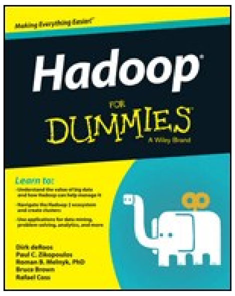

Hadoop for Dummies

by Dirk deRoos et al.

Chapter 10: Developing and Scheduling Application Workflows with Oozie

In This Chapter

- Setting up the Oozie server
- Developing and running an Oozie workflow
- Scheduling and coordinating Oozie workflows

Moving data and running different kinds of applications in Hadoop is great stuff, but it's only half the battle. For Hadoop's efficiencies to truly start paying off for you, start thinking about how you can tie together a number of these actions to form a cohesive workflow. This idea is appealing, especially after you and your colleagues have built a number of Hadoop applications and you need to mix and match them for different purposes. At the same time, you inevitably need to prepare or move data as you progress through your workflows and make decisions based on the output of your jobs or other factors. Of course, you can always write your own logic or hack an existing workflow tool to do this in a Hadoop setting — but that's a lot of work. Your best bet is to use Apache *Oozie*, a workflow engine and scheduling facility designed specifically for Hadoop.

As a workflow engine, Oozie enables you to run a set of Hadoop applications in a specified sequence known as a *workflow*. You define this sequence in the form of a directed acyclic graph (DAG) of actions. In this workflow, the nodes are actions and decision points (where the control flow will go in one direction, or another), while the connecting lines show the sequence of these actions and the directions of the control flow. Oozie graphs are acyclic (no cycles, in other words), which means you can't use loops in your workflows. In terms of the actions you can schedule, Oozie supports a wide range of job types, including Pig, Hive, and MapReduce, as well as jobs coming from Java programs and Shell scripts.

Oozie also provides a handy scheduling facility. An Oozie *coordinator job*, for example, enables you to schedule any workflows you've already created. You can schedule them to run based on specific time intervals, or even based on data availability. At an even higher level, you can create an Oozie *bundle job* to manage your coordinator jobs. Using a bundle job, you can easily apply policies against a set of coordinator jobs by using a *bundle* job.

For all three kinds of Oozie jobs (workflow, coordinator, and bundle), you start out by defining them using individual `.xml` files, and then you configure them using a combination of properties files and command-line options.

Figure 10-1 gives an overview of all the components you'd usually find in an Oozie server. Don't expect to understand all the elements in one fell swoop. We help you work through the various parts shown here throughout this chapter as we explain how all the components work together.

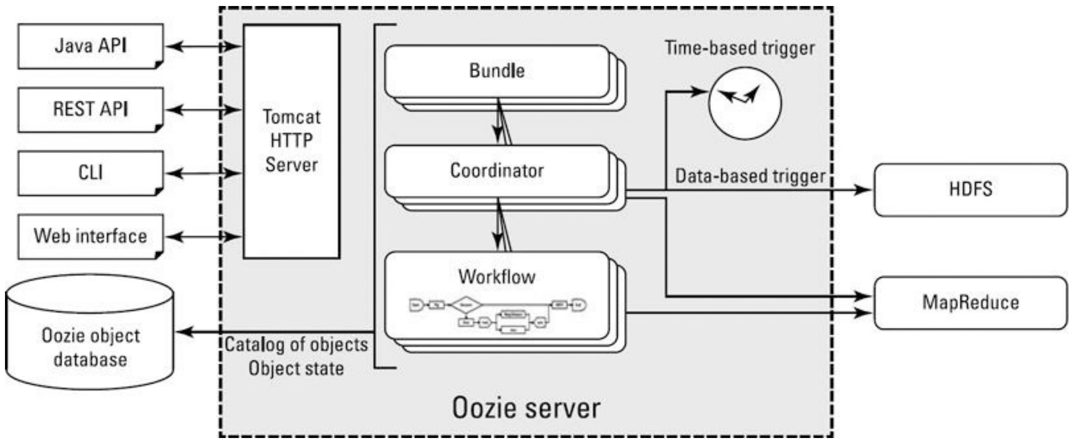

Figure 10-1: Oozie server components

Getting Oozie in Place

Apache Oozie is included in every major Hadoop distribution, including Apache Bigtop, which is the basis of the distribution used by this book. In your Hadoop cluster, install the Oozie server on an edge node, where you would also run other client applications against the cluster's data, as shown in Figure 10-2.

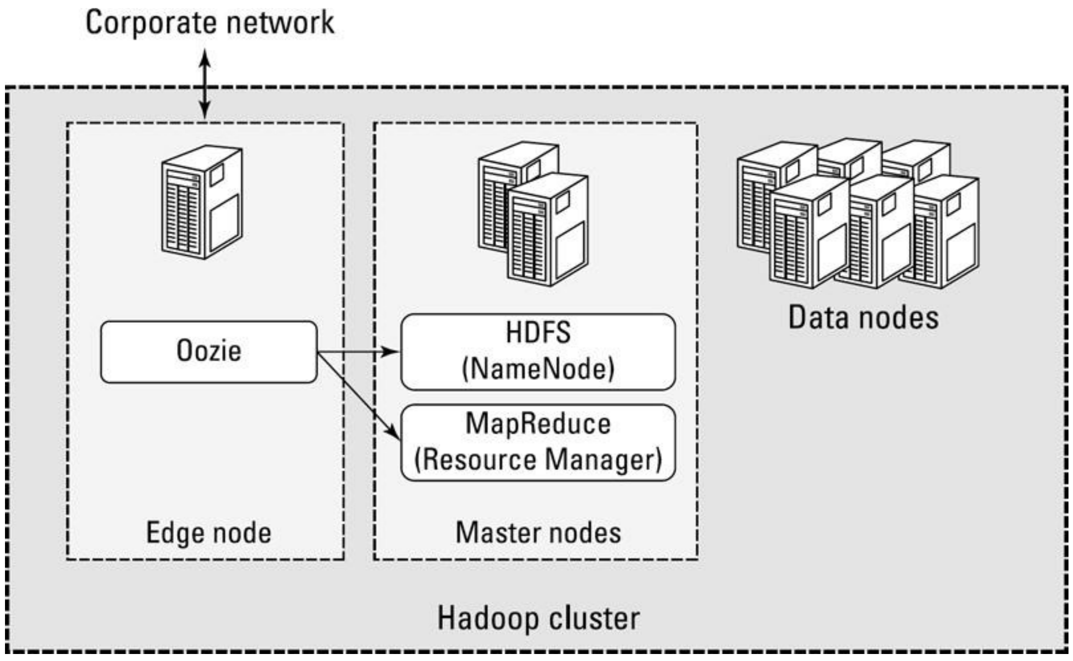

Figure 10-2: Oozie server deployment

Edge nodes are designed to be a gateway for the outside network to the Hadoop cluster. This makes them ideal for data transfer technologies (Flume, for example), but also client applications and other application infrastructure like Oozie. Oozie does not need a dedicated server, and can easily coexist with other services that are ideally suited for edge nodes, like Pig and Hive. For more information on Hadoop deployments, see Chapter 16.

After Oozie is deployed, you're ready to start the Oozie server. Oozie's infrastructure is installed in the `$OOZIE_HOME` directory. From there, run the `oozie-start.sh` command to start the server. (As you might expect, stopping the server involves typing `oozie-stop.sh`.) You can test the status of your Oozie instance by running the command

```
oozie admin -status
```

After you have the Oozie server deployed and started, you can catalog and run your various workflow, coordinator, or bundle jobs. When working with your jobs, Oozie stores the catalog definitions — the data describing all the Oozie objects (workflow, coordinator, and bundle jobs) — as well as their states in a dedicated database.

TECHNICAL STUFF By default, Oozie is configured to use the embedded Derby database, but you can use MySQL, Oracle, or PostgreSQL, if you need to.

A quick look at Figure 10-1 tells you that you have four options for interacting with the Oozie server:

- **The Java API**: This option is useful in situations where you have your own scheduling code in Java applications, and you need to control the execution of your Oozie workflows, coordinators, or bundles from within your application.
- **The REST API**: Again, this option works well in those cases where you want to use your own scheduling code as the basis of your Oozie workflows, coordinators, or bundles, or if you want to build your own interface or extend an existing one for administering the Oozie server.
- **Command Line Interface (CLI)**: It's the traditional Linux command line interface for Oozie.
- **The Oozie Web Console**: Okay, maybe you can't do much interacting here, but the Oozie Web Console gives you a (read-only) view of the state of the Oozie server, which is useful for monitoring your Oozie jobs.

Hue, a Hadoop administration interface, provides another tool for working with Oozie. Oozie workflows, coordinators, and bundles are all defined using XML, which can be tedious to edit, especially for complex situations. Hue provides a GUI designer tool to graphically build workflows and other Oozie objects.

TECHNICAL STUFF Underneath the covers, Oozie includes an embedded Tomcat web server, which handles its input and output.

Developing and Running an Oozie Workflow

Oozie workflows are, at their core, directed graphs, where you can define actions (Hadoop applications) and data flow, but with no looping — meaning you can't define a structure where you'd run a specific operation over and over until some condition is met (a `for` loop, for example). Oozie workflows are quite flexible in that you can define condition-based decisions and forked paths for parallel execution. You can also execute a wide range of actions.

Figure 10-3 shows a sample Oozie workflow.

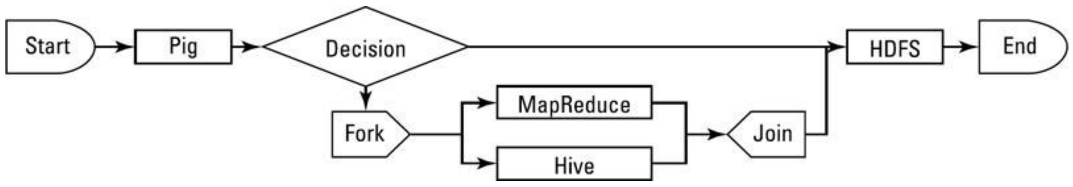

Figure 10-3: A sample Oozie workflow

In this figure, we see a workflow showing the basic capabilities of Oozie workflows. First, a Pig script is run, and is immediately followed by a decision tree. Depending on the state of the output, the control flow can either go directly to an HDFS file operation (for example, a copyToLocal operation) or to a fork action. If the control flow passes to the fork action, two jobs are run concurrently: a MapReduce job, and a Hive query. The control flow then goes to the HDFS operation once both the MapReduce job and Hive query are finished running. After the HDFS operation, the workflow is complete.

Writing Oozie Workflow Definitions

Oozie workflow definitions are written in XML, based on the Hadoop Process Definition Language (hPDL) schema. This particular schema is, in turn, based on the XML Process Definition Language (XPDL) schema, which is a product-independent standard for modeling business process definitions.

An Oozie workflow is composed of a series of actions, which are encoded by XML nodes. There are different kinds of nodes, representing different kinds of actions or control flow directives. Each Oozie workflow has its own XML file, where every node and its interconnections are defined. Workflow nodes all require unique identifiers because they're used to identify the next node to be processed in the workflow. This means that the order in which the actions are executed depends on where an action's node appears in the workflow XML. To see how this concept would look, check out Listing 10-1, which shows an example of the basic structure of an Oozie workflow's XML file.

Listing 10-1: A Sample Oozie XML File

```
<workflow-app name="SampleWorkflow" xmlns="uri:oozie:workflow:0.1">
  <start to="firstJob"/>
  <action name="firstJob">
    <pig>...</pig>
    <ok to="secondJob"/>
    <error to="kill"/>
  </action>
  <action name="secondJob">
    <map-reduce>...</map-reduce>
    <ok to="end" />
    <error to="kill" />
  </action>
  <end name="end"/>
  <kill name="kill">
    <message>"Killed job."</message>
  </kill>
</workflow-app>
```

In this example, aside from the start, end, and kill nodes, you have two action nodes. Each action node represents an application or a command being executed. The next few sections look a bit closer at each node type.

Start and End Nodes

Each workflow XML file must have one matched pair of start and end nodes. The sole purpose of the start node is to direct the workflow to the first node, which is done using the `to` attribute. Because it's the automatic starting point for the workflow, no `name` identifier is required.

REMEMBER Action nodes need `name` identifiers, as the Oozie server uses them to track the current position of the control flow as well as to specify which action to execute next.

The sole purpose of the end node is to provide a termination point for the workflow. A `name` identifier is required, but there's no need for a `to` attribute.

Kill Nodes

Oozie workflows can include *kill* nodes, which are a special kind of node dedicated to handling error conditions. Kill nodes are optional, and you can define multiple instances of them for cases where you need specialized handling for different kinds of errors. Action nodes can include error transition tags, which direct the control flow to the named kill node in case of an error. You can also direct decision nodes to point to a kill node based on the results of decision predicates, if needed. Like an end node, a kill node results in the workflow ending, and it does not need a `to` attribute.

Decision Nodes

Decision nodes enable you to define conditional logic to determine the next step to be taken in the workflow — Listing 10-2 gives some examples:

Listing 10-2: A Sample Oozie XML File

```
<workflow-app name="SampleWorkflow" xmlns="uri:oozie:workflow:0.1">
  <start to="firstDecision"/>
  @@1
  <decision name="firstDecision"≫
  <switch>
   @@2
    <case to="firstJob">
    ${fs:fileSize('usr/dirk/ny-flights') gt 10 * GB}
    </case>
   @@3
    <case to="secondJob"≫
    ${fs:filSize('usr/dirk/ny-flights') lt 100 * MB}
    </case>
   @@4
    <default to="thirdJob"/>
  </switch>
  </decision>
  <action name="firstJob">...</action>
  <action name="secondJob">...</action>
  <action name="thirdJob">...</action>
  <end name="end"/>
</workflow-app>
```

In this workflow, we begin with a decision node (see the code following the bold @@1), which includes a case statement (called `switch`), where, depending on the size of the files in the `‘usr/dirk/ny-flights’` directory, a different action is taken. Here, if the size of the files in the `‘usr/dirk/ny-flights’` directory is greater than 10GB (see the code following the bold @@2), the control flow runs the action named `firstJob` next. If the size of the files in the `‘usr/dirk/ny-flights’` directory is less than 100MB (see the code following the bold @@3), the control flow runs the action named `secondJob` next. And if neither case we've seen so far is true (in this case, if the size of the files in the `‘usr/dirk/ny-flights’` directory is greater than 100MB and less than 10GB), we want the action named `thirdJob` to run.

REMEMBER Case statements (seen here as `switch`) are quite common in control flow programming languages. (We talk about the difference between *control flow* and *data flow* languages in Chapter 8.) They enable you to define the flow of a program based on a series of decisions. They're called case statements, because they're really a set of cases: for example, *in case* the first comparison is true, we'll run one function, or *in case* the second comparison is true, we'll run a different function.

As we just saw, a decision node consists of a `switch` operation, where you can define one or more cases and a single default case, which is mandatory. This is to ensure the workflow always has a next action. *Predicates* for the case statements — the logic inside the `<case>` tags — are written as JSP Expression Language (EL) expressions, which resolve to either a `true` or `false` value.

TIP For the full range of EL expressions that are bundled in the Oozie, check out the related Oozie workflows specifications at this site:

```
http://oozie.apache.org/docs/4.0.0/WorkflowFunctionalSpec.
html#a4.2_Expression_Language_Functions
```

Action Nodes

Action nodes are where the actual work performed by the workflow is completed. You have a wide variety of actions to choose from — Hadoop applications (like Pig, Hive, and MapReduce), Java applications, HDFS operations, and even sending e-mail, to name just a few examples. You can also configure custom action types for operations that have no existing action.

Depending on the kind of action being used, a number of different tags need to be used. All actions, however, require transition tags: one for defining the next node after then successful completion of the action, and one for defining the next node if the action fails. In the following list, we describe the more commonly used action node types:

- **MapReduce**: MapReduce, as we discuss in Chapter 6, is a framework for distributed applications to run on Hadoop. For a MapReduce workflow to be successful, a couple things need to happen. MapReduce actions, for example, require that you specify the addresses of the processing and storage servers for your Hadoop cluster. We also need to specify the master services for both the processing and storage systems in Hadoop so that Oozie can properly submit this job for execution on the Hadoop cluster, and so that the input files can be found. Listing 10-3 shows the tagging for a MapReduce action:

  Listing 10-3: A Sample Oozie XML File to Run a MapReduce Job

```
<workflow-app name=" SampleWorkflow" xmlns="uri:oozie:workflow:0.1">
   ...
   <action name="firstJob">
      <map-reduce>
      @@1     <job-tracker>serverName:8021</job-tracker>
         <name-node>serverName:8020</name-node>
      @@2      <prepare>
            <delete path="hdfs://clientName:8020/usr/sample/output-data"/>
         </prepare>
      @@3     <job-xml>jobConfig.xml</job-xml>
         <configuration>
           ...
            <property>
               <name>mapreduce.map.class</name>
               <value>dummies.oozie.FlightMilesMapper</value>
            </property>
            <property>
               <name>mapreduce.reduce.class</name>
               <value>dummies.oozie.FlightMilesReducer </value>
            </property>
            <property>
               <name>mapred.mapoutput.key.class</name>
               <value>org.apache.hadoop.io.Text</value>
            </property>
            <property>
               <name>mapred.mapoutput.value.class</name>
               <value>org.apache.hadoop.io.IntWritable</value>
            </property>
            <property>
               <name>mapred.output.key.class</name>
               <value>org.apache.hadoop.io.Text</value>
            </property>
            <property>
               <name>mapred.output.value.class</name>
               <value>org.apache.hadoop.io.IntWritable</value>
            </property>
            <property>
               <name>mapred.input.dir</name>
               <value>'/usr/dirk/flightdata'</value>
            </property>
            <property>
               <name>mapred.output.dir</name>
               <value>'/usr/dirk/flightmiles'</value>
            </property>
            ...
         </configuration>
      </map-reduce>
      <ok to="end"/>
      <error to="end"/>
   </action>
   ...
</workflow-app>
```

In this code, we just have a single action to illustrate how to invoke a MapReduce job from an Oozie workflow. In the code following the bold @@1, we need to define the master servers for the storage and processing systems in Hadoop. For the processing side, the old JobTracker term is used, but you can enter the name for the Region Server if you're using YARN to manage the processing in your cluster. (See Chapter 7 for more information on the JobTracker and the Region Server and how they manage the processing for Hadoop, both in Hadoop 1 and in Hadoop 2.) Note that we also specify the server and port number for the NameNode (again, so the MapReduce job can find its files).

In the code following the bold @@2, the `<prepare>` tag is used to delete any residual information from previous runs of the same application. You can also do other file movement operations here if needed.

All the definitions for the MapReduce applications are specified in configuration details. In the code following the bold @@3, we can see the first of two options: the `<job-xml>` tag, which is optional, can point to a Hadoop `JobConf` file, where you can define all your configuration details outside the Oozie workflow XML document. This can be useful if you need to run the same MapReduce application in many of your workflows, so if configurations need to change you only need to adjust the settings in one place. You can also enter configuration details in the `<configuration>` tag, as we've done in the example above. In the example, you can see that we define all the key touch points for the MapReduce application: the data types of the key/value pairs as they input and output the map and reduce phases, the class names for the map and reduce code you have written, and the paths for the input and output files. It's important to note that configuration settings specified here would override any settings defined in the file identified in the `<job-xml>` tag.

- **Hive**: Similar to MapReduce actions, as just described, Hive actions require that you specify the addresses of the processing and storage servers for your Hadoop cluster. Hive enables you to submit SQL-like queries against data in HDFS that you've cataloged as a Hive table. (For more information on Hive, see Chapter 13.) As Hive does its work, Hive queries get turned into MapReduce jobs, so we will need to specify the names of the processing and storage systems used in your Hadoop cluster. The following example shows the tagging for a Hive action:

Listing 10-4: A Sample Oozie XML File to Run a Hive Query

```
<workflow-app name="SampleWorkflow" xmlns="uri:oozie:workflow:0.2">
    ...
    <action name="firstJob">
        <hive>
            <job-tracker>serverName:8021</job-tracker>
            <name-node>serverName:8020</name-node>
            <prepare>
                <delete path="hdfs://clientName:8020/usr/sample/output-data"/>
            </prepare>
            <job-xml>jobConfig.xml</job-xml>
            <configuration>...</configuration>
        @@1 <script>firstJob.hql</script>
        </hive>
        <ok to="end"/>
        <error to="end"/>
    </action>
    ...
</workflow-app>
```

In the code in Listing 10-4, we have defined similar definitions as we've done with the MapReduce action. The key difference here is that we can avoid the extensive configuration tags defining the MapReduce details and simply specify the location and name of the file containing the Hive query. (See the code following the bold @@1.)

TIP To specify the Hive script being used, enter the filename and path in the `<script>` tag. Aside from this tag, the remaining tags shown are the same as for MapReduce.

- **Pig**: Pig scripts enable you to define a data flow (a series of actions you can apply to data) and the Pig compiler turns that code into MapReduce. (See Chapter 8 for more on Pig in general.) Pig actions require that you specify the addresses of the processing and storage servers for your Hadoop cluster. Since the Hadoop processing and storage systems are used here, as they are in the Hive action, we need to specify their names here as well. Listing 10-5 shows the tagging for a Pig action:

Listing 10-5: A Sample Oozie XML File to Run a Pig Script

```
<workflow-apfp name="SampleWorkflow" xmlns="uri:oozie:workflow:0.2">
    ...
    <action name="firstJob">
        <pig>
            <job-tracker>serverName:8021</job-tracker>
            <name-node>serverName:8020</name-node>
            <prepare>
                <delete path="hdfs://clientName:8020/usr/sample/output-data"/>
            </prepare>
            <job-xml>jobConfig.xml</job-xml>
            <configuration>...</configuration>
        @@1 <script>firstJob.pig</script>
        </pig>
        <ok to="end"/>
        <error to="end"/>
    </action>
    ...
</workflow-app>
```

Listing 10-5 looks a lot like Listing 10-4. Once again, we have defined similar definitions as we've done with the MapReduce action and once again the key difference here is that we can avoid the extensive configuration tags defining the MapReduce details. All we have to do is specify the location and name of the file containing the Pig script query. (See the code following the bold @@1.) To specify the `.pig` script being used, enter the filename and path in the `<script>` tag.

- **File System (FS)**: The File System action enables you to run HDFS commands as part of your workflow, which is tremendously useful as you post-process and pre-preprocess data. ***Note***: The HDFS commands enable you to perform the typical file movement operations people need to do when manipulating data inputs and outputs, like deleting, copying, renaming, and moving files. Listing 10-6 shows the tagging for a file system action where a file is deleted, a directory is created, a file is moved, and permissions are changed:

Listing 10-6: A Sample Oozie XML File to Run File System Commands

```
<workflow-app name="SampleWorkflow" xmlns="uri:oozie:workflow:0.1">
    ...
    <action name="firstJob">
         <fs>
            <delete path="hdfs://servername:8020/usr/sample/temp-data"/>
            <mkdir path="archives/${wf:id()}"/>
            <move source="${jobInput}"
           target="archives/${wf:id()}/processed-input"/>
            <chmod path="${jobOutput}" permissions="-rwxrw-rw-" dir-files="true"><recursive/></ch
        </fs>
        <ok to="end"/>
        <error to="end"/>
    </action>
    ...
</workflow-app>
```

Fork and Join Nodes

You can define parallel execution tracks for your workflows by using fork and join nodes together. This structure, which begins with a fork, can spawn two or more workflow paths, which would then be executed in parallel. Use the join node to merge the control flow back to a single path. See the code in Listing 10-7:

Listing 10-7: A Sample Oozie XML File to Fork a Control Flow

```
<workflow-app name="SampleWorkflow" xmlns="uri:oozie:workflow:0.1">
  <start to="fork"/>
  <fork name="fork">
    <path start="firstJob" />
    <path start="secondJob" />
  </fork>
  <action name="firstJob">
    ...
    <ok to="join" />
    <error to="end" />
  </action>
  <action name="secondJob">
    ...
    <ok to="join" />
    <error to="end" />
  </action>
  <join name="join" to="end" />
  <end name="end"/>
</workflow-app>
```

The actions and other control flow nodes must point to the join node to terminate the individual workflow paths that were spawned with the fork operation. Before the next node pointed to by the join node can be executed, all the actions and control flows in each of the paths must be finished.

Configuring Oozie Workflows

You can configure Oozie workflows in one of three ways, depending on your particular circumstances. You can use

- **The `config-default.xml` file**: Defines parameters that don't change for the workflow.

- **The `job.properties` file**: Defines parameters that are common for a particular deployment of the workflow. Definitions here override those made in the `config-default.xml` file.
- **The command-line parameters**: Defines parameters that are specific for the workflow invocation. Definitions here override those made in the `job.properties` file and the `config-default.xml` file.

The configuration details will differ, depending on the action they're associated with. For example, as we saw in the MapReduce action (map-action) in Listing 10-3, you have many more things to configure there, as opposed to a file system (`fs`) action like the one shown in Listing 10-6.

Running Oozie Workflows

Before running your Oozie workflows, all its components need to exist within a specified directory structure. Specifically, the workflow itself should have its own, dedicated directory, where `workflow.xml` is in the root directory, and any code libraries exist in the subdirectory named `lib`. The workflow directory and all its files must exist in HDFS for it to be executed.

TIP If you'll be using the Oozie command-line interface to work with various jobs, be sure to set the `OOZIE_URL` environment variable. (This is easily done from a command line in a Linux terminal.) You can save yourself a lot of typing because the Oozie server's URL will now automatically be included with your requests. Here's a sample command one could use to set the `OOZIE_URL` environment variable from the command line:

```
export OOZIE_URL="http://localhost:8080/oozie"
```

TIP To run an Oozie workload from the Oozie command-line interface, issue a command like the following, while ensuring that the `job.properties` file is *locally accessible* — meaning the account you're using can see it, meaning it has to be on the same system where you're running Oozie commands:

```
$ oozie job -config sampleWorkload/job.properties -run
```

After you submit a job, the workload is stored in the Oozie object database. (Refer to Figure 10-1.) On submission, Oozie returns an identifier to enable you to monitor and administer your workflow — `job: 0000001-00000001234567-oozie-W`, for example:

To check the status of this job, you'd run the command

```
oozie job -info 0000001-00000001234567-oozie-W
```

Scheduling and Coordinating Oozie Workflows

After you've created a set of workflows, you can use a series of Oozie coordinator jobs to schedule when they're executed. You have two scheduling options for execution: a specific time and the availability of data in conjunction with a certain time. The following three sections take a look at each option.

Time-Based Scheduling for Oozie Coordinator Jobs

Oozie coordinator jobs can be scheduled to execute at a certain time, but after they're started, they can then be configured to run at specified intervals. The following example shows a coordinator job that starts running at a specified start time and date:

Listing 10-8: A Sample Oozie XML File to Schedule a Workflow by Time

```
<coordinator-app name="sampleCoordinator"
                 frequency="${coord:days(1)}"
                 start="2014-06-01T00:01Z"
                 end="2014-06-01T01:00Z"
                 timezone="UTC"
                 xmlns="uri:oozie:coordinator:0.1">
   <controls>...</controls>
   <action>
      <workflow>
         <app-path>${workflowAppPath}</app-path>
      </workflow>
   </action>
</coordinator-app>
```

Time and Data Availability-Based Scheduling for Oozie Coordinator Jobs

Oozie coordinator jobs can also be scheduled to execute at a certain time if specified data files or directories are available. Listing 10-9 shows an example of a coordinator that starts running at a specified start time and date, is executed once a day if the data set identified by `triggerDatasetDir` exists, and runs until the specified end time:

Listing 10-9: A Sample Oozie XML File to Schedule a Workflow by Time and Data Availability

```
<coordinator-app name="sampleCoordinator"
                 frequency="${coord:days(1)}"
                 start="${startTime}"
                 end="${endTime}"
                 timezone="${timeZoneDef}"
                 xmlns="uri:oozie:coordinator:0.1">
   <controls>...</controls>
   <datasets>
      <dataset name="input" frequency="${coord:days(1)}" initial-
                  instance="${startTime}" timezone="${timeZoneDef}">
         <uri-template>${triggerDatasetDir}</uri-template>
      </dataset>
   </datasets>
   <input-events>
         <data-in name="sampleInput" dataset="input">
         <instance>${startTime}</instance>
      </data-in>
   </input-events>
   <action>
      <workflow>
         <app-path>${workflowAppPath}</app-path>
      </workflow>
   </action>
</coordinator-app>
```

Running Oozie Coordinator Jobs

Similar to Oozie workflow jobs, coordinator jobs require a `job.properties` file, and the coordinator.xml file needs to be loaded in the HDFS. To run an Oozie coordinator job from the Oozie command-line interface, issue a command like the following while ensuring that the `job.properties` file is locally accessible:

```
$ oozie job -config sampleCoordinator/job.properties -run
```

After you submit the job, the coordinator is stored in the Oozie object database. (Refer to Figure 10-1.) On submission, Oozie returns an identifier to enable you to monitor and administer your coordinator — `job: 0000001-00000001234567-oozie-C`, for example:

To check the status of this job, run the command

```
oozie job -info 0000001-00000001234567-oozie-C
```