



Activiti in Action

Tijs Rademakers and Ron van Liempd

In this article, based on chapter 3 of <u>Activiti in Action</u>, the authors describe different Activiti libraries you can use and explain setting up a Maven project structure.

You may also be interested in...

Creating an Activiti Development Environment

In this article, you will be introduced into the different Activiti libraries you can use and setting up a Maven project structure.

Logging is an important vehicle to understand what's going on in a complex framework as a process engine. We'll see how we can tune the log levels for our needs. At the end, we'll look at a couple of options you have to run the Activiti engine. But, first, it's time to find out a little bit more about the Activiti library structure.

Getting familiar with the Activiti libraries

The Activiti distribution consists of several modules, including Activiti Modeler, Cycle, Probe, and Explorer. Each of these modules has its dependency and project structure. We'll only focus on the Activiti Engine module, which provides the core component of the project.

But the Activiti engine also consists of at least two layers. The first layer, which provides core process engine logic, is the process virtual machine. The second layer is the engine, which provides the engine interfaces and implements the BPMN 2.0 specification. The engine and the process virtual machine are implemented in the same archive file, which can be found in the lib directory of the Activiti installation. In early versions of Activiti, the engine and process virtual machine layers were separated in different JARs but, for convenience reasons, the two layers were merged in one JAR in the current release.

An optional layer is the Spring container integration for the Activiti engine. 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 archive that is also available in the lib directory of the Activiti distribution. The 3 layers for the Activiti Engine are shown in figure 1.

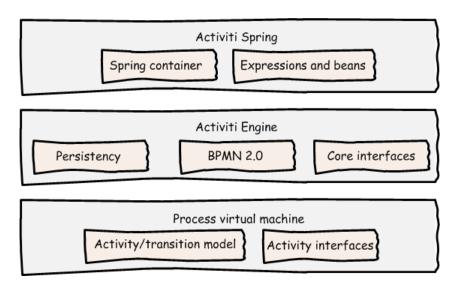


Figure 1 Overview of the different layers of the Activiti Engine, including the process virtual machine, the engine and the Spring integration layers

As you can see in figure 1, each layer of the Activiti Engine adds a specific set of functionalities. Obviously, most of the process engine functionalities are provided by the Engine layer. The process virtual machine can be considered the foundational level of the Activiti Engine, which provides a simple state or activity and transition model. The Spring layer adds the capability of running Activiti inside the Spring container with support for using Spring beans within service tasks and expressions.

Before we can use the Activiti Engine in the development environment, the dependent libraries must also be available. Next, we show you a Maven-based project structure, which will provide you with the necessary dependencies. But you can also reference the library dependencies from the Activiti installation directory. Notice that you then have to run the setup first, because the setup script puts the libraries in right places of the workspace directory.

In the workspace directory, there are several projects created for the Engine, Spring, and other component examples. 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 here. But next up, we'll see that a Maven project structure makes life a bit easier.

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 very easy to set up your project with Maven. The examples in the Activiti distribution have a Maven structure and a pom.xml. But, let's first create a Maven project for our example code with the following command.

```
\label{lem:monostative} \verb|monostative| archetype:generate -DgroupId=org.bpmnwithactiviti -DartifactId=bpmn-examples -DinteractiveMode=false \\
```

This will create a standard Maven project layout with a base directory of bpmn-examples. A standard App Java class and an AppTest unit test are also generated but, since we won't need those, you can delete them right away. The generated pom.xml in the root of the bpmn-examples directory needs some work because we have to add the Activiti dependencies. The full Maven pom.xml is shown in listing 1.

Listing 1 A standard Maven configuration for an Activiti project

```
<version>1.0 </version>
  <packaging>jar</packaging>
  <name>BPMN 2.0 with Activiti - Examples
  properties>
   <activiti-version>5.0</activiti-version>
  </properties>
  <dependencies>
   <dependency>
                                                                          #1
     <groupId>org.activiti</groupId>
      <artifactId>activiti-engine</artifactId>
                                                                          #1
      <version>${activiti-version}</version>
                                                                          #1
   </dependency>
   <dependency>
      <groupId>com.h2database
                                                                          #2
      <artifactId>h2</artifactId>
                                                                          #2
     <version>1.2.132
     <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.1
     <scope>test</scope>
   </dependency>
  </dependencies>
  <repositories>
   <repository>
      <id>Activiti</id>
                                                                          £З
      <url>http://maven.alfresco.com/nexus/content/
        [CA]repositories/activiti</url>
   </repository>
 </repositories>
</project>
#1 Gets the Engine and PVM archives
#2 H2 database driver
#3 Alfresco Maven repository for Activiti
```

The Maven namespace declarations are left out of the listing to make the configuration more readable. To include the Activiti Engine and PVM libraries we 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 we didn't include the Activiti Spring module.

To be able to test with an in-memory H2 database, we 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 is not yet available from a central Maven repository, we also need to add the Alfresco Maven repository for the Activiti project (#3). If you are using the Eclipse development tool, you can now run the mvn eclipse:eclipse command, which will create the necessary Eclipse project and classpath files. But, of course, similar Maven archetypes are available for IntelliJ IDEA and other IDEs.

Now all the Java libraries needed to run the Activiti Engine are available inside the IDE. Now let's discuss how we can tune logging in the Activiti Engine.

Logging in the Activiti Engine

Logging statements can help a lot when you are debugging but are also essential for getting good error descriptions from a production system. When you are using multiple open-source frameworks in one project, you may run into different logging systems. And that's exactly the case with Activiti and Spring.

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 SLF4J (Simple Logging Façade for Java—www.slf4j.org) framework, which can translate logging statements from different frameworks into the logging statement of your choice.

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

We'll be using a lot of unit tests to work with the Activiti BPM platform and, therefore, the next code snippet shows the AbstractTest class we'll be extending from in a unit test to initialize our logging framework.

public abstract class AbstractTest {

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

This abstract unit test class ensures 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 we want. In our logging.properties the log level is set to FINEST to get all logging information out of the Activiti Engine when we want to.

Next in the code snippet, the log handlers are removed from the <code>java.util.logging.Logger</code> class because, otherwise, the JDK 1.4 logging framework would still perform the logging. At the end, we can invoke the SLF4J bridge, which will direct all JDK 1.4 logging output to SLF4J. And, since we have the SLF4J Log4J library on the classpath, we can now define the log level of the Activiti Engine, the Spring framework, and other external frameworks in a standard log4j.xml file.

This means we can define a log level of DEBUG when we want to do some debugging, and we can set the level to ERROR when we don't want extra information logged in our unit tests. With our logging configuration in place, let's discuss the options we have to run the Activiti Engine.

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, for example, deploy new process definitions, start new process instances, and query for user tasks. But what are our options to run the Activiti Engine during development? In the following subsections we discuss the following 3 options:

- 1. Runing the Activiti Engine in the JVM with an in-memory database (H2).
- 2. Runing the Activiti Engine in the JVM with a standalone database (H2).
- 3. Runing the Activiti Engine on an application server (Apache Tomcat) with a standalone database (H2).

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 with an inmemory 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 2 shows this way of deployment in a graphical overview.

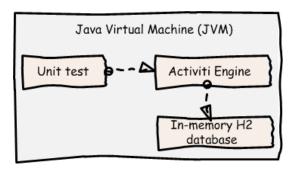


Figure 2 An overview of the deployment scenario where the Activiti Engine runs within the Java Virtual Machine with an in-memory database

This deployment alternative is used because it's the easiest to use from within an IDE. Next up, we look at another option, which is to use a standalone 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, another deployment alternative is needed. You must be able to run the Activiti Engine connected to a standalone database. This enables possibilities to, for example, query the standalone database for specific running process instances. In a schematic way this type of deployment looks like figure 3.

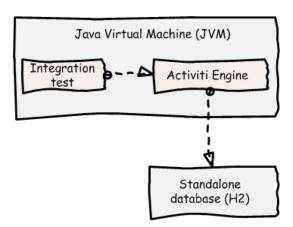


Figure 3 A schematic picture of the deployment alternative where the Activiti Engine runs within the same JVM as the unit test and connects to a standalone database

An Activiti Engine can be created from within a unit test and the H2 database that installed and started as part of the Activiti installation setup used as the process engine database. This type of setup is not very suitable for unit testing but can be handy for integration testing where you also want to use the Activiti Explorer and Probe together in 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 very useful for unit and integration testing. But, eventually, we'll want to deploy our business processes on a production-like environment and do some basic testing there also. This means that we can't start an Activiti Engine from within a unit test because it runs on a separate application server environment.

What we can do is use the REST API provided with the Activiti Engine to interact with the process engine. And the deployment of a new process definition must then be done via Activiti Probe or an Ant script by deploying a Business ARchive (BAR) file. Let's have a look at this alternative in figure 4.

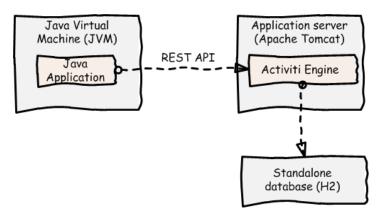


Figure 4 A typical Activiti Engine environment where the process engine runs on an application server like Apache Tomcat with a standalone database. The REST API provides the necessary interface to interact with the process engine.

In an environment like the one shown in figure 4, the need for unit tests is typically low, because the deployment alternatives we discussed earlier are more likely to be used for unit and integration testing. But there will still be a need to communicate with the process engine when tools like the Activiti Explorer or Probe don't provide all the information you need or in cases where you want to communicate with the process engine from other applications. The REST API provides a great way to implement the communication necessary.

Summary

The Activiti Engine provides a service task that invokes a Java class, and we looked at how to use this BPMN construct within a BPMN 2.0 process definition. This provides a powerful feature if there's a need for process logic inside your business process.

Here are some other Manning titles you might be interested in:



Open-Source ESBs in Action
Tijs Rademakers and Jos Dirksen



Open Source SOA Jeff Davis



Mule in Action
David Dossot and John D'Emic

Last updated: April 8, 2011