This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-12-2005/jw-1205-maven.html

An introduction to Maven 2

How applied best practices can optimize the Java build process

By John Ferguson Smart, JavaWorld.com, 12/05/05

Maven is a popular open source build tool for enterprise Java projects, designed to take much of the hard work out of the build process. Maven uses a declarative approach, where the project structure and contents are described, rather then the task-based approach used in Ant or in traditional make files, for example. This helps enforce company-wide development standards and reduces the time needed to write and maintain build scripts.

The declarative, lifecycle-based approach used by Maven 1 is, for many, a radical departure from more traditional build techniques, and Maven 2 goes even further in this regard. In this article, I go through some of the basic principals behind Maven 2 and then step through a working example. Let's start by reviewing the fundamentals of Maven 2.

The project object model

The heart of a Maven 2 project is the project object model (or POM for short). It contains a detailed description of your project, including information about versioning and configuration management, dependencies, application and testing resources, team members and structure, and much more. The POM takes the form of an XML file (*pom.xml*), which is placed in your project home directory. A simple pom.xml file is shown here:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javaworld.hotels</groupId>
  <artifactId>HotelDatabase</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
     <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
     </dependency>
  </dependencies>
</project>
```

The Maven 2 directory structure

Much of Maven's power comes from the standard practices it encourages. A developer who has previously worked on a Maven project will immediately feel familiar with the structure and organization of a new one. Time need not be wasted reinventing directory structures, conventions, and customized Ant build scripts for each project. Although you can override any particular directory location for your own specific ends, you really should respect the standard Maven 2 directory structure as much as possible, for several reasons:

- It makes your POM file smaller and simpler
- It makes the project easier to understand and makes life easier for the poor guy who must maintain the project when you leave
- It makes it easier to integrate plug-ins

The standard Maven 2 directory structure is illustrated in Figure 1. In the project home directory goes the POM (pom.xml) and two subdirectories: src for all source code and target for generated artifacts.

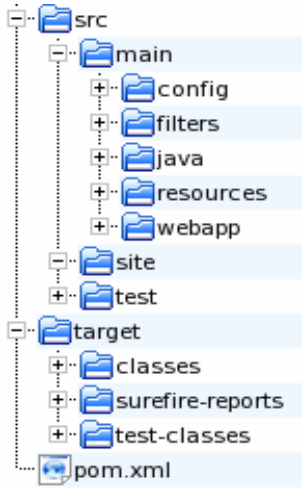

Figure 1. The standard Maven 2 directory layout

The src directory has a number of subdirectories, each of which has a clearly defined purpose:

- **src/main/java:** Your Java source code goes here (strangely enough!)
- **src/main/resources:** Other resources your application needs
- **src/main/filters:**
  Resource filters, in the form of properties files, which may be used to define variables only known at runtime
- **src/main/config:** Configuration files
- **src/main/webapp:** The Web application directory for a WAR project
- **src/test/java:** Unit tests
- **src/test/resources:** Resources to be used for unit tests, but will not be deployed
- **src/test/filters:** Resources filters to be used for unit tests, but will not be deployed
- **src/site:** Files used to generate the Maven project Website

Project lifecycles

Project lifecycles are central to Maven 2. Most developers are familiar with the notion of build phases such as compile, test, and deploy. Ant has targets with names like those. In Maven 1, corresponding plug-ins are called directly. To compile Java source code, for instance, the `java` plug-in is used:

```
$maven java:compile
```

In Maven 2, this notion is standardized into a set of well-known and well-defined lifecycle phases (see Figure

2). Instead of invoking plug-ins, the Maven 2 developer invokes a lifecycle phase: `$mvn compile`.

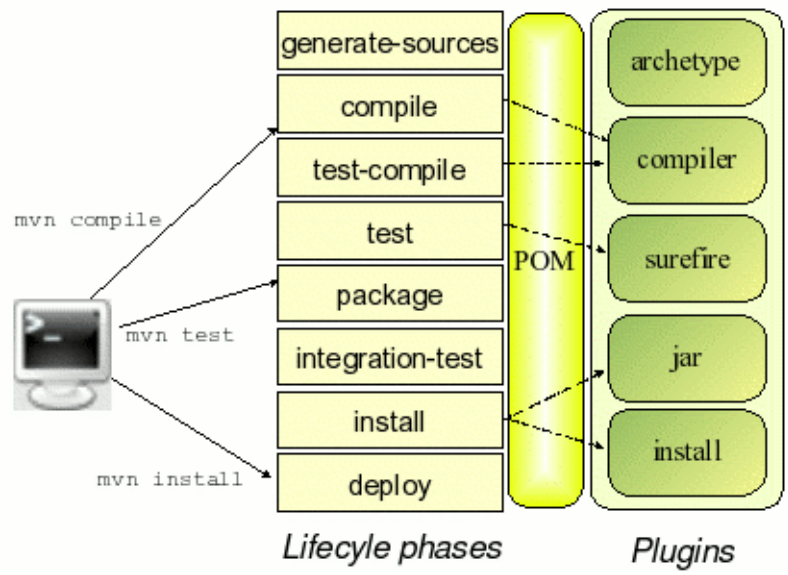

Figure 2. Maven 2 lifecycle phases

Some of the more useful Maven 2 lifecycle phases are the following:

- `generate-sources`: Generates any extra source code needed for the application, which is generally accomplished using the appropriate plug-ins
- `compile`: Compiles the project source code
- `test-compile`: Compiles the project unit tests
- `test`: Runs the unit tests (typically using JUnit) in the src/test directory
- `package`: Packages the compiled code in its distributable format (JAR, WAR, etc.)
- `integration-test`: Processes and deploys the package if necessary into an environment where integration tests can be run
- `install`: Installs the package into the local repository for use as a dependency in other projects on your local machine
- `deploy`: Done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects

Many other lifecycle phases are available. See Resources for more details.

These phases illustrate the benefits of the recommended practices encouraged by Maven 2: once a developer is familiar with the main Maven 2 lifecycle phases, he should feel at ease with the lifecycle phases of any Maven project.

The lifecycle phase invokes the plug-ins it needs to do the job. Invoking a lifecycle phase automatically invokes any previous lifecycle phases as well. Since the lifecycle phases are limited in number, easy to understand, and well organized, becoming familiar with the lifecycle of a new Maven 2 project is easy.

Transitive dependencies

One of the highlights of Maven 2 is transitive dependency management. If you have ever used a tool like *urpmi*
on a Linux box, you'll know what transitive dependencies are. With Maven 1, you have to declare each and every JAR that will be needed, directly or indirectly, by your application. For example, can you list the JARs needed by a Hibernate application? With Maven 2, you don't have to. You just tell Maven which libraries *you* need, and Maven will take care of the libraries that your libraries need (and so on).

Suppose you want to use Hibernate in your project. You would simply add a new dependency to the `dependencies` section in pom.xml, as follows:

```
<dependency>
    <groupId>hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.0.3</version>
    <scope>compile</scope>
 </dependency>
```

And that's it! You don't have to hunt around to know in which other JARs (and in which versions) you need to run Hibernate 3.0.3; Maven will do it for you!

The XML structure for dependencies in Maven 2 is similar to the one used in Maven 1. The main difference is the `scope` tag, which is explained in the following section.

Dependency scopes

In a real-world enterprise application, you may not need to include all the dependencies in the deployed application. Some JARs are needed only for unit testing, while others will be provided at runtime by the application server. Using a technique called *dependency scoping*, Maven 2 lets you use certain JARs only when you really need them and excludes them from the classpath when you don't.

Maven provides four dependency scopes:

- `compile`: A compile-scope dependency is available in all phases. This is the default value.
- `provided`: A provided dependency is used to compile the application, but will not be deployed. You would use this scope when you expect the JDK or application server to provide the JAR. The servlet APIs are a good example.
- `runtime`: Runtime-scope dependencies are not needed for compilation, only for execution, such as JDBC (Java Database Connectivity) drivers.
- `test`: Test-scope dependencies are needed only to compile and run tests (JUnit, for example).

Project communication

An important part of any project is internal communication. While it is not a silver bullet, a centralized technical project Website can go a long way towards improving visibility within the team. With minimal effort, you can have a professional-quality project Website up and running in very little time.

This takes a whole new dimension when the Maven site generation is integrated into a build process using continuous integration or even automatic nightly builds. A typical Maven site can publish, on a daily basis:

- General project information such as source repositories, defect tracking, team members, etc.
- Unit test and test coverage reports
- Automatic code reviews and with Checkstyle and PMD
- Configuration and versioning information
- Dependencies
- Javadoc
- Source code in indexed and cross-referenced HTML format
- Team member list
- And much more

Once again, any Maven-savvy developer will immediately know where to look to become familiar with a new Maven 2 project.

A practical example

Now that we have seen a few of the basic notions used in Maven 2, let's see how it works in the real world. The rest of this tutorial examines how we would use Maven 2 on a simple Java Enterprise Edition project. The

demo application involves an imaginary (and simplified) hotel database system. To demonstrate how Maven handles dependencies between projects and components, this application will be built using two components (see Figure 3):

- A business logic component: HotelDatabase.jar
- A Web application component: HotelWebApp.war

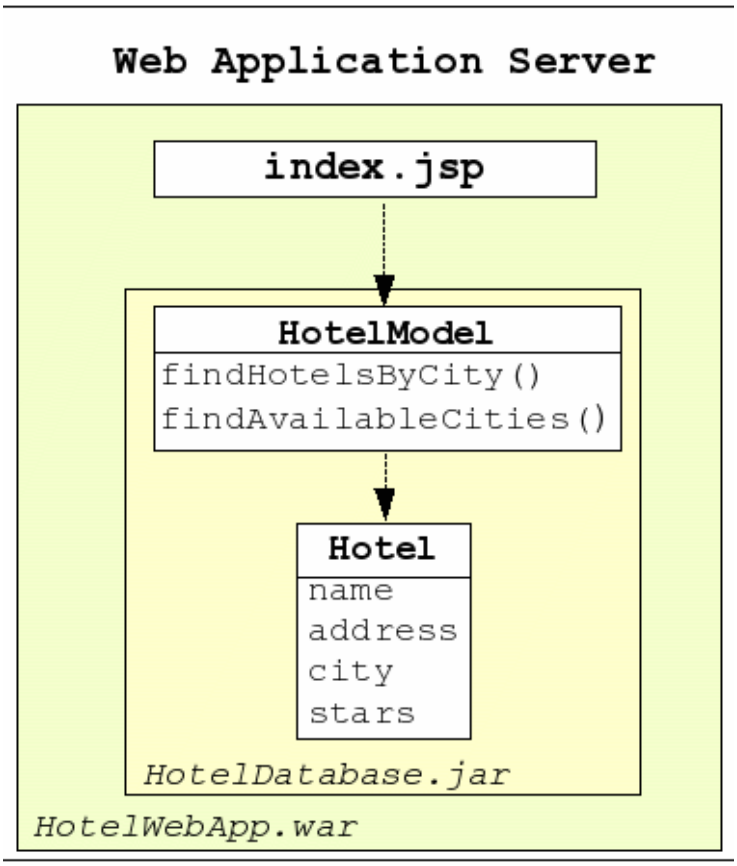

Figure 3. The tutorial application architecture involves two simple components: a JAR (HotelDatabase.jar) and a WAR (HotelWebapp.war)

You can download the source code to follow along with the tutorial in Resources.

Set up your project environment

We start by configuring your work environment. In real-world projects, you will often need to define and configure environment or user-specific parameters that should not be distributed to all users. If you are behind a firewall with a proxy, for example, you need to configure the proxy settings so that Maven can download JARs from repositories on the Web. For Maven 1 users, the build.properties and project.properties files do this job. In Maven 2, they have been replaced by a settings.xml file, which goes in the $HOME/.m2 directory. Here is an example:

```
 <?xml version="1.0" encoding="UTF-8"?>
<settings>
  <proxies>
    <proxy>
      <active/>
      <protocol>http</protocol>
      <username>scott</username>
      <password>tiger</password>
      <port>8080</port>
      <host>my.proxy.url</host>
      <id/>
    </proxy>
  </proxies>
</settings>
```

Create a new project with the archetype plug-in

The next step is to create a new Maven 2 project template for the business logic component. Maven 2 provides the `archetype` plug-in, which builds an empty Maven 2-compatible project directory structure. This plug-in proves convenient for getting a basic project environment up and running quickly. The default archetype model will produce a JAR library project. Several other artifact types are available for other specific project types, including Web applications, Maven plug-ins, and others.

Run the following command to set up your HotelDatabase.jar project:

```
mvn archetype:create -DgroupId=com.javaworld.hotels -
  DartifactId=HotelDatabase -Dpackagename=com.javaworld.hotels
```

Now you have a brand new Maven 2 project directory structure. Switch to the `HotelDatabase` directory to continue the tutorial.

Implementing the business logic

Now we implement the business logic. The `Hotel` class is a simple JavaBean. The `HotelModel` class implements two services: the `findAvailableCities()` method, which lists available cities, and the `findHotelsByCity()` method, which lists all hotels in a given city. A simple, memory-based implementation of the `HotelModel` class is presented here:

```
package com.javaworld.hotels.model;


import java.util.ArrayList;
import java.util.List;


import com.javaworld.hotels.businessobjects.Hotel;


public class HotelModel {


   /**
    * The list of all known cities in the database.
    */
   private static String[] cities =
   {
      "Paris",
      "London",
   };

   /**
    * The list of all hotels in the database.
    */
   private static Hotel[] hotels = {
      new Hotel("Hotel Latin","Quartier latin","Paris",3),
      new Hotel("Hotel Etoile","Place de l'Etoile","Paris",4),
      new Hotel("Hotel Vendome","Place Vendome","Paris",5),
      new Hotel("Hotel Hilton","Trafalgar Square","London",4),
      new Hotel("Hotel Ibis","The City","London",3),
   };

   /**
    * Returns the hotels in a given city.
    * @param city the name of the city
```

```
     * @return a list of Hotel objects
     */
    public List<Hotel> findHotelsByCity(String city){
       List<Hotel> hotelsFound = new ArrayList<Hotel>();

       for(Hotel hotel : hotels) {
          if (hotel.getCity().equalsIgnoreCase(city)) {
             hotelsFound.add(hotel);
          }
       }
       return hotelsFound;
    }

    /**
     * Returns the list of cities in the database which have a hotel.
     * @return a list of city names
     */
    public String[] findAvailableCities() {
       return cities;
    }
}
```

Unit testing with Maven 2

Now let's test the application. A few simple test classes can be found in the source code. Unit testing is (or should be!) an important part of any enterprise Java application. Maven completely integrates unit testing into the development lifecycle. To run all your unit tests, you invoke the `test` lifecycle phase:

```
mvn test
```

If you want to run only one test, you can use the `test` parameter:

```
mvn test -Dtest=HotelModelTest
```

A nice feature of Maven 2 is its use of regular expressions and the `test` parameter to control the tests you want to run. If you want to run only one test, you just indicate the name of the test class:

```
mvn test -Dtest=HotelModelTest
```

If you want to run only a subset of the unit tests, you can use a standard regular expression. For example, to test all `ModelTest` classes:

```
mvn test -Dtest=*ModelTest
```

Building and deploying the JAR

Once you're happy with the tests, you can build and deploy your new JAR. The `install` command compiles, tests, and bundles your classes into a jar file and deploys it to your local Maven 2 repository, where it can be seen by other projects:

```
 mvn install
```

Create a Web application

Now we want to use this library in a Web application. For simplicity, our Web application will consist of a JavaServer Pages (JSP) file that directly invokes the `HotelModel` class. First, we create a new Web application project using the `archetype` plug-in:

```
 mvn archetype:create -DgroupId=com.javaworld.hotels -
DartifactId=HotelWebapp -Dpackagename=com.javaworld.hotels -
DarchetypeArtifactId=maven-archetype-webapp
```

Next, we need to include our business logic JAR in this application. All we need to do is to add a dependency to the new pom.xml, pointing to our `HotelDatabase` component:

```
   <dependency>
      <groupId>com.javaworld.hotels</groupId>
      <artifactId>HotelDatabase</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
```

Now we implement the main (and only) JSP page. It simply lists the available cities and, if a city is chosen, lists the corresponding hotels:

```
 <html>
<body>
<h2>Hotel database tutorial application</h2>
<%@ page import="
   java.util.List,
    com.javaworld.hotels.businessobjects.Hotel,
    com.javaworld.hotels.model.HotelModel"
%>
<%
   HotelModel model = new HotelModel();


   String[] cityList = model.findAvailableCities();


   String selectedCity = request.getParameter("city");
   List<Hotel> hotelList = model.findHotelsByCity(selectedCity);
%>


   <h3>Choose a destination</h3>
   <form action="index.jsp" method="get">
     Please choose a city:
      <SELECT name="city">
         <OPTION value="">---Any city---</OPTION>
         <%
         for(String cityName : cityList){
         %>
          <OPTION value="<%=cityName%>"><%=cityName%></OPTION>
         <%
         }
         %>
      </SELECT>
      <BUTTON type="submit">GO</BUTTON>
   </form>
   <% if (hotelList.size() > 0) { %>
      <h3>Available hotels in <%=selectedCity%> </h3>
```

```
      <table border="1">
           <tr>
             <th>Name</th>
             <th>Address</th>
             <th>City</th>
             <th>Stars</th>
            </tr>
         <%
         for(Hotel hotel : hotelList){
         %>
           <tr>
             <td><%=hotel.getName()%></td>
             <td><%=hotel.getAddress()%></td>
             <td><%=hotel.getCity()%></td>
             <td><%=hotel.getStars()%> stars</td>
            </tr>
         <%
         }
         %>
       </table>
    <%}%>
</body>
</html>
```

Now run `mvn install` from the `HotelWebapp`
directory; this will compile, bundle, and deploy the HotelWebapp.war file to your local repository (you can also find it in the `target`
directory if you need to). Now you can deploy this war file to your favorite application server and see what you get (see Figure 4).

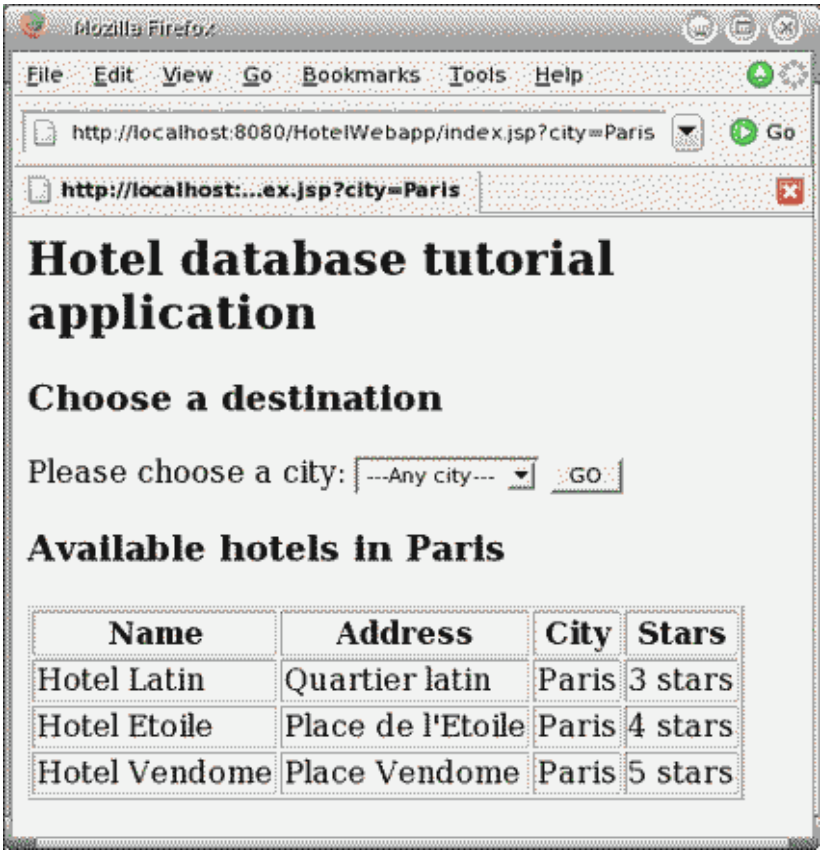

Figure 4. The tutorial application in action

Working with plug-ins

Maven 2 comes with an ever-increasing number of plug-ins that add extra functions to your build process with little effort. To use a plug-in, you bind it to a lifecycle phase. Maven will then figure out when (and how) to

use it. Some plug-ins are already used by Maven behind the scenes, so you just have to declare them in the `plugins` section of your pom.xml file. The following plug-in, for example, is used to compile with J2SE 1.5 source code:

```
...
  <build>
     ...
     <plugins>
        <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
           <configuration>
              <source>1.5</source>
              <target>1.5</target>
           </configuration>
        </plugin>
     </plugins>
  </build>
```

In other cases, you bind the plug-in to a lifecycle phase so that Maven will know when to use it. In the following example, we want to run some standard Ant tasks. To do this, we bind the `maven-antrun-plugin` to the `generate-sources` phase, and add the Ant tasks between the `tasks` tags, as shown here:

```
...
 <build>
   ...
   <plugins>
      <plugin>
         <artifactId>maven-antrun-plugin</artifactId>
         <executions>
           <execution>
             <phase>generate-sources</phase>
             <configuration>
               <tasks>
                 <!-- Ant tasks go here -->
               </tasks>
             </configuration>
             <goals>
               <goal>run</goal>
             </goals>
           </execution>
         </executions>
       </plugin>
    </plugins>
 </build>
```

Conclusion

Maven 2.0 is a powerful tool that greatly simplifies and standardizes the build process. By promoting a standard project organization and recommended best practices, Maven handles much of the grunt work. And standard plug-ins such as the site generator provide valuable team-oriented project tools with little extra effort. Check it out!

Author Bio

John Ferguson Smart has been involved in the IT industry since 1991, and in J2EE development since 1999. His specialties are J2EE architecture and development and IT project management, including offshore project management. He has wide experience in open source Java technologies. He has worked on many large-scale J2EE projects for governments and businesses in both hemispheres, involving international and offshore

teams, and also writes technical articles in the J2EE field. His technical blog can be found at http://www.jroller.com/page/wakaleo.