a file to the repository  
s) Search the repository using extracted metadata  
  
d) Display statistics  
  
?) Show this menu  
q) Quit  
>s  
  
1 image was found:  
Media 1  
  Name: caution.png  
  Path: /images/caution.png  
  Type: image  
  image:numberOfImages: 1  
  image:physicalHeightInches: -1.0  
  image:width: 48  
  image:physicalWidthDpi: -1  
  image:progressive: false  
  image:physicalWidthInches: -1.0  
  jcr:primaryType: image:metadata  
  image:physicalHeightDpi: -1  
  image:formatName: PNG  
  image:bitsPerPixel: 24  
  jcr:mimeType: image/png  
  image:height: 48
```

Figure 3.3. Searching for media using the Example Client

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

```
-----
Menu:

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

d) Display statistics

?) Show this menu
q) Quit
>s

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

Figure 3.4. Searching for media using the Example Client

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

```
-----
Menu:

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

d) Display statistics

?) Show this menu
q) Quit
>d

# nodes sequenced: 2
# nodes skipped: 10

>_
```

Figure 3.5. Sequencing statistics using the Example Client

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



Note

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

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

3. Summarizing what we just did

In this chapter you downloaded and installed the example application and used it to upload files into a JCR repository. JBoss DNA automatically sequenced the image and/or MP3 files you uploaded, extracted the metadata from the files, and stored that metadata inside the repository. The application allowed you to see this metadata and the sequencing statistics.

This application was very simplistic. In fact, running through the example probably only took you a minute or two. So while this application won't win any awards, it does show the basics of what JBoss DNA can do.

In the [next chapter](#) we'll venture into the code to get an understanding of how JBoss DNA actually works and how you can use it in your own applications.

Using JBoss DNA

As we've mentioned before, JBoss DNA is able to work with existing JCR repositories. Your client applications make changes to the information in those repositories, and JBoss DNA automatically uses its sequencers to extract additional information from the uploaded files.



Note

Configuring JBoss DNA sequencers is a bit more manual than is ideal. As you'll see, JBoss DNA uses dependency injection to allow a great deal of flexibility in how it can be configured and customized. However, the next release will provide a much easier mechanism for configuring not only the sequencer service but also the upcoming federation engine and JCR implementation.

1. Configuring the Sequencing Service

The JBoss DNA *sequencing service* is the component that manages the *sequencers* and that reacts to changes in JCR repositories and then running the appropriate sequencers. This involves processing the changes on a node, determine which (if any) sequencer should be run on that node, and for each sequencer constructing the execution environment, calling the sequencer, and saving the information generated by the sequencer.

To set up the sequencing service, an instance is created and dependent components are injected into the object. This includes among other things:

- An *execution context* that defines the context in which the service runs, including a factory for JCR sessions given names of the repository and workspace. This factory must be configured, and is how JBoss DNA knows about your JCR repositories and how to connect to them. More on this a bit later.
- An optional *factory for class loaders* used to load sequencers. If no factory is supplied, the service uses the current thread's context class loader (or if that is null the class loader that loaded the sequencing service class).
- An `java.util.concurrent.ExecutorService` used to execute the sequencing activities. If none is supplied, a new single-threaded executor is created by calling `Executors.newSingleThreadExecutor()`. (This can easily be changed by subclassing and overriding the `SequencerService.createDefaultExecutorService()` method.)
- Filters for sequencers and events. By default, all sequencers are considered for "node added", "property added" and "property changed" events.

As mentioned above, the `ExecutionContext` provides access to a `SessionFactory` that is used by JBoss DNA to establish sessions to your JCR repositories. Two implementations are available:

- The `JndiSessionFactory` looks up JCR `Repository` instances in JNDI using names that are supplied when creating sessions. This implementation also has methods to set the JCR `Credentials` for a given workspace name.
- The `SimpleSessionFactory` has methods to register the JCR `Repository` instances with names, as well as methods to set the JCR `Credentials` for a given workspace name.

You can use the `SimpleExecutionContext` implementation of `ExecutionContext` and supply a `SessionFactory` instance, or you can provide your own implementation.

Here's an example of how to instantiate and configure the `SequencingService`:

```
SimpleSessionFactory sessionFactory = new SimpleSessionFactory();
sessionFactory.registerRepository("Main Repository",
    this.repository);
Credentials credentials = new SimpleCredentials("jsmith",
    "secret".toCharArray());
sessionFactory.registerCredentials("Main Repository/Workspace1",
    credentials);
ExecutionContext executionContext = new
    SimpleExecutionContext(sessionFactory);

// Create the sequencing service, passing in the execution context
...
SequencingService sequencingService = new SequencingService();
sequencingService.setExecutionContext(executionContext);
```

After the sequencing service is created and configured, it must be started. The `SequencingService` has an *administration object* (that is an instance of `ServiceAdministrator`) with `start()`, `pause()`, and `shutdown()` methods. The latter method will close the queue for sequencing, but will allow sequencing operations already running to complete normally. To wait until all sequencing operations have completed, simply call the `awaitTermination` method and pass it the maximum amount of time you want to wait.

```
sequencingService.getAdministrator().start();
```

The sequencing service must also be configured with the sequencers that it will use. This is done using the `addSequencer(SequencerConfig)` method and passing a `SequencerConfig` instance that can create. Here's an example:

```
String name = "Image Sequencer";
String desc = "Sequences image files to extract the characteristics
  of the image";
String classname =
  "org.jboss.dna.sequencer.images.ImageMetadataSequencer";
String[] classpath = null; // Use the current classpath
String[] pathExpressions =
  {"/*.(jpg|jpeg|gif|bmp|pcx|png))[*]/jcr:content[@jcr:data] =>
  /images/$1"};
SequencerConfig imageSequencerConfig = new SequencerConfig(name,
  desc, classname, classpath, pathExpressions);
sequencingService.addSequencer(imageSequencerConfig);

name = "Mp3 Sequencer";
desc = "Sequences mp3 files to extract the id3 tags of the audio
  file";
classname = "org.jboss.dna.sequencer.mp3.Mp3MetadataSequencer";
String[] mp3PathExpressions = {"/*.(mp3)[*]/jcr:content[@jcr:data]
  => /mp3s/$1"};
SequencerConfig mp3SequencerConfig = new SequencerConfig(name,
  desc, classname, classpath, mp3PathExpressions);
sequencingService.addSequencer(mp3SequencerConfig);
```

This is pretty self-explanatory, except for the `classpath` and `pathExpression` parameters. The `classpath` parameter defines the classpath that is passed to the class loader factory mentioned above. Our sequencer is on the classpath, so we can simply use `null` here.

The path expression is more complicated. Sequencer path expressions are used by the sequencing service to determine whether a particular changed node should be sequenced. The expressions consist of two parts: a selection criteria and an output expression. Here's a simple example:

```
/a/b/c@title => /d/e/f
```

Here, the `/a/b/c@title` is the selection criteria that applies when the `/a/b/c` node has a `title` property that is added or changed. When the selection criteria matches a change event, the sequencer will be run and any generated output will be inserted into the repository described by the output expression. In this example, the generated output would be placed at the `/d/e/f` node.



Note

Sequencer path expressions can be fairly complex and may use wildcards, specify same-name sibling indexes, provide optional and

choice elements, and may capture parts of the selection criteria for use in the output expression. The path expression used in the image sequencer configuration example above shows a more complex example:

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

This uses "/" to select any node at any level in the repository whose name ends with "." and one of the extensions (e.g., ".jpg", ".jpeg", etc.) and that has a child node named "jcr:content" that has a "jcr:data" property. It also selects the file name as the first capture group (the first set of parentheses) for use in the output expression. In this example, any sequencer output is placed on a node with that same file name under the "/images" node.

Other things are possible, too. For example, the name of the repository/workspace (as used by the `SessionFactory`) may be specified at the beginning of the select criteria and/or the output expression. This means it's possible to place the sequencer output in a different repository than the node being sequenced.

For more detail about sequencer path expressions, see the

`org.jboss.dna.repository.sequencer.SequencerPathExpression` class and the corresponding `org.jboss.dna.repository.sequencer.S` test case.

After the service is started, it is ready to start reacting to changes in the repository. But it first must be wired to the repositories using listener. This is accomplished using the `ObservationService` described in the [next section](#).

2. Configuring the Observation Service

The JBoss DNA `ObservationService` is responsible for listening to one or more JCR repositories and multiplexing the events to its listeners. Unlike JCR events, this framework embeds in the events the name of the repository and workspace that can be passed to a `SessionFactory` to obtain a session to the repository in which the change occurred. This simple design makes it very easy for JBoss DNA to concurrently work with multiple JCR repositories.

Configuring an `ObservationService` is pretty easy, especially if you reuse the same `SessionFactory` supplied to the `SequencingService`. Here's an example:

```
this.observationService = new ObservationService(sessionFactory);
this.observationService.getAdministrator().start();
```



Note

Both the `ObservationService` implement `AdministeredService`, which has a `ServiceAdministrator` used to start, pause, and shutdown the service. In other words, the lifecycle of the services are managed in the same way.

After the observation service is started, listeners can be added. The `SequencingService` implements the required interface, and so it may be registered directly:

```
observationService.addListener(sequencingService);
```

Finally, the observation service must be wired to monitor one or your JCR repositories. This is done with one of the `monitor(...)` methods:

```
int eventTypes = Event.NODE_ADDED | Event.PROPERTY_ADDED |
    Event.PROPERTY_CHANGED;
observationService.monitor("Main Repository/Workspace1",
    eventTypes);
```

At this point, the observation service is listening to a JCR repository, and forwarding the appropriate events to the sequencing service, which will asynchronously process the changes and sequence the information added to or changed in the repository.

3. Shutting down JBoss DNA services

The JBoss DNA services are utilizing resources and threads that must be released your application is ready to shut down. The safe way to do this is to simply obtain the `ServiceAdministrator` for each service (via the `getServiceAdministrator()` method) and call `shutdown()`. As previously mentioned, the shutdown method will simply prevent new work from being process and will not wait for existing work to be completed. If you want to wait until the service completes all its work, you must wait until the service terminates. Here's an example that shows how this is done:

```
// Shut down the service and wait until it's all shut down ...
sequencingService.getAdministrator().shutdown();
```

```
sequencingService.getAdministrator().awaitTermination(5,
    TimeUnit.SECONDS);

// Shut down the observation service ...
observationService.getAdministrator().shutdown();
observationService.getAdministrator().awaitTermination(5,
    TimeUnit.SECONDS);
```

At this point, we've covered how to configure and use the JBoss DNA services in your application. The next chapter goes back to the [sample application](#) to show how all these pieces fit together.

4. Reviewing the example application

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

The example is comprised of 3 classes and 1 interface, located in the `src/main/java` directory:

```
org/jboss/example/dna/sequencers/ConsoleInput.java
                               /MediaInfo.java
                               /SequencingClient.java
                               /UserInterface.java
```

`SequencingClient` is the class that contains the main application. `MediaInfo` is a simple Java object that encapsulates metadata about a media file (as generated by the sequencer), and used by the client to pass information to the `UserInterface`, which is an interface with methods that will be called at runtime to request data from the user. `ConsoleInput` is an implementation of this that creates a text user interface, allowing the user to operate the client from the command line. We can easily create a graphical implementation of `UserInterface` at a later date. We can also create a mock implementation for testing purposes that simulates a user entering data. This allows us to check the behaviour of the client automatically using conventional JUnit test cases, as demonstrated by the code in the `src/test/java` directory:

```
org/jboss/example/dna/sequencers/SequencingClientTest.java
                               /MockUserInterface.java
```

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



Note

To keep the code shown in this book as readable as possible, some of the comments and error handling have been removed.

The `startRepository()` method starts up an in-memory Jackrabbit JCR repository. The bulk of this method is simply gathering and passing the information required by Jackrabbit. Because Jackrabbit's `TransientRepository` implementation shuts down after the last session is closed, the application maintains a session to ensure that the repository remains open throughout the application's lifetime. And finally, the node type needed by the image sequencer is registered with Jackrabbit.

```
public void startRepository() throws Exception {
    if (this.repository == null) {
        try {

            // Load the Jackrabbit configuration ...
            File configFile = new File(this.jackrabbitConfigPath);
            String pathToConfig = configFile.getAbsolutePath();

            // Find the directory where the Jackrabbit repository
            data will be stored ...
            File workingDirectory = new
            File(this.workingDirectory);
            String workingDirectoryPath =
            workingDirectory.getAbsolutePath();

            // Get the Jackrabbit custom node definition (CND) file
            ...
            URL cndFile =
            Thread.currentThread().getContextClassLoader().getResource("jackrabbitNodeTypes.cnd");

            // Create the Jackrabbit repository instance and
            establish a session to keep the repository alive ...
            this.repository = new TransientRepository(pathToConfig,
            workingDirectoryPath);
            if (this.username != null) {
                Credentials credentials = new
                SimpleCredentials(this.username, this.password);
                this.keepAliveSession =
                this.repository.login(credentials, this.workspaceName);
            } else {
                this.keepAliveSession = this.repository.login();
            }
        }
    }
}
```

```
    }

    try {
        // Register the node types (only valid the first
time) ...

        JackrabbitNodeTypeManager
mgr =
        (JackrabbitNodeTypeManager)this.keepAliveSession.getWorkspace().getNodeTypeManager
        mgr.registerNodeTypes(cndFile.openStream(),
        JackrabbitNodeTypeManager.TEXT_X_JCR_CND);
    } catch (RepositoryException e) {
        if (!e.getMessage().contains("already exists"))
throw e;
    }

    } catch (Exception e) {
        this.repository = null;
        this.keepAliveSession = null;
        throw e;
    }
}
}
```

As you can see, this method really has nothing to do with JBoss DNA, other than setting up a JCR repository that JBoss DNA will use.

The `shutdownRepository()` method shuts down the Jackrabbit transient repository by closing the "keep alive session". Again, this method really does nothing specifically with JBoss DNA, but is needed to manage the JCR repository that JBoss DNA uses.

```
public void shutdownRepository() throws Exception {
    if (this.repository != null) {
        try {
            this.keepAliveSession.logout();
        } finally {
            this.repository = null;
            this.keepAliveSession = null;
        }
    }
}
```

The `startDnaServices()` method first starts the JCR repository (if it were not already started), and proceeds to create and configure the `SequencingService` as described [earlier](#). This involves setting up the `SessionFactory`, `ExecutionContext`, creating the `SequencingService` instance, and configuring the image sequencer.

The method then continues by setting up the `ObservationService` as described [earlier](#) and starting the service.

```
public void startDnaServices() throws Exception {
    if (this.repository == null) this.startRepository();
    if (this.sequencingService == null) {

        SimpleSessionFactory sessionFactory = new
SimpleSessionFactory();
        sessionFactory.registerRepository(this.repositoryName,
this.repository);
        if (this.username != null) {
            Credentials credentials = new
SimpleCredentials(this.username, this.password);
            sessionFactory.registerCredentials(this.repositoryName
+ "/" + this.workspaceName, credentials);
        }
        this.executionContext = new
SimpleExecutionContext(sessionFactory);

        // Create the sequencing service, passing in the execution
context ...
        this.sequencingService = new SequencingService();

this.sequencingService.setExecutionContext(executionContext);

        // Configure the sequencers.
        String name = "Image Sequencer";
        String desc = "Sequences image files to extract the
characteristics of the image";
        String classname =
"org.jboss.dna.sequencer.images.ImageMetadataSequencer";
        String[] classpath = null; // Use the current classpath
        String[] pathExpressions
=
{ "/*.(jpg|jpeg|gif|bmp|pcx|png|iff|ras|pbm|pgm|ppm|psd))[*]/
jcr:content[@jcr:data] => /images/$1"};
        SequencerConfig imageSequencerConfig = new
SequencerConfig(name, desc, classname, classpath,
pathExpressions);
        this.sequencingService.addSequencer(imageSequencerConfig);

        // Set up the MP3 sequencer ...
        name = "Mp3 Sequencer";
        desc = "Sequences mp3 files to extract the id3 tags of the
audio file";
        classname =
"org.jboss.dna.sequencer.mp3.Mp3MetadataSequencer";
```

```
String[] mp3PathExpressions =
{"//(*.mp3)[*]/jcr:content[@jcr:data] => /mp3s/$1"};
SequencerConfig mp3SequencerConfig = new
SequencerConfig(name, desc, classname, classpath,
mp3PathExpressions);
this.sequencingService.addSequencer(mp3SequencerConfig);

// Use the DNA observation service to listen to the JCR
repository (or multiple ones), and
// then register the sequencing service as a listener to
this observation service...
this.observationService = new
ObservationService(this.executionContext.getSessionFactory());
this.observationService.getAdministrator().start();

this.observationService.addListener(this.sequencingService);
this.observationService.monitor(this.repositoryName + "/"
+ this.workspaceName, Event.NODE_ADDED | Event.PROPERTY_ADDED |
Event.PROPERTY_CHANGED);
}
// Start up the sequencing service ...
this.sequencingService.getAdministrator().start();
}
```

The `shutdownDnaServices()` method is pretty straightforward: it just calls `shutdown` on each of the services and waits until they terminate.

```
public void shutdownDnaServices() throws Exception {
    if (this.sequencingService == null) return;

    // Shut down the service and wait until it's all shut down ...
    this.sequencingService.getAdministrator().shutdown();
    this.sequencingService.getAdministrator().awaitTermination(5,
    TimeUnit.SECONDS);

    // Shut down the observation service ...
    this.observationService.getAdministrator().shutdown();
    this.observationService.getAdministrator().awaitTermination(5,
    TimeUnit.SECONDS);
}
```

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

The main method of the `SequencingClient` class creates a `SequencingClient` instance, and passes a new `ConsoleInput` instance:

```
public static void main( String[] args ) throws Exception {
    SequencingClient client = new SequencingClient();
    client.setRepositoryInformation("repo", "default", "jsmith",
    "secret".toCharArray());
    client.setUserInterface(new ConsoleInput(client));
}
```

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

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

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

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

5. Summarizing what we just did

In this chapter we covered the different JBoss DNA components and how they can be used in your application. Specifically, we described how the `SequencingService` and `ObservationService` can be configured and used. And we ended the chapter by reviewing the example application, which not only uses JBoss DNA, but also the repository via the JCR API.

Creating custom sequencers

The current release of JBoss DNA comes with two sequencers: one that extracts metadata from a variety of image file formats, and another that extracts some of the ID3 metadata from MP3 audio files. However, it's very easy to create your own sequencers and to then configure JBoss DNA to use them in your own application.

Creating a custom sequencer involves the following steps:

- Create a Maven 2 project for your sequencer;
- Implement the `org.jboss.dna.spi.sequencers.StreamSequencer` interface with your own implementation, and create unit tests to verify the functionality and expected behavior;
- Add the sequencer configuration to the JBoss DNA `SequencingService` in your application, as described in the [previous chapter](#), and
- Deploy the JAR file with your implementation (as well as any dependencies) and make them available to JBoss DNA in your application.

It's that simple.

1. Creating the Maven 2 project

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



Note

JBoss DNA 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](#).



Note

The `dna-sequencer-images` project is a small, self-contained sequencer implementation that has only the minimal dependencies.

Starting with this project's source and modifying it to suit your needs may be the easiest way to get started. See the subversion repository:
<http://anonsvn.jboss.org/repos/dna/trunk/sequencers/dna-sequencer-images/>

You can create your Maven project any way you'd like. For examples, see the [Maven 2 documentation](http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project) [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.jboss.dna</groupId>
  <artifactId>dna-common</artifactId>
  <version>0.1</version>
</dependency>
<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-spi</artifactId>
  <version>0.1</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
</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:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.4</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-library</artifactId>
  <version>1.1</version>
```

```

    <scope>test</scope>
  </dependency>
  <!-- Logging with Log4J -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.4.3</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
    <scope>test</scope>
  </dependency>

```

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

```

<dependency>
  <groupId>org.jboss.dna</groupId>
  <artifactId>dna-repository</artifactId>
  <version>0.1</version>
  <scope>test</scope>
</dependency>
<!-- Java Content Repository API -->
<dependency>
  <groupId>javax.jcr</groupId>
  <artifactId>jcr</artifactId>
  <version>1.0.1</version>
  <scope>test</scope>
</dependency>
<!-- Apache Jackrabbit (JCR Implementation) -->
<dependency>
  <groupId>org.apache.jackrabbit</groupId>
  <artifactId>jackrabbit-api</artifactId>
  <version>1.3.3</version>
  <scope>test</scope>
  <!-- Exclude these since they are included in JDK 1.5 -->
  <exclusions>
    <exclusion>
      <groupId>xml-apis</groupId>
      <artifactId>xml-apis</artifactId>
    </exclusion>
    <exclusion>
      <groupId>xerces</groupId>

```

```
        <artifactId>xercesImpl</artifactId>
    </exclusion>
</exclusions>
</dependency>
<dependency>
    <groupId>org.apache.jackrabbit</groupId>
    <artifactId>jackrabbit-core</artifactId>
    <version>1.3.3</version>
    <scope>test</scope>
    <!-- Exclude these since they are included in JDK 1.5 -->
    <exclusions>
        <exclusion>
            <groupId>xml-apis</groupId>
            <artifactId>xml-apis</artifactId>
        </exclusion>
        <exclusion>
            <groupId>xerces</groupId>
            <artifactId>xercesImpl</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

At this point, your project should be set up correctly, and you're ready to move on to [writing the Java implementation](#) for your sequencer.

2. Implementing the StreamSequencer interface

After creating the project and setting up the dependencies, the next step is to create a Java class that implements the `org.jboss.dna.spi.sequencers.StreamSequencer` interface. This interface is very straightforward, and involves a single method:

```
public interface StreamSequencer {

    /**
     * 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 progressMonitor the progress monitor that should be
     * kept
     * updated with the sequencer's progress and that should be
     * frequently consulted as to whether this operation has been
     * cancelled.
     */
}
```

```
*/  
void sequence( InputStream stream, SequencerOutput output,  
              ProgressMonitor progressMonitor );
```

The job of a stream sequencer is to process the data in the supplied stream, and place into the `SequencerOutput` any information that is to go into the JCR repository. JBoss DNA figures out when your sequencer should be called (of course using the sequencing configuration you'll add in a bit), and then makes sure the generated information is saved in the correct place in the repository.

The `SequencerOutput` class is fairly easy to use. There are basically two methods you need to call. One method sets the property values, 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. JBoss DNA will ensure that nodes are created or updated whenever they're needed.

```
public interface SequencerOutput {  
  
    /**  
     * 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 );  
  
    /**  
     * 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 );
}
```

JBoss DNA 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.

For a complete example of a sequencer, let's look at the `org.jboss.dna.sequencers.image.ImageMetadataSequencer` implementation:

```
public class ImageMetadataSequencer implements StreamSequencer {

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

    /**
     * {@inheritDoc}
     */
    public void sequence( InputStream stream, SequencerOutput
        output,
                           ProgressMonitor progressMonitor ) {
        progressMonitor.beginTask(10,
            ImageSequencerI18n.sequencerTaskName);
```

```

        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;
        }
        progressMonitor.worked(5);
        if (progressMonitor.isCancelled()) return;

        // Generate the output graph if we found useful metadata
        ...
        if (metadata != null) {
            // Place the image metadata into the output map ...
            output.setProperty(METADATA_NODE, IMAGE_PRIMARY_TYPE,
                "image:metadata");
            // output.psetProperty(METADATA_NODE, IMAGE_MIXINS,
            "");

            output.setProperty(METADATA_NODE, IMAGE_MIME_TYPE,
                metadata.getMimeType());
            // output.setProperty(METADATA_NODE, IMAGE_ENCODING,
            "");

            output.setProperty(METADATA_NODE, IMAGE_FORMAT_NAME,
                metadata.getFormatName());
            output.setProperty(METADATA_NODE, IMAGE_WIDTH,
                metadata.getWidth());
            output.setProperty(METADATA_NODE, IMAGE_HEIGHT,
                metadata.getHeight());
            output.setProperty(METADATA_NODE, IMAGE_BITS_PER_PIXEL,
                metadata.getBitsPerPixel());
            output.setProperty(METADATA_NODE, IMAGE_PROGRESSIVE,
                metadata.isProgressive());
            output.setProperty(METADATA_NODE,
                IMAGE_NUMBER_OF_IMAGES, metadata.getNumberOfImages());
            output.setProperty(METADATA_NODE,
                IMAGE_PHYSICAL_WIDTH_DPI, metadata.getPhysicalWidthDpi());
            output.setProperty(METADATA_NODE,
                IMAGE_PHYSICAL_HEIGHT_DPI, metadata.getPhysicalHeightDpi());
            output.setProperty(METADATA_NODE,
                IMAGE_PHYSICAL_WIDTH_INCHES, metadata.getPhysicalWidthInch());
            output.setProperty(METADATA_NODE,
                IMAGE_PHYSICAL_HEIGHT_INCHES, metadata.getPhysicalHeightInch());
        }

        progressMonitor.done();
    }

```

```
}
```

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.)

Also note how the progress monitor is used. Reporting progress through the supplied `ProgressMonitor` is very easy, and it ensures that JBoss DNA can accurately monitor and report the status of sequencing activities to the users. At the beginning of the operation, call `beginTask(...)` with a meaningful message describing the operation and a total for the amount of work that will be done by this sequencer. Then perform the sequencing work, periodically reporting work by specifying the incremental amount of work with the `worked(double)` method, or by creating a subtask with the `createSubtask(double)` method and reporting work against that subtask monitor.

Your method should periodically use the `ProgressMonitor`'s `isCancelled()` method to check whether the operation has been cancelled.. If this method returns true, the implementation should abort all work as soon as possible and close any resources that were acquired or opened.

Finally, when your sequencing operation is completed, it should call `done()` on the progress monitor.

3. Testing custom sequencers

The sequencing framework was designed to make testing sequencers much easier. In particular, the `StreamSequencer` interface does not make use of the JCR API. So instead of requiring a fully-configured JCR repository and JBoss DNA 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.jboss.dna.sequencer.images` package of the `dna-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 JBoss DNA. Of course, this example code does not do any error handling and does not make all the assertions a real test would.

```

Sequencer sequencer = new ImageMetadataSequencer();
MockSequencerOutput output = new MockSequencerOutput();
ProgressMonitor progress = new SimpleProgressMonitor("Test
    activity");
InputStream stream = null;
try {
    stream =
    this.getClass().getClassLoader().getResource("caution.gif").openStream();
    sequencer.sequence(stream,output,progress);    // 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 sequencer = new ImageMetadataSequencer();
MockSequencerOutput output = new MockSequencerOutput();
ProgressMonitor progress = new SimpleProgressMonitor("Test
    activity");
InputStream stream = null;
try {
    stream =
    this.getClass().getClassLoader().getResource("caution.pict").openStream();
    sequencer.sequence(stream,output,progress);    // writes to
    'output'
    assertThat(output.getProperties(), 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 are produced.

It may also be useful to create some integration tests that [configures JBoss DNA](#) to use the custom sequencer, and to then upload content using the JCR API,

verifying that the custom sequencer did run. However, remember that JBoss DNA 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. (One way of doing this is to wait for a second after uploading your content, shutdown the `SequencingService` and await its termination, and then check that the sequencer output has been saved to the JCR repository. For an example of this technique, see the `SequencingClientTest` unit test in the example application.)

4. Deploying custom sequencers

The first step of deploying a sequencer consists of adding/changing the sequencer configuration (e.g., `SequencerConfig`) in the `SequencingService`. This was covered in the [previous chapter](#).

The second step is to make the sequencer implementation available to JBoss DNA. At this time, the JAR containing your new sequencer, as well as any JARs that your sequencer depends on, should be placed on your application classpath.



Note

A future goal of JBoss DNA is to allow sequencers, connectors and other extensions to be easily deployed into a runtime repository. This process will not only be much simpler, but it will also provide JBoss DNA with the information necessary to update configurations and create the appropriate class loaders for each extension. Having separate class loaders for each extension helps prevent the pollution of the common classpath, facilitates an isolated runtime environment to eliminate any dependency conflicts, and may potentially enable hot redeployment of newer extension versions.

Looking to the future

What's next for JBoss DNA? Well, the sequencing system is just the beginning. With this release, the sequencing system is stable enough so that more [sequencers](#) can be developed and used within your own applications. If you're interested in getting involved with the JBoss DNA project, consider picking up one of the sequencers on our [roadmap](#) [<http://jira.jboss.org/jira/browse/DNA?report=com.atlassian.jira.plugin.system.project:roadmap-panel>]. Or, check out [JIRA](#) [<http://jira.jboss.org/jira/secure/IssueNavigator.jspa?reset=true&mode=hide&pid=12310520&sorter/order=DESC&sorter/field=priority&resolution=-1&component=12311436>] for the list of sequencers we've thought of. If you think of one that's not there, please add it to JIRA!

The next release will focus on creating the [federation engine](#) and connectors for several popular and ubiquitous systems. The 0.2 release will likely only federate information in a read-only manner, but updates will soon follow. Also, during the early part of the next release, the JBoss DNA project will switch to use JDK 6. Java 5 is being end-of-lifed, so we want to move to a supported JDK. However, a number of JBoss projects and products continue to require Java 5, so our next release will most likely use JDK 6 with Java 5 compatibility.

Other components on our roadmap include a web user interface, a REST-ful server, and a view system that allows domain-specific views of information in the repository. These components are farther out on our roadmap, and at this time have not been targeted to a particular release. If any of these are of interest to you, please [get involved](#) in the community.
