Example 11-34. Defining a Quartz scheduler to execute HDFS scripts and MapReduce jobs

```
<!-- job definition as before -->
<hdp:job id="wordcountJob" … />

<!-- script definition as before -->
<hdp:script id="setupScript" … />

<!-- simple job runner as before -->
<hdp:job-runner

<bean id="jobDetail"
      class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="runner"/>
  <property name="targetMethod" value="run"/>
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
  <property name="jobDetail" ref="jobDetail"/>
  <property name="cronExpression" value="3/30 * * * * ?"/>
</bean>

<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers" ref="cronTrigger"/>
</bean>
```

Quartz's `JobDetail` class encapsulates what code will execute when the trigger condition is satisfied. Spring's helper class `MethodInvokingJobDetailFactoryBean` will create a `JobDetail` object whose behavior is delegated to invoking the specified method on a Spring-managed object. This application can be found in the directory *hadoop/scheduling-quartz*. Running the application gives similar output to the previous Spring `TaskScheduler`-based example.

CHAPTER 12

Analyzing Data with Hadoop

While the MapReduce programming model is at the heart of Hadoop, it is low-level and as such becomes a unproductive way for developers to write complex analysis jobs. To increase developer productivity, several higher-level languages and APIs have been created that abstract away the low-level details of the MapReduce programming model. There are several choices available for writing data analysis jobs. The Hive and Pig projects are popular choices that provide SQL-like and procedural data flow-like languages, respectively. HBase is also a popular way to store and analyze data in HDFS. It is a column-oriented database, and unlike MapReduce, provides random read and write access to data with low latency. MapReduce jobs can read and write data in HBase's table format, but data processing is often done via HBase's own client API. In this chapter, we will show how to use Spring for Apache Hadoop to write Java applications that use these Hadoop technologies.

Using Hive

The previous chapter used the MapReduce API to analyze data stored in HDFS. While counting the frequency of words is relatively straightforward with the MapReduce API, more complex analysis tasks don't fit the MapReduce model as well and thus reduce developer productivity. In response to this difficulty, Facebook developed Hive as a means to interact with Hadoop in a more declarative, SQL-like manner. Hive provides a language called HiveQL to analyze data stored in HDFS, and it is easy to learn since it is similar to SQL. Under the covers, HiveQL queries are translated into multiple jobs based on the MapReduce API. Hive is now a top-level Apache project and is still heavily developed by Facebook.

While providing a deep understanding of Hive is beyond the scope of this book, the basic programming model is to create a Hive table schema that provides structure on top of the data stored in HDFS. HiveQL queries are then pared by the Hive engine, translating them into MapReduce jobs in order to execute the queries. HiveQL statements can be submitted to the Hive engine through the command line or through a component called the Hive Server, which provides access via JDBC, ODBC, or Thrift.

For more details on how to install, run, and develop with Hive and HiveQL, refer to the project website as well as the book *Programming Hive* (O'Reilly).

As with MapReduce jobs, Spring for Apache Hadoop aims to simplify Hive programming by removing the need to use command-line tools to develop and execute Hive applications. Instead, Spring for Apache Hadoop makes it easy to write Java applications that connect to a Hive server (optionally embedded), create Hive Thrift clients, and use Spring's rich JDBC support (`JdbcTemplate`) via the Hive JDBC driver.

Hello World

As an introduction to using Hive, in this section we will perform a small analysis on the Unix password file using the Hive command line. The goal of the analysis is to create a report on the number of users of a particular shell (e.g., bash or sh). To install Hive, download it from the main Hive website. After installing the Hive distribution, add its *bin* directory to your path. Now, as shown in Example 12-1, we start the Hive command-line console to execute some HiveQL commands.

Example 12-1. Analyzing a password file in the Hive command-line interface

```
$ hive
hive> drop table passwords;
hive> create table passwords (user string, passwd string, uid int, gid int,
                              userinfo string, home string, shell string)
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY ':' LINES TERMINATED BY '10';
hive> load data local inpath '/etc/passwd' into table passwords;
Copying data from file:/etc/passwd
Copying file: file:/etc/passwd
Loading data to table default.passwords
OK
hive> drop table grpshell;
hive> create table grpshell (shell string, count int);
hive> INSERT OVERWRITE TABLE grpshell SELECT p.shell, count(*)
                                      FROM passwords p GROUP BY p.shell;
Total MapReduce jobs = 1
Launching Job 1 out of 1
…
Total MapReduce CPU Time Spent: 1 seconds 980 msec
hive> select * from grpshell;
OK
/bin/bash 5
/bin/false 16
/bin/sh 18
/bin/sync 1
/usr/sbin/nologin 1
Time taken: 0.122 seconds
```

You can also put the HiveQL commands in a file and execute that from the command line (Example 12-2).

Example 12-2. Executing Hive from the command line

```
$ hive -f password-analysis.hql
$ hadoop dfs -cat /user/hive/warehouse/grpshell/000000_0

/bin/bash 5
/bin/false 16
/bin/sh 18
/bin/sync 1
/usr/sbin/nologin 1
```

The Hive command line passes commands directly to the Hive engine. Hive also supports variable substitution using the notation `${hiveconf:varName}` inside the script and the command line argument `-hiveconfg varName=varValue`. To interact with Hive outside the command line, you need to connect to a Hive server using a Thrift client or over JDBC. The next section shows how you can start a Hive server on the command line or bootstrap an embedded server in your Java application.

Running a Hive Server

In a production environment, it is most common to run a Hive server as a standalone server process—potentially multiple Hive servers behind a HAProxy—to avoid some known issues with handling many concurrent client connections.[1]

If you want to run a standalone server for use in the sample application, start Hive using the command line:

```
hive --service hiveserver -hiveconf fs.default.name=hdfs://localhost:9000 \
     -hiveconf mapred.job.tracker=localhost:9001
```

Another alternative, useful for development or to avoid having to run another server, is to bootstrap the Hive server in the same Java process that will run Hive client applications. The Hadoop namespace makes embedding the Hive server a one-line configuration task, as shown in Example 12-3.

Example 12-3. Creating a Hive server with default options

```
<hive-server/>
```

By default, the hostname is `localhost` and the port is 10000. You can change those values using the `host` and `port` attributes. You can also provide additional options to the Hive server by referencing a properties file with the `properties-location` attribute or by inlining properties inside the `<hive-server/>` XML element. When the `ApplicationContext` is created, the Hive server is started automatically. If you wish to override this behavior, set the `auto-startup` element to `false`. Lastly, you can reference a specific Hadoop configuration object, allowing you to create multiple Hive servers that connect to different Hadoop clusters. These options are shown in Example 12-4.

1. *https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Thrift+API*

Example 12-4. Creating and configuring a Hive server

```
<context:property-placeholder location="hadoop.properties,hive.properties" />

<configuration id="hadoopConfiguration">
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>

<hive-server port="${hive.port}" auto-startup="false"
             configuration-ref="hadoopConfiguration"
             properties-location="hive-server.properties">
  hive.exec.scratchdir=/tmp/hive/
</hive-server>
```

The files *hadoop.properties* and *hive.properties* are loaded from the classpath. Their combined values are shown in Example 12-5. We can use the property file *hive-server-config.properties* to configure the server; these values are the same as those you would put inside *hive-site.xml*.

Example 12-5. Properties used to configure a simple Hive application

```
hd.fs=hdfs://localhost:9000
mapred.job.tracker=localhost:9001
hive.host=localhost
hive.port=10000
hive.table=passwords
```

Using the Hive Thrift Client

The Hadoop namespace supports creating a Thrift client, as shown in Example 12-6.

Example 12-6. Creating and configuring a Hive Thrift client

```
<hive-client-factory host="${hive.host}" port="${hive.port}"/>
```

The namespace creates an instance of the class `HiveClientFactory`. Calling the method `getHiveClient` on `HiveClientFactory` will return a new instance of the `HiveClient`. This is a convenient pattern that Spring provides since the `HiveClient` is not a thread-safe class, so a new instance needs to be created inside methods that are shared across multiple threads. Some of the other parameters that we can set on the `HiveClient` through the XML namespace are the connection timeout and a collection of scripts to execute once the client connects. To use the `HiveClient`, we create a `HivePasswordRepository` class to execute the *password-analysis.hql* script used in the previous section and then execute a query against the passwords table. Adding a `<context:component-scan/>` element to the configuration for the Hive server shown earlier will automatically register the `HivePasswordRepository` class with the container by scanning the classpath for classes annotated with the Spring stereotype `@Repository` annotation. See Example 12-7.

Example 12-7. Using the Thrift HiveClient in a data access layer

```
@Repository
public class HivePasswordRepository implements PasswordRepository {

  private static final Log logger = LogFactory.getLog(HivePasswordRepository.class);

  private HiveClientFactory hiveClientFactory;
  private String tableName;

  // constructor and setters omitted

  @Override
  public Long count() {
    HiveClient hiveClient = hiveClientFactory.getHiveClient();
    try {
      hiveClient.execute("select count(*) from " + tableName);
      return Long.parseLong(hiveClient.fetchOne());
      // checked exceptions
    } catch (HiveServerException ex) {
      throw translateExcpetion(ex);
    } catch (org.apache.thrift.TException tex) {
      throw translateExcpetion(tex);
    } finally {
      try {
        hiveClient.shutdown();
      } catch (org.apache.thrift.TException tex) {
        logger.debug("Unexpected exception on shutting down HiveClient", tex);
      }
    }
  }

  @Override
  public void processPasswordFile(String inputFile) {
    // Implementation not shown
  }

  private RuntimeException translateExcpetion(Exception ex) {
    return new RuntimeException(ex);
  }
}
```

The sample code for this application is located in *./hadoop/hive*. Refer to the *readme* file in the sample's directory for more information on running the sample application. The driver for the sample application will call `HivePasswordRepository`'s `processPasswordFile` method and then its count method, returning the value `41` for our dataset. The error handling is shown in this example to highlight the data access layer development best practice of avoiding throwing checked exceptions to the calling code.

The helper class `HiveTemplate`, which provides a number of benefits that can simplify the development of using Hive programmatically. It translates the `HiveClient`'s checked exceptions and error codes into Spring's portable DAO exception hierarchy. This means that calling code does not have to be aware of Hive. The `HiveClient` is also not

thread-safe, so as with other template classes in Spring, the `HiveTemplate` provides thread-safe access to the underlying resources so you don't have to deal with the incidental complexity of the `HiveClient`'s API. You can instead focus on executing HSQL and getting results. To create a `HiveTemplate`, use the XML namespace and optionally pass in a reference to the name of the `HiveClientFactory`. Example 12-8 is a minimal configuration for the use of a new implementation of `PasswordRepository` that uses the `HiveTemplate`.

Example 12-8. Configuring a HiveTemplate

```
<context:property-placeholder location="hadoop.properties,hive.properties"/>

<configuration>
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>

<hive-client-factory host="${hive.host}" port="${hive.port}"/>

<hive-template/>
```

The XML namespace for `<hive-template/>` will also let you explicitly reference a `HiveClientFactory` by name using the `hive-client-factory-ref` element. Using `HiveTemplate`, the `HiveTemplatePasswordRepository` class is now much more simply implemented. See Example 12-9.

Example 12-9. PersonRepository implementation using HiveTemplate

```
@Repository
public class HiveTemplatePasswordRepository implements PasswordRepository {

  private HiveOperations hiveOperations;
  private String tableName;

  // constructor and setters omitted

  @Override
  public Long count() {
    return hiveOperations.queryForLong("select count(*) from " + tableName);
  }

  @Override
  public void processPasswordFile(String inputFile) {
    Map parameters = new HashMap();
    parameters.put("inputFile", inputFile);
    hiveOperations.query("classpath:password-analysis.hql", parameters);
  }
}
```

Note that the `HiveTemplate` class implements the `HiveOperations` interface. This is a common implementation style of Spring template classes since it facilitates unit testing, as interfaces can be easily mocked or stubbed. The helper method `queryForLong` makes

it a one liner to retrieve simple values from Hive queries. `HiveTemplate`'s query methods also let you pass a reference to a script location using Spring's `Resource` abstraction, which provides great flexibility for loading an `InputStream` from the classpath, a file, or over HTTP. The query method's second argument is used to replace substitution variables in the script with values. `HiveTemplate` also provides an execute callback method that will hand you a managed `HiveClient` instance. As with other template classes in Spring, this will let you get at the lower-level API if any of the convenience methods do not meet your needs but you will still benefit from the template's exception, translation, and resource management features.

Spring for Apache Hadoop also provides a `HiveRunner` helper class that like the `JobRunner`, lets you execute HDFS script operations before and after running a HiveQL script. You can configure the runner using the XML namespace element `<hive-runner/>`.

Using the Hive JDBC Client

The JDBC support for Hive lets you use your existing Spring knowledge of `JdbcTemplate` to interact with Hive. Hive provides a `HiveDriver` class that can be passed into Spring's `SimpleDriverDataSource`, as shown in Example 12-10.

Example 12-10. Creating and configuring a Hive JDBC-based access

```
<bean id="hiveDriver" class="org.apache.hadoop.hive.jdbc.HiveDriver" />

<bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
  <constructor-arg name="driver" ref="hiveDriver" />
  <constructor-arg name="url" value="${hive.url}"/>
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.simple.JdbcTemplate">
  <constructor-arg ref="dataSource" />
</bean>
```

`SimpleDriverDataSource` provides a simple implementation of the standard JDBC `DataSource` interface given a `java.sql.Driver` implementation. It returns a new connection for each call to the `DataSource`'s `getConnection` method. That should be sufficient for most Hive JDBC applications, since the overhead of creating the connection is low compared to the length of time for executing the Hive operation. If a connection pool is needed, it is easy to change the configuration to use Apache Commons DBCP or c3p0 connection pools.

`JdbcTemplate` brings a wide range of `ResultSet` to POJO mapping functionality as well as translating error codes into Spring's portable DAO (data access object) exception hierarchy. As of Hive 0.10, the JDBC driver supports generating meaningful error codes. This allows you to easily distinguish between catching Spring's `TransientDataAccessException` and `NonTransientDataAccessException`. *Transient* exceptions indicate that the operation can be retried and will probably succeed, whereas a *nontransient* exception indicates that retrying the operation will not succeed.

An implementation of the `PasswordRepository` using `JdbcTemplate` is shown in Example 12-11.

Example 12-11. PersonRepository implementation using JdbcTemplate

```
@Repository
public class JdbcPasswordRepository implements PasswordRepository {

  private JdbcOperations jdbcOperations;
  private String tableName;

  // constructor and setters omitted

  @Override
  public Long count() {
    return jdbcOperations.queryForLong("select count(*) from " + tableName);
  }

  @Override
  public void processPasswordFile(String inputFile) {
    // Implementation not shown.
  }
}
```

The implementation of the method processPasswordFile is somewhat lengthy due to the need to replace substitution variables in the script. Refer to the sample code for more details. Note that Spring provides the utility class `SimpleJdbcTestUtils` is part of the testing package; it's often used to execute DDL scripts for relational databases but can come in handy when you need to execute HiveQL scripts without variable substitution.

Apache Logfile Analysis Using Hive

Next, we will perform a simple analysis on Apache HTTPD logfiles using Hive. The structure of the configuration to run this analysis is similar to the one used previously to analyze the `password` file with the `HiveTemplate`. The HiveQL script shown in Example 12-12 generates a file that contains the cumulative number of hits for each URL. It also extracts the minimum and maximum hit numbers and a simple table that can be used to show the distribution of hits in a simple chart.

Example 12-12. HiveQL for basic Apache HTTPD log analysis

```
ADD JAR ${hiveconf:hiveContribJar};

DROP TABLE IF EXISTS apachelog;
CREATE TABLE apachelog(remoteHost STRING, remoteLogname STRING, user STRING, time STRING,
                       method STRING, uri STRING, proto STRING, status STRING,
                       bytes STRING, referer STRING, userAgent STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
```

```
  "input.regex" = "^([^ ]*) +([^ ]*) +([^ ]*) +\\[([^]]*)\\] +\\\"([^ ]*) ([^ ]*)
 ([^ ]*)\\\" ([^ ]*) ([^ ]*) (?:\\\"-\\\")*\\\"(.*)\\\" (.*)$",
  "output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s %9$s %10$s %11$s")
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH "${hiveconf:localInPath}" INTO TABLE apachelog;

-- basic filtering
-- SELECT a.uri FROM apachelog a WHERE a.method='GET' AND a.status='200';

-- determine popular URLs (for caching purposes)

INSERT OVERWRITE LOCAL DIRECTORY 'hive_uri_hits' SELECT a.uri, "\t", COUNT(*)
 FROM apachelog a GROUP BY a.uri ORDER BY uri;

-- create histogram data for charting, view book sample code for details
```

This example uses the utility library *hive-contrib.jar*, which contains a serializer/deserializer that can read and parse the file format of Apache logfiles. The *hive-contrib.jar* can be downloaded from Maven central or built directly from the source. While we have parameterized the location of the hive-contrib.jar another option is to put a copy of the jar into the Hadoop library directory on all task tracker machines. The results are placed in local directories. The sample code for this application is located in *./hadoop/hive*. Refer to the *readme* file in the sample's directory for more information on running the application. A sample of the contents of the data in the *hive_uri_hits* directory is shown in Example 12-13.

Example 12-13. The cumulative number of hits for each URI

```
/archives.html 3
/archives/000005.html 2
/archives/000021.html 1
…
/archives/000055.html 1
/archives/000064.html 2
```

The contents of the *hive_histogram* directory show that there is 1 URL that has been requested 22 times, 3 URLs were each hit 4 times, and 74 URLs have been hit only once. This gives us an indication of which URLs would benefit from being cached. The sample application shows two ways to execute the Hive script, using the `HiveTemplate` and the `HiveRunner`. The configuration for the `HiveRunner` is shown in Example 12-14.

Example 12-14. Using a HiveRunner to run the Apache Log file analysis

```
<context:property-placeholder location="hadoop.properties,hive.properties"/>

<configuration>
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>
```

```
<hive-server port="${hive.port}"
             properties-location="hive-server.properties"/>

<hive-client-factory host="${hive.host}" port="${hive.port}"/>

<hive-runner id="hiveRunner" run-at-startup="false" >
  <script location="apache-log-simple.hql">
    <arguments>
      hiveContribJar=${hiveContribJar}
      localInPath="./data/apache.log"
    </arguments>
  </script>
</hive-runner>
```

While the size of this dataset was very small and we could have analyzed it using Unix command-line utilities, using Hadoop lets us scale the analysis over very large sets of data. Hadoop also lets us cheaply store the raw data so that we can redo the analysis without the information loss that would normally result from keeping summaries of historical data.

Using Pig

Pig provides an alternative to writing MapReduce applications to analyze data stored in HDFS. Pig applications are written in the Pig Latin language, a high-level data processing language that is more in the spirit of using `sed` or `awk` than the SQL-like language that Hive provides. A Pig Latin script describes a sequence of steps, where each step performs a transformation on items of data in a collection. A simple sequence of steps would be to load, filter, and save data, but more complex operation—such as joining two data items based on common values—are also available. Pig can be extended by user-defined functions (UDFs) that encapsulate commonly used functionality such as algorithms or support for reading and writing well-known data formats such as Apache HTTPD logfiles. A `PigServer` is responsible for translating Pig Latin scripts into multiple jobs based on the MapReduce API and executing them.

A common way to start developing a Pig Latin script is to use the interactive console that ships with Pig, called Grunt. You can execute scripts in two different run modes. The first is the LOCAL mode, which works with data stored on the local filesystem and runs MapReduce jobs locally using an embedded version of Hadoop. The second mode, MAPREDUCE, uses HDFS and runs MapReduce jobs on the Hadoop cluster. By using the local filesystem, you can work on a small set of the data and develop your scripts iteratively. When you are satisfied with your script's functionality, you can easily switch to running the same script on the cluster over the full dataset. As an alternative to using the interactive console or running Pig from the command line, you can embed the Pig in your application. The `PigServer` class encapsulates how you can programmatically connect to Pig, execute scripts, and register functions.

Spring for Apache Hadoop makes it very easy to embed the `PigServer` in your application and to run Pig Latin scripts programmatically. Since Pig Latin does not have control flow statements such as conditional branches (if-else) or loops, Java can be useful to fill in those gaps. Using Pig programmatically also allows you to execute Pig scripts in response to event-driven activities using Spring Integration, or to take part in a larger workflow using Spring Batch.

To get familiar with Pig, we will first write a basic application to analyze the Unix password files using Pig's command-line tools. Then we show how you can use Spring for Apache Hadoop to develop Java applications that make use of Pig. For more details on how to install, run, and develop with Pig and Pig Latin, refer to the project website as well as the book *Programming Pig* (O'Reilly).

Hello World

As a Hello World exercise, we will perform a small analysis on the Unix password file. The goal of the analysis is to create a report on the number of users of a particular shell (e.g., bash or sh). Using familiar Unix utilities, you can easily see how many people are using the bash shell (Example 12-15).

Example 12-15. Using Unix utilities to count users of the bash shell

```
$ $ more /etc/passwd | grep /bin/bash
root:x:0:0:root:/root:/bin/bash
couchdb:x:105:113:CouchDB Administrator,,,:/var/lib/couchdb:/bin/bash
mpollack:x:1000:1000:Mark Pollack,,,:/home/mpollack:/bin/bash
postgres:x:116:123:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
testuser:x:1001:1001:testuser,,,,:/home/testuser:/bin/bash

$ more /etc/passwd | grep /bin/bash | wc -l
5
```

To perform a similar analysis using Pig, we first load the */etc/password* file into HDFS (Example 12-16).

Example 12-16. Copying /etc/password into HDFS

```
$ hadoop dfs -copyFromLocal /etc/passwd /test/passwd
$ hadoop dfs -cat /test/passwd

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
…
```

To install Pig, download it from the main Pig website. After installing the distribution, you should add the Pig distribution's *bin* directory to your path and also set the environment variable `PIG_CLASSPATH` to point to the Hadoop configuration directory (e.g., `export PIG_CLASSPATH=$HADOOP_INSTALL/conf/`).

Now we start the Pig interactive console, Grunt, in LOCAL mode and execute some Pig Latin commands (Example 12-17).

Example 12-17. Executing Pig Latin commands using Grunt

```
$ pig -x local
grunt> passwd = LOAD '/test/passwd' USING PigStorage(':') \
       AS (username:chararray, password:chararray, uid:int, gid:int, userinfo:chararray,
           home_dir:chararray, shell:chararray);
grunt> grouped_by_shell = GROUP passwd BY shell;
grunt> password_count = FOREACH grouped_by_shell GENERATE group, COUNT(passwd);
grunt> STORE password_count into '/tmp/passwordAnalysis';
grunt> quit
```

Since the example dataset is small, all the results fit in a single tab-delimited file, as shown in Example 12-18.

Example 12-18. Command execution result

```
$ hadoop dfs -cat /tmp/passwordAnalysis/part-r-00000

/bin/sh 18
/bin/bash 5
/bin/sync 1
/bin/false 16
/usr/sbin/nologin 1
```

The general flow of the data transformations taking place in the Pig Latin script is as follows. The first line loads the data from the HDFS file, */test/passwd*, into the variable named `passwd`. The `LOAD` command takes the location of the file in HDFS, as well as the format by which the lines in the file should be broken up, in order to create a dataset (aka, a Pig relation). In this example, we are using the `PigStorage` function to load the text file and separate the fields based on a colon character. Pig can apply a schema to the columns that are in the file by defining a name and a data type to each column. With the input dataset assigned to the variable `passwd`, we can now perform operations on the dataset to transform it into other derived datasets. We create the dataset `grouped_by_shell` using the `GROUP` operation. The `grouped_by_shell` dataset will have the shell name as the key and a collection of all records in the `passwd` dataset that have that shell value. If we had used the `DUMP` operation to view the contents of the `grouped_by_shell` dataset for the `/bin/bash` key, we would see the result shown in Example 12-19.

Example 12-19. Group-by-shell dataset

```
(/bin/bash,{(testuser,x,1001,1001,testuser,,,,,/home/testuser,/bin/bash),
            (root,x,0,0,root,/root,/bin/bash),
            (couchdb,x,105,113,CouchDB Administrator,,,,/var/lib/couchdb,/bin/bash),
            (mpollack,x,1000,1000,Mark Pollack,,,,/home/mpollack,/bin/bash),
            (postgres,x,116,123,PostgreSQL administrator,,,,/var/lib/postgresql,/bin/bash)
           })
```

The key is the shell name and the value is a collection, or bag, of password records that have the same key. In the next line, the expression `FOREACH grouped_by_shell GENERATE` will apply an operation on each record of the `grouped_by_shell` dataset and generate a new record. The new dataset that is created will group together all the records with the same key and count them.

We can also parameterize the script to avoid hardcoding values—for example, the input and output locations. In Example 12-20, we put all the commands entered into the interactive console into a file named *password-analysis.pig*, parameterized by the variables `inputFile` and `outputDir`.

Example 12-20. Parameterized Pig script

```
passwd = LOAD '$inputFile' USING PigStorage(':')
         AS (username:chararray, password:chararray, uid:int, gid:int, userinfo:chararray,
             home_dir:chararray, shell:chararray);
grouped_by_shell = GROUP passwd BY shell;
password_count = FOREACH grouped_by_shell GENERATE group, COUNT(passwd);
STORE password_count into '$outputDir';
```

To run this script in the interactive console, use the `run` command, as shown in Example 12-21.

Example 12-21. Running a parameterized Pig script in Grunt

```
grunt> run -param inputFile=/test/passwd outputDir=/tmp/passwordAnalysis
  password-analysis.pig
grunt> exit

$ hadoop dfs -cat /tmp/passwordAnalysis/part-r-00000

/bin/sh 18
/bin/bash 5
/bin/sync 1
/bin/false 16
/usr/sbin/nologin 1
```

Or, to run the script directly in the command line, see Example 12-22.

Example 12-22. Running a parameterized Pig script on the command line

```
$ pig -file password-analysis.pig -param inputFile=/test/passwd
  -param outputDir=/tmp/passwordAnalysis
```

Running a PigServer

Now we'll shift to using a more structured and programmatic way of running Pig scripts. Spring for Apache Hadoop makes it easy to declaratively configure and create a `PigServer` in a Java application, much like the Grunt shell does under the covers. Running a `PigServer` inside the Spring container will let you parameterize and externalize properties that control what Hadoop cluster to execute your scripts against,

properties of the PigServer, and arguments passed into the script. Spring's XML namespace for Hadoop makes it very easy to create and configure a PigServer. As Example 12-23 demonstrates, this configuration is done in the same way as the rest of your application configuration. The location of the optional Pig initialization script can be any Spring resource URL, located on the filesystem, classpath, HDFS, or HTTP.

Example 12-23. Configuring a PigServer

```
<context:property-placeholder location="hadoop.properties" />

<configuration>
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>

<pig-factory properties-location="pig-server.properties"
  <script location="initialization.pig">
    <arguments>
      inputFile=${initInputFile}
    </arguments>
  </script>
<pig-factory/>
```

In Example 12-24, the script is located from the classpath and the variables to be replaced are contained in the property file *hadoop.properties*.

Example 12-24. hadoop.properties

```
hd.fs=hdfs://localhost:9000
mapred.job.tracker=localhost:9001
inputFile=/test/passwd
outputDir=/tmp/passwordAnalysis
```

Some of the other attributes on the <pig-factory/> namespace are properties-location, which references a properties file to configure properties of the PigServer; job-tracker, which sets the location of the job tracker used to a value different than that used in the Hadoop configuration; and job-name, which sets the root name of the MapReduce jobs created by Pig so they can be easily identified as belonging to this script.

Since the PigServer class is not a thread-safe object and there is state created after each execution that needs to be cleaned up, the <pig-factory/> namespace creates an instance of a PigServerFactory so that you can easily create new PigServer instances as needed. Similar in purpose to JobRunner and HiveRunner, the PigRunner helper class to provide a convenient way to repeatedly execute Pig jobs and also execute HDFS scripts before and after their execution. The configuration of the PigRunner is shown in Example 12-25.

Example 12-25. Configuring a PigRunner

```
<pig-runner id="pigRunner"
            pre-action="hdfsScript"
            run-at-startup="true" >
```

```
  <script location="password-analysis.pig">
    <arguments>
      inputDir=${inputDir}
      outputDir=${outputDir}
    </arguments>
  </script>
</pig-runner>
```

We set the `run-at-startup` element to `true`, enabling the Pig script to be executed when the Spring `ApplicationContext` is started (the default is `false`). The sample application is located in the directory *hadoop/pig*. To run the sample password analysis application, run the commands shown in Example 12-26.

Example 12-26. Command to build and run the Pig scripting example

```
$ cd hadoop/pig
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/pigApp
```

Since the `PigServerFactory` and `PigRunner` classes are Spring-managed objects, they can also be injected into any other object managed by Spring. It is often convenient to inject the `PigRunner` helper class to ensure that a new instance of the `PigServer` is created for each execution of the script and that its resources used are cleaned up after execution. For example, to run a Pig job asynchronously as part of a service layer in an application, we inject the `PigRunner` and use Spring's `@Async` annotation (Example 12-27).

Example 12-27. Dependency injection of a PigRunner to execute a Pig job asynchronously

```
@Component
public class AnalysisService {

  private PigRunner pigRunner;

  @Autowired
  public AnalysisService(PigRunner pigRunner)
    this.pigRunner = pigRunner;
  }

  @Async
  public void performAnalysis() {
    pigRunner.call();
  }
}
```

Controlling Runtime Script Execution

To have more runtime control over what Pig scripts are executed and the arguments passed into them, we can use the `PigTemplate` class. As with other template classes in Spring, `PigTemplate` manages the underlying resources on your behalf, is thread-safe once configured, and will translate Pig errors and exceptions into Spring's portable DAO exception hierarchy. Spring's DAO exception hierarchy makes it easier to work

across different data access technologies without having to catch exceptions or look for return codes, which are specific to each technology. Spring's DAO exception hierarchy also helps to separate out nontransient and transient exceptions. In the case of a transient exception being thrown, the failed operation might be able to succeed if it is retried again. Using retry advice on a data access layer via Spring's AOP (aspect-oriented programming) support is one way to implement this functionality. Since Spring's JDBC helper classes also perform the same exception translation, exceptions thrown in any Hive-based data access that uses Spring's JDBC support will also map into the DAO hierarchy. While switching between Hive and Pig is not a trivial task, since analysis scripts need to be rewritten, you can at least insulate calling code from the differences in implementation between a Hive-based and a Pig-based data access layer. It will also allow you to more easily mix calls to Hive- and Pig-based data access classes and handle errors in a consistent way.

To configure a `PigTemplate`, create a `PigServerFactory` definition as before and add a `<pig-template/>` element (Example 12-28). We can define common configuration properties of the `PigServer` in a properties file specified by the `properties-location` element. Then reference the template inside a DAO or repository class—in this case, `PigPasswordRepository`.

Example 12-28. Configuring a PigTemplate

```
<pig-factory properties-location="pig-server.properties"/>

<pig-template/>

<beans:bean id="passwordRepository"
            class="com.oreilly.springdata.hadoop.pig.PigPasswordRepository">
  <beans:constructor-arg ref="pigTemplate"/>
</beans:bean>
```

The `PigPasswordRepository` (Example 12-29) lets you pass in the input file at runtime. The method `processPasswordFiles` shows you one way to programmatically process multiple files in Java. For example, you may want to select a group of input files based on a complex set of criteria that cannot be specified inside Pig Latin or a Pig user-defined function. Note that the `PigTemplate` class implements the `PigOperations` interface. This is a common implementation style of Spring template classes since it facilitates unit testing, as interfaces can be easily mocked or stubbed.

Example 12-29. Pig-based PasswordRepository

```
public class PigPasswordRepository implements PasswordRepository {

  private PigOperations pigOperations;
  private String pigScript = "classpath:password-analysis.pig";

  // constructor and setters omitted

  @Override
  public void processPasswordFile(String inputFile) {
```

```
    Assert.notNull(inputFile);
    String outputDir =
      PathUtils.format("/data/password-repo/output/%1$tY/%1$tm/%1$td/%1$tH/%1$tM/%1$tS");
    Properties scriptParameters = new Properties();
    scriptParameters.put("inputDir", inputFile);
    scriptParameters.put("outputDir", outputDir);
    pigOperations.executeScript(pigScript, scriptParameters);
  }

  @Override
  public void processPasswordFiles(Collection<String> inputFiles) {
    for (String inputFile : inputFiles) {
      processPasswordFile(inputFile);
    }
  }
}
```

The pig script *password-analysis.pig* is loaded via Spring's resource abstraction, which in this case is loaded from the classpath. To run an application that uses the `PigPasswordRepository`, use the commands in Example 12-30.

Example 12-30. Building and running the Pig scripting example that uses PigPasswordRepository

```
$ cd hadoop/pig
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/pigAppWithRepository
```

The essential pieces of code that are executed by this application are shown in Example 12-31.

Example 12-31. Using the PigPasswordRepository

```
PasswordRepository repo = context.getBean(PigPasswordRepository.class);
repo.processPasswordFile("/data/passwd/input")

Collection<String> files = new ArrayList<String>();
files.add("/data/passwd/input");
files.add("/data/passwd/input2");
repo.processPasswordFiles(files);
```

Calling Pig Scripts Inside Spring Integration Data Pipelines

To run a Pig Latin script inside of a Spring Integration data pipeline, we can reference the `PigPasswordRepository` in a Spring Integration service activator definition (Example 12-32).

Example 12-32. Invoking a Pig script within a Spring Integration data pipeline

```
<bean id="passwordService" class="com.oreilly.springdata.hadoop.pig.PasswordService">
  <constructor-arg ref="passwordRepository" />
</bean>

<int:service-activator input-channel="exampleChannel" ref="passwordService" />
```

Whether the service activator is executed asynchronously or synchronously depends on the type of input channel used. If it is a `DirectChannel` (the default), then it will be executed synchronously; if it is an `ExecutorChannel`, it will be executed asynchronously, delegating the execution to a `TaskExecutor`. The service activator class, `PasswordService`, is shown in Example 12-33.

Example 12-33. Spring Integration service activator to execute a Pig analysis job

```
public class PasswordService {

  private PasswordRepository passwordRepository;

  // constructor omitted

  @ServiceActivator
  public void process(@Header("hdfs_path") String inputDir) {
    passwordRepository.processPasswordFile(inputDir);
  }
}
```

The process method's argument will be taken from the header of the Spring Integration message. The value of the method argument is the value associated with the key `hdfs_path` in the message header. This header value is populated by a `MessageHandler` implementation, such as the `FsShellWritingMessageHandler` used previously, and needs to be called before the service activator in the data processing pipeline.

The examples in this section show that there is a steady progression from creating a simple Pig-based application that runs a script, to executing scripts with runtime parameter substitution, to executing scripts within a Spring Integration data pipeline. The section "Hadoop Workflows" on page 238 will show you how to orchestrate the execution of a Pig script inside a larger collection of steps using Spring Batch.

Apache Logfile Analysis Using Pig

Next, we will perform a simple analysis on an Apache HTTPD logfile and show the use of a custom loader for Apache logfiles. As you can see in Example 12-34, the structure of the configuration to run this analysis is similar to the one used previously to analyze the `password` file.

Example 12-34. Configuration to analyze an Apache HTTD logfile using Pig

```
<context:property-placeholder location="hadoop.properties,pig-analysis.properties"/>

<configuration>
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>

<pig-factory/>

<script id="hdfsScript" location="copy-files.groovy">
```

```
  <property name="localSourceFile" value="${pig.localSourceFile}"/>
  <property name="inputDir" value="${pig.inputPath}"/>
  <property name="outputDir" value="${pig.outputPath}"/>
</script>
```

The *copy-files.groovy* script is responsible for copying a sample logfile into HDFS, removing the content of the output path.

The Pig script generates a file that contains the cumulative number of hits for each URL and is intended to be a starting point for a more comprehensive analysis. The script also extracts minimum and maximum hit numbers and a simple table that can be used to show the distribution of hits in a simple chart. Pig makes it easy to filter and preprocess the data on a variety of criteria—for example, retain only GET requests that were successful and remove GET requests for images. The Pig script and the `PigRunner` configuration that will run the script when the Spring `ApplicationContext` starts are shown in Example 12-35 and Example 12-36, respectively.

Example 12-35. Pig script for basic Apache HTTPD log analysis

```
REGISTER $piggybanklib;
DEFINE LogLoader org.apache.pig.piggybank.storage.apachelog.CombinedLogLoader();
logs = LOAD '$inputPath' USING LogLoader AS (remoteHost, remoteLogname, user, time, \
       method, uri, proto, status, bytes, referer, userAgent);

-- determine popular URLs (for caching purposes for example)
byUri = ORDER logs BY uri;
byUri = GROUP logs BY uri;

uriHits = FOREACH byUri GENERATE group AS uri, COUNT(logs.uri) AS numHits;
STORE uriHits into '$outputPath/pig_uri_hits';

-- create histogram data for charting, view book sample code for details
```

Example 12-36. PigRunner configuration to analyze Apache HTTPD files

```
<pig-runner id="pigRunner"
            pre-action="hdfsScript"
            run-at-startup="true" >
  <script location="apache-log-simple.pig">
    <arguments>
      piggybanklib=${pig.piggybanklib}
      inputPath=${pig.inputPath}
      outputPath=${pig.outputPath}
    </arguments>
  </script>
</pig-runner>
```

The arguments to the Pig script specify the location of a jar file that contains the custom loader to read and parse Apache logfiles and the location of the input and output paths. To parse the Apache HTTPD logfiles, we will use a custom loader provided as part of the Pig distribution. It is distributed as source code as part of the Piggybank project. The compiled Piggybank jar file is provided in the sample application's *lib* directory.

Using HBase

HBase is a distributed column-oriented database. It models data as tables, which are then stored in HDFS and can scale to support tables with billions of rows and millions of columns. Like Hadoop's HDFS and MapReduce, HBase is modeled on technology developed at Google. In the case of HBase, it is Google's BigTable technology, which was described in a research paper in 2006. Unlike MapReduce, HBase provides near real-time key-based access to data, and therefore can be used in interactive, non-batch-based applications.

The HBase data model consists of a table identified by a unique key with associated columns. These columns are grouped together into column families so that data that is often accessed together can be stored together on disk to increase I/O performance. The data stored in a column is a collection of key/value pairs, not a single value as is commonly the case in a relational database. A schema is used to describe the column families and needs to be defined in advance, but the collection of key/value pairs stored as values does not. This gives the system a great amount of flexibility to evolve. There are many more details to the data model, which you must thoroughly understand in order to use HBase effectively. The book *HBase: The Definitive Guide* (O'Reilly) is an excellent reference to HBase and goes into great detail about the data model, architecture, API, and administration of HBase.

Spring for Apache Hadoop provides some basic, but quite handy, support for developing HBase applications, allowing you to easily configure your connection to HBase and provide thread-safe data access to HBase tables, as well as a lightweight object-to-column data mapping functionality.

Hello World

To install HBase, download it from the main Hive website. After installing the distribution, you start the HBase server by executing the *start-hbase.sh* script in the *bin* directory. As with Pig and Hive, HBase comes with an interactive console, which you can start by executing the command `hbase shell` in the *bin* directory.

Once inside the interactive console, you can start to create tables, define column families, and add rows of data to a specific column family. Example 12-37 demonstrates creating a user table with two column families, inserting some sample data, retrieving data by key, and deleting a row.

Example 12-37. Using the HBase interactive console

```
$ ./bin/hbase shell
> create 'users', { NAME => 'cfInfo'}, { NAME => 'cfStatus' }
> put 'users', 'row-1', 'cfInfo:qUser', 'user1'
> put 'users', 'row-1', 'cfInfo:qEmail', 'user1@yahoo.com'
> put 'users', 'row-1', 'cfInfo:qPassword', 'user1pwd'
> put 'users', 'row-1', 'cfStatus:qEmailValidated', 'true'
```

```
> scan 'users'
ROW                   COLUMN+CELL
 row-1                column=cfInfo:qEmail, timestamp=1346326115599, value=user1
                      @yahoo.com
 row-1                column=cfInfo:qPassword, timestamp=1346326128125, value=us
                      er1pwd
 row-1                column=cfInfo:qUser, timestamp=1346326078830, value=user1
 row-1                column=cfStatus:qEmailValidated, timestamp=1346326146784,
                      value=true
1 row(s) in 0.0520 seconds
> get 'users', 'row-1'
COLUMN                CELL
 cfInfo:qEmail        timestamp=1346326115599, value=user1@yahoo.com
 cfInfo:qPassword     timestamp=1346326128125, value=user1pwd
 cfInfo:qUser         timestamp=1346326078830, value=user1
 cfStatus:qEmailValid timestamp=1346326146784, value=true
 ated
4 row(s) in 0.0120 seconds

> deleteall 'users', 'row-1'
```

The two column families created are named `cfInfo` and `cfStatus`. The key names, called *qualifiers* in HBase, that are stored in the `cfInfo` column family are the username, email, and password. The `cfStatus` column family stores other information that we do not frequently need to access, along with the data stored in the `cfInfo` column family. As an example, we place the status of the email address validation process in the `cfStatus` column family, but other data—such as whether the user has participated in any online surveys—is also a candidate for inclusion. The `deleteall` command deletes all data for the specified table and row.

Using the HBase Java Client

There are many client API options to interact with HBase. The Java client is what we will use in this section but REST, Thrift, and Avro clients are also available. The `HTable` class is the main way in Java to interact with HBase. It allows you to put data into a table using a `Put` class, get data by key using a `Get` class, and delete data using a `Delete` class. You query that data using a `Scan` class, which lets you specify key ranges as well as filter criteria. Example 12-38 puts a row of user data under the key `user1` into the user table from the previous section.

Example 12-38. HBase Put API

```
Configuration configuration = new Configuration(); // Hadoop configuration object
HTable table = new HTable(configuration, "users");

Put p = new Put(Bytes.toBytes("user1"));
p.add(Bytes.toBytes("cfInfo"), Bytes.toBytes("qUser"), Bytes.toBytes("user1"));
p.add(Bytes.toBytes("cfInfo"), Bytes.toBytes("qEmail"), Bytes.toBytes("user1@yahoo.com"));
p.add(Bytes.toBytes("cfInfo"), Bytes.toBytes("qPassword"), Bytes.toBytes("user1pwd"));
table.put(p);
```

The HBase API requires that you work with the data as byte arrays and not other primitive types. The `HTable` class is also not thread safe, and requires you to carefully manage the underlying resources it uses and catch HBase-specific exceptions. Spring's `HBaseTemplate` class provides a higher-level abstraction for interacting with HBase. As with other Spring template classes, it is thread-safe once created and provides exception translation into Spring's portable data access exception hierarchy. Similar to `JdbcTemplate`, it provides several callback interfaces, such as `TableCallback`, `RowMapper`, and `ResultsExtractor`, that let you encapsulate commonly used functionality, such as mapping HBase result objects to POJOs.

The `TableCallback` callback interface provides the foundation for the functionality of `HBaseTemplate`. It performs the table lookup, applies configuration settings (such as when to flush data), closes the table, and translates any thrown exceptions into Spring's DAO exception hierarchy. The `RowMapper` callback interface is used to map one row from the HBase query `ResultScanner` into a POJO. `HBaseTemplate` has several overloaded `find` methods that take additional criteria and that automatically loop over HBase's `ResultScanner` "result set" object, converting each row to a POJO, and return a list of mapped objects. See the Javadoc API for more details. For more control over the mapping process—for example, when one row does not directly map onto one POJO—the `ResultsExtractor` interface hands you the `ResultScanner` object so you can perform the iterative result set processing yourself.

To create and configure the `HBaseTemplate`, create a `HBaseConfiguration` object and pass it to `HBaseTemplate`. Configuring a `HBaseTemplate` using Spring's Hadoop XML namespace is demonstrated in Example 12-39, but it is also easy to achieve programmatically in pure Java code.

Example 12-39. Configuring an HBaseTemplate

```
<configuration>
  fs.default.name=hdfs://localhost:9000
</configuration>

<hbase-configuration configuration-ref="hadoopConfiguration" />

<beans:bean id="hbaseTemplate" class="org.springframework.data.hadoop.hbase.HbaseTemplate">
  <beans:property name="configuration" ref="hbaseConfiguration" />
</beans:bean>
```

A `HBaseTemplate`-based `UserRepository` class that finds all users and also adds users to HBase is shown in Example 12-40.

Example 12-40. HBaseTemplate-based UserRepository class

```
@Repository
public class UserRepository {

  public static final byte[] CF_INFO = Bytes.toBytes("cfInfo");

  private HbaseTemplate hbaseTemplate;
```

```
  private String tableName = "users";
  private byte[] qUser = Bytes.toBytes("user");
  private byte[] qEmail = Bytes.toBytes("email");
  private byte[] qPassword = Bytes.toBytes("password");

  // constructor omitted

  public List<User> findAll() {
    return hbaseTemplate.find(tableName, "cfInfo", new RowMapper<User>() {
      @Override
      public User mapRow(Result result, int rowNum) throws Exception {
        return new User(Bytes.toString(result.getValue(CF_INFO, qUser)),
                        Bytes.toString(result.getValue(CF_INFO, qEmail)),
                        Bytes.toString(result.getValue(CF_INFO, qPassword)));
      }
    });
  }

  public User save(final String userName, final String email, final String password) {
    return hbaseTemplate.execute(tableName, new TableCallback<User>() {
      public User doInTable(HTable table) throws Throwable {
        User user = new User(userName, email, password);
        Put p = new Put(Bytes.toBytes(user.getName()));
        p.add(CF_INFO, qUser, Bytes.toBytes(user.getName()));
        p.add(CF_INFO, qEmail, Bytes.toBytes(user.getEmail()));
        p.add(CF_INFO, qPassword, Bytes.toBytes(user.getPassword()));
        table.put(p);
        return user;
      }
    });
  }
}
```

In this example, we used an anonymous inner class to implement the `TableCallback` and `RowMapper` interfaces, but creating standalone classes is a common implementation strategy that lets you reuse mapping logic across various parts of your application. While you can develop far more functionality with HBase to make it as feature-rich as Spring's MongoDB support, we've seen that the basic plumbing for interacting with HBase available with Spring Hadoop at the time of this writing simplifies HBase application development. In addition, HBase allows for a consistent configuration and programming model that you can further use and extend across Spring Data and other Spring-related projects.

CHAPTER 13

Creating Big Data Pipelines with Spring Batch and Spring Integration

The goal of Spring for Apache Hadoop is to simplify the development of Hadoop applications. Hadoop applications involve much more than just executing a single MapReduce job and moving a few files into and out of HDFS as in the wordcount example. There is a wide range of functionality needed to create a real-world Hadoop application. This includes collecting event-driven data, writing data analysis jobs using programming languages such as Pig, scheduling, chaining together multiple analysis jobs, and moving large amounts of data between HDFS and other systems such as databases and traditional filesystems.

Spring Integration provides the foundation to coordinate event-driven activities—for example, the shipping of logfiles, processing of event streams, real-time analysis, or triggering the execution of batch data analysis jobs. Spring Batch provides the framework to coordinate coarse-grained steps in a workflow, both Hadoop-based steps and those outside of Hadoop. Spring Batch also provides efficient data processing capabilities to move data into and out of HDFS from diverse sources such as flat files, relational databases, or NoSQL databases.

Spring for Apache Hadoop in conjunction with Spring Integration and Spring Batch provides a comprehensive and consistent programming model that can be used to implement Hadoop applications that span this wide range of functionality. Another product, Splunk, also requires a wide range of functionality to create real-world big data pipeline solutions. Spring's support for Splunk helps you to create complex Splunk applications and opens the door for solutions that mix these two technologies.

Collecting and Loading Data into HDFS

The examples demonstrated until now have relied on a fixed set of data files existing in a local directory that get copied into HDFS. In practice, files that are to be loaded into HDFS are continuously generated by another process, such as a web server. The

contents of a local directory get filled up with rolled-over logfiles, usually following a naming convention such as *myapp-timestamp.log*. It is also common that logfiles are being continuously created on remote machines, such as a web farm, and need to be transferred to a separate machine and loaded into HDFS. We can implement these use cases by using Spring Integration in combination with Spring for Apache Hadoop.

In this section, we will provide a brief introduction to Spring Integration and then implement an application for each of the use cases just described. In addition, we will show how Spring Integration can be used to process and load into HDFS data that comes from an event stream. Lastly, we will show the features available in Spring Integration that enable rich runtime management of these applications through JMX (Java management extensions) and over HTTP.

An Introduction to Spring Integration

Spring Integration is an open source Apache 2.0 licensed project, started in 2007, that supports writing applications based on established enterprise integration patterns. These patterns provide the key building blocks to develop integration applications that tie new and existing system together. The patterns are based upon a messaging model in which messages are exchanged within an application as well as between external systems. Adopting a messaging model brings many benefits, such as the logical decoupling between components as well as physical decoupling; the consumer of messages does not need to be directly aware of the producer. This decoupling makes it easier to build integration applications, as they can be developed by assembling individual building blocks together. The messaging model also makes it easier to test the application, since individual blocks can be tested first in isolation from other components. This allows bugs to be found earlier in the development process rather than later during distributed system testing, where tracking down the root cause of a failure can be very difficult. The key building blocks of a Spring Integration application and how they relate to each other is shown in Figure 13-1.

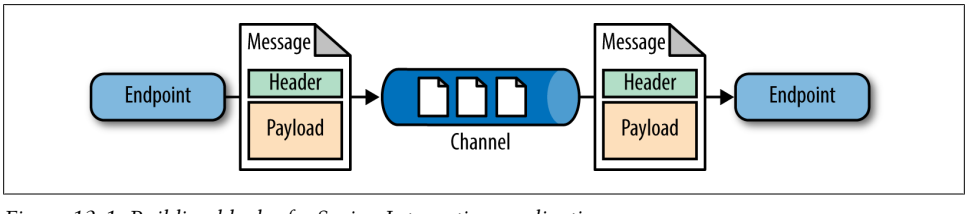

Figure 13-1. Building block of a Spring Integration application

Endpoints are producers or consumers of messages that are connected through channels. Messages are a simple data structure that contains key/value pairs in a header and an arbitrary object type for the payload. Endpoints can be adapters that communicate with external systems such as email, FTP, TCP, JMS, RabbitMQ, or syslog, but can also be operations that act on a message as it moves from one channel to another.

Common messaging operations that are supported in Spring Integration are routing to one or more channels based on the headers of message, transforming the payload from a string to a rich data type, and filtering messages so that only those that pass the filter criteria are passed along to a downstream channel. Figure 13-2 is an example taken from a joint Spring/C24 project in the financial services industry that shows the type of data processing pipelines that can be created with Spring Integration.

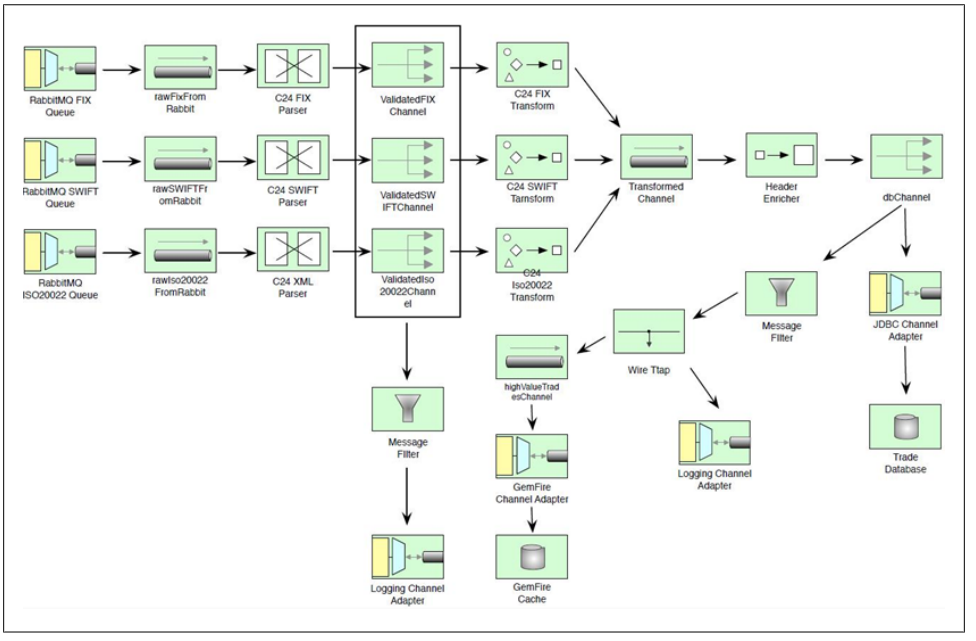

Figure 13-2. A Spring Integration processing pipeline

This diagram shows financial trade messages being received on the left via three RabbitMQ adapters that correspond to three external sources of trade data. The messages are then parsed, validated, and transformed into a canonical data format. Note that this format is not required to be XML and is often a POJO. The message header is then enriched, and the trade is stored into a relational database and also passed into a filter. The filter selects only high-value trades that are subsequently placed into a GemFire-based data grid where real-time processing can occur. We can define this processing pipeline declaratively using XML or Scala, but while most of the application can be declaratively configured, any components that you may need to write are POJOs that can be easily unit-tested.

In addition to endpoints, channels, and messages, another key component of Spring Integration is its management functionality. You can easily expose all components in a data pipeline via JMX, where you can perform operations such as stopping and starting adapters. The control bus component allows you to send in small fragments of code—for example, using Groovy—that can take complex actions to modify the state

of the system, such as changing filter criteria or starting and stopping adapters. The control bus is then connected to a middleware adapter so it can receive code to execute; HTTP and message-oriented middleware adapters are common choices.

We will not be able to dive into the inner workings of Spring Integration in great depth, nor cover every feature of the adapters that are used, but you should end up with a good feel for how you can use Spring Integration in conjunction with Spring for Apache Hadoop to create very rich data pipeline solutions. The example applications developed here contain some custom code for working with HDFS that is planned to be incorporated into the Spring Integration project. For additional information on Spring Integration, consult the project website, which contains links to extensive reference documentation, sample applications, and links to several books on Spring Integration.

Copying Logfiles

Copying logfiles into Hadoop as they are continuously generated is a common task. We will create two applications that continuously load generated logfiles into HDFS. One application will use an inbound file adapter to poll a directory for files, and the other will poll an FTP site. The outbound adapter writes to HDFS, and its implementation uses the `FsShell` class provided by Spring for Apache Hadoop, which was described in "Scripting HDFS on the JVM" on page 187. The diagram for this data pipeline is shown in Figure 13-3.

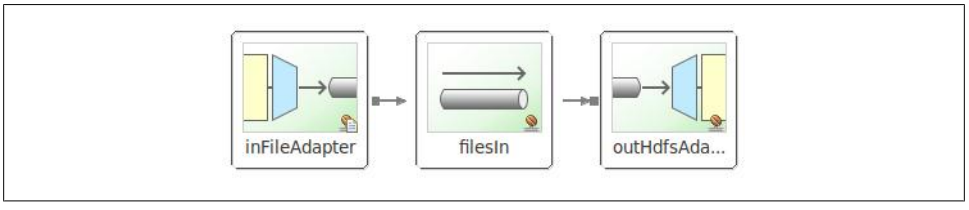

Figure 13-3. A Spring Integration data pipeline that polls a directory for files and copies them into HDFS

The file inbound adapter is configured with the directory to poll for files as well as the filename pattern that determines what files will be detected by the adapter. These values are externalized into a properties file so they can easily be changed across different runtime environments. The adapter uses a poller to check the directory since the filesystem is not an event-driven source. There are several ways you can configure the poller, but the most common are to use a fixed delay, a fixed rate, or a cron expression. In this example, we do not make use of any additional operations in the pipeline that would sit between the two adapters, but we could easily add that functionality if required. The configuration file to configure this data pipeline is shown in Example 13-1.

Example 13-1. Defining a data pipeline that polls for files in a directory and loads them into HDFS

```
<context:property-placeholder location="hadoop.properties,polling.properties"/>

<hdp:configuration id="hadoopConfiguration">fs.default.name=${hd.fs}</hdp:configuration>

<int:channel id="filesIn"/>

<file:inbound-channel-adapter id="inFileAdapter"
                              channel="filesIn"
                              directory="${polling.directory}"
                              filename-pattern="${polling.fileNamePattern}">
  <int:poller id="poller" fixed-delay="${polling.fixedDelay}"/>
</file:inbound-channel-adapter>


<int:outbound-channel-adapter id="outHdfsAdapter"
                              channel="filesIn"
                              ref="fsShellWritingMessagingHandler" >

<bean id="fsShellWritingMessagingHandler"
      class="com.oreilly.springdata.hadoop.filepolling.FsShellWritingMessageHandler">
  <constructor-arg value="${polling.destinationHdfsDirectory}"/>
  <constructor-arg ref="hadoopConfiguration"/>
</bean>
```

The relevant configuration parameters for the pipeline are externalized in the `polling.properties` file, as shown in Example 13-2.

Example 13-2. The externalized properties for polling a directory and loading them into HDFS

```
polling.directory=/opt/application/logs
polling.fixedDelay=5000
polling.fileNamePattern=*.txt
polling.destinationHdfsDirectory=/data/application/logs
```

This configuration will poll the directory */opt/application/logs* every five seconds and look for files that match the pattern **.txt*. By default, duplicate files are prevented when we specify a `filename-pattern`; the state is kept in memory. A future enhancement of the file adapter is to persistently store this application state. The `FsShellWritingMessageHandler` class is responsible for copying the file into HDFS using `FsShell`'s `copyFromLocal` method. If you want to remove the files from the polling directory after the transfer, then you set the property `deleteSourceFiles` on `FsShellWritingMessageHandler` to `true`. You can also lock files to prevent them from being picked up concurrently if more than one process is reading from the same directory. See the Spring Integration reference guide for more information.

To build and run this application, use the commands shown in Example 13-3.

Example 13-3. Command to build and run the file polling example

```
$ cd hadoop/file-polling
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/filepolling
```

The relevant parts of the output are shown in Example 13-4.

Example 13-4. Output from running the file polling example

```
03:48:44.187 [main] INFO
  c.o.s.hadoop.filepolling.FilePolling - File Polling Application Running
03:48:44.191 [task-scheduler-1] DEBUG o.s.i.file.FileReadingMessageSource - \
Added to queue: [/opt/application/logs/file_1.txt]
03:48:44.215 [task-scheduler-1] INFO  o.s.i.file.FileReadingMessageSource - \
Created message: [[Payload=/opt/application/logs/file_1.txt]
03:48:44.215 [task-scheduler-1] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Poll resulted in Message: [Payload=/opt/application/logs/file_1.txt]
03:48:44.215 [task-scheduler-1] DEBUG o.s.i.channel.DirectChannel - \
preSend on channel 'filesIn', message: [Payload=/opt/application/logs/file_1.txt]
03:48:44.310 [task-scheduler-1] INFO  c.o.s.h.f.FsShellWritingMessageHandler - \
sourceFile = /opt/application/logs/file_1.txt
03:48:44.310 [task-scheduler-1] INFO  c.o.s.h.f.FsShellWritingMessageHandler - \
resultFile = /data/application/logs/file_1.txt
03:48:44.462 [task-scheduler-1] DEBUG o.s.i.channel.DirectChannel - \
postSend (sent=true) on channel 'filesIn', \
message: [Payload=/opt/application/logs/file_1.txt]
03:48:49.465 [task-scheduler-2] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Poll resulted in Message: null
03:48:49.465 [task-scheduler-2] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Received no Message during the poll, returning 'false'
03:48:54.466 [task-scheduler-1] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Poll resulted in Message: null
03:48:54.467 [task-scheduler-1] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Received no Message during the poll, returning 'false'
```

In this log, we can see that the first time around the poller detects the one file that was in the directory and then afterward considers it processed, so the file inbound adapter does not process it a second time. There are additional options in `FsShellWritingMessageHandler` to enable the generation of an additional directory path that contains an embedded date or a UUID (universally unique identifier). To enable the output to have an additional dated directory path using the default path format (*year/month/day/hour/minute/second*), set the property `generateDestinationDirectory` to `true`. Setting `generateDestinationDirectory` to `true` would result in the file written into HDFS, as shown in Example 13-5.

Example 13-5. Partial output from running the file polling example with generateDestinationDirectory set to true

```
03:48:44.187 [main] INFO  c.o.s.hadoop.filepolling.FilePolling - \
File Polling Application Running
...
04:02:32.843 [task-scheduler-1] INFO  c.o.s.h.f.FsShellWritingMessageHandler - \
```

```
            sourceFile = /opt/application/logs/file_1.txt
04:02:32.843 [task-scheduler-1] INFO  c.o.s.h.f.FsShellWritingMessageHandler - \
            resultFile = /data/application/logs/2012/08/09/04/02/32/file_1.txt
```

Another way to move files into HDFS is to collect them via FTP from remote machines, as illustrated in Figure 13-4.

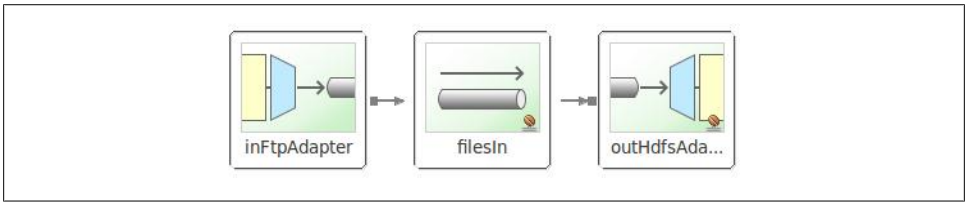

Figure 13-4. A Spring Integration data pipeline that polls an FTP site for files and copies them into HDFS

The configuration in Example 13-6 is similar to the one for file polling, only the configuration of the inbound adapter is changed.

Example 13-6. Defining a data pipeline that polls for files on an FTP site and loads them into HDFS

```
<context:property-placeholder location="ftp.properties,hadoop.properties"/>

<hdp:configuration>fs.default.name=${hd.fs}</hdp:configuration>

<bean id="ftpClientFactory"
      class="org.springframework.integration.ftp.session.DefaultFtpSessionFactory">
  <property name="host" value="${ftp.host}"/>
  <property name="port" value="${ftp.port}"/>
  <property name="username" value="${ftp.username}"/>
  <property name="password" value="${ftp.password}"/>
</bean>

<int:channel id="filesIn"/>

<int-ftp:inbound-channel-adapter id="inFtpAdapter"
    channel="filesIn"
    cache-sessions="false"
    session-factory="ftpClientFactory"
    filename-pattern="*.txt"
    auto-create-local-directory="true"
    delete-remote-files="false"
    remote-directory="${ftp.remoteDirectory}"
    local-directory="${ftp.localDirectory}">
  <int:poller fixed-rate="5000"/>
</int-ftp:inbound-channel-adapter>

<int:outbound-channel-adapter id="outHdfsAdapter"
    channel="filesIn" ref="fsShellWritingMessagingHandler"/>

<bean id="fsShellWritingMessagingHandler"
      class="com.oreilly.springdata.hadoop.ftp.FsShellWritingMessageHandler">
```

```
  <constructor-arg value="${ftp.destinationHdfsDirectory}"/>
  <constructor-arg ref="hadoopConfiguration"/>
</bean>
```

You can build and run this application using the commands shown in Example 13-7.

Example 13-7. Command to build and run the file polling example

```
$ cd hadoop/ftp
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/ftp
```

The configuration assumes there is a *testuser* account on the FTP host machine. Once you place a file in the outgoing FTP directory, you will see the data pipeline in action, copying the file to a local directory and then copying it into HDFS.

Event Streams

Streams are another common source of data that you might want to store into HDFS and optionally perform real-time analysis as it flows into the system. To meet this need, Spring Integration provides several inbound adapters that we can use to process streams of data. Once inside a Spring Integration, the data can be passed through a processing chain and stored into HDFS. The pipeline can also take parts of the stream and write data to other databases, both relational and NoSQL, in addition to forwarding the stream to other systems using one of the many outbound adapters. Figure 13-2 showed one example of this type of data pipeline. Next, we will use the TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) inbound adapters to consume data produced by syslog and then write the data into HDFS.

The configuration that sets up a TCP-syslog-to-HDFS processing chain is shown in Example 13-8.

Example 13-8. Defining a data pipeline that receives syslog data over TCP and loads it into HDFS

```
<context:property-placeholder location="hadoop.properties, syslog.properties"/>

<hdp:configuration register-url-handler="false">
  fs.default.name=${hd.fs}
</hdp:configuration>

<hdp:file-system id="hadoopFs"/>

<int-ip:tcp-connection-factory id="syslogListener"
                               type="server"
                               port="${syslog.tcp.port}"
                               deserializer="lfDeserializer"/>

<bean id="lfDeserializer"
      class="com.oreilly.springdata.integration.ip.syslog.ByteArrayLfSerializer"/>
```

```
<int-ip:tcp-inbound-channel-adapter id="tcpAdapter"
                                    channel="syslogChannel"
                                    connection-factory="syslogListener"/>

<!-- processing chain -->
<int:chain input-channel="syslogChannel">
  <int:transformer ref="sysLogToMapTransformer"/>
  <int:object-to-string-transformer/>
  <int:outbound-channel-adapter ref="hdfsWritingMessageHandler"/>
</int:chain>

<bean id="sysLogToMapTransformer"
      class="com.oreilly.springdata.integration.ip.syslog.SyslogToMapTransformer"/>

<bean id="hdfsWritingMessageHandler"
      class="com.oreilly.springdata.hadoop.streaming.HdfsWritingMessageHandler">
  <constructor-arg ref="hdfsWriterFactory"/>
</bean>

<bean id="hdfsWriterFactory"
      class="com.oreilly.springdata.hadoop.streaming.HdfsTextFileWriterFactory">
  <constructor-arg ref="hadoopFs"/>
  <property name="basePath" value="${syslog.hdfs.basePath}"/>
  <property name="baseFilename" value="${syslog.hdfs.baseFilename}"/>
  <property name="fileSuffix" value="${syslog.hdfs.fileSuffix}"/>
  <property name="rolloverThresholdInBytes"
      value="${syslog.hdfs.rolloverThresholdInBytes}"/>
</bean>
```

The relevant configuration parameters for the pipeline are externalized in the `streaming.properties` file, as shown in Example 13-9.

Example 13-9. The externalized properties for streaming data from syslog into HDFS

```
syslog.tcp.port=1514
syslog.udp.port=1513
syslog.hdfs.basePath=/data/
syslog.hdfs.baseFilename=syslog
syslog.hdfs.fileSuffix=log
syslog.hdfs.rolloverThresholdInBytes=500
```

The diagram for this data pipeline is shown in Figure 13-5.

This configuration will create a connection factory that listens for an incoming TCP connection on port 1514. The serializer segments the incoming byte stream based on the newline character in order to break up the incoming syslog stream into events. Note that this lower-level serializer configuration will be encapsulated in a syslog XML namespace in the future so as to simplify the configuration. The inbound channel adapter takes the syslog message off the TCP data stream and parses it into a byte array, which is set as the payload of the incoming message.

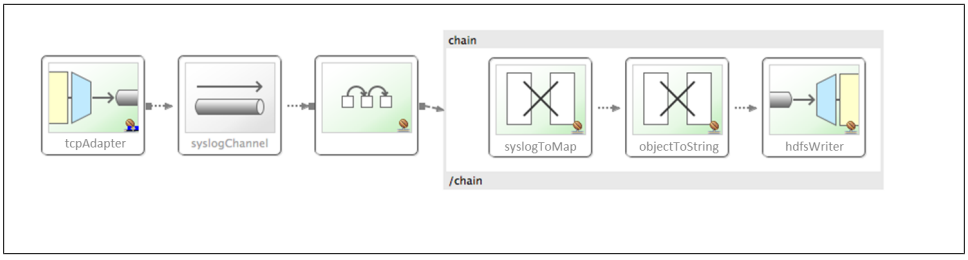

Figure 13-5. A Spring Integration data pipeline that streams data from syslog into HDFS

Spring Integration's chain component groups together a sequence of endpoints without our having to explicitly declare the channels that connect them. The first element in the chain parses the `byte[]` array and converts it to a `java.util.Map` containing the key/value pairs of the syslog message. At this stage, you could perform additional operations on the data, such as filtering, enrichment, real-time analysis, or routing to other databases. In this example, we have simply transformed the payload (now a `Map`) to a `String` using the built-in object-to-string transformer. This string is then passed into the `HdfsWritingMessageHandler` that writes the data into HDFS. `HdfsWritingMessageHandler` lets you configure the HDFS directory to write the files, the file naming policy, and the file size rollover policy. In this example, the rollover threshold was set artificially low (500 bytes versus the 10 MB default) to highlight the rollover capabilities in a simple test usage case.

To build and run this application, use the commands shown in Example 13-10.

Example 13-10. Commands to build and run the Syslog streaming example

```
$ cd hadoop/streaming
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/streaming
```

To send a test message, use the logger utility demonstrated in Example 13-11.

Example 13-11. Sending a message to syslog

```
$ logger -p local3.info -t TESTING "Test Syslog Message"
```

Since we set `HdfsWritingMessageHandler`'s `rolloverThresholdInBytes` property so low, after sending a few of these messages or just waiting for messages to come in from the operating system, you will see inside HDFS the files shown in Example 13-12.

Example 13-12. Syslog data in HDFS

```
$ hadoop dfs -ls /data
-rw-r--r-- 3 mpollack supergroup 711 2012-08-09 13:19 /data/syslog-0.log
-rw-r--r-- 3 mpollack supergroup 202 2012-08-09 13:22 /data/syslog-1.log
-rw-r--r-- 3 mpollack supergroup 240 2012-08-09 13:22 /data/syslog-2.log
-rw-r--r-- 3 mpollack supergroup 119 2012-08-09 15:04 /data/syslog-3.log
…
```

```
$ hadoop dfs -cat /data/syslog-2.log
{HOST=ubuntu, MESSAGE=Test Syslog Message, SEVERITY=6, FACILITY=19, \
 TIMESTAMP=Thu Aug 09 13:22:44 EDT 2012, TAG=TESTING}
{HOST=ubuntu, MESSAGE=Test Syslog Message, SEVERITY=6, FACILITY=19, \
 TIMESTAMP=Thu Aug 09 13:22:55 EDT 2012, TAG=TESTING}
```

To use UDP instead of TCP, remove the TCP-related definitions and add the commands shown in Example 13-13.

Example 13-13. Configuration to use UDP to consume syslog data

```
<int-ip:udp-inbound-channel-adapter id="udpAdapter"
        channel="syslogChannel" port="${syslog.udp.port}"/>
```

Event Forwarding

When you need to process a large amount of data from several different machines, it can be useful to forward the data from where it is produced to another server (as opposed to processing the data locally). The TCP inbound and outbound adapters can be paired together in an application so that they forward data from one server to another. The channel that connects the two adapters can be backed by several persistent message stores. Message stores are represented in Spring Integration by the interface `MessageStore`, and implementations are available for JDBC, Redis, MongoDB, and GemFire. Pairing inbound and outbound adapters together in an application affects the message processing flow such that the message is persisted in the message store of the producer application before the message is sent to the consumer application. The message is removed from the producer's message store once the acknowledgment from the consumer is received. The consumer sends its acknowledgment once it has successfully put the received message in its own message-store-backed channel. This configuration enables an additional level of "store and forward" guarantee via TCP normally found in messaging middleware such as JMS or RabbitMQ.

Example 13-14 is a simple demonstration of forwarding TCP traffic and using Spring's support to easily bootstrap an embedded HSQL database to serve as the message store.

Example 13-14. Store and forwarding of data across processes using TCP adapters

```
<int:channel id="dataChannel">
  <int:queue message-store="messageStore"/>
</int:channel>

<int-jdbc:message-store id="messageStore" data-source="dataSource"/>

<jdbc:embedded-database id="dataSource"/>

<int-ip:tcp-inbound-channel-adapter id="tcpInAdapter"
        channel="dataChannel" port="${syslog.tcp.in.port}"/>

<int-ip:tcp-outbound-channel-adapter id="tcpOutAdapter"
        channel="dataChannel"  port="${syslog.tcp.out.port}"/>
```

Management

Spring Integration provides two key features that let you manage data pipelines at runtime: the exporting of channels and endpoints to JMX and a control bus. Much like JMX, the control bus lets you invoke operations and view metric information related to each component, but it is more general-purpose because it allows you to run small programs inside the running application to change its state and behavior.

Exporting channels and endpoints to JMX is as simple as adding the lines of XML configuration shown in Example 13-15.

Example 13-15. Exporting channels and endpoints to JMX

```
<int-jmx:mbean-export default-domain="streaming-syslog"/>

<context:mbean-server/>
```

Running the TCP streaming example in the previous section and then starting JConsole shows the JMX metrics and operations that are available (Figure 13-6). Some examples are to start and stop the TCP adapter, and to get the min, max, and mean duration of processing in a `MessageHandler`.

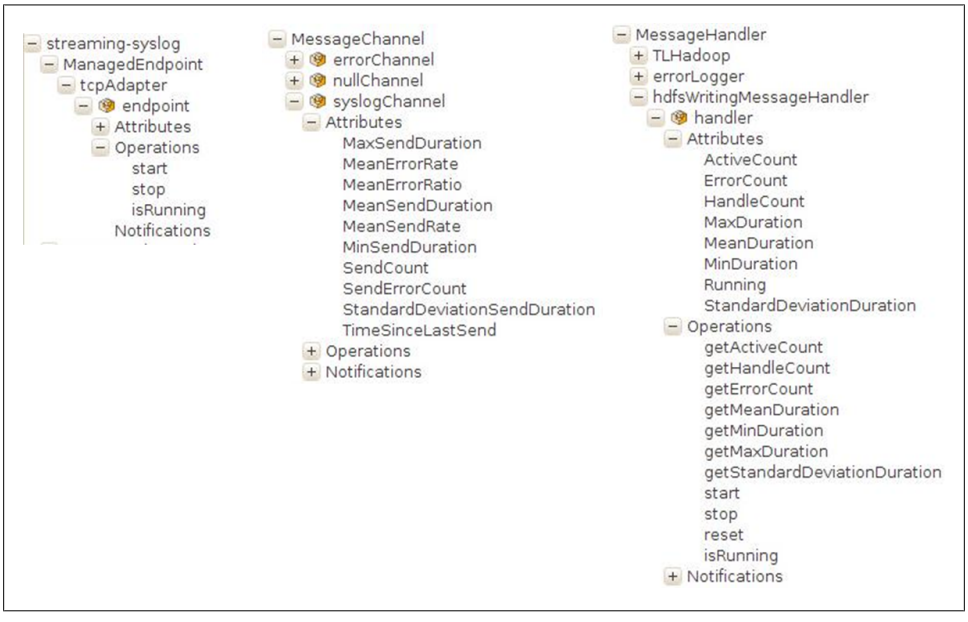

Figure 13-6. Screenshots of the JConsole JMX application showing the operations and properties available on the TcpAdapter, channels, and HdfsWritingMessageHandler

A control bus can execute Groovy scripts or Spring Expression Language (SpEL) expressions, allowing you to manipulate the state of components inside the application

programmatically. By default, Spring Integration exposes all of its components to be accessed through a control bus. The syntax for a SpEL expression that stops the TCP inbound adapter would be `@tcpAdapter.stop()`. The `@` prefix is an operator that will retrieve an object by name from the Spring `ApplicationContext`; in this case, the name is `tcpAdapter`, and the method to invoke is `stop`. A Groovy script to perform the same action would not have the `@` prefix. To declare a control bus, add the configuration shown in Example 13-16.

Example 13-16. Configuring a Groovy-based control bus

```
<int:channel id="inOperationChannel"/>

<int-groovy:control-bus input-channel="inOperationChannel"/>
```

By attaching an inbound channel adapter or gateway to the control bus's input channel, you can execute scripts remotely. It is also possible to create a Spring MVC application and have the controller send the message to the control bus's input channel, as shown in Example 13-17. This approach might be more natural if you want to provide additional web application functionality, such as security or additional views. Example 13-17 shows a Spring MVC controller that forwards the body of the incoming web request to the control bus and returns a `String`-based response.

Example 13-17. Spring MVC control to send message to the control bus

```
@Controller
public class ControlBusController {

  private @Autowired MessageChannel inOperationChannel;

  @RequestMapping("/admin")
  public @ResponseBody String simple(@RequestBody String message) {

    Message<String> operation = MessageBuilder.withPayload(message).build();
    MessagingTemplate template = new MessagingTemplate();
    Message response = template.sendAndReceive(inOperationChannel, operation);
    return response != null ? response.getPayload().toString() : null;

  }
}
```

Running the sample application again, we can interact with the control bus over HTTP using curl to query and modify the state of the inbound TCP adapter (Example 13-18).

Example 13-18. Configuring the control bus

```
$ cd streaming
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/streaming
… output omitted …
$ curl -X GET --data "tcpAdapter.isRunning()" http://localhost:8080/admin
true
$ curl -X GET --data "tcpAdapter.stop()" http://localhost:8080/admin
```

```
$ curl -X GET --data "tcpAdapter.isRunning()" http://localhost:8080/admin
false
$ curl -X GET --data "tcpAdapter.start()" http://localhost:8080/admin
$ curl -X GET --data "tcpAdapter.isRunning()" http://localhost:8080/admin
true
```

An Introduction to Spring Batch

The Spring Batch project was started in 2007 as a collaboration between SpringSource and Accenture to provide a comprehensive batch framework to support the development of robust batch applications. These batch applications require performing bulk processing of large amounts of data that are critical to the operation of a business. Spring Batch has been widely adopted and used in thousands of enterprise applications worldwide. Batch jobs have their own set of best practices and domain concepts, gathered over many years of building up Accenture's consulting business and encapsulated into the Spring Batch project. Thus, Spring Batch supports the processing of large volumes of data with features such as automatic retries after failure, skipping of records, job restarting from the point of last failure, periodic batch commits to a transactional database, reusable components (such as parsers, mappers, readers, processors, writers, and validators), and workflow definitions. As part of the Spring ecosystem, the Spring Batch project builds upon the core features of the Spring Framework, such as the use of the Spring Expression Language. Spring Batch also carries over the design philosophy of the Spring Framework, which emphasizes a POJO-based development approach and promotes the creation of maintainable, testable code.

The concept of a workflow in Spring Batch translates to a Spring Batch `Job` (not to be confused with a MapReduce job). A batch `Job` is a directed graph, each node of the graph being a processing `Step`. `Step`s can be executed sequentially or in parallel, depending on the configuration. `Job`s can be started, stopped, and restarted. Restarting jobs is possible since the progress of executed steps in a `Job` is persisted in a database via a `JobRepository`. Jobs are composable as well, so you can have a job of jobs. Figure 13-7 shows the basic components in a Spring Batch application. The `JobLauncher` is responsible for starting a job and is often triggered via a scheduler. The Spring Framework provides basic scheduling functionality as well as integration with Quartz; however, often enterprises use their own scheduling software such as Tivoli or Control-M. Other options to launch a job are through a RESTful administration API, a web application, or programmatically in response to an external event. In the latter case, using the Spring Integration project and its many channel adapters for communicating with external systems is a common choice. You can read more about combining Spring Integration and Spring Batch in the book *Spring Integration in Action* [Fisher12].For additional information on Spring Batch, consult the project website, which contains links to extensive reference documentation, sample applications, and links to several books.

The processing performed in a step is broken down into three stages—an `ItemReader`, `ItemProcessor`, and `ItemWriter`—as shown in Figure 13-8. Using an `ItemProcessor` is optional.

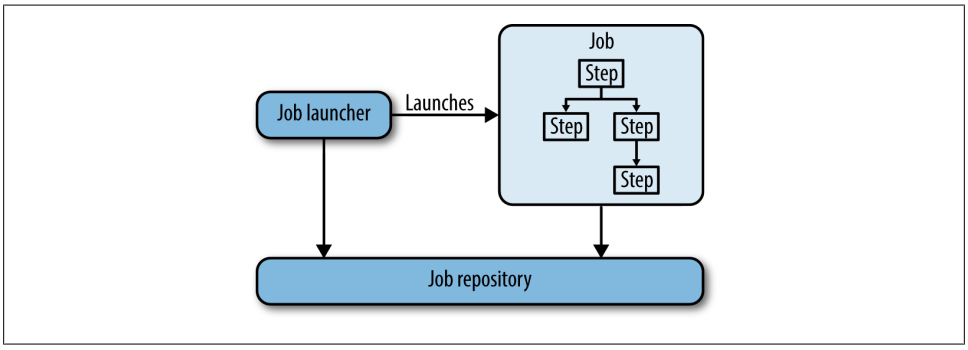

Figure 13-7. Spring Batch overview

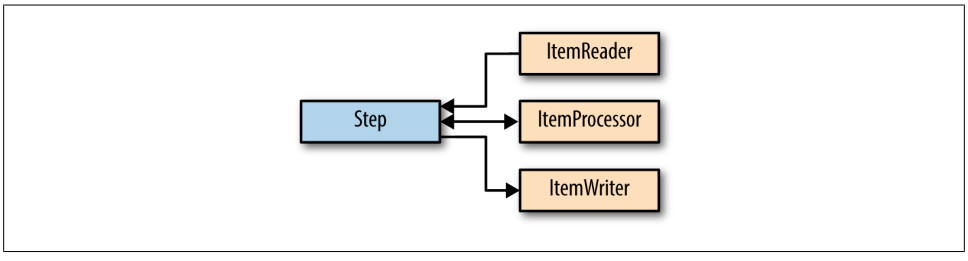

Figure 13-8. Spring Batch step components

One of the primary use cases for Spring Batch is to process the contents of large files and load the data into a relational database. In this case, a `FlatFileItemReader` and a `JdbcItemWriter` are used along with custom logic either configured declaratively or coded directly in an `ItemProcessor`. To increase the performance, the `Step` is "chunked," meaning that chunks of data, say 100 rows, are aggregated together and then passed to the `ItemWriter`. This allows us to efficiently process a group of records by using the batch APIs available in many databases to insert data. A snippet of configuration using the Spring Batch XML namespace that reads from a file and writes to a database is shown in Example 13-19. In subsequent sections, we will dive into the configuration of readers, writers, and processors.

Example 13-19. Configuring a Spring Batch step to process flat file data and copy it into a database

```
<step id="simpleStep">
  <tasklet>
    <chunk reader="flatFileItemReader" processor="itemProcessor" writer="jdbcItemWriter"
           commit-interval="100"/>
    </chunk>
```

```
  </tasklet>
</step>
```

Additional features available in Spring Batch allow you to scale up and out the job execution in order to handle the requirements of high-volume and high-performance batch jobs. For more information on these topics, refer to the Spring Batch reference guide or one of the Spring Batch books ([CoTeGreBa11], [Minella11]).

It is important to note that the execution model of a Spring Batch application takes place outside of the Hadoop cluster. Spring Batch applications can scale up by using different threads to concurrently process different files or scale out using Spring Batch's own master-slave remote partitioning model. In practice, scaling up with threads has been sufficient to meet the performance requirements of most users. You should try this option as a first strategy to scale before using remote partitioning. Another execution model that will be developed in the future is to run a Spring Batch job inside the Hadoop cluster itself, taking advantage of the cluster's resource management functionality to scale out processing across the nodes of the cluster, taking into account the locality of data stored in HDFS. Both models have their advantages, and performance isn't the only criteria to decide which execution model to use. Executing a batch job outside of the Hadoop cluster often enables easier data movement between different systems and multiple Hadoop clusters.

In the following sections, we will use the Spring Batch framework to process and load data into HDFS from a relational database. In the section "Exporting Data from HDFS" on page 243, we will export data from HDFS into a relational database and the MongoDB document database.

Processing and Loading Data from a Database

To process and load data from a relational database to HDFS, we need to configure a Spring Batch tasklet with a `JdbcItemReader` and a `HdfsTextItemWriter`. The sample application for this section is located in *./hadoop/batch-import* and is based on the sample code that comes from the book *Spring Batch in Action*. The domain for the sample application is an online store that needs to maintain a catalog of the products it sells. We have modified the example only slightly to write to HDFS instead of a flat file system. The configuration of the Spring Batch tasklet is shown in Example 13-20.

Example 13-20. Configuring a Spring Batch step to read from a database and write to HDFS

```
<job id="importProducts" xmlns="http://www.springframework.org/schema/batch">
  <step id="readWriteProducts">
    <tasklet>
      <chunk reader="jdbcReader" writer="hdfsWriter" commit-interval="100"/>
   </tasklet>
  </step>
</job>

<bean id="jdbcReader" class="org.springframework.batch.item.database.JdbcCursorItemReader">
```

```
  <property name="dataSource" ref="dataSource"/>
  <property name="sql" value="select id, name, description, price from product"/>
  <property name="rowMapper" ref="productRowMapper"/>
</bean>

<bean id="productRowMapper" class="com.oreilly.springdata.domain.ProductRowMapper"/>
```

We configure the `JdbcCursorItemReader` with a standard JDBC `DataSource` along with the SQL statement that will select the data from the product table that will be loaded into HDFS. To start and initialize the database with sample data, run the commands shown in Example 13-21.

Example 13-21. Commands to initalize and run the H2 database for a Spring Batch application

```
$ cd hadoop/batch-import
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/start-database
```

A browser will pop up that lets you browse the contents of the database containing the product table in addition to the tables for Spring Batch used to implement the job repository.

The commit interval is set to 100, which is more than the amount of data available in this simple application, but represents a typical number to use. For each 100 records read from the database, the transaction that updates the job execution metadata will be committed to the database. This allows for a restart of the job upon a failure to pick up where it left off.

The `rowMapper` property of `JdbcCursorItemReader` is an implementation of Spring's `RowMapper` interface, which is part of Spring's JDBC feature set. The `RowMapper` interface provides a simple way to convert a JDBC `ResultSet` to a POJO when a single row in a `ResultSet` maps onto a single POJO instance. Iteration over the `ResultSet` as well as exception handling (which is normally quite verbose and error-prone) is encapsulated by Spring, letting you focus on writing only the required mapping code. The `ProductRowMapper` used in this application converts each row of the `ResultSet` object to a `Product` Java object and is shown in Example 13-22. The `Product` class is a simple POJO with getters and setters that correspond to the columns selected from the product table.

Example 13-22. The ProductRowMapper that converts a row in a ResultSet to a Product object

```
public class ProductRowMapper implements RowMapper<Product> {

  public Product mapRow(ResultSet rs, int rowNum) throws SQLException {
    Product product = new Product();
    product.setId(rs.getString("id"));
    product.setName(rs.getString("name"));
    product.setDescription(rs.getString("description"));
    product.setPrice(rs.getBigDecimal("price"));
    return product;
  }
}
```

The `JdbcCursorItemReader` class relies on the streaming functionality of the underlying JDBC driver to iterate through the result set in an efficient manner. You can set the property `fetchSize` to give a hint to the driver to load only a certain amount of data into the driver that runs in the client process. The value to set the `fetchSize` to depends on the JDBC driver. For example, in the case of MySQL, the documentation suggests setting `fetchSize` to `Integer.MIN_VALUE`, a nonobvious choice for handling large result sets efficiently. Of note, Spring Batch also provides the class `JdbcPagingItemReader` as another strategy to control how much data is loaded from the database into the client process as well as the ability to load data from stored procedures.

The last part of the configuration for the application to run is the `hdfsWriter` shown in Example 13-23.

Example 13-23. The configuration of the HdfsTextItemWriter

```
<context:property-placeholder location="hadoop.properties"/>

<hdp:configuration>fs.default.name=${hd.fs}</hdp:configuration>
<hdp:file-system id="hadoopFs"/>

<bean id="hdfsWriter" class="com.oreilly.springdata.batch.item.HdfsTextItemWriter">
  <constructor-arg ref="hadoopFs"/>
  <property name="basePath" value="/import/data/products/"/>
  <property name="baseFilename" value="product"/>
  <property name="fileSuffix" value="txt"/>
  <property name="rolloverThresholdInBytes" value="100"/>
  <property name="lineAggregator">
   <bean class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>
  </property>
</bean>
```

The Hadoop configuration is as we have seen in the previous sections, but the `<hdp:file-system/>` is new. It is responsible for creating the appropriate `org.apache.hadoop.fs.FileSystem` implementation based on the Hadoop configuration. The possible options are implementations that communicate with HDFS using the standard HDFS protocol (*hdfs://*), HFTP (*hftp://*), or WebHDFS (*webhdfs://*). The `FileSystem` is used by `HdfsTextItemWriter` to write plain-text files to HDFS. The configuration of the `HdfsTextItemWriter`'s properties—`basePath`, `baseFileName`, and `fileSuffix`—will result in files being written into the */import/data/products* directory with names such as *product-0.txt* and *product-1.txt*. We set `rolloverThresholdInBytes` to a very low value for the purposes of demonstrating the rollover behavior.

`ItemWriters` often require a collaborating object, an implementation of the `LineAggregator` interface that is responsible for converting the item being processed into a string. In this example, we are using the `PassThroughFieldExtractor` provided by Spring Batch, which will delegate to the `toString()` method of the `Product` class to create the string. The `toString()` method of `Product` is a simple comma-delimited concatenation of the ID, name, description, and price values.

To run the application and import from a database to HDFS, execute the commands shown in Example 13-24.

Example 13-24. Commands to import data from a database to HDFS

```
$ cd hadoop/batch-import
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/import
```

Example 13-25 shows the resulting content in HDFS.

Example 13-25. The imported product data in HDFS

```
$ hadoop dfs -ls /import/data/products

Found 6 items
-rw-r--r-- 3 mpollack supergroup 114 2012-08-21 11:40 /import/data/products/product-0.txt
-rw-r--r-- 3 mpollack supergroup 113 2012-08-21 11:40 /import/data/products/product-1.txt
-rw-r--r-- 3 mpollack supergroup 122 2012-08-21 11:40 /import/data/products/product-2.txt
-rw-r--r-- 3 mpollack supergroup 119 2012-08-21 11:40 /import/data/products/product-3.txt
-rw-r--r-- 3 mpollack supergroup 136 2012-08-21 11:40 /import/data/products/product-4.txt
-rw-r--r-- 3 mpollack supergroup  51 2012-08-21 11:40 /import/data/products/product-5.txt

$ hadoop dfs -cat /import/data/products/

PR1...210,BlackBerry 8100 Pearl,,124.6
PR1...211,Sony Ericsson W810i,,139.45
PR1...212,Samsung MM-A900M Ace,,97.8
```

There are many other `LineAggregator` implementations available in Spring Batch that give you a great deal of declarative control over what fields are written to the file and what characters are used to delimit each field. Example 13-26 shows one such implementation.

Example 13-26. Specifying JavaBean property names to create the String written to HDFS for each product object

```
<property name="lineAggregator">
 <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
  <property name="fieldExtractor">
   <bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
    <property name="names" value="id,price,name"/>
   </bean>
  </property>
 </bean>
</property>
```

The `DelimitedLineAggregator` will use a comma to separate fields by default, and the `BeanWrapperFieldExtractor` is passed in the JavaBean property names that it will select from the `Product` object to write a line of output to HDFS.

Running the application again with this configuration of a `LineAggregator` will create files in HDFS that have the content shown in Example 13-27.

Example 13-27. The imported product data in HDFS using an alternative formatting

```
$ hadoop dfs -cat /import/data/products/products-0.txt

PR1...210,124.6,BlackBerry 8100 Pearl
PR1...211,139.45,Sony Ericsson W810i
PR1...212,97.8,Samsung MM-A900M Ace
```

Hadoop Workflows

Hadoop applications rarely consist of one MapReduce job or Hive script. Analysis logic is usually broken up into several steps that are composed together into a chain of execution to perform the complete analysis task. In the previous MapReduce and Apache weblog examples, we used `JobRunner`s, `HiveRunner`s, and `PigRunner`s to execute HDFS operations and MapReduce, Hive, or Pig jobs, but that is not a completely satisfactory solution. As the number of steps in an analysis chain increases, the flow of execution is hard to visualize and not naturally structured as a graph structure in the XML namespace. There is also no tracking of the execution steps in the analysis chain when we're using the various Runner classes. This means that if one step in the chain fails, we must restart it (manually) from the beginning, making the overall "wall clock" time for the analysis task significantly larger as well as inefficient. In this section, we will introduce extensions to the Spring Batch project that will provide structure for chaining together multiple Hadoop operations into what are loosely called *workflows*.

Spring Batch Support for Hadoop

Because Hadoop is a batch-oriented system, Spring Batch's domain concepts and workflow provide a strong foundation to structure Hadoop-based analysis tasks. To make Spring Batch "Hadoop aware," we take advantage of the fact that the processing actions that compose a Spring Batch `Step` are pluggable. The plug-in point for a `Step` is known as a `Tasklet`. Spring for Apache Hadoop provides custom `Tasklet`s for HDFS operations as well as for all types of Hadoop jobs: MapReduce, Streaming, Hive, and Pig. This allows for creating workflows as shown in Figure 13-9.

There is support in the Eclipse-based Spring Tool Suite (STS) to support the visual authoring of Spring Batch jobs. Figure 13-10 shows the equivalent diagram to Figure 13-9 inside of STS.

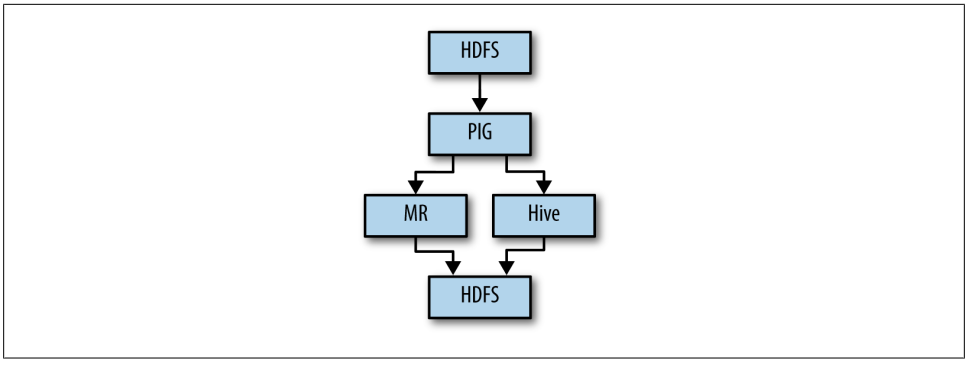

Figure 13-9. Steps in a Spring Batch application that execute Hadoop HDFS operations and run Pig, MapReduce, and Hive jobs

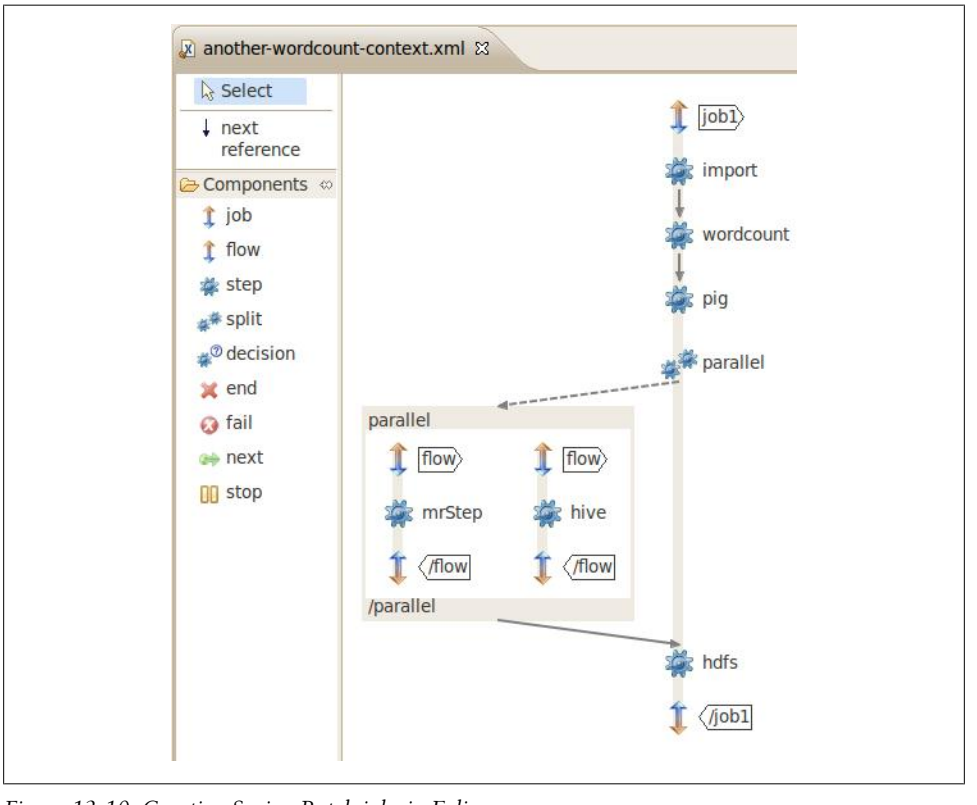

Figure 13-10. Creating Spring Batch jobs in Eclipse

The underlying XML for a Spring Batch job with a linear flow of steps has the general pattern shown in Example 13-28.

Example 13-28. Spring Batch job definition for a sequence of steps

```
<job id="job">
    <step id="stepA" next="stepB"/>
    <step id="stepB" next="stepC"/>
    <step id="stepC"/>
</job>
```

You can also make the flow conditional based on the exit status of the step. There are several well-known `ExitStatus` codes that a step returns, the most common of which are `COMPLETED` and `FAILED`. To create a conditional flow, you use the nested `next` element of the step, as shown in Example 13-29.

Example 13-29. Spring Batch job definition for a conditional sequence of steps

```
<job id="job">
    <step id="stepA">
        <next on="FAILED" to="stepC"/>
        <next on="*" to="stepB"/>
    </step>
    <step id="stepB" next="stepC"/>
    <step id="stepC"/>
</job>
```

In this example, if the exit code matches `FAILED`, the next step executed is `stepC`; otherwise, `stepB` is executed followed by `stepC`.

There is a wide range of ways to configure the flow of a Spring that we will not cover in this section. To learn more about how to configure more advanced job flows, see the reference documentation or one of the aforementioned Spring Batch books.

To configure a Hadoop-related step, you can use the XML namespace provided by Spring for Apache Hadoop. Next, we'll show how we can configure the wordcount example as a Spring Batch application, reusing the existing configuration of a MapReduce and HDFS script that were part of the standalone Hadoop application examples used previously. Then we will show how to configure other Hadoop-related steps, such as for Hive and Pig.

Wordcount as a Spring Batch Application

The wordcount example has two steps: importing data into HDFS and then running a MapReduce job. Example 13-30 shows the Spring Batch job representing the workflow using the Spring Batch XML namespace. We use the namespace prefix `batch` to distinguish the batch configuration from the Hadoop configuration.

Example 13-30. Setting up the Spring Batch job to perform HDFS and MapReduce steps

```
<batch:job id="job1">
  <batch:step id="import" next="wordcount">
    <batch:tasklet ref="scriptTasklet"/>
  </batch:step>

  <batch:step id="wordcount">
    <batch:tasklet ref="wordcountTasklet"/>
  </batch:step>
</batch:job>

<tasklet id="wordcountTasklet" job-ref="wordcountJob"/>
<script-tasklet id="scriptTasklet" script-ref="hdfsScript">

<!-- MapReduce job and HDFS script as defined in previous examples -->
<job id="wordcountJob"
     input-path="${wordcount.input.path}"
     output-path="${wordcount.output.path}"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<script id="hdfsScript" location="copy-data.groovy" >
  <property name="inputPath" value="${wordcount.input.path}"/>
  <property name="outputPath" value="${wordcount.output.path}"/>
  <property name="localResource" value="${local.data}"/>
</script>
```

It is common to parameterize a batch application by providing job parameters that are passed in when the job is launched. To change the batch job to reference these job parameters instead of ones that comes from static property files, we need to make a few changes to the configuration. We can retrieve batch job parameters using SpEL, much like how we currently reference variables using `${...}` syntax.

The syntax of SpEL is similar to Java, and expressions are generally just one line of code that gets evaluated. Instead of using the syntax `${...}` to reference a variable, use the syntax `#{...}` to evaluate an expression. To access the Spring Batch job parameters, we'd use the expression `#{jobParameters['mr.input']}`. The variable `jobParameters` is available by default when a bean is placed in Spring Batch's step scope. The configuration of the MapReduce job and HDFS script, then, looks like Example 13-31.

Example 13-31. Linking the Spring Batch Tasklets to Hadoop Job and Script components

```
<job id="wordcount-job" scope="step"
     input-path="#{jobParameters['mr.input']}"
     output-path="#{jobParameters['mr.output']}"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<script id="hdfsScript" location="copy-files.groovy" scope="step">
  <property name="localSourceFile" value="#{jobParameters['localData']}"/>
  <property name="hdfsInputDir" value="#{jobParameters['mr.input']}"/>
```

```
  <property name="hdfsOutputDir" value="#{jobParameters['mr.output']}"/>
</script>
```

The main application that runs the batch application passes in values for these parameters, but we could also set them using other ways to launch a Spring Batch application. The `CommandLineJobRunner`, administrative REST API, or the administrative web application are common choices. Example 13-32 is the main driver class for the sample application.

Example 13-32. Main application that launched a batch job

```
ApplicationContext context =
  new ClassPathXmlApplicationContext("classpath:/META-INF/spring/*-context.xml");
JobLauncher jobLauncher = context.getBean(JobLauncher.class);
Job job = context.getBean(Job.class);
jobLauncher.run(job, new JobParametersBuilder()
  .addString("mr.input", "/user/gutenberg/input/word/")
  .addString("mr.output", "/user/gutenberg/output/word/")
  .addString("localData", "./data/nietzsche-chapter-1.txt")
  .addDate("date", new Date()).toJobParameters());
```

To run the batch application, execute the commands shown in Example 13-33.

Example 13-33. Commands to run the wordcount batch application

```
$ cd hadoop/batch-wordcount
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/batch-wordcount
```

Hive and Pig Steps

To execute a Hive script as part of a Spring Batch workflow, use the Hive `Tasklet` element, as shown in Example 13-34.

Example 13-34. Configuring a Hive Tasklet

```
<job id="job1" xmlns="http://www.springframework.org/schema/batch">
  <step id="import" next="hive">
    <tasklet ref="scriptTasklet"/>
  </step>

  <step id="hive">
    <tasklet ref="hiveTasklet"/>
  </step>
</job>

<hdp:hive-client-factory host="${hive.host}" port="${hive.port}"/>

<hive-tasklet id="hiveTasklet">
   <hdp:script location="analysis.hsql"/>
</hdp:hive-tasklet>
```

To execute a Pig script as part of a Spring Batch workflow, use the Pig `Tasklet` element (Example 13-35).

Example 13-35. Configuring a Pig Tasklet

```
<job id="job1" xmlns="http://www.springframework.org/schema/batch">
  <step id="import" next="pig">
    <tasklet ref="scriptTasklet"/>
  </step>

  <step id="pig">
    <tasklet ref="pigTasklet"/>
  </step>
</job>

<pig-factory/>

<pig-tasklet id="pigTasklet">
  <script location="analysis.pig">
    <arguments>
      piggybanklib=${pig.piggybanklib}
      inputPath=${pig.inputPath}
      outputPath=${pig.outputPath}
    </arguments>
  </script>
</pig-tasklet>
```

Exporting Data from HDFS

The results from data analysis in Hadoop are often copied into structured data stores, such as a relational or NoSQL database, for presentation purposes or further analysis. One of the main use cases for Spring Batch is moving data back between files and databases and processing it along the way. In this section, we will use Spring Batch to export data from HDFS, perform some basic processing on the data, and then store the data outside of HDFS. The target data stores are a relational database and MongoDB.

From HDFS to JDBC

Moving the result data created from MapReduce jobs located in HDFS into a relational database is very common. Spring Batch provides many out-of-the-box components that you can configure to perform this activity. The sample application for this section is located in *./hadoop/batch-extract* and is based on the sample code that comes from the book *Spring Batch in Action*. The domain for the sample application is an online store that needs to maintain a catalog of the products it sells. The application as it was originally written reads product data from flat files on a local filesystem and then writes the product data into a product table in a relational database. We have modified the example to read from HDFS and also added error handling to show an additional Spring Batch feature.

To read from HDFS instead of a local filesystem, we need to register a `HdfsResourceLoader` with Spring to read data from HDFS using Spring's `Resource` abstraction. This lets us use the Spring Batch's existing `FlatFileItemReader` class, as it is based on the `Resource` abstraction. Spring's `Resource` abstraction provides a uniform way to read an `InputStream` from a variety of sources such as a URL (*http*, *ftp*), the Java `ClassPath`, or the standard filesystem. The `Resource` abstraction also supports reading from multiple source locations through the use of Ant-style regular expressions.To configure Spring to be able to read from HDFS as well as make HDFS the default resource type that will be used (e.g., *hdfs://hostname:port* versus *file://*), add the lines of XML shown in Example 13-36 to a Spring configuration file.

Example 13-36. Configuring the default resource loader to use HDFS

```
<context:property-placeholder location="hadoop.properties"/>

<hdp:configuration>fs.default.name=${hd.fs}</hdp:configuration>
<hdp:resource-loader id="hadoopResourceLoader"/>

<bean id="defaultResourceLoader"
  class="org.springframework.data.hadoop.fs.CustomResourceLoaderRegistrar">
  <property name="loader" ref="hadoopResourceLoader"/>
</bean>
```

The basic concepts of Spring Batch—such as jobs, steps, `ItemReader`, processors, and writers—were explained in "An Introduction to Spring Batch" on page 232. In this section, we will configure these components and discuss some of their configuration properties. However, we can't go into detail on all the ways to configure and run Spring Batch applications; there is a great deal of richness in Spring Batch relating to error handling, notifications, data validation, data processing, and scaling that we simply can't cover here. For additional information, you should consult the Spring Reference manual or one of the books on Spring Batch mentioned earlier.

Example 13-37 is the top-level configuration to create a Spring Batch job with a single step that processes the output files of a MapReduce job in HDFS and writes them to a database.

Example 13-37. Configuration of a Spring Batch job to read from HDFS and write to a relational database

```
<job id="exportProducts">
  <step id="readWriteProducts">
    <tasklet>
      <chunk reader="hdfsReader" processor="processor" writer="jdbcWriter"
             commit-interval="100" skip-limit="5">
        <skippable-exception-classes>
          <include class="org.springframework.batch.item.file.FlatFileParseException"/>
        </skippable-exception-classes>
      </chunk>
      <listeners>
        <listener ref="jdbcSkipListener"/>
      </listeners>
```

```
    </tasklet>
  </step>
</job>
```

The job defines only one step, which contains a reader, processor, and a writer. The commit interval refers to the number of items to process and aggregate before committing them to the database. In practice, the commit interval value is varied to determine which value will result in the highest performance. Values between 10 and a few hundred are what you can expect to use for this property. One of Spring Batch's features relating to resilient handling of errors is shown here: the use of the `skip-limit` and `skippable-exception-classes` properties. These properties determine how many times a specific error in processing will be allowed to occur before failing the step. The `skip-limit` determines how many times an exception can be thrown before failing the job. In this case, we are tolerating up to five throws of a `FlatFileParseException`. To keep track of all the lines that were not processed correctly, we configure a listener that will write the offending data into a separate database table. The listener extends the Spring Batch class `SkipListenerSupport`, and we override the `onSkipInRead(Throwable t)` that provides information on the failed import line.

Since we are going to read many files created from a MapReduce job (e.g., part-r-00001 and part-r-00002), we use Spring Batch's `MultiResourceItemReader`, passing in the HDFS directory name as a job parameter and a reference to a `FlatFileItemReader` that does the actual work of reading individual files from HDFS. See Example 13-38.

Example 13-38. Configuration of a Spring Batch HDFS reader

```
<bean id="hdfsReader"
  class="org.springframework.batch.item.file.MultiResourceItemReader" scope="step">
  <property name="resources" value="#{jobParameters['hdfsSourceDirectory']}"/>
  <property name="delegate" ref="flatFileItemReader"/>
</bean>

<bean id="flatFileItemReader"
       class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="lineMapper">
    <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean
          class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
          <property name="names" value="id, name, description, price"/>
        </bean>
      </property>
      <property name="fieldSetMapper">
        <bean class="com.oreilly.springdata.batch.item.file.ProductFieldSetMapper"/>
      </property>
   </bean>
  </property>
</bean>
```

When launching the job, we provide the job parameter named `hdfsSourceDirectory` either programmatically, or through the REST API/web application if using Spring

Batch's administration features. Setting the scope of the bean to `step` enables resolution of `jobParameter` variables. Example 13-39 uses a main Java class to load the Spring Batch configuration and launch the job.

Example 13-39. Launching a Spring Batch job with parameters

```
public static void main(String[] args) throws Exception {

  ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath*:/META-INF/spring/*.xml")
  JobLauncher jobLauncher = ctx.getBean(JobLauncher.class);
  Job job = ctx.getBean(Job.class);

  jobLauncher.run(job, new JobParametersBuilder()
    .addString("hdfsSourceDirectory", "/data/analysis/results/part-*")
    .addDate("date", new Date())
    .toJobParameters());
}
```

The `MultiResourceItemReader` class is what allows multiple files to be processed from the HDFS directory */data/analysis/results* that have a filename matching the expression `part-*`. Each file that is found by the `MultiResourceItemReader` is processed by the `FlatFileItemReader`. A sample of the content in each file is shown in Example 13-40. There are four columns in this data, representing the product ID, name, description, and price. (The description is empty in the sample data.)

Example 13-40. Sample content of HDFS file being exported into a database

```
PR1...210,BlackBerry 8100 Pearl,,124.60cl
PR1...211,Sony Ericsson W810i,,139.45
PR1...212,Samsung MM-A900M Ace,,97.80
PR1...213,Toshiba M285-E 14,,166.20
PR1...214,Nokia 2610 Phone,,145.50
…
PR2...315,Sony BDP-S590 3D Blu-ray Disk Player,,86.99
PR2...316,GoPro HD HERO2,,241.14
PR2...317,Toshiba 32C120U 32-Inch LCD HDTV,,239.99
```

We specify `FlatFileItemReader`'s functionality by configuring two collaborating objects of the `DefaultLineMapper`. The first is the `DelimitedLineTokenizer` provided by Spring Batch, which reads a line of input, and by default, tokenizes it based on commas. The field names and each value of the token are placed into Spring Batch's `FieldSet` object. The `FieldSet` object is similar to a JDBC result set but for data read from files. It allows you to access fields by name or position and to convert the values of those fields to Java types such as `String`, `Integer`, or `BigDecimal`. The second collaborating object is the `ProductFieldSetMapper`, provided by us, which converts the `FieldSet` to a custom domain object, in this case the `Product` class.

The `ProductFieldSetMapper` is extremely similar to the `ProductRowMapper` used in the previous section when reading from the database and writing to HDFS. See Example 13-41.