4
Using OSWorkflow in your Application

As seen in Chapter 2, one way to use OSWorkflow is embedding it into your application. This chapter covers the main API features needed to successfully use OSWorkflow inside your application.

This chapter covers all the details of the OSWorkflow configuration and persistence options. It also talks about unit testing for workflow definitions, which is an indispensable feature for quickly validating and checking descriptors.

We also integrate OSWorkflow with Spring—a popular lightweight object container with features such as declarative transactions and AOP programming. This gives OSWorkflow features to your Spring application.

The chapter ends with OSWorkflow's security mechanisms for restricting access to actions and `Workflow` instances, and for extending the security model with our own user and group directory such as LDAP.

OSWorkflow Configuration

This section will cover the configuration options available for OSWorkflow. We'll show you how to register the workflow definition created in the previous two chapters and generate new workflow instances through programming. After that, we will see the workflow persistence options offered by OSWorkflow and PropertySet.

Registering our Process Descriptors

Before tackling the aspects of persistence and the workflow factory concept, we must see how we can configure OSWorkflow.

Firstly, the framework configures itself by parsing and reading the `osworkflow.xml` file in the classpath. This file contains several settings, such as the `WorkflowStore` implementation class name, the `WorkflowFactory` handler class name, and so on.

The default `osworkflow.xml` file is as follows:

```
<osworkflow>
  <persistence class="com.opensymphony.workflow.spi.memory
.                                             MemoryWorkflowStore"/>
  <factory class= "com.opensymphony.workflow.loader.
                                                 XMLWorkflowFactory">
  <property key="resource" value="workflows.xml" />
  </factory>
</osworkflow>
```

OSWorkflow delegates the persistence features—such as loading and saving individual instances—of the engine to an interface named `WorkflowStore`. There are several built-in implementations such as EJB, JDBC, etc. The default `osworkflow.xml` file uses the `MemoryWorkflowStore` implementation for persisting instances in memory.

The `WorkflowFactory` is the interface responsible for reading workflow definitions in any format and giving them to the engine in a proper format. The default implementation is `XMLWorkflowFactory`; obviously this reads XML workflow definitions.

As you can see, the default `osworkflow.xml` file is configured to use an `XMLWorkflowFactory` and points to a resource file called `workflows.xml`. The `workflows.xml` file looks like this:

```
<workflows>
  <workflow name="holiday" type="resource" location="holiday.xml"/>
  <workflow name="loan" type="resource" location="loan.xml"/>
</workflows>
```

The `workflows.xml` file describes which workflow definitions are available and in which file the engine can find the XML data. You can see we have two different processes to initiate—`holiday` and `loan`.

Embedding OSWorkflow into your Application

OSWorkflow can be embedded in any application whether it's J2SE or JEE based by simply including the `osworkflow.jar` file into the classpath of the application. In any case, you must have the `osworkflow.xml` file, any referenced resources such as `workflows.xml`, and the XML of the process in the classpath. In this case, the workflow descriptor name is `holiday.xml`.

Starting a Workflow

Imagine you have an application that interfaces with OSWorkflow, and you'd like to instantiate a new `Workflow`. This is as easy as:

```
Workflow wf = new BasicWorkflow("johndoe");
Long id = Wf.initialize("holiday", 1, null);
```

The first line creates a new `BasicWorkflow` with the current username as parameter. `BasicWorkflow` objects are heavyweight and it is reasonable to have one instance per user to avoid the creation cost.

The second line executes the `initialize()` method with the workflow name as the first parameter, the initial action number as the second parameter, and the actions input map as the third parameter. In this case, the workflow name is the definition name as stated in the `workflows.xml` file. We send `null` as the third parameter because we need no input parameters to instantiate this particular workflow.

The returned `Long` is the workflow identification number assigned by the engine. The initial number is defined in the `WorkflowDescriptor`. If this ID is incorrect, the engine will throw an `InvalidActionException`.

[This code snippet doesn't call the `Configuration` object. This is very important if you plan to use differently configured Workflows in the same JVM.]

Before initializing a workflow instance, you can test it by calling the `Workflow` interface method, `canInitialize()`. If this method returns true, then you can safely execute the `initialize()` method.

```
boolean canInit = wf.canInitialize("Holiday", 1, null)
```

Executing Actions

We now have a newly created instance; let's execute some actions. We need to invoke the `doAction` method of the `Workflow` interface. The code is as follows:

```
Wf.doAction(id, 1, null);
```

The parameters are the workflow identifier number, the action number ID (now you can see why actions must be uniquely numbered within a definition), and a map with inputs for the workflow. We send `null` as the third parameter indicating that there is no need of external inputs for this workflow type and action.

Every call to the `initialize()` and `doAction()` methods takes a map as an input parameter. The transient variables map is merged with this input map, so you can also find the input content. This is the main mechanism to send information to the `Workflow` instance from the caller. The input map key name is preserved in the transient variables map.

What's the Status?

To get the current steps of the workflow, you must call the `getCurrentSteps()` method of the `Workflow` interface. The code snippet is as follows:

```
List steps = wf.getCurrentSteps(id);
```

This method returns a list of `StepDescriptor`, one for each current step of the `Workflow` instance. To see the step information, we must call the `WorkflowDescriptor`. The following code snippet shows you how to do that:

```
for (Iterator iterator = steps.iterator(); iterator.hasNext();)
{
  Step step = (Step) iterator.next();
  StepDescriptor sd = wd.getStep(step.getStepId());
}
```

By iterating the current step list and looking up a `StepDescriptor` from the `WorkflowDescriptor`, we can get detailed step information, such as the ID, start date, finish date, and name of the step. If you want to see the history steps, call the `getHistorySteps()` method. The code is as follows:

```
List steps = wf.getHistorySteps(id);
```

Similarly to its current steps counterpart, `getHistorySteps` returns a list of `StepDescriptor`, this time with the completed steps of the `Workflow` instance. To describe the history steps, you can use the code snippet mentioned earlier to describe the current steps.

What can I Do?

Typically in user interfaces, you must see the actions available for the current steps and the current user. The `Workflow` interface has a method called `getAvailableActions()` for that purpose. The following code fragment shows how to invoke it:

```
int[] actions = wf.getAvailableActions(id, null);
```

The parameters are the workflow instance identifier and the parameters map again. For some action to show as available, it must satisfy some condition such as the existence of external data. The passing of the map allows for this sort of scenario to happen. The method returns an array of action IDs. To retrieve the action names, you must use the `WorkflowDescriptor`. See the following snippet:

```
WorkflowDescriptor wd =
                    wf.getWorkflowDescriptor(wf.getWorkflowName(id));
for (int i = 0; i < actions.length; i++)
{
  String name = wd.getAction(actions[i]).getName();
}
```

The code iterates over the action ID array and calls the `getAction()` descriptor method. This method returns an `ActionDescriptor`—an object that describes an action in the definition. Finally, it calls the `ActionDescriptor.getName()` method to obtain the name of the action.

The Useful Abstract States

Besides the current status and old status values you provide in the definition, OSWorkflow has the concept of an abstract state, which is a state every workflow has implicitly. These states are as follows:

State	Description
Activated	Workflow is live.
Completed	Workflow has been finished without any problem.
Created	Workflow has been initialized but no actions have been executed yet.
Killed	Workflow has been canceled.
Suspended	Workflow has been suspended.
Unknown	The state of the workflow is unknown.

Programmatically you can know the abstract state for a particular `Workflow` instance by calling the `getEntryState()` method.

```
int abstract = wf.getEntryState(long id)
```

You can change the abstract state of the instance by calling the `changeEntryState()` method of the `Workflow` interface. Be sure to check the abstract state constants present in the `Workflow` interface.

Querying the Workflow Store

Human-oriented BPMS have GUIs that let the user realize tasks and search for work items. OSWorkflow permits searching the workflow store via a `WorkflowExpressionQuery`. This class is a GOF composite design pattern, so you can nest expressions into expressions for complex queries.

This search is a very generic one including only the fields in OSWorkflow. For more powerful searches, you should create a domain concept that can be attached to the workflow ID. For example, the `holiday` workflow uses the domain concept of an Employee Request.

You should have an Employee Request table with all the important domain data, such as department, dates, etc. This is the table to be searched when domain data is needed. If you can survive with only the workflow default data, the following search is very useful.

The following code example searches the store for `Workflows` having current steps with the `OWNER` equal to `johndoe`. Don't worry about the owner; we'll see this concept in the security section of this chapter.

```
WorkflowExpressionQuery q = new WorkflowExpressionQuery
                         (new FieldExpression(FieldExpression.OWNER,
                          FieldExpression.CURRENT_STEPS,
                          FieldExpression.EQUALS, "johndoe"));
List wfs = wf.query(q);
```

The following fields are available for searching:

Field	Description
`ACTION`	The action that triggered the transition to the step.
`CALLER`	The caller of the action.
`DUE_DATE`	The due date of the step.
`FINISH_DATE`	The finish date of the step. It is Null if the step is not yet finished.
`OWNER`	The owner of the step.
`START_DATE`	The start date of the step.
`STATE`	The state of the workflow.
`STATUS`	The status of the step.
`STEP`	The step.
`NAME`	The name of the business process.

The contexts are as follows:

Context	**Description**
`CURRENT_STEPS`	The current steps of the workflow
`ENTRY`	The workflow entry that is the header information
`HISTORY_STEPS`	The history steps of the workflow

The operators are as follows:

Operator	**Description**
`EQUALS`	Equals operator
`GT`	Greater than operator
`LT`	Less than operator
`NOT_EQUALS`	Not Equals operator

The workflow store drains performance from transactional activity, so use it with care. Some workflow stores don't support querying while others don't support nested expressions; so be sure to check your store. For example, the `HibernateStore` included in OSWorkflow currently doesn't support nested expressions.

Querying the Workflow Factory

The `WorkflowFactory` interface has the following methods to inspect the available workflows descriptors.

The `getWorkflowName()` method returns the workflow name of a particular `Workflow` instance. The `getWorkflowNames()` method returns an array of strings with all the available workflow definition names. Check the following snippet for the usage of these methods:

```
String wfName = workflow.getWorkflowName(workflowId);
System.out.println("available workflows:" +
                        Arrays.toString(workflow.getWorkflowNames()));
```

This code is needed when there's more than one `WorkflowDescriptor` in your system and you want to programmatically query their names. Once you have their names, you can instantiate new workflows or inspect their descriptors.

Inspecting the Workflow Descriptor from Code

The `XMLWorkflowDescriptor` describes a business process in a human-readable format. When OSWorkflow parses and validates the XML, it builds a memory structure called the `WorkflowDescriptor`. This descriptor has all the information that the process engine needs to follow the process and to create a new instance of the `Workflow`. We can get a hold of the descriptor of any factory-registered `Workflow` by calling the `getWorkflowDescriptor()` method of the `Workflow` interface. The following code fragment shows a sample invocation:

```
WorkflowDescriptor wd = wf.getWorkflowDescriptor("holiday");
```

This code will return an object representation of the XML workflow descriptor that we built in Chapter 2. By traversing the descriptor, we can analyze the process structure and get the steps, actions, results, etc. of the `WorkflowDescriptor`.

Don't confuse the `WorkflowDescriptor` with its instances.

You can also build a `WorkflowDescriptor` programmatically; it is useful for dynamic on-the-fly processes.

Using the Workflow Configuration Object

The examples of instantiating a new workflow that have been discussed so far didn't make any reference to the `Configuration` object. If you don't call the `Configuration` object, OSWorkflow assumes a unique configuration for workflows in the JVM. If you plan to use workflows with different store and factory options, you must call the `Configuration` object. The following code fragment shows you how:

```
Workflow workflow = new BasicWorkflow("Holiday");
Configuration config = new DefaultConfiguration();
workflow.setConfiguration(config);
```

It is recommended to call the `Configuration` object for flexibility. Be sure to call the `setConfiguration` method of the `Workflow` interface to use the per-instance configuration model.

Workflow Interface Implementations

OSWorkflow offers great extensibility by giving us different implementations of the `Workflow` interface. The following table summarizes the relevant features of each one:

Implementation	Features
`BasicWorkflow`	Basic implementation. It doesn't support transactions.
`OfBiz`	Based on OfBiz, it supports local transactions.
`EJBWorkflow`	Using JTA and CMP, it supports global XA transactions. It must be used only inside J2EE-compliant application servers, like JBoss.

The `OfBiz` implementation is based on the transaction components of the OfBiz suite (an open-source ERP) to implement local transactions i.e., JDBC transactions. Use it only if you are storing the `Workflow` instance data in a JDBC-compliant database.

On the other hand, the `EJBWorkflow` alternative will use the JTA J2EE API to create a global transaction in each method invocation, causing an unnecessary overhead for simple applications. Use it only if you need distributed transactions, i.e. a workflow action and another database inserted in one transaction.

Implementations supporting transactions like `OfBiz` and `EJBWorkflow` can roll back the current transaction by calling the `setRollbackOnly()` method. Also, in the case of an exception, OSWorkflow will roll back the current transaction to preserve data consistency.

To change your current implementation just instantiate the implementation class instead of `BasicWorkflow`.

[Remember that `BasicWorkflow` doesn't support transactions!]

If none of the options suits your needs, you can create your own `Workflow` class by implementing the `Workflow` interface.

Loading the Descriptors—the Workflow Factory

As we have seen before, OSWorkflow delegates the responsibility of loading workflow definitions to a `WorkflowFactory` implementation. There are three built-in implementations to choose from, namely, `XMLWorkflowFactory`, `JDBCWorkflowFactory`, and `SpringHibernateWorkflowFactory`.

Don't confuse the `WorkflowFactory` with the `WorkflowStore`. The first one manages the descriptors while the latter manages the workflow instance data.

`XMLWorkflowFactory` loads the process definition from an XML file in the file system. This is the default implementation. It takes only one parameter called `resource`, which specifies the `workflow` XML file name. This file is loaded from the classpath. A slightly modified variation is the `SpringWorkflowFactory`, which looks up the XML files from a Spring resource.

Loading Descriptors from a Database

`JDBCWorkflowFactory` uses the database to load the workflow descriptors. This is done with a `BLOB` or `LONG VARCHAR` column. To use this factory, you must declare it in `osworkflow.xml` file; it takes a mandatory parameter called `datasource`, which an is the JNDI name of the JDBC datasource to be used. See the following `osworkflow.xml` file:

```
<osworkflow>
  <factory class=
           "com.opensymphony.workflow.loader.JDBCWorkflowFactory">
  <arg name="datasource" value="jdbc/Defaultds"/>
  </factory>
</osworkflow>
```

This sample file will try to look up a JNDI resource called `jdbc/Defaultds` and then try to get a connection from it. Finally, it will try to use a database table named `OS_WORKFLOWDEFS` to find the workflow descriptors. This table is composed of two columns, the first `WF_NAME`, which is the workflow name and is of the `CHAR` or `VARCHAR` type while the second column is `WF_DEFINITION`. The whole XML will be stored in this column as a `BINARY` or `TEXT` type.

`SpringHibernateWorkflowFactory` is the same as above, but uses the Hibernate framework, benefiting from caching and the high-performance ORM features.

Each `WorkflowFactory` retrieves a `WorkflowDescriptor`, a class representing the structure of the definition in an object-oriented way. Implementing your own `WorkflowFactory` enables you to build workflow definitions on the fly. You can implement a template definition and customize it on the fly using some rules.

You can build your own implementation if none of the options fits your requirement by implementing the `WorkflowFactory` interface.

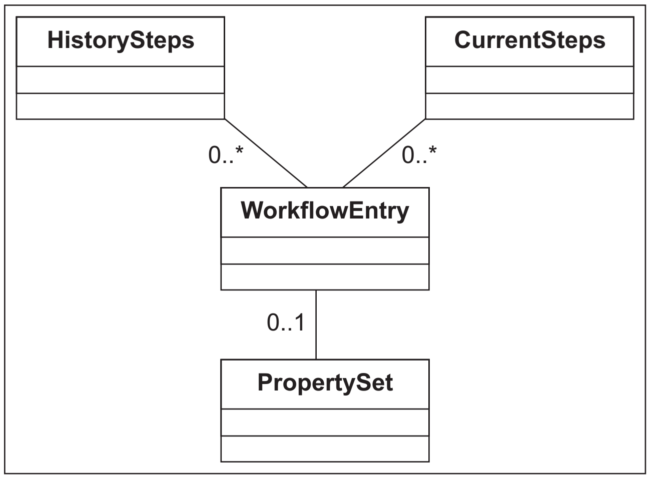
Persisting Instance Information

OSWorkflow delegates the responsibility of loading and storing instance data to the `WorkflowStore`. Later in the section we'll see the different built-in alternatives. First, we'll take a look at exactly which data is made persistent.

The data that is made persistent in OSWorkflow when you use one of the database-backed alternatives is as follows:

- Workflow entry: The instance header data, such as process name, abstract state, etc.
- Current and history steps: The steps that were travelled and the one that the workflow is in.
- PropertySet: The persistent instance-specific data.

The following figure displays the relationships between them.



OSWorkflow gives several choices for storing this information. It's very important that you chose your strategy carefully for maintenance and performance reasons. You can configure the persistence strategy in the `osworkflow.xml` file.

Persistence Choices

The following are the built-in `WorkflowStore` implementations in OSWorkflow:

Memory Store

This store strategy saves the workflow information in RAM, useful for testing your business processes.

JDBC

This uses plain old JDBC to access and save the data. It lacks the caching and fetching strategies of Hibernate. The JDBC store is a more basic strategy than Hibernate and is suitable only for very simple workflow applications.

Hibernate

The Hibernate store uses this ORM framework to manage the persistence of `Workflows`. It has the advantage of high performance ORM features such as caching.

Hibernate uses XML files to map objects to relational concepts. In the mapping files, you can tune parameters such as fetching strategies and lazy loading. For more information about Hibernate, take a look at `www.hibernate.org`.

This store is highly recommended as the default production store of OSWorkflow.

Other Stores

OSWorkflow is bundled with other store implementations like `MySQLWorkflowStore`, `SerializableStore`, `OfBizStore`, and `EJBStore`. Be aware that they are designed for very specific scenarios.

If none of the previous strategies suits your needs, then you can build one by implementing the `WorkflowStore` interface.

Configuring the JDBC Store

In this section, we'll configure the JDBC Store for basic usage. This store expects several configuration parameters in the `osworkflow.xml` file:

```
<persistence class=
              "com.opensymphony.workflow.spi.jdbc.JDBCWorkflowStore">
<!-- For jdbc persistence, all are required. -->
  <property key="datasource" value="jdbc/DefaultDS"/>
  <property key="entry.sequence"
                           value="SELECT nextVal('seq_os_wfentry')"/>
  <property key="entry.table" value="OS_WFENTRY"/>
  <property key="entry.id" value="ID"/>
  <property key="entry.name" value="NAME"/>
  <property key="entry.state" value="STATE"/>
  <property key="step.sequence"
                      value="SELECT nextVal('seq_os_currentsteps')"/>
  <property key="history.table" value="OS_HISTORYSTEP"/>
  <property key="current.table" value="OS_CURRENTSTEP"/>
  <property key="historyPrev.table" value="OS_HISTORYSTEP_PREV"/>
  <property key="currentPrev.table" value="OS_CURRENTSTEP_PREV"/>
  <property key="step.id" value="ID"/>
  <property key="step.entryId" value="ENTRY_ID"/>
```

```
    <property key="step.stepId" value="STEP_ID"/>
    <property key="step.actionId" value="ACTION_ID"/>
    <property key="step.owner" value="OWNER"/>
    <property key="step.caller" value="CALLER"/>
    <property key="step.startDate" value="START_DATE"/>
    <property key="step.finishDate" value="FINISH_DATE"/>
    <property key="step.dueDate" value="DUE_DATE"/>
    <property key="step.status" value="STATUS"/>
    <property key="step.previousId" value="PREVIOUS_ID"/>
  </persistence>
```

The most important configuration setting is the JNDI datasource, in this case `jdbc/Defaultds`. You must use the same name that you've used to configure the application server's datasource.

The other parameters map the tables, column names, and sequences to be used with the JDBC-backed store. If you didn't modify the included SQL scripts, then these values will work right out of the box. Then you should take a look at the `entry` and `step` sequences, which vary a lot across database vendors.

Search for SQL scripts tailored to different databases in the `src\etc\deployment\jdbc` directory.

After you have executed the SQL script and configured the `osworkflow.xml` file, all the new `Workflow` instance and associated data will be stored in the database.

PropertySet Persistence

If you're going to use the `PropertySet` functionality within your `WorkflowDescriptor` and you want to persist the variables, then you must configure the `PropertySet` to use the database instead of the system's memory.

You can configure the `PropertySet` persistence mechanism in the `propertyset.xml` file in the classpath. If you are using a JDBC-based store for persisting instance data, then you should use the `PropertySet`'s counterpart, the `JDBCPropertySet`. As you might have guessed, it stores the `PropertySet` data in a couple of database tables.

Configuring the JDBC PropertySet

Here is the configuration (`propertyset.xml`) to use a JDBC-based `PropertySet`:

```
  <propertysets>
    <propertyset name="jdbc" class=
        "com.opensymphony.module.propertyset.database.JDBCPropertySet">
      <arg name="datasource" value="jdbc/DefaultDS"/>
      <arg name="table.name" value="OS_PROPERTYENTRY"/>
      <arg name="col.globalKey" value="GLOBAL_KEY"/>
      <arg name="col.itemKey" value="ITEM_KEY"/>
      <arg name="col.itemType" value="ITEM_TYPE"/>
      <arg name="col.string" value="STRING_VALUE"/>
      <arg name="col.date" value="DATE_VALUE"/>
      <arg name="col.data" value="DATA_VALUE"/>
      <arg name="col.float" value="FLOAT_VALUE"/>
      <arg name="col.number" value="NUMBER_VALUE"/>
    </propertyset>
  </propertysets>
```

This snippet is self-descriptive and includes the usual JNDI datasource name, the table, and each of the column names.

Unit Testing your Workflow

After constructing and changing your business processes, you will want to validate the functionality and flow. This section explains how to use the JUnit framework to verify the correctness and completeness of your business process. This verification is called *unit testing*.

What is JUnit?

JUnit is a unit-testing framework for Java. JUnit is based on a `TestCase` concept: each `TestCase` contains a set of assertions; if any of these assertions fail, the `TestCase` fails. To run unit tests you need to download JUnit from `http://junit.org/index.html`. Unpack the distribution and copy `junit.jar` to your classpath; that's the only file we need to run our example (in addition to the OSWorkflow libraries we've used before).

For this example, we'll build a JUnit `TestCase` with a set of assertions about the current steps and available actions of a sample `WorkflowDescriptor`. You can extend this example with your own set of assertions, as they vary across business processes. Here's the `TestCase` code:

```
package packtpub.osw;

import java.util.Collection;
import java.util.HashMap;
import junit.framework.TestCase;
import com.opensymphony.workflow.Workflow;
import com.opensymphony.workflow.basic.BasicWorkflow;
import com.opensymphony.workflow.config.Configuration;
import com.opensymphony.workflow.config.DefaultConfiguration;
import com.opensymphony.workflow.spi.Step;
```

```
/**
 * Basic workflow testcase
 */
public class WorkflowTestCase extends TestCase
{
  private Workflow workflow;
  private long workflowId;
    /** Creates a workflow instance for testing. **/
    public void setUp()
    {
      final String wfName = "holiday2";
      workflow = new BasicWorkflow("test");
      Configuration config = new DefaultConfiguration();
      workflow.setConfiguration(config);
      try
      {
        workflowId = workflow.initialize(wfName, 100, new HashMap());
      } catch (Exception e)
        {
          e.printStackTrace();
        }
    }

    public void testWorkflow()
    {
      try
        {
          int[] availableActions =
                       workflow.getAvailableActions(workflowId,null);
          assertEquals("Unexpected number of available actions", 1,
                                          availableActions.length);
          assertEquals("Unexpected available action", 1,
                                              availableActions[0]);
          Collection currentSteps =
                                workflow.getCurrentSteps(workflowId);
          assertEquals("Unexpected number of current steps", 1,
                                              currentSteps.size());
          Step currentStep = (Step) currentSteps.iterator().next();
          assertEquals("Unexpected current step", 1,
                                          currentStep.getStepId());
        } catch (Exception e)
          {
            e.printStackTrace();
            fail();
          }
    }
}
```

The `setup()` method is the first thing to be executed by JUnit by convention. Our test extends from `TestCase` as every JUnit test does, and the `testWorkflow()` method is the one executed by the framework after the `setup()` method. All methods that start with the name "test" will be executed as part of the `TestCase`. In the `testWorkflow()` method, you'll notice several `assertEquals` invocations; these are the JUnit assertions. For example, take the following block of code:

```
int[] availableActions =
             workflow.getAvailableActions(workflowId,null);
assertEquals("Unexpected number of available actions", 1,
                                   availableActions.length);
assertEquals("Unexpected available action", 1,
                                       availableActions[0]);
```

First, we will query the available actions of the `Workflow` instance, which we created in the `setUp()` method. Then, we will test the assertion of the number of available actions (in this case, it's just one) and the identifier of the available action (in this case, `1`).

The second block checks that the new instance is in exactly one current step and this step has the identifier `1`:

```
Collection currentSteps =
                      workflow.getCurrentSteps(workflowId);
assertEquals("Unexpected number of current steps", 1,
                                       currentSteps.size());
Step currentStep = (Step) currentSteps.iterator().next();
assertEquals("Unexpected current step", 1,
                                   currentStep.getStepId());
```

Finally in the catch block, there's one `fail()` method to cancel the test if anything goes wrong.

```
} catch (Exception e)
  {
    e.printStackTrace();
    fail();
  }
```

When we are done with coding the unit test, it's time to run it, and verify the assertion, thus validating the user requirements about the business process.

Running the Tests

The JUnit testing framework is made up of only one JAR file, `junit.jar`. To run the `TestCase`, you must have this JAR in your classpath and must execute the following command:

```
C:\org.junit_3.8.1>java -cp junit.jar;osworkflow-2.8.0.jar;
commons-logging.jar;
propertyset-1.4.jar
                  junit.textui.TestRunner.packtpub.osw.WorkflowTestCase;
```

This command will invoke the JUnit default `TestRunner` on our `packtpub.osw.WorkflowTestCase`. `TestRunner` is a class responsible for executing each `TestCase` and returning the success or failure code of each one. JUnit has several `TestRunners`, some text-based and others graphical. Refer to the JUnit documentation for more details.

The output of the previous command is as follows:

```
C:\org.junit_3.8.1>java -cp junit.jar;osworkflow-2.8.0.jar;
commons-logging.jar;
propertyset-1.4.jar
                  junit.textui.TestRunner.packtpub.osw.WorkflowTestCase;
Time: 0,25
OK (1 test)
C: \org.junit_3.8.1>
```

The `TestRunner` tells us that the test finished OK with no failures. This indicates that the process definition is complete enough to cover all the user requirements. You should run this unit test every time you make changes to the business process descriptor. This assures that the requirements are fulfilled and serves as a regression testing.

Integrating with Spring

In this section we'll discuss the integration of OSWorkflow with the Spring lightweight object container.

Spring is an object container, specifically an **Inversion of Control** (**IoC**) container. IoC containers manage their component's dependencies and lifecycle. Component dependencies are managed declaratively via injection. This way each component only knows its dependency interface but not its implementation. The implementation is the one instantiated by the container and set to the component as an interface, so you don't need to create new object dependencies inside your code. This means no more use of the `new` keyword in Java.

The Object Registry—BeanFactory

Spring uses the concept of a BeanFactory. This BeanFactory is an application-wide registry and manages components. It is responsible for instantiating and injecting objects and their dependencies.

OSWorkflow can be integrated with the Spring container as a bean in the BeanFactory. In this way you can declaratively manage OSWorkflow dependencies.

In addition to this native integration, OSWorkflow can utilize Spring-managed beans for `Functions`, `Conditions`, and other components.

The current Spring version supported by OSWorkflow is 2.0. To download the Spring Framework, go to `www.springframework.org`. To include Spring in your application, just put the `spring.jar` file in the classpath. For each module you use, several different third-party libraries are required; in this example, only the `hibernate3.jar` file is needed.

The Spring BeanFactory's beans are usually defined in an XML file called `BeanFactory.xml`. This file must reside in the classpath and contains each bean and its dependencies declarations. A sample `BeanFactory.xml` file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
               "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="Employee" class="packtpub.osw.Employee">
    <property name="salary">
      <value>1000</value>
    </property>
  </bean>
</beans>
```

The `beans` tag is the root element of the XML; all `bean` tags must be nested inside it. The `bean` tag declares a new bean inside the BeanFactory. The `id` attribute serves as a unique identifier inside the BeanFactory and the `class` attribute marks the Java class to be instantiated by reflection.

The `property` tag tells Spring to set a value to the JavaBean `name` attribute, in this case `salary`. The `value` element nested inside the `property` tag defines the value to be set, in this case `1000`. This value can be converted automatically by Spring, depending on the type of the target JavaBean `property`.

Note that the Spring convention is to create only once instance per component—a singleton model. To override this behavior, you must set the bean element's `singleton` attribute to false.

OSWorkflow includes a sample `BeanFactory.xml` named `osworkflow-spring.xml` along with its distribution to show how to incorporate OSWorkflow functionality into your Spring-based application. This file lacks a Hibernate `SessionFactory`, so it's not functional out of the box. It's important to understand each bean in this file, so let's go fragment by fragment:

```
<bean id="workflowStore" class=
                              "com.opensymphony.workflow.spi.hibernate.
                                         SpringHibernateWorkflowStore">
    <property name="sessionFactory">
      <ref bean="sessionFactory"/>
    </property>
</bean>
```

The first bean is the `WorkflowStore`. Its implementation, the `SpringHibernateWorkflowStore` uses Hibernate for persistence and joins the current Spring transaction by default. It has one mandatory JavaBean `property` to be set, which is the Hibernate 3 `SessionFactory`.

After the `WorkflowStore` bean, comes the `SpringWorkflowFactory` that extends the default `XMLWorkflowFactory` and enables the loading of the configuration directly from the container. The definition is as follows:

```
<bean id="workflowFactory" class="com.opensymphony.workflow.loader.
                             SpringWorkflowFactory" init-method="init">
  <property name="resource">
    <value>workflows.xml</value>
  </property>
  <property name="reload">
    <value>true</value>
  </property>
</bean>
```

You will notice an `init-method` attribute. This tells Spring to call the method with the same name as the attribute immediately after creating the bean, in this case the `init` method. The following fragment below shows the definition of the `SpringConfiguration`:

```
<bean id="osworkflowConfiguration" class=
              "com.opensymphony.workflow.config.SpringConfiguration">
  <property name="store">
    <ref local="workflowStore"/>
  </property>
  <property name="factory">
    <ref local="workflowFactory"/>
  </property>
</bean>
```

Remember that the `Configuration` interface plays a coordination role between the `WorkflowStore` (which manages instance data) and the `WorkflowFactory` (which loads the template definitions). So it's natural to see the two mandatory properties of the `SpringConfiguration`, a `WorkflowStore` and a `WorkflowFactory`. The two previous bean definitions are referenced using the `ref` element.

Lastly, you must let Spring manage the `Workflow` implementation of your choice. In the following code snippet, we will define the `BasicWorkflow` implementation.

```
<bean id="workflow" class="com.opensymphony.workflow.basic.
                                BasicWorkflow" singleton="false">
  <property name="configuration">
    <ref local="osworkflowConfiguration"/>
  </property>
</bean>
```

Note that the bean definition is a prototype one, that is, a bean with the `singleton` attribute set to false. It is created every time your code calls the `BeanFactory` and requests the `workflow` bean. This is a very important concept for you to grasp: Spring creates only one instance of each bean by default.

The two JavaBean properties are the `configuration` (mandatory) and the `typeresolver` (optional).

Lastly, we will add a Hibernate `SessionFactory` declaration to the XML:

```
<bean id="dataSource" class=
       "org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
      <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
      <value>jdbc:hsqldb:data/osworkflow</value>
    </property>
    <property name="username">
      <value>sa</value>
    </property>
    <property name="password">
      <value></value>
    </property>
</bean>

<bean id="sessionFactory" class=
        "org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource">
      <ref local="dataSource"/>
    </property>
    <property name="mappingResources">
      <list>
        <value>com/opensymphony/workflow/spi/hibernate3/
                                HibernateCurrentStep.hbm.xml</value>
        <value>com/opensymphony/workflow/spi/hibernate3/
                                HibernateHistoryStep.hbm.xml</value>
        <value>com/opensymphony/workflow/spi/hibernate3/
                              HibernateWorkflowEntry.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
        </prop>
      </props>
    </property>
</bean>
```

This creates a new Hibernate `SessionFactory` available to the `WorkflowStore` defined in the first fragment. But before we declare a `SessionFactory`, we must define a datasource (in this case using the HSQL database) on which the OSWorkflow instance data resides; this is the purpose of the `dataSource` bean.

The `SessionFactory` defines some Hibernate mapping files (using the `list` element) included with OSWorkflow to map the `entry` and `step` objects to the corresponding tables.

This definition is not currently included in the OSWorkflow distribution, and you must manually merge the `osworkflow-spring.xml` file with it.

Using our BeanFactory

Once the XML of the `BeanFactory` has been set up, you can invoke OSWorkflow functionality inside your Spring application. The following code snippet shows you how:

```
package packtpub.osw;

import java.util.Collections;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.opensymphony.workflow.Workflow;

public class SpringExample
```

```
{
  public static void main(String[] args)
  {
    XmlBeanFactory beanFactory =new XmlBeanFactory(
    new ClassPathResource("osworkflow-spring-hibernate3.xml");
    Workflow workflow = (Workflow) beanFactory.getBean("workflow");
    try
    {
      workflow.initialize("example", 100, null);
    } catch (Exception e)
      {
        e.printStackTrace();
      }
  }
}
```

This example initializes the Spring `BeanFactory`, and then gets a `workflow` bean. Note that the code is using the `Workflow` interface and never calls the actual `BasicWorkflow` implementation. This really decouples our code from the implementation details, leaving more room for the more important things such as business logic.

This code is much simpler than the other versions shown before. You don't have to care about creating or looking up `DataSource`, `OSWorkflow Configuration`, and `SessionFactory` or instantiating new `Workflow` implementations. We also get rid of the `workflows.xml` file by uniting all under the same XML file.

Transaction Support in Spring

Before we use Spring and OSWorkflow in production, we must define some important things such as transactions.

Spring's transaction manager can use different strategies, such as JTA global transactions, JDBC local transactions, or no transactions at all.

JTA is a standard J2EE API capable of creating and synchronizing transactions across different systems. Most of the popular J2EE application servers, such as JBoss, include a JTA subsystem.

The Spring container can also make use of AOP. AOP is a new programming technique, which simplifies the programming of applications by factoring out cross concerns such as logging code, transaction handling code, etc.

If you need to make use of transactions during use of Workflow instances, then you must include a transactional aspect and weave it into the OSWorkflow code. This weaving is done transparently by the container. The transactional aspect is another bean referencing the OSWorkflow `Workflow` bean. Aspects are also called interceptors.

```
<bean id="transactionInterceptor" class="org.springframework.
                     transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager">
    <ref local="transactionManager"/>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>

<bean id="workflow" class=
                "org.springframework.aop.framework.ProxyFactoryBean">
  <property name="singleton">
    <value>false</value>
  </property>
  <property name="proxyInterfaces">
    <value>com.opensymphony.workflow.Workflow</value>
  </property>
  <property name="interceptorNames">
    <list>
      <value>transactionInterceptor</value>
      <value>workflowTarget</value>
    </list>
  </property>
</bean>

<bean id="workflowTarget" class=
   "com.opensymphony.workflow.basic.BasicWorkflow" singleton="false">
  <constructor-arg>
    <value>test</value>
  </constructor-arg>
  <property name="configuration">
    <ref local="osworkflowConfiguration"/>
  </property>
</bean>
```

Now the original bean is substituted by a proxy, which wraps each `Workflow` interface method with transaction handling code (the `TransactionInterceptor` bean). The original `workflow` bean is now called the target bean of the interceptor.

With this addition, the `initialize()` method of the example code block in the previous section would run on its own transaction, due to the `PROPAGATION_REQUIRED` transaction attribute for all the `Workflow` interface methods.

The SpringTypeResolver

By using a `SpringTypeResolver` you can describe the beans that implement the `Function` or other resolver-supported interfaces and reference them by name inside the `WorkflowDescriptor`. To define a resolver, add this bean element to the BeanFactory XML:

```
<bean id="workflowTypeResolver"
          class="com.opensymphony.workflow.util.SpringTypeResolver"/>
```

Then, modify the `workflow` bean definition by adding a new `property` element:

```
<bean id="workflow"
                class="com.opensymphony.workflow.basic.BasicWorkflow"
                                                  singleton="false">
    <property name="configuration">
      <ref local="osworkflowConfiguration"/>
    </property>
    <property name="resolver">
      <ref local="workflowTypeResolver"/>
    </property>
</bean>
```

After that, add a `Function` bean that looks like this in the BeanFactory XML:

```
<bean id="SampleBusinessLogicFunction"
       class="com.packtpub.logic.SampleBizLogic" singleton="false" />
```

It can be declared and used in the `WorkflowDescriptor` in the following way:

```
<function type="spring">
  <arg name="bean.name">SampleBusinessLogicFunction</arg>
</function>
```

The new "spring" function type signals the `WorkflowDescriptor` to resolve the bean name with the Spring BeanFactory via a call to the `BeanFactory.getBean()` method with the bean name as the method parameter. The `SampleBusinessLogicFunction` would obviously have to implement the classic `FunctionProvider` interface.

The resolver is very useful for decoupling the function definition from the actual function implementation.

This section doesn't try to be a tutorial for Spring. On the contrary it hardly brushes the surface of all Spring features. Refer to the Spring project documentation for more details.

Workflow Security

Every business process defines proper roles for each activity or step; for example only managers can sign a check over 10, 000 dollars, or only the person who initiated the process can finish it by approving or rejecting something.

OSWorkflow makes security very flexible for the programmer by discriminating step permissions and actions restrictions independently, and using the concept of step ownership to assign tasks directly to some users.

In addition to this, OSWorkflow relies on the OSUser open-source component to manage user authentication and authorization. OSUser has very powerful extension mechanisms; but you are not bound to it, OSWorkflow can use any security package by using `Conditions` for instance.

First we'll cover step permissions, which allow us to define status or group conditions to restrict entering any workflow process step.

Step Permissions

The first and basic security measure is the step permission. The step permission denies or allows entry to the step to the current user by means of one or more `Conditions`. Let's see an example:

```
<step id="1" name="First Draft">
  <external-permissions>
    <permission name="permA">
      <restrict-to>
        <conditions type="AND">
          <condition type="class">
            <arg name="class.name">
            com.opensymphony.workflow.util.StatusCondition
            </arg>
            <arg name="status">Underway
            </arg>
          </condition>
            <condition type="class">
```

```
                 <arg name="class.name">
                 com.opensymphony.workflow.util.AllowOwnerOnlyCondition
                 </arg>
               </condition>
           </conditions>
         </restrict-to>
       </permission>
     </external-permissions>
       <actions>
   …
   </step>
```

The external-permissions element is applicable inside the step element. It contains one or more named permissions, which are restrictions nesting conditions. These conditions are evaluated; if they are true, the user has permission, otherwise the user cannot enter the step. Also, if the permission fails, the user has no available actions from that step.

In this example, if the workflow status is Underway and the owner is invoking the process, the permission predicate evaluates to true, enabling access to the user.

You can query the current permissions needed for the execution of the step by calling the getSecurityPermissions() method, which receives the workflow identifier and an inputs map:

```
List perms = workflow.getSecurityPermissions(workflowId, null);
```

This method returns a java.util.List of permission names in string form.

Action Restrictions

Sometimes a lot of users have access to the step, but each role has an action dedicated to it. For securing individual actions there are action restrictions. A restriction is simply a condition that must be met for the user to execute the action. Take a look at the following descriptor snippet:

```
<action id="2" name="Sign Up For Editing">
  <restrict-to>
    <conditions type="AND">
      <condition type="class">
        <arg name="class.name">
        com.opensymphony.workflow.util.StatusCondition
        </arg>
        <arg name="status">Queued</arg>
      </condition>
```

```
            <condition type="class">
            <arg name="class.name">
            com.opensymphony.workflow.util.OSUserGroupCondition
            </arg>
            <arg name="group">bars
            </arg>
            </condition>
        </conditions>
      </restrict-to>
    </action>
```

In this example, all the conditions must evaluate to true (`AND` operator) and the action will became available to the user when `getAvailableActions()` is called.

Step Ownership

Every step has an attribute called the `owner`. This attribute is useful for assigning ownership of a step to a user. In this way you can define `Conditions` that require access to the step owner or you can query the step by its owner.

The `owner` attribute for the step is set in the `result` that provoked the transition to it. The `owner` attribute too is subjected to variable interpolation. The next descriptor fragment shows an `unconditional-result` that tells the engine to go to step 2 and set the owner of step 2 to the same name as that of the current user:

```
<results>
<unconditional-result old-status="Finished" step="2"
                                                owner="${caller}"/>
</results>
```

Extending User and Group Authentication and Authorization

By default, OSWorkflow will look up an OSUser user object. It also has several built-in conditions to handle this type of users.

```
UserManager um = UserManager.getInstance();
User test = um.createUser("jdoe");
test.setPassword("test");
Group  foos = um.createGroup("foos");
Group  bars = um.createGroup("bars");
Group  bazs = um.createGroup("bazs");

   test.addToGroup(foos);
```

```
        test.addToGroup(bars);
        test.addToGroup(bazs);

    workflow = new BasicWorkflow("jdoe");
```

OSUser has a singleton `UserManager` responsible for managing users and groups. This code snippet creates a user named `jdoe` with a password, and assigns it to three groups.

After that a new workflow is instantiated with the user's name. OSWorkflow automatically binds the OSUser user and his or her built-in user.

For more advanced security requirements and to follow the security architecture your company has you have to extend OSUser. OSUser supports a very large range of pluggable providers for each function:

- **Credentials**: The process of verifying that the user is authentic.
- **Access Control**: This is used for determining whether a user is allowed to perform a certain task.
- **Profile**: This has personal details and data associated with the user.
- **Management**: This allows the underlying data to be modified.

The `osuser.xml` file is the main configuration file for OSUser. Here, you can configure the different built-in pluggable providers or a custom made one.

```
<opensymphony-user>
    <provider class="com.opensymphony.user.provider.memory.
                                            MemoryAccessProvider" />
    <provider class="com.opensymphony.user.provider.memory.
                                       MemoryCredentialsProvider" />
    <provider class="com.opensymphony.user.provider.memory.
                                           MemoryProfileProvider" />
    <authenticator class="com.opensymphony.user.authenticator.
                                              SmartAuthenticator" />
</opensymphony-user>
```

OSUser has built-in providers for LDAP, plain files, UNIX, and Windows NT users, PAM, JAAS, and JDBC.

By using OSUser you can extend the security of OSWorkflow. If none of the security providers suits your needs, you can create a new one or alternatively create a new security mechanism inside OSWorkflow.

Summary

This chapter covered a lot of ground; first we learned how to configure OSWorkflow to load XML descriptors, and then we took a very through view of the OSWorkflow API to use it inside our applications as an embedded workflow engine.

Later we saw the persistence alternatives OSWorkflow has to store the workflow descriptor and instance data. We also saw the JUnit unit-testing framework that allows us to verify the correctness and validate the functional requirements of our business processes.

We saw that Spring enables us to decouple our application with clear separation of concerns and declarative transactions and security. OSWorkflow integrates seamlessly with Spring benefiting from of all it features.

The chapter ended with the description of the different built-in security mechanisms of OSWorkflow such as actions and step restrictions. We also learned how to extend the OSWorkflow user and group directory by using OSUser.

The next chapter is about the JBoss Rules engine, a very efficient way to decouple and reuse the business logic inside our business processes.