Executable business processes in BPMN 2.0

Activiti IN ACTION

Tijs Rademakers

Forewords by Tom Baeyens
and Joram Barrez

SAMPLE CHAPTER

MANNING

Activiti in Action

by Tijs Rademakers

Chapter 4

brief contents

4 Working with the Activiti process engine

This chapter covers

- Setting up a development environment
- Understanding the Activiti API
- Implementing processes with plain Java
- Using Spring with Activiti

It's time to take a look at the core asset of the Activiti platform, the Activiti process engine. We already looked at a simple example in chapter 1 and at the Activiti tool stack in chapter 3, but, in this chapter, we'll discuss how you can use the Activiti Java API to interact with and use the process engine in a lot more detail.

To develop business process applications, you first have to set up a decent development environment, including a Maven configuration. We'll cover this first. Then, we'll take a look at the Activiti API, which will provide the necessary interfaces to start new processes, claim user tasks, and query the process engine for specific process instances, for example. After that, we'll explore the Java service tasks of Activiti, which provide a way to implement BPMN 2.0 processes with plain Java logic. When there's no need for web service interfaces or other external interfaces, the Java service tasks provide an easy-to-use framework to build processes. We'll also discuss how to execute these Java service tasks asynchronously. Finally, we'll look at how to apply Spring beans inside the BPMN 2.0 processes and even run the whole Activiti engine within a Spring container.

Let's get started by setting up a development environment so you can work with Activiti and explore some examples.

4.1 Creating an Activiti development environment

In this chapter, you'll be developing a lot of code snippets and unit tests. Instead of using a simple text editor, you might like to use your favorite development tool to develop processes, process logic, and unit tests. In this section, you'll be introduced to the different Activiti libraries you can use and how to set up a Maven project structure.

Logging is an important tool for understanding what's going on in a complex framework, like a process engine. First you'll learn how you can tune the log levels for your needs. Then you'll see a couple of options for running the Activiti engine.

Let's begin by taking a closer look at the Activiti library structure.

4.1.1 Getting familiar with the Activiti libraries

In chapter 3, you saw that the Activiti distribution consists of several modules, including the Activiti Explorer and the add-on components Activiti Modeler and camunda fox cycle. Each of these modules have their dependencies and project structure. In this section, we'll only focus on the Activiti Engine module, which provides the core component of the project.

But the Activiti engine also consists of several layers, as shown in figure 4.1. The first layer is the engine itself, which provides the engine interfaces we'll discuss in section 4.2 and which implements the BPMN 2.0 specification. The engine component also includes a process virtual machine abstraction, which translates the BPMN 2.0 engine logic into a state machine model, as discussed in chapter 1. This process virtual machine, therefore, is capable of supporting other process languages and provides the foundational layer of the Activiti Engine. The engine component is implemented in the activiti-engine-*version* JAR file.

An optional layer is the Spring container integration for the Activiti engine, which we'll discuss in detail in section 4.4. This layer makes the Activiti engine available for use inside a Spring container and provides functionality to invoke Spring beans directly from service tasks. This layer is provided with the activiti-spring-version JAR file that's available in the workspace/activiti-spring-examples/libs-runtime directory of the Activiti distribution.

As you can see in figure 4.1, each layer of the Activiti Engine adds a specific set of functionality. Before you can use the Activiti Engine in the development environment, the dependent libraries must also be available. In the next section, you'll see a Maven-based project structure that will provide you with the necessary dependencies. But you can also reference the library dependencies from the Activiti workspace directory. Notice that you then have to start by running the setup as described in chapter 1.

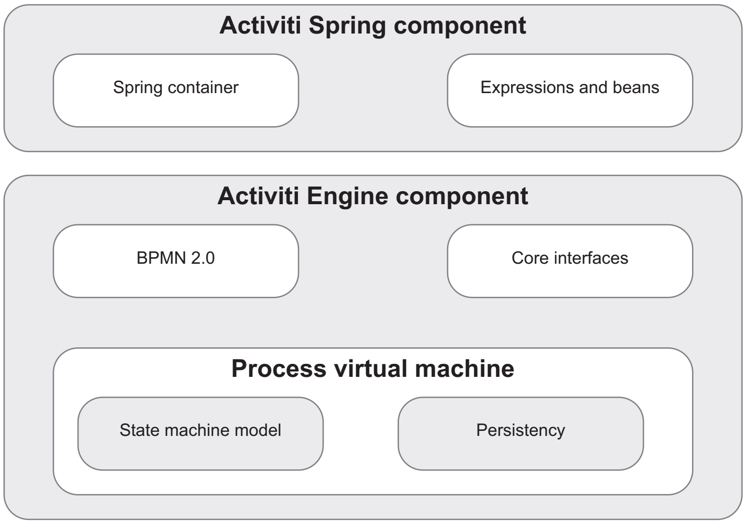

Figure 4.1 Overview of the different layers of the Activiti Engine, including the process virtual machine, the engine, and the Spring integration layers

After the Activiti installation script (see chapter 1) has been executed, you can find the Activiti Engine libraries in the workspace/activiti-engine-examples directory. The runtime libraries can be found in the libs-runtime directory, and the libraries necessary to test the examples are provided in the libs-test directory of every example project. If you don't want to use Maven for your project, you can retrieve the necessary libraries from the workspace/activiti-spring-examples directory, but, you'll see in the next section that a Maven project structure makes life a bit easier.

4.1.2 Mavenizing your Activiti project

Apache Maven can be considered the default choice for dependency management and project build management in general, and Activiti makes it easy to set up your project with Maven. In this section, you'll learn about the Maven configuration that's used in the source code of this book's examples. The examples in the Activiti distribution also have a Maven structure and a pom.xml file. To create a new Activiti project with a Maven configuration from scratch, you can create a new Maven project in the Eclipse IDE. In the presented wizard, you can fill in the necessary group and artifact identifier and choose a project name.

The pom.xml in the root of the new project needs some work; you have to add the Activiti dependencies. The following listing shows a Maven pom.xml that contains the minimal set of dependencies you need when starting an Activiti project. For a full list of all the dependencies you'll use throughout this book, you can look at the pom.xml file in the root of the `bpmn-examples` project in the book's source code.

Listing 4.1 A standard Maven configuration for an Activiti project

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.bpmnwithactiviti</groupId>
  <artifactId>your-project</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>your-project</name>

  <properties>
    <activiti-version>5.9</activiti-version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.activiti</groupId>                      1 Activiti
      <artifactId>activiti-engine</artifactId>               Engine
      <version>${activiti-version}</version>                 dependencies
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>                    2 H2 database
      <artifactId>h2</artifactId>                            driver
      <version>1.2.132</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
     <repositories>                                        3 Alfresco Maven
    <repository>                                             repository for
      <id>Activiti</id>                                      Activiti
      <url>http://maven.alfresco.com/nexus/content/
          ➥ repositories/activiti</url>
    </repository>
  </repositories>
</project>
```

In the listing, the Maven namespace declarations are left out to make the configuration more readable. To include the Activiti Engine dependencies, you only have to include the `activiti-engine` dependency 1. This dependency will also get all the third-party libraries, which are necessary to run the Activiti Engine. Notice that, in this Maven configuration, the Activiti Spring module isn't included because you don't need it for your first examples, but the module is included in the book's example source code.

To be able to test with an in-memory H2 database, you must also add the H2 database dependency 2. The H2 database dependency also provides the database driver to connect to both the standalone H2 database provided with the Activiti distribution as well as the in-memory H2 database.

Because the Activiti Engine dependency isn't yet available from a central Maven repository, you also need to add the Alfresco Maven repository for the Activiti project 3. If you're using the Eclipse development tool, you can now use the Maven Update Project Configuration menu item to create the necessary Eclipse project and classpath files.

NOTE Similar Maven archetypes are available for IntelliJ IDEA and other IDEs, and you can still use the good old command line to execute Maven commands.

Now all the Java libraries needed to run the Activiti Engine are available inside the IDE. This means that you can start implementing Activiti logic in your project. In the `bpmn-examples` project available in the book's source code, you can see that we also used a Maven configuration and defined the Activiti Engine dependencies. Now let's discuss how to tune logging in the Activiti Engine.

4.1.3 Logging in the Activiti Engine

Logging statements can help a lot when you're debugging, but they're also essential for getting good error descriptions from a production system. When you're using multiple open source frameworks in one project—like you do in this chapter with Activiti and Spring—you may run into different logging systems.

Activiti uses the standard Java java.util.logging API, also known as JDK 1.4 logging, and Spring uses Apache commons logging. This means that, by default, it's not possible to have one logging configuration file. Luckily, there's the Simple Logging Façade for Java (SLF4J—http://www.slf4j.org) framework that can translate log messages from different frameworks into the log message of your choice.

In this book, we'll use Log4J (http://logging.apache.org/log4j/1.2) as the logging system of choice, but you can easily change this to Apache Commons Logging, for example. SLF4J provides support for Log4J as well as Apache Commons Logging. For JDK 1.4 logging statements to be translated by SLF4J to Log4J, you have to do some coding.

You'll be using a lot of unit tests to work with the Activiti BPM platform, so the next code snippet shows the `AbstractTest` class you'll be extending from in every unit test to initialize your logging framework:

```
import java.util.logging.Handler;
import java.util.logging.LogManager;
import java.util.logging.Logger;

import org.activiti.engine.impl.util.LogUtil;
import org.slf4j.bridge.SLF4JBridgeHandler;

public abstract class AbstractTest {

  @BeforeClass
  public static void routeLoggingToSlf4j() {
    LogUtil.readJavaUtilLoggingConfigFromClasspath();
    Logger rootLogger =
        LogManager.getLogManager().getLogger("");
    Handler[] handlers = rootLogger.getHandlers();
    for (int i = 0; i < handlers.length; i++) {
      rootLogger.removeHandler(handlers[i]);
    }
    SLF4JBridgeHandler.install();
  }
}
```

This abstract unit test class first makes sure that the `logging.properties` for the JDK 1.4 logging of the Activiti Engine are read from the classpath. By default, the JDK 1.4 logging reads the log configuration of your JAVA_HOME/lib/logging.properties—and that's not what you want. In the logging.properties file of the `bpmn-examples` project, the log level is set to FINEST so you can get all the logging information out of the Activiti Engine when you want to.

Next in the code snippet, the log handlers are removed from the `java.util.logging.Logger` class; otherwise, the JDK 1.4 logging framework still performs the logging. At the end of the code snippet, the `install` method of the SLF4J bridge is invoked, which will direct all JDK 1.4 logging output to SLF4J. Because you have the SLF4J Log4J library on the classpath, you can now define the log level of the Activiti Engine, the Spring framework, and other external frameworks in a default log4j.xml file. With this configuration, all logging is redirected to Log4j and you can define the desired logging level in the `log4j.xml` that's available on the classpath.

This means you can define a log level of `DEBUG` when you want to do some debugging, and you can set the level to `ERROR` when you don't want extra information logged in your unit tests. In the source code examples implemented in the `bpmn-examples` project, you extend the `AbstractTest` class in all unit test classes.

Now that the logging configuration is in place, let's discuss the options available for running the Activiti Engine.

4.1.4 *Developing and testing with the Activiti Engine*

The primary component you have to deal with when designing and developing BPMN 2.0 processes with Activiti is the Activiti Engine. The engine is your entry point to deploying new process definitions, starting new process instances, querying for user tasks, and so on. But what are the options for running the Activiti Engine during development? In the following subsections, we'll discuss the following three options:

- Running the Activiti Engine in the JVM with an in-memory database (H2)
- Running the Activiti Engine in the JVM with a standalone database (H2)
- Running the Activiti Engine on an application server (Apache Tomcat) with a standalone database (H2)

Let's look at the first of these options now.

RUNNING THE ACTIVITI ENGINE WITH AN IN-MEMORY DATABASE

A good way to test a BPMN 2.0 process is to run the Activiti Engine inside the Java Virtual Machine (JVM) with an in-memory database. In this deployment scenario, the unit tests can also be run within a continuous build environment without the need for external server components. The whole process engine environment runs from within the JVM and the unit test. Figure 4.2 illustrates this method of deployment.

In the source code examples we'll discuss in the rest of this chapter, this deployment alternative is used because it's the easiest to use from within an IDE. In the next subsection, we'll take a look at another option: using a standalone database.

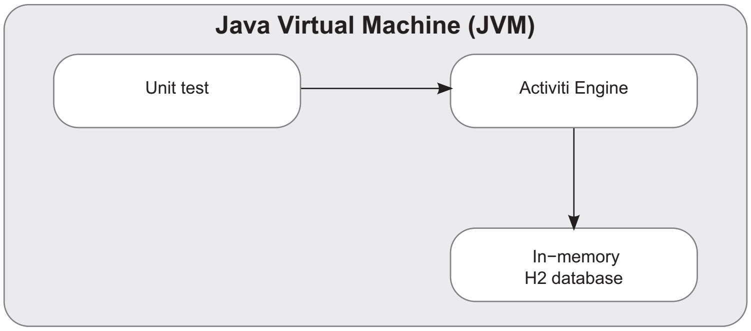

Figure 4.2 The deployment scenario where the Activiti Engine runs within the JVM with an in-memory database

RUNNING THE ACTIVITI ENGINE WITH A STANDALONE DATABASE

If you want to work with process definitions or instances that are deployed and running on a standalone environment, you need another deployment alternative. You must be able to run the Activiti Engine connected to a standalone database. This enables possibilities, such as querying the standalone database for specific running process instances. This type of deployment is shown in figure 4.3.

In the first example of this book, shown in chapter 1, you used this deployment option; an Activiti Engine is created from within a unit test and connected to a standalone H2 database. The H2 database is already installed and started as part of the Activiti installation setup. This type of setup isn't suitable for unit testing because the outcome of the unit test may vary with each run depending on what's already present in the database unless you clean the database before each run. But, it can be handy for integration testing, where you also want to use the Activiti Explorer together with a process you create from your local development environment.

RUNNING THE ACTIVITI ENGINE ON APACHE TOMCAT WITH A STANDALONE DATABASE

The previous deployment options are useful for unit and integration testing. But, eventually, you'll want to deploy your business processes on a production-like environment and do some basic testing there, too. This means that you can't start an Activiti Engine from within a unit test, because it runs on a separate application server environment.

What you can do is use the REST API provided with the Activiti Engine to interact with the process engine. The deployment of a new process definition must then be

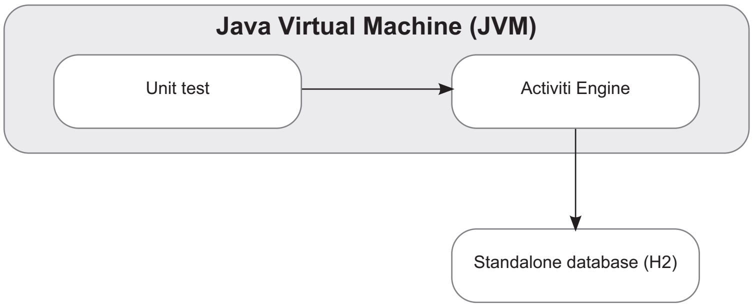

Figure 4.3 The deployment alternative where the Activiti Engine runs within the same JVM as the unit test and connects to a standalone database

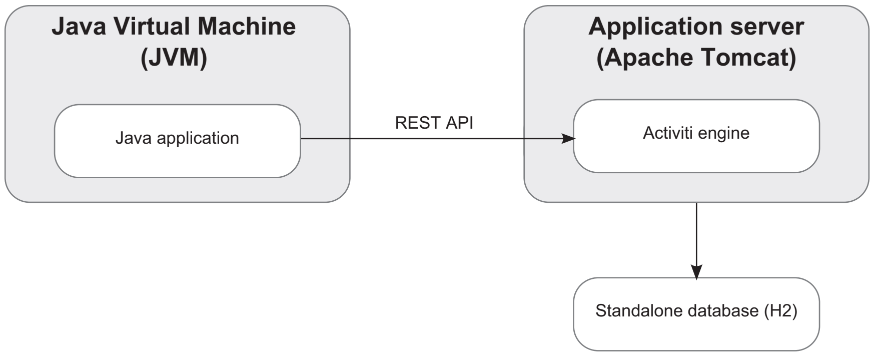

Figure 4.4 A typical Activiti Engine environment where the process engine runs on an application server, such as Apache Tomcat, with a standalone database. The REST API provides the necessary interface to interact with the process engine.

done via Activiti Explorer (like you did in chapter 3) or an Ant script by deploying a Business Archive file. This alternative is shown in figure 4.4.

In an environment like the one shown in figure 4.4, the need for unit tests is typically low because the deployment alternatives discussed earlier are more likely to be used for unit and integration testing. But, you will still need to communicate with the process engine when tools like the Activiti Explorer don't provide all the information you need or in cases where you want to communicate with the process engine from other applications. An example would be when you want to build a custom user interface for users to interact with the user tasks in a particular process. The REST API provides a great way to implement the necessary communication.

In chapter 8, we'll take close look at the possibilities of the REST API. But first, it's time to learn about the Java interfaces you can use to talk with the Activiti Engine.

4.2 Using the Activiti Engine API

The Activiti Engine API is divided into seven core interfaces, each targeted at interacting with different functionality of the process engine. Table 4.1 summarizes the core interfaces.

Table 4.1 Overview of the seven core interfaces of the Activiti API

Interface	Description
`FormService`	To work with the user task forms generated by the Activiti form engine, the form service provides several methods.
`HistoryService`	To retrieve information about completed process instances, you can use the history service interface.
`IdentityService`	The identity service provides an interface on the authentication component of the Activiti process engine.

Table 4.1 Overview of the seven core interfaces of the Activiti API *(continued)*

Interface	Description
`ManagementService`	The management service can be used to query the Activiti tables and execute jobs.
`RepositoryService`	The repository service provides functionality to deploy, query, delete, and retrieve process definitions.
`RuntimeService`	The runtime service provides an interface to start and query process instances. In addition, process variables can be retrieved and set, and processes can be signaled to leave a wait state.
`TaskService`	With the task service you can do a lot of things with user tasks. For example, you can create a new task and query Activiti for a list of tasks that a specific user can claim.

In this section, we'll discuss most of these interfaces with small and easy-to-use code examples, starting with the `RuntimeService`. We won't be covering the `FormService` and the `ManagementService` here because they provide specific functionality. The `FormService` can be used to interact with a user task or start event forms (we'll discuss this in chapter 5), and the `ManagementService` can be used to access jobs and query the Activiti tables (the job architecture is discussed in chapter 15).

4.2.1 Starting process instances with the RuntimeService

The primary usage of the `RuntimeService` is to start new process instances based on a specific process definition. But this isn't the sole purpose of this interface; it also provides simple query functionality and methods to set and retrieve process variables, among other operations.

Let's first look at how to use the `RuntimeService` to start a new process instance.

Listing 4.2 Start a new process instance with the `RuntimeService`

```
public class RuntimeServiceTest extends AbstractTest {

  private static RuntimeService runtimeService;

  @BeforeClass
  public static void init() {
    ProcessEngine processEngine =                                   (1) Creates Activiti
        ProcessEngineConfiguration                                      engine
          .createStandaloneInMemProcessEngineConfiguration()
          .buildProcessEngine();

    RepositoryService repositoryService =
        processEngine.getRepositoryService();
    repositoryService.createDeployment()                            (2) Deploys
        .addClasspathResource(                                          bookorder
            "chapter4/bookorder.bpmn20.xml")                            process
        .deploy();
    runtimeService = processEngine.getRuntimeService();
  }

  @Test
  public void startProcessInstance() {
    Map<String, Object> variableMap =
        new HashMap<String, Object>();
    variableMap.put("isbn", "123456");
    ProcessInstance processInstance =
        runtimeService.startProcessInstanceByKey(
            "bookorder", variableMap);
    assertNotNull(processInstance.getId());
    System.out.println("id " + processInstance.getId() + " "
        + processInstance.getProcessDefinitionId());
}

  @Test
  public void queryProcessInstance() {
    List<ProcessInstance> instanceList = runtimeService
        .createProcessInstanceQuery()
        .processDefinitionKey("bookorder")
        .list();

    for (ProcessInstance queryProcessInstance : instanceList) {
      assertEquals(false, queryProcessInstance.isEnded());
      System.out.println("id " + queryProcessInstance.getId() +
          ", ended=" + queryProcessInstance.isEnded());
    }
  }
}
```

3 Starts new process instance

4 Queries for running bookorder instances

To implement a unit test class with multiple test methods, it's a good practice to create the Activiti engine 1 in an `init` method annotated with `@BeforeClass`. This makes the Activiti engine available in every test method. Then the book order process used in chapter 1 is deployed on the engine 2. Figure 4.5 shows the simple book order process.

To start a new process instance for the book order process shown in figure 4.5, you can use the `startProcessInstanceByKey` method 3 of the `RuntimeService` interface. With this method, the latest version of the specified process definition name is started. You can optionally provide a map of process variables as in this example. The other way to start a new process instance is to use the `startProcessInstanceById` method, which starts a specific version of a process definition. The process definition identifier is stored within the Activiti Engine database and is provided when you deploy a process definition with the `RepositoryService`. But, most of the time, you'll want to use the `startProcessInstanceByKey` method because you want to use the latest version of the process.

In the last step of the unit test, you query the Activiti engine for running process instances of the book order process 4. Note that you use the `processDefinitionKey`

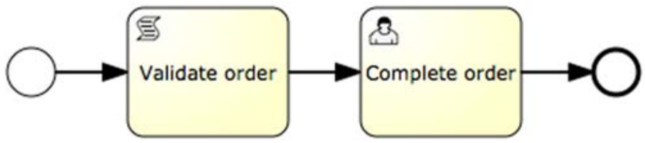

Figure 4.5 The process diagram of a simple book order process, containing a "Validate order script" task and a "Complete order user" task.

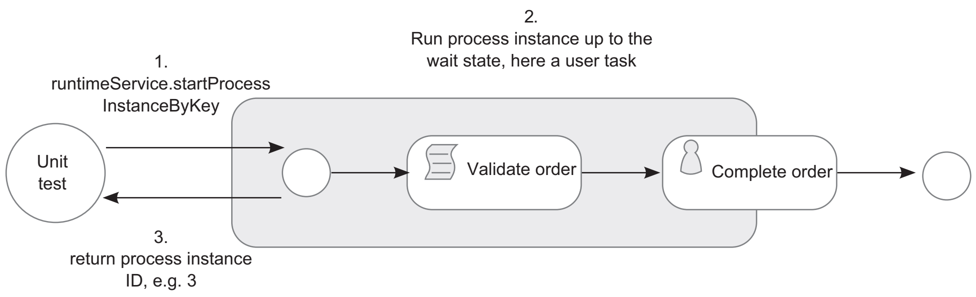

Figure 4.6 Overview of what happens with the process instance after you've started the process instance in the unit test

method here, which means all running process instances for all versions of the book order process are returned. To retrieve only the running process instances of a specific version of the book order process definition, you have to use the `processDefinitionId` method and provide the process definition identifier value.

When you run this unit test, you'll see one running process instance with the process instance query because you only started one new process instance. Let's see what happened inside the Activiti engine with the book order process instance after you started the process instance (figure 4.6).

The Activiti Engine executes the process instance immediately after the book order process is started with the `startProcessInstanceByKey` method. Because this is a synchronous execution in a single transaction and thread, the unit test will wait until the process instance identifier is returned. Activiti executes the process until a wait state is encountered. A user task is an example of such a wait state because somebody has to claim and complete the task before the process instance will proceed to the next activity. "Validate order" is a script task, which isn't a wait state and, therefore, it executes synchronously in the current thread. In section 4.3, you'll see how to define an automatic task, like a script or a Java service task, to run asynchronously.

For the unit test example, the Activiti engine executes the "Validate order" script task and initiates the "Complete order" user task. Then the wait state is activated and the process instance identifier is returned to the unit test. When you query the Activiti engine for running processes, you find exactly one instance, which has a current activity of the type `user task`. Now let's look at how to deal with this user task using the `TaskService`.

4.2.2 Working with user tasks via the TaskService

The `TaskService` provides a lot of functionality surrounding user tasks for the Activiti engine. You can, for example, use the `TaskService` to query the engine for specific tasks or to create a new standalone task for a specific user. In this section, we'll walk through most of the functionality the `TaskService` interface provides, starting with querying for running user tasks.

QUERYING FOR USER TASKS WITH THE TASKSERVICE

In the previous section, we looked at the usage of the `RuntimeService` with a rather large unit test. From now on, you'll use the unit testing functionality that the Activiti framework provides, as you'll see listing 4.3.

Unit testing with Activiti

When you want to test your process definition without a lot of plumbing code, you can use the unit testing functionality of the Activiti framework. The unit testing functionality of Activiti using JUnit 4 is centered on the use of the `ActivitiRule` class. The `ActivitiRule` class is a subclass of the JUnit `TestWatchman` class, which intercepts test method calls so it can provide the setup and teardown functionality. If you want to use JUnit 3 to create your tests, you can use the `AbstractActivitiTestCase` abstract base class.

But, first things first. At the creation of an `ActivitiRule` instance, an Activiti Engine is created using the activiti.cfg.xml configuration file found on the classpath by default. This can be overwritten if you want to. The activiti.cfg.xml configuration file contains, for example, the definition of the Activiti Engine and configures the database (embedded or standalone) that's used by the Activiti Engine. We'll discuss all the configuration options of the activiti.cfg.xml file in chapter 8.

The main usage of the `ActivitiRule` instance is that you can deploy a process definition before a test method is executed. This can be done by including the `@Deployment` annotation. By default, a process definition with the name testclassname.testmethodname.bpmn20.xml in the same package as the test class is deployed, but this can be overridden by specifying one or more bpmn20.xml files with the `resources` element of the `@Deployment` annotation. The `@Deployment` annotation also makes sure that after the test method has executed, running process instances, user tasks, and jobs are deleted. This is handy for keeping the database clean while running your unit tests.

In addition, the `ActivitiRule` instance can be used to retrieve the seven core interfaces we discuss in this section (section 4.2). You can also specify a specific `java.util.Date` with the `setCurrentTime` method, which can be used to test timers and due dates.

Listing 4.3 Querying for user tasks with the `TaskService` interface

```
public class TaskServiceTest extends AbstractTest {

  @Rule                                                    1 Initiates
  public ActivitiRule activitiRule = new ActivitiRule(       Activiti unit
      "activiti.cfg-mem.xml");                               testing

  private void startProcessInstance() {
    RuntimeService runtimeService =
        activitiRule.getRuntimeService();
    Map<String, Object> variableMap =
        new HashMap<String, Object>();
    variableMap.put("isbn", "123456");
    runtimeService.startProcessInstanceByKey(        2 Starts new
        "bookorder", variableMap);                     process instance
  }

  @Test
  @Deployment(resources={                                       ◁ ③ Deploys book
      ➥ "chapter4/bookorder.bpmn20.xml"})                          order process
  public void queryTask() {
    startProcessInstance();
    TaskService taskService = activitiRule.getTaskService();
    Task task = taskService.createTaskQuery()                    ④ Queries
        .taskCandidateGroup("sales")                                for user
        .singleResult();                                            tasks
    assertEquals("Complete order", task.getName());
    System.out.println("task id " + task.getId() +
        ", name " + task.getName() +
        ", def key " + task.getTaskDefinitionKey());
  }
}
```

This unit test makes use of the powerful unit testing functionality Activiti provides with the ActivitiRule class ①. This reduces the plumbing code necessary to test a process definition to a minimum. With the @Deployment annotation ③, the book order process definition is deployed to the Activiti engine. But, you still have to start a new process instance ② before you can proceed with the actual testing logic.

Querying for user tasks is done via the TaskQuery interface where you can, for example, specify candidate user or group criteria and define ordering instructions. In this example, you query for user tasks that can be completed by users belonging to the sales group ④. The user task defined in the book order process definition has a group definition that's equal to the name sales:

```
<userTask id="usertask1"
    name="Complete order"
    activiti:candidateGroups="sales">
  <documentation>book order user task</documentation>
</userTask>
```

Because you know that only one user task is running inside the unit test, you can use the singleResult method to return one Task instance.

Now let's move on to creating a new task and completing it.

CREATING AND COMPLETING USER TASKS VIA THE TASKSERVICE

The most common functionalities of the TaskService that you'll be using are the claim and complete methods. When a user task is created for a process instance, a person has to claim and complete the user task before the process instance proceeds to the next activity. Claiming a task means that the person who claims the task becomes the owner (assignee) of the task. It also means that the claimed task isn't available anymore for the other potential task owners to claim or complete it.

> **NOTE** The TaskService doesn't prohibit you from completing a task before it has been claimed. But, it's a best practice to claim a task with a particular user first and then complete it. This ensures that a full audit trail, including the name of the user who completed the task, is available.

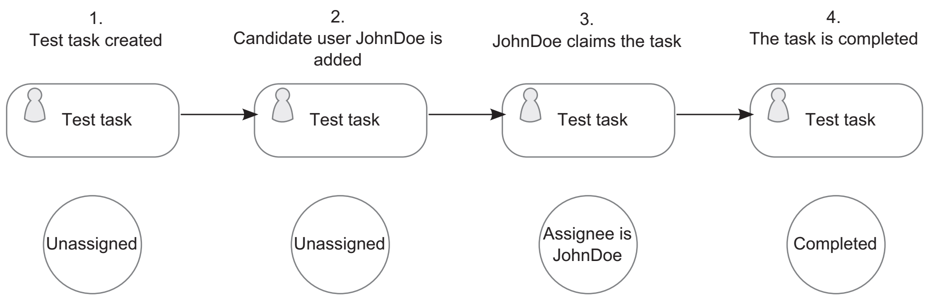

Figure 4.7 An overview of the different states a user task goes through when it's created, claimed, and completed

In the next example, you're going to create a standalone user task and claim and complete it. A standalone user task isn't bound to a specific process instance and can be created at any point in time in the Activiti Engine. The claim and complete functionality is no different for a process-bound user task. Figure 4.7 shows the different states the user task will have in this example.

The next listing implements the different states shown in figure 4.7 in a unit test method. The `createTask` test method is implemented in the same unit test class as the previous listing.

Listing 4.4 Implementation of the claim and complete functionality

```
public class TaskServiceTest extends AbstractTest {

  @Rule
  public ActivitiRule activitiRule = new ActivitiRule(
      "activiti.cfg-mem.xml");

  @Test
  public void createTask() {
    TaskService taskService = activitiRule.getTaskService();
    Task task = taskService.newTask();                          ❶ New user task created
    task.setName("Test task");
    task.setPriority(100);
    taskService.saveTask(task);
    assertNull(task.getAssignee());

    IdentityService identityService =
        activitiRule.getIdentityService();
    User user = identityService.newUser("JohnDoe");             ❷ New user added
    identityService.saveUser(user);

    taskService.addCandidateUser(task.getId(), "JohnDoe");      ❸ User task gets candidate user
    task = taskService.createTaskQuery()
        .taskCandidateUser("JohnDoe")
        .singleResult();
    assertNotNull(task);
    assertEquals("Test task", task.getName());
    assertNull(task.getAssignee());
    taskService.claim(task.getId(), "JohnDoe");          4 User task claimed
    task = taskService.createTaskQuery()
        .taskAssignee("JohnDoe")
        .singleResult();
    assertEquals("JohnDoe", task.getAssignee());

    taskService.complete(task.getId());                  5 User task completed
    task = taskService.createTaskQuery()
        .taskAssignee("JohnDoe")
        .singleResult();
    assertNull(task);
  }
}
```

To create a standalone user task, you can use the `newTask` method 1 of the `TaskService` interface. In this example, the user task name and priority are set and the task is saved with these new values.

TIP The priority attribute of a user task can be used to define the urgency of the work to be done. By default, this value is 50, but you define any value from 0 to 100 (where 100 is the highest priority level and 0 the lowest). The Activiti Engine itself doesn't use the priority attribute, but it can be used by your workflow application.

When the new user task is created, the `assignee` attribute is empty, meaning that there is no specific user allocated yet to do the work associated with the user task. To be able to claim the user task with a specific user, you first add a new user to the Activiti Engine. This is done via the `IdentityService` and the `newUser` method 2. Note that the user isn't created before the `saveUser` method is invoked. Once the John Doe user is created, you can add a candidate user to the user task 3. This means that John Doe is the candidate who'll execute the work associated with the user task.

NOTE In Activiti, there's no validation if the user who claims the user task is also part of the candidate user or group. The Engine doesn't even validate whether the user is known. This makes it easy to plug in your own identity management solution, which can, for example, be an LDAP repository. We'll take a look at various options to implement identity management in chapter 10. It's a best practice to define a list of candidate users or groups and only claim the user task with a user that's on this list. The validation logic that checks if a user exists in your identity management system and whether the user is part of a specific group must be implemented by you.

In the next step of the unit test, the user task is claimed with the John Doe user 4. Now the assignee attribute of the user task is filled with the user identifier of the claimer, which in this example, is John Doe. To complete the user task, the `complete` method is used 5. When the user task is completed, it can't be found with a task query anymore. The only way to retrieve the user task at this point is via the `HistoricActivityInstanceQuery`, which we'll discuss in section 4.2.5. First, let's look at how to delete a process definition via the `RepositoryService`.

4.2.3 Deleting process definitions with the RepositoryService

You already used the `RepositoryService` interface in section 4.2.1 to deploy a process definition; it can also be used to query the Activiti engine for deployment artifacts and process definitions. In this section, you'll also use the delete functionality, which the `RepositoryService` interface provides. Let's work through an example where you deploy a new process definition and delete it at the end (see the next listing).

Listing 4.5 Deleting a deployment with the `RepositoryService`

```
public class RepositoryServiceTest extends AbstractTest {

  @Rule
  public ActivitiRule activitiRule = new ActivitiRule(
      "activiti.cfg-mem.xml");

  @Test
  public void deleteDeployment() {
    RepositoryService repositoryService =
        activitiRule.getRepositoryService();
    String deploymentID = repositoryService.createDeployment()
        .addClasspathResource("chapter4/bookorder.bpmn20.xml")
        .deploy()
        .getId();                                           ① Deploys new process definition

    Deployment deployment = repositoryService
        .createDeploymentQuery()
        .singleResult();                                    ② Queries engine for deployments
    assertNotNull(deployment);
    assertEquals(deploymentID, deployment.getId());
    System.out.println("Found deployment " + deployment.getId()
        + ", deployed at " + deployment.getDeploymentTime());

    ProcessDefinition processDefinition = repositoryService
        .createProcessDefinitionQuery()
        .latestVersion()
        .singleResult();                                    ③ Retrieves the deployed process definition
    assertNotNull(processDefinition);
    assertEquals("bookorder", processDefinition.getKey());
    System.out.println("Found process definition " +
        processDefinition.getId());

    RuntimeService runtimeService =
        activitiRule.getRuntimeService();
    Map<String, Object> variableMap =
        new HashMap<String, Object>();
    variableMap.put("isbn", "123456");
    runtimeService.startProcessInstanceByKey(
        "bookorder", variableMap);                          ④ Starts new process instance

    ProcessInstance processInstance = runtimeService
        .createProcessInstanceQuery()
        .singleResult();
    assertNotNull(processInstance);
    assertEquals(processDefinition.getId(),
        processInstance.getProcessDefinitionId());
    repositoryService.deleteDeployment(deploymentID, true);    ◁─ Deletes process definition and instances 5

    deployment = repositoryService
        .createDeploymentQuery()
        .singleResult();
    assertNull(deployment);
    processDefinition = repositoryService
        .createProcessDefinitionQuery()
        .singleResult();
    assertNull(processDefinition);
    processInstance = runtimeService
        .createProcessInstanceQuery()
        .singleResult();
    assertNull(processInstance);
  }
}
```

This is quite a bit of coding, but, as you can see, you do a lot of querying to validate the results of the deployment activities. First, you start with deploying the book order process definition 1 like you did in section 4.2.1. The difference here is that you keep track of the deployment identifier that's generated by the Activiti Engine. This deployment identifier will be needed later on.

When the deployment has been executed, you can query the process engine for deployment artifacts with the `DeploymentQuery` 2. Because you're using an in-memory database, you'll expect to find only the deployment done in this unit test. In addition to the deployment artifact query, you can also query the engine for the latest version of the deployed process definitions via the `ProcessDefinitionQuery` 3.

NOTE A deployment can contain multiple resources, including a process definition. But, it can also contain other resources, such as a business rule and a process definition image.

Because you want to show the ability to delete the process definition, including possible running process instances and process history information, a new process instance is started 4 in the unit test. The `RepositoryService` interface provides two types of `delete` methods:

- The `deleteDeployment` method with a deployment identifier and a false input parameter, which only deletes the deployment and not the corresponding process instance data. When there are still running process instances, you'll get an exception when running this method.
- The `deleteDeployment` method with a deployment identifier and a true input parameter, which deletes all information regarding the process definition, running instances, and history. If you want all process data, including running process instances, to be deleted, you should use a Boolean value of `true` for the second input parameter.

Because you've been running process instances in this unit test, you must use the Boolean value of `true` 5, or you'll receive an exception. In the last part of the unit test, you validate that the deployment, process definition, and instance are deleted. You can execute this unit test to ensure it runs successfully. Let's move on to the IdentityService interface and see how to create new group memberships.

4.2.4 Creating users, groups, and memberships with the IdentityService

In section 4.2.2, we talked about assigning, claiming, and completing user tasks with the TaskService interface. You've already seen how to create a new user within the Activiti identity module by using the IdentityService. The IdentityService interface does provide a lot more functionality, including query functions and group membership functions. This can be handy if you want to query the Activiti Engine for users belonging to a specific group or assign users a new group membership.

In the next listing, a new user, group, and group membership are created and the newly created group membership is tested using the book order process example you've seen before.

Listing 4.6 Creating and testing a group membership

```
public class IdentityServiceTest extends AbstractTest {

  @Rule
  public ActivitiRule activitiRule =
      new ActivitiRule("activiti.cfg-mem.xml");

 @Deployment(resources = {"chapter4/bookorder.bpmn20.xml"})
  public void testMembership() {
    IdentityService identityService =
        activitiRule.getIdentityService();

    User newUser = identityService
        .newUser("John Doe");
    identityService.saveUser(newUser);                   1 Creates a new user
    User user = identityService
        .createUserQuery()                               2 Queries for all registered users
        .singleResult();
    assertEquals("John Doe", user.getId());

    Group newGroup = identityService
        .newGroup("sales");
    newGroup.setName("Sales");
    identityService.saveGroup(newGroup);                 3 Creates a new group
    Group group = identityService
        .createGroupQuery()
        .singleResult();
    assertEquals("Sales", group.getName());

    identityService.createMembership("John Doe", "sales");   4 Creates a group membership

    identityService.setAuthenticatedUserId("John Doe");      5 Sets process initiator

    RuntimeService runtimeService =
        activitiRule.getRuntimeService();
    Map<String, Object> variableMap =
        new HashMap<String, Object>();
    variableMap.put("isbn", "123456");
    runtimeService.startProcessInstanceByKey(
        "bookorder", variableMap);
    TaskService taskService = activitiRule.getTaskService();
    Task task = taskService.createTaskQuery()
        .taskCandidateUser("John Doe")
        .singleResult();
    assertNotNull(task);
    assertEquals("Complete order", task.getName());
  }
}
```

6 Queries to validate group membership

You again create a new user 1, but now you also query the Engine to see if the user was created correctly 2. Because you eventually want to test whether the newly created `John Doe` user will be a candidate user for the user task in the book order process, you also create a new group 3, `sales`, which is used in the group assignment in that process.

Having created the user and group, you can now create a group membership of `John Doe` for the `sales` group 4. As you can see, this is all easy to do when using the `IdentityService` interface. The `IdentityService` also enables you to set the authenticated user 5; in this example, it's used to set the user who starts or initiates the process instance. In the process definition, a process variable can be configured whereby this user identifier will be available during process execution:

```
<startEvent id="startEvent" activiti:initiator="starter" />
```

When you start a book order process instance, you can now test whether your new user is a candidate user for the `Complete order` user task. To test this, you use a task query with a candidate user criterion, which is equal to `John Doe` 6. When the group membership has been created successfully, the retrieved user task name should be equal to `Complete order`.

It's nice to be able to start new process instances and work with user tasks using the Activiti API. But, what happens if a process instance is finished or terminated? Will you still be able to retrieve information about these process instances, such as for reporting? Yes; and that's what the next section about the `HistoryService` interface is about.

4.2.5 A sneak peek into the past with the HistoryService

When information about ended process instances is needed, or previous activities from a running execution must be retrieved, the `HistoryService` provides an interface to query this kind of data. But before we dive into a code example of how to use the `HistoryService` interface, let's first look at how the historic data about process instances and activities is stored inside the Activiti engine database. You'll again use the book order process for this and start a new process instance, as shown in figure 4.8.

Note that a historic process instance is stored right away when a new process instance is started. A query on historic process instances will also give results when all created process instances are still running; they don't have an end time yet. The database table in which you can find the historic process instances is the ACT_HI_PROCINST table. When the process instance enters its first activity state, such as `Validate order`, a record in the

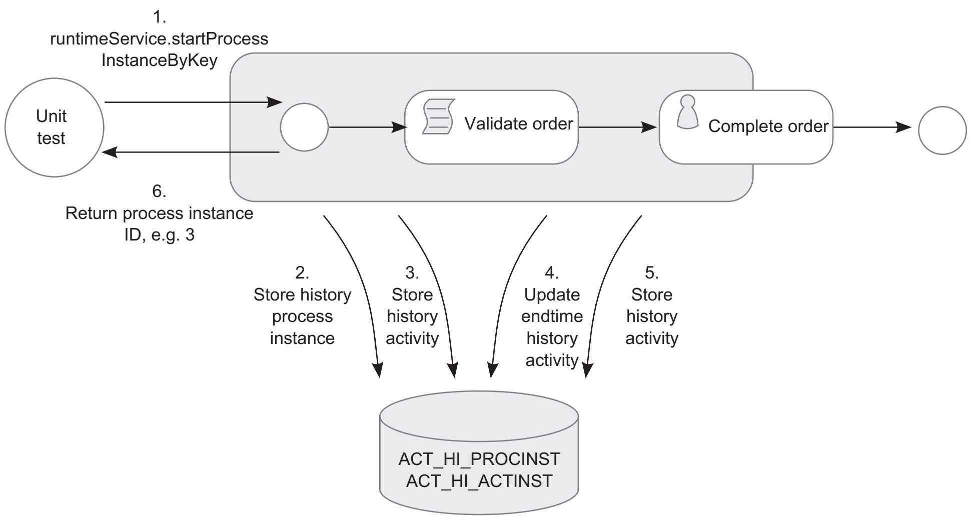

Figure 4.8 Overview of the historic data of process instances and activities, which is stored by the Activiti engine in the database when starting the example book order process

historic activity table (ACT_HI_ACTINST) is made. When the activity is finished, the record is updated with the end time of the activity. Figure 4.9 finishes the book order process example and shows what happens if the "Complete order" user task is completed.

When the user task is completed, the end time of the corresponding history activity instance is updated with the time at completion. Then the book order process instance reaches its final end state and the end time of the historic process instance is filled in. Because the process instance has finished its execution at this point, the Activiti Engine

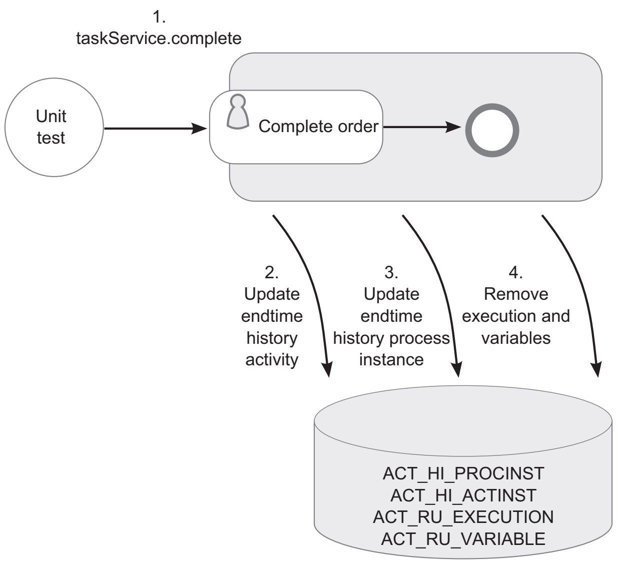

Figure 4.9 The complete story of what's stored in the Activiti engine database when the book order process user task is completed

will also delete the runtime execution information other than the history from the database. The deleted data is the execution instance stored in the ACT_RU_EXECUTION table and the process variables persisted in the ACT_RU_VARIABLE table.

The execution data is deleted to reduce the number of rows in the running process instance tables to improve performance. The historic tables don't have any foreign keys so they can be backed up easily. Note that the process variables aren't automatically stored in the history tables. The updates to process variables are only stored in a historic table, named ACT_HI_DETAIL, when you set the level of historic information to keep to `full`. We'll discuss how to configure the level of historic information that's logged shortly.

First, let's look at a unit test that uses the `HistoryService` interface to query the Activiti Engine for history process instances and history activities.

Listing 4.7 Querying for process instances and historic activities

```
public class HistoryServiceTest extends AbstractTest {

  @Rule
  public ActivitiRule activitiRule = new ActivitiRule(
      "activiti.cfg-mem-fullhistory.xml");

  private String startAndComplete() {
    RuntimeService runtimeService =
        activitiRule.getRuntimeService();
    Map<String, Object> variableMap =
        new HashMap<String, Object>();
    variableMap.put("isbn", "123456");
    String processInstanceID = runtimeService                  ① Starts a
        .startProcessInstanceByKey(                              new process
            "bookorder", variableMap)                            instance
        .getId();

    TaskService taskService = activitiRule.getTaskService();
    Task task = taskService.createTaskQuery()
        .taskCandidateGroup("sales")
        .singleResult();
    variableMap = new HashMap<String, Object>();
    variableMap.put("extraInfo", "Extra information");      ② Completes a
    variableMap.put("isbn", "654321");                          user task with
    taskService.complete(task.getId(), variableMap);            variables
    return processInstanceID;
  }

  @Test
  @Deployment(resources={"chapter4/bookorder.bpmn20.xml"})
  public void queryHistoricInstances() {
    String processInstanceID = startAndcomplete();
    HistoryService historyService =
        activitiRule.getHistoryService();
    HistoricProcessInstance historicProcessInstance =
        historyService                                        ③ Queries for
            .createHistoricProcessInstanceQuery()               historic
            .processInstanceId(processInstanceID)               process
            .singleResult();                                    instances
    assertNotNull(historicProcessInstance);
    assertEquals(processInstanceID, historicProcessInstance
        .getId());
    System.out.println("history process with definition id " +
        historicProcessInstance.getProcessDefinitionId() +
        ", started at " +
            historicProcessInstance.getStartTime() +
        ", ended at " + historicProcessInstance.getEndTime() +
        ", duration was " +
           historicProcessInstance.getDurationInMillis());
  }

  @Test
  @Deployment(resources={"chapter4/bookorder.bpmn20.xml"})
  public void queryHistoricActivities() {
    startAndcomplete();
    HistoryService historyService =
        activitiRule.getHistoryService();
    List<HistoricActivityInstance> activityList =
        historyService
            .createHistoricActivityInstanceQuery()
            .list();
    assertEquals(3, activityList.size());
    for (HistoricActivityInstance historicActivityInstance :
        activityList) {
      assertNotNull(historicActivityInstance.getActivityId());
      System.out.println("history activity " +
          historicActivityInstance.getActivityName() +
          ", type " +
              historicActivityInstance.getActivityType() +
          ", duration was " +
              historicActivityInstance.getDurationInMillis());
    }
  }
}
```

4 Queries for historic activities

The `startAndComplete` method starts a new process instance 1 and completes the user task with an update to the `isbn` process variable and the addition of a new process variable `extraInfo` 2. This corresponds to the execution logic in figures 4.8 and 4.9. The variables that are passed onto the process instance at the user task completion will be used later on, in listing 4.8.

In the first unit test implemented using the `HistoricProcessInstanceQuery`, the historic process instance started and completed in the `startAndComplete` method is retrieved 3. Note that `HistoricProcessInstanceQuery` would also have returned the book order process instance if the user task wasn't completed and the process instance was still running, as illustrated in figure 4.4. The information that can be retrieved from a `HistoricProcessInstance` is basic; for example, the start and end times.

More interesting is the information that can be retrieved via the `HistoricActivity-InstanceQuery` 4, which can provide a list of activities that have been executed by the Activiti Engine. In this example, the query will return three activities: the start event plus the “Validate order” and “Complete order” tasks from the book order process definition. This kind of information can be handy when you want to see the audit trail whose route has been executed in a specific process instance.

In this example, the default history settings of Activiti were used. But you can configure four levels of history archiving:

- *None*—No history information is archived.
- *Activity*—All process and activity instance information is archived.
- *Audit (default)*—All process, activity instance, and form properties information is archived.
- *Full*—The highest level of archiving; all audit information is archived and, additionally, the updates to process variables and user task form properties are stored.

When you don’t want to use the default setting of `audit` for history archiving, you can specify an alternative value in the Activiti configuration file, which by default is activiti.cfg.xml. To do this, add the following property to the process engine configuration:

```
<property name="history" value="full" />
```

In this example, you’ve specified the highest level of history archiving, but this can be any of the four levels mentioned previously. In the highest level, the updates to process variables are logged in the history table ACT_HI_DETAIL. The next listing shows a unit test method—the same `HistoryServiceTest` class used in listing 4.7—which retrieves these process variable updates.

Listing 4.8 Retrieving process variable updates with the `HistoryService` interface

```
@Test
@Deployment(resources={"chapter4/bookorder.bpmn20.xml"})
public void queryHistoricVariableUpdates() {
  startAndComplete();
  HistoryService historyService =
      activitiRule.getHistoryService();
  List<HistoricDetail> historicVariableUpdateList =
      historyService
          .createHistoricDetailQuery()
          .variableUpdates()
          .list();                                    ① Queries process variable updates
  assertNotNull(historicVariableUpdateList);
  assertEquals(3, historicVariableUpdateList.size());
  for (HistoricDetail historicDetail :
      historicVariableUpdateList) {
    assertTrue(historicDetail instanceof
        HistoricVariableUpdate);                      ② HistoricVariableUpdate for process variable updates
    HistoricVariableUpdate historicVariableUpdate =
        (HistoricVariableUpdate) historicDetail;
    assertNotNull(historicVariableUpdate.getExecutionId());
    System.out.println("historic variable update,
        revision " +
            historicVariableUpdate.getRevision() +
        ", variable type name " +
            historicVariableUpdate.getVariableTypeName() +
        ", variable name " +
            historicVariableUpdate.getVariableName() +
        ", Variable value '" +
            historicVariableUpdate.getValue()+"'");          ③ Gets new process variable value
    }
}
```

When the history level is set to full, the historic detail query can be used to retrieve process variable updates ①. This means that, when a new process variable is created, an update row is created. But an update row also is created when the value of a process variable is changed, as you saw with the isbn variable in listing 4.7.

The process variable updates can be retrieved using HistoricVariableUpdate instances ②. In this unit test, you don't do a lot of testing, but you print all the variable update information for learning purposes, like the new process variable value ③. When you run this unit test, you should see the following console output:

```
historic variable update, revision 0, variable type name string, variable
     name isbn, Variable value '123456'
historic variable update, revision 1, variable type name string, variable
     name isbn, Variable value '654321'
historic variable update, revision 0, variable type name string, variable
     name extraInfo, Variable value 'Extra information'
```

The first entry is created at the start of the process instance, when the isbn process variable is set. The second entry shows a new revision of the isbn variable, created when the user task is completed. And, the same goes for the last process variable update entry. This completes our detailed discussion of the history service interface.

We haven't covered the ManagementService and FormService interfaces yet. In the previous sections, you worked with the most frequently used interfaces of the Activiti Engine. These two, less common interfaces will be discussed in chapter 5, when you'll use a task form and a boundary timer event.

Now, though, let's look at developing Java service tasks.

4.3 Using plain Java to do BPM

By now, you're familiar with the Activiti Engine API, but we haven't discussed the use of Java inside a process definition yet. In addition to the script, web service, and user tasks available to define a process, you can also use Java classes to implement the process logic. When there's no web service that can be executed to deal with business logic, you can use a Java service task to do that work.

NOTE The use of Java to implement a service task isn't standard BPMN 2.0 functionality but is provided as an add-on by the Activiti framework.

The Java service task can be used in four ways:

- Java service task class
- Java service task class with field extensions
- Java service task with method or value expressions
- A delegate expression that defines a variable that is resolved to a Java bean at runtime

In the following sections, we'll look at each of these four options with short code examples.

4.3.1 Java service task with class definition

The simplest way of using a Java service task is to create a simple Java class that extends the `JavaDelegate` convenience class and defines the fully qualified class name (package name and the class name) in the service task of the process definition.

Let's use the book order process example again and implement the validate order functionality in a Java class.

Listing 4.9 A validate order class that extends the `JavaDelegate` class

```
public class ValidateService implements JavaDelegate {

  @Override
  public void execute(DelegateExecution execution) {
    System.out.println("execution id " + execution.getId());
    Long isbn = (Long) execution.getVariable("isbn");
    System.out.println("received isbn " + isbn);
    execution.setVariable("validatetime", new Date());
  }
}
```

A typical Java service task must implement the `JavaDelegate` class, which makes it easy to implement a bit of process logic. This convenience class takes care of leaving the Java service task when it has finished to all the outgoing transitions for which the sequence flow condition, if present, doesn't evaluate to false. When the Java service task is executed in the process instance, the `execute` method will be invoked by the Activiti Engine. The `DelegateExecution` instance provides an interface to retrieve and set the process variables. In this simple listing, the `isbn` process variable is retrieved and the `validatetime` variable is set with the current date and time.

The only thing you have to change in the process definition is the service task for the validate order step; but, in the following listing, the full process definition is included to make it more comprehensible.

Listing 4.10 The book order process definition with a Java service task class

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
    targetNamespace="http://www.bpmnwithactiviti.org"
    xmlns:activiti="http://activiti.org/bpmn">
```

```
  <process id="bookorder" name="bookorder">
    <startEvent id="startevent1" name="Start"/>
    <serviceTask id="serviceTask1"
        name="Validate order"
        activiti:class="org.bpmnwithactiviti.
            ➥ chapter4.java.ValidateService"/>
    <sequenceFlow id="sequenceflow1"
        name="Validate order"
        sourceRef="startevent1"
        targetRef="serviceTask1"/>
    <userTask id="usertask1" name="Complete order"
        activiti:candidateGroups="sales"/>
    <sequenceFlow id="sequenceflow2"
        name="Sending to management"
        sourceRef="serviceTask1"
        targetRef="usertask1"/>
    <endEvent id="endevent1" name="End"/>
    <sequenceFlow id="sequenceflow3"
        name="flow"
        sourceRef="usertask1"
        targetRef="endevent1"/>
  </process>
</definitions>
```

1 Definition of service task

2 Configures a fully qualified class name

3 Shorthand to configure candidate groups

The service task **1** is configured with a `class` attribute **2** that's part of the Activiti BPMN extensions namespace. Note that you configured the `ValidateService` class shown in listing 4.8. The user task is changed a bit because you use the shorthand `candidateGroups` attribute here **3**. Activiti provides an easier way to define candidate users and groups with extension attributes because the BPMN 2.0 specification is a little bit verbose on this point. The same candidate group assignment would look like the following code snippet with BPMN 2.0–compliant XML:

```
<userTask id="usertask1" name="Complete order">
  <documentation>book order user task</documentation>
  <potentialOwner>
    <resourceAssignmentExpression>
      <formalExpression>sales</formalExpression>
    </resourceAssignmentExpression>
  </potentialOwner>
</userTask>
```

To test your book order process with a Java service task, you can develop a simple unit test like the one shown in the next listing.

Listing 4.11 Unit test that tests the book order process with Java service task

```
public class JavaBpmnTest extends AbstractTest {

  @Rule
  public ActivitiRule activitiRule = new ActivitiRule(
      "activiti.cfg-mem.xml");

  private ProcessInstance startProcessInstance() {
    RuntimeService runtimeService =
        activitiRule.getRuntimeService();
    Map<String, Object> variableMap =
        new HashMap<String, Object>();
    variableMap.put("isbn", 123456L);                          ① Starts new
    return runtimeService.startProcessInstanceByKey(              process
        "bookorder", variableMap);                                instance
  }

  @Test                                                    ② Deploys Java book
  @Deployment(resources={                                     order process
      "chapter4/bookorder.java.bpmn20.xml"})
  public void executeJavaService() {
    ProcessInstance processInstance = startProcessInstance();
    RuntimeService runtimeService =
        activitiRule.getRuntimeService();                  ③ Gets
    Date validatetime = (Date) runtimeService.getVariable(    validatetime
        processInstance.getId(), "validatetime");             process variable
    assertNotNull(validatetime);
    System.out.println("validatetime is " + validatetime);
  }
}
```

To test the execution of the Java service task, you first have to start a new process instance ① of the book order process in listing 4.10. To deploy the process definition with only one line of coding, you use the `@Deployment` annotation ②. Because the `ValidateService` class invoked in the Java service task sets a process variable with the name `validatetime`, you test if that variable is set ③. This shows you don't need a large unit test to verify a successful execution of a process definition.

Up to this point, you've been executing processes in a synchronous manner, until you encounter non-automatic tasks, such as a user task. In the next section, you'll see how to use `async` continuations to execute a service task asynchronously.

4.3.2 Introducing asynchronous behavior

In figure 4.6, you saw that Activiti executes automatic tasks like a service task in the same transaction and thread as the transaction and thread the process was started in. This means that the Java class that starts a process instance will have to wait until all automatic tasks have been executed in a process definition. When a service task contains long-running logic, like the invocation of an external web service or the construction of a large PDF document, this may not be the desired behavior.

Activiti provides a solution for these cases in the form of async continuations. From a BPMN 2.0 XML perspective, the definition of an asynchronous service task (or another type of task) is easy. You only have to add an `async` attribute to the service task configuration:

```
<serviceTask id="serviceTask1"
  name="Validate order"
  activiti:async="true"
  activiti:class="org.bpmnwithactiviti.chapter4.java.LongValidateService"/>
```

When we define a service task with the `async` attribute set to `true`, the execution of the service task logic will be executed in a separate transaction and thread. The process

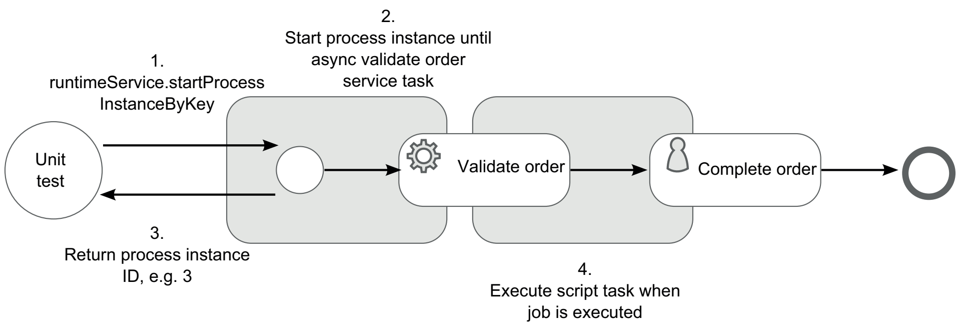

Figure 4.10 Asynchronous execution of the validate order service task using an Activiti async continuation

state is persisted to the Activiti database and a job is created to handle the service task execution. Figure 4.10 shows the book order process definition with the asynchronous validate order service task.

As you can see in figure 4.10, the unit test class (`JavaBpmnTest`) that starts the process instance will get a response right after the Activiti Engine stores the process state and creates a job to execute the "Validate order" service task. The Activiti job executor component that executes these jobs will be discussed in detail in chapter 15. For now, think of it as a standalone component that executes jobs in a separate transaction and thread. In the `executeAsyncService` method of the `JavaBpmnTest` class, you can find a unit test that executes the book order process as described in figure 4.10.

Besides a service task, you can also configure async continuations on other automatic tasks like a business rule task, call activity, or script task, and even on a subprocess. Furthermore, you can configure an async continuation on a non-automatic task like a user task or a receive task, which results in the execution listener being executed in a separate thread. (Execution listeners are introduced in chapter 6.)

You can also enhance the Java service task by injecting process variables or string values. In the next section, the book order example is changed a bit to include field extensions.

4.3.3 Java service task with class definition and field extensions

In section 4.4, you'll learn how to use the Activiti Engine inside a Spring container, which provides many ways to implement dependency injection. But the Activiti engine also provides some simple functionality regarding dependency injection. To be able to implement dependency injection, you'll have to change the `ValidateService` a bit, like the example in the following listing.

Listing 4.12 Java service task class with dependency injection

```
public class ValidateServiceWithFields
    implements JavaDelegate {

  private Expression validatetext;
  private Expression isbn;

  @Override
  public void execute(DelegateExecution execution) {
    System.out.println("execution id " + execution.getId());
    System.out.println("received isbn " +
        (Long) isbn.getValue(execution));                          ① Get isbn expression value
    execution.setVariable("validatetime", new Date());
    System.out.println(
        validatetext.getValue(execution).toString() +
            execution.getVariable("validatetime"));
  }
}
```

The `ValidateServiceWithFields` class defines two attributes that can be injected by the Activiti Engine: the `isbn` and `validatetext` attributes. Notice that the attributes are of type `org.activiti.engine.impl.el.Expression`. The `Expression` class is used by Activiti to support simple string attribute values as well as complex expressions.

> **NOTE** You might have expected a String type attribute for the `validatetext` parameter. But a service task has only one instance inside the Activiti Engine, which is reused for every process instance. Therefore, multiple threads can access a service task class at the same time, and class level attributes aren't thread safe. Activiti introduces an `Expression` class and the attribute value is retrieved by passing a `DelegateExecution` instance to the `Expression` instance, which can then evaluate the `Expression` value for that specific process instance.

In this example, the `isbn` expression consists of some logic to give the `isbn` at least a value that consists of more than six digits, as you'll see in the next code snippet. To get the value of the `isbn` number with the expression calculated, you can invoke the `getValue` method with the `DelegateExecution` instance as a parameter ①.

Now you only have to change the service task definition of the book order process, shown in listing 4.10, according to the following code snippet:

```
<serviceTask id="serviceTask1" name="Validate order"
    activiti:class="org.bpmnwithactiviti.chapter4.
         ➥ java.ValidateServiceWithFields">
  <extensionElements>
    <activiti:field name="validatetext"
        stringValue="Validaton done at "/>
       <activiti:field name="isbn">
      <activiti:expression>
        ${isbn > 999999 ? isbn : 1000000 + isbn}
      </activiti:expression>
    </activiti:field>
  </extensionElements>
</serviceTask>
```

With this `extensionElements` XML element, fields to be injected into the `ValidateServiceWithFields` class can be specified. This can be a simple `String` value, like the `validatetext` field, or an expression using, for example, process variables like the isbn field. Note that you can directly use process variables in a process definition for the Activiti engine. You don't need additional coding. Because this new process definition can be tested with a unit test similar to the one shown in listing 4.11, we won't cover this in more detail.

In addition to using classes inside a Java service task, you can also use method or value expressions; this is what we'll explore in the next section.

4.3.4 Java service task with method and value expressions

When you don't want to be dependent on the JavaDelegate interface in your service class, you can define a method or value expression for a Java service task. Let's look at two simple examples to get you introduced to this type of Java service task.

When you have a BookOrder class with a validate method like the following code snippet, you can use a method expression:

```
public class BookOrder implements Serializable {

  private static final long serialVersionUID = 1L;

  public Date validate(Long isbn) {
    System.out.println("received isbn " + isbn);
    return new Date();
  }
}
```

The method expression will invoke the validate method and proceed to next transition. Note that you now have an isbn instance as a parameter in the validate method. How the isbn instance is passed on is defined in the method expression of the Java service task:

```
<serviceTask id="serviceTask1" name="Validate order"
    activiti:expression="#{bookOrder.validate(isbn)}"
    activiti:resultVariableName="validatetime"/>
```

If necessary, you can still pass a DelegateExecution instance as a parameter into the method by using the implicit execution variable, as illustrated in section 4.4.2. The attribute resultVariableName is used to make the return value of the method available as a process variable with the name validatetime. To be able to use the BookOrder instance inside the process definition, you must make sure the class is made available as a process variable with a name of bookOrder. This can be done when the process is started, like you did in the unit test of listing 4.11.

> **NOTE** When you use a Java bean as a process variable, make sure the bean implements the Serializable interface because the process variable will be persisted to the Activiti Engine database.

Another use of expressions inside a Java service task is a value expression. A value expression defines an attribute inside a Java bean for which the corresponding getter method will be invoked. This isn't a common use of Java service tasks, but it looks like the following XML snippet:

```
<serviceTask id="serviceTask1" name="Validate order"
    activiti:expression="#{bookOrder.isbn}"
    activiti:resultVariableName="isbn"/>
```

In this example, the `getIsbn` method will be invoked on the `BookOrder` process variable and the resulting value is assigned to the `isbn` process variable.

We haven't discussed the `delegateExpression` attribute yet, which is the fourth way to define a Java service task. With a delegate expression, you can configure a variable that is evaluated at runtime to a Java class that must implement the `JavaDelegate` interface. Here's a simple example:

```
<serviceTask id="serviceTask1" name="Validate order"
    activiti:delegateExpression="#{orderValidator}"/>
```

The `orderValidator` variable should evaluate to a bean name that is defined in the Spring configuration or to a fully qualified class name.

In the next section, we'll explore richer functionality and the use of the Spring container with the Activiti Engine.

4.4 Using Spring with Activiti

Activiti is able to run on various platforms, including the plain Java approach we've taken until now and on a servlet container or application server like Apache Tomcat. But, it's also easy to run the Activiti Engine within a Spring application context. By using the Spring container to execute the Activiti Engine, you can, for example, use the Spring dependency injection functionality and invoke a Spring bean from a service task in the BPMN process. In the second subsection, you'll see that it's easy to develop unit tests with Spring and Activiti; but, first, you must define the Spring configuration to integrate with the Activiti Engine.

4.4.1 Creating a generic Spring configuration for Activiti

To set up the Spring container to start up the Activiti engine, you need a generic application context configuration. You can use the Spring configuration shown in the following listing every time you want to use a Spring container to start up the Activiti Engine. For convenience reasons, the namespace declarations that are part of the root element `beans` are left out of the listing, but they can be found in the source code of the book.

Listing 4.13 Generic Spring configuration to start up the Activiti Engine

```
<beans>
  <bean id="dataSource" class="org.springframework.jdbc.
          ➥ datasource.TransactionAwareDataSourceProxy">      ① Defines H2
    <property name="targetDataSource">                           datasource
      <bean class="org.springframework.jdbc.
                ➥ datasource.SimpleDriverDataSource">
        <property name="driverClass" value="org.h2.Driver" />
        <property name="url"
            value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
        <property name="username" value="sa" />
        <property name="password" value="" />
      </bean>
    </property>
  </bean>
                                                              ② Wraps
  <bean id="transactionManager"                                  transaction
      class="org.springframework.jdbc.                           manager
          ➥ datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <bean id="processEngineConfiguration"                       ③ Creates Activiti
        class="org.activiti.spring.                              process engine
            ➥ SpringProcessEngineConfiguration">                 configuration
    <property name="databaseType" value="h2" />
    <property name="dataSource" ref="dataSource" />
    <property name="transactionManager"
              ref="transactionManager" />
    <property name="databaseSchemaUpdate" value="true" />      ④ Deploys
    <property name="deploymentResources"                          book order
        value="classpath*:chapter4/bookorder.spring.bpmn20.xml" />  process
        <property name="jobExecutorActivate" value="false" />
  </bean>

  <bean id="processEngine"
      class="org.activiti.spring.ProcessEngineFactoryBean">
    <property name="processEngineConfiguration"
              ref="processEngineConfiguration" />
  </bean>

  <bean id="repositoryService"
        factory-bean="processEngine"
        factory-method="getRepositoryService" />              ⑤ Creates a
  <bean id="runtimeService"                                      RuntimeService
        factory-bean="processEngine"                             instance
        factory-method="getRuntimeService" />
  <bean id="taskService"
        factory-bean="processEngine"
        factory-method="getTaskService" />
  <bean id="historyService"
        factory-bean="processEngine"
        factory-method="getHistoryService" />
  <bean id="managementService"
        factory-bean="processEngine"
        factory-method="getManagementService" />
</beans>
```

In this listing, you can see that many things you developed programmatically in Java in the previous examples are now defined in the Spring configuration file. For example, you have to define a process engine configuration ③, which will be used to define the configuration options of the Activiti Engine.

In this Spring configuration, you defined an in-memory H2 data source ① in a so-called transaction-aware data source definition. Because the data source is wrapped in a transaction manager ②, you can use the standard Spring JDBC transaction manager.

With the data source and the transaction manager defined, you can instantiate the `SpringProcessEngineConfiguration` with these components (3). This means the Activiti Engine configuration is created with an in-memory data source when the Spring container is started. You can also specify a number of processes or task forms that have to be deployed to the Activiti Engine when it has started with the `deploymentResources` property (4). You'll see how this makes unit testing even easier in a moment. Note that this property definition is specific to the example you'll implement in this section. The `SpringProcessEngineConfiguration` is used to instantiate the `ProcessEngineFactoryBean` that starts the Activiti Engine with the configured resources and settings.

In addition to the instantiation of the Activiti Engine, the Spring container can also create the core interface classes to the Activiti Engine for you. For example, the `RuntimeService` is created via the `getRuntimeService` method of the `processEngine` bean (5). With this generic Spring configuration defined, you can now proceed to define a unit test that uses this Spring configuration to test a specific process.

4.4.2 Implementing a Spring-enabled unit test for Activiti

Because you've already defined all the necessary configuration of the Activiti engine, your unit test can be kept simple. The next listing shows a unit test that starts a new process instance of the book order process definition and completes the user task.

Listing 4.14 A unit test that takes advantage of the Spring configuration

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:chapter4/
         ➥ spring-test-application-context.xml")          (1) Loads Spring configuration
public class SpringTest extends AbstractTest {

  @Autowired
  private RuntimeService runtimeService;                    (2) Injects RuntimeService instance

  @Autowired
  private TaskService taskService;

  @Test
  public void simpleSpringTest() {
    Map<String, Object> variableMap =
        new HashMap<String, Object>();
    variableMap.put("isbn", 123456L);
    runtimeService.startProcessInstanceByKey(               (3) Starts a process instance
        "bookorder", variableMap);
    Task task = taskService
        .createTaskQuery()
        .singleResult();
    assertEquals("Complete order", task.getName());
    taskService.complete(task.getId());
    assertEquals(0, runtimeService.
        createProcessInstanceQuery().count());
  }
}
```

As you can see, the unit test is simple because you don't have to create the Activiti engine yourself. With the standard Spring annotations `@RunWith` and `@Context-Configuration` (1), the Spring configuration you defined in listing 4.13 is used as part of this unit test.

With the `@Autowired` annotation, you can let the Spring container inject an instance of the `RuntimeService` in your unit test class (2). This means you don't have to do any plumbing before you can start a new process instance of the book order process definition (3). Because the book order process is already deployed as part of the Activiti Engine creation in the Spring configuration, you don't have to deploy the process first, either.

To complete the unit test, you query the Activiti engine for any running user tasks. Because you run this unit test with an in-memory database, you can be sure that no user task is running other than the `Complete Order` user task defined in the book order process. When this task is completed, you can make sure that there's no running process instance anymore by running a process instance query.

To make this unit test work, you have to implement the process definition of the `bookorder.spring.bpmn20.xml` file. This process definition has some small differences when compared to the bookorder.bpmn20.xml file you've used before. In this process definition, a Spring bean is used to implement the validation order activity that was first implemented with a script task. Let's take a quick look at the revised XML definition of the service task:

```
<serviceTask id="serviceTask1"
             name="Validate order"
             activiti:expression="#{order.validate(execution)}"/>
```

Because you run the Activiti Engine within the Spring container, you can directly reference Spring beans from a service task.

NOTE A Spring service task isn't standard BPMN 2.0 functionality, but is implemented as an add-on by the Activiti framework.

The `expression` attribute can be used to define a Spring bean name with the method that must be invoked, which, in this case, is the `order` Spring bean and the `validate` method. As you saw in section 4.3, you can pass on a `DelegateExecution` instance with the reserved keyword `execution`. Because you didn't configure the Spring bean `order` in section 4.3.1, the following code snippet must be added to the generic Spring configuration from listing 4.13:

```
<bean id="order"
    class="org.bpmnwithactiviti.chapter4.spring.OrderService" />
```

The last step is to implement the Spring bean class `OrderService` before you can finally run the unit test. This class is really simple and only prints a message to the system console:

```
public class OrderService {
  public void validate(DelegateExecution execution) {
    System.out.println("validating order for isbn " +
        execution.getVariable("isbn"));
  }
}
```

As you can see, this needs no explaining. Now you can run the unit test provided in listing 4.14 and see that it runs successfully.

A more flexible deployment strategy

In addition to the definition of deployment resources in the Spring configuration used in the previous example, you can also define a deployment per test method. To implement this strategy, an `ActivitiRule` Spring bean must be added to the Spring configuration:

```
<bean id="activitiRule" class="org.activiti.engine.test.ActivitiRule">
  <property name="processEngine" ref="processEngine" />
</bean>
```

The `ActivitiRule` Spring bean must also be injected into the Spring-enabled unit test via the `@Autowire` annotation:

```
@Autowired
@Rule
public ActivitiRule activitiSpringRule;
```

Now you can add a `@Deployment` annotation to every test method where you want to deploy a specific process definition to test the process logic. The advantage of this deployment strategy is that it's finer grained. You can define a specific process definition to be deployed before a test method is executed, and it will be undeployed afterwards. When you define the deployment resources as part of the Spring configuration, they will be available for every test method.

The unit test shown in the next listing can be rewritten to use the more flexible deployment strategy by using the `@Deployment` annotation.

Listing 4.15 Use of the `@Deployment` annotation in an unit test

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:chapter4/
        ➥ spring-nodeployment-application-context.xml")
public class SpringWithDeploymentTest extends AbstractTest {

  @Autowired
  private RuntimeService runtimeService;

  @Autowired
  private TaskService taskService;

  @Autowired
  @Rule
  public ActivitiRule activitiSpringRule;
```

1 Spring configuration without deployment resources

2 ActivitiRule instance for test convenience

```
  @Test
  @Deployment(resources = {
      "chapter4/bookorder.spring.bpmn20.xml" })
  public void simpleProcessTest() {
    runtimeService.startProcessInstanceByKey("bookorder");
    Task task = taskService.createTaskQuery().singleResult();
    assertEquals("Complete order", task.getName());
    taskService.complete(task.getId());
    assertEquals(0, runtimeService
        .createProcessInstanceQuery()
        .count());
  }
}
```

3 **Process definition to be deployed**

In this unit test, you use a Spring configuration that has no deployment resources defined 1—unlike the Spring configuration you saw in section 4.4.1. To be able to use the `@Deployment` annotation, you have to inject the `ActivitiRule` instance 2, which provides a hook into the Activiti Engine to deploy and undeploy process definitions. As shown with the `simpleProcessTest` method 3, you can now configure a process definition file as part of the `@Deployment` annotation.

There are plenty of possibilities for using the strength of the Spring framework together with the Activiti process engine, such as using Spring's transaction handling. The information provided in this section should get you started. In chapter 6, you'll see how Spring can be used to retrieve and update entity objects from a database.

4.5 Summary

You now know a lot about the different ways you can develop and test with the Activiti Engine. You can query the process engine with all kinds of criteria to retrieve process definitions, instances, and user tasks. Because the Activiti Engine provides a service task that invokes a Java class, we also took a look at how to use this BPMN construct within a BPMN 2.0 process definition. This provides a powerful feature if you need process logic inside your business process. You also saw that you can make a service task asynchronous by adding an `async` continuation attribute to its definition.

We also covered the Spring integration module, which provides functionality to run the Activiti Engine within a Spring container. Running the Activiti Engine in a Spring container makes it possible to use Spring beans from a service task or expressions inside conditions or variable assignments. Because Spring provides functionality like transaction and security management and easy hooks to implement data access and messaging logic (among other things), the integration of Spring with Activiti provides lots of possibilities.

In the next chapter, we'll move away from the short code examples you've seen so far, and we'll look at a larger business process that you can implement using the Activiti Designer and the Eclipse IDE.

METHODOLOGY/JAVA

Activiti IN ACTION

Tijs Rademakers

Activiti streamlines the implemention of your business processes: with Activiti Designer you draw your business process using BPMN. Its XML output goes to the Activiti Engine which then creates the web forms and performs the communications that implement your process. It's as simple as that. Activiti is lightweight, integrates seamlessly with standard frameworks, and includes easy-to-use design and management tools.

Activiti in Action introduces developers to business process modeling with Activiti. You'll start by exploring BPMN 2.0 from a developer's perspective. Then, you'll quickly move to examples that show you how to implement processes with Activiti. You'll dive into key areas of process modeling, including workflow, ESB usage, process monitoring, event handling, business rule engines, and document management integration.

What's Inside

- Activiti from the ground up
- Dozens of real-world examples
- Integrate with standard Java tooling

Written for business application developers. Familiarity with Java and BPMN is helpful but not required.

Tijs Rademakers is a senior software engineer specializing in open source BPM, lead developer of Activiti Designer, and member of the core Activiti development team. He's the coauthor of Manning's *Open Source ESBs in Action*.

To download their free eBook in PDF, ePub and Kindle formats, owners of this book should visit manning.com/ActivitiinAction

Free eBook
SEE INSERT

"A comprehensive overview of the Activiti framework, the Activiti Engine, and BPMN."
—From the Foreword by Tom Baeyens, Founder of jBPM

"A superb book. Best source of knowledge on Activiti and BPMN 2.0. Period."
—From the Foreword by Joram Barrez, Cofounder of Activiti

"The very first book on Actviti ... immediately sets the bar high."
—Roy Prins, CIBER Netherlands

"Just enough theory to let you get right down to coding."
—Gil Goldman
Dalet Digital Media Systems

ISBN 13: 978-1-617290-12-1
ISBN 10: 1-617290-12-2
54999
9 781617 290121

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]