IBM

developerWorks®

5 things you didn't know about ...: Apache Maven

Tips for managing the project life cycle with Maven

Steven Haines
Alex Theedom

May 17, 2017
(First published October 05, 2010)

You might be familiar with profiles, but did you know that you can use them in Maven to execute specific behaviors in different environments? This installment in the *5 things* series looks beyond Maven's build features, and even its basic tools for managing the project life cycle, delivering five tips that will improve the productivity and ease with which you manage applications in Maven.

Maven is an excellent build tool for Java™ developers, and you can use it to manage the life cycle of your projects as well. As a life-cycle management tool, Maven operates against phases rather than Ant-style build "tasks." Maven handles all phases of the project life cycle, including validation, code generation, compilation, testing, packaging, integration testing, verification, installation, deployment, and project site creation and deployment.

To understand the difference between Maven and a traditional build tool, consider the process of building a JAR file and an EAR file. Using Ant, you would need to define specific tasks to assemble each artifact. Maven, on the other hand, does most of that work for you: you just tell it whether the project is a JAR or EAR file, then instruct it to process the "package" phase. Maven will find the required resources and construct the files.

You'll find plenty of introductory tutorials for getting started with Maven, including some listed in this article's Related topics section. The five tips here are intended to help you with what comes next: the programming scenarios that arise when using Maven to manage the life cycle of your applications.

1. Executable JAR files

About this series

So you think you know about Java programming? The fact is, most developers scratch the surface of the Java platform, learning just enough to get the job done. In this ongoing series, Java technology sleuths dig beneath the core functionality of the Java platform, turning up tips and tricks that could help you solve even your stickiest programming challenges.

Building a JAR file with Maven is pretty easy: just define the project packaging as "jar" and then execute the package life-cycle phase. But defining an executable JAR file is more tricky. Doing this effectively involves the following steps:

1. Define a `main` class in your JAR's MANIFEST.MF file that defines the executable class. (MANIFEST.MF is the file that Maven generates when packaging your application.)
2. Find all of the libraries on which your project depends.
3. Include those libraries in your MANIFEST.MF file so that your application classes can find them.

You can do all of this manually, or you can do it more efficiently with the help of two Maven plug-ins: `maven-jar-plugin` and `maven-dependency-plugin`.

maven-jar-plugin

The `maven-jar-plugin` does many things, but here we're interested in using it to modify the contents of a default MANIFEST.MF file. In the plug-ins section of your POM file, add the code shown in Listing 1:

Listing 1. Using maven-jar-plugin to modify MANIFEST.MF

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <classpathPrefix>lib/</classpathPrefix>
                            <mainClass>com.mypackage.MyClass</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
```

All Maven plug-ins expose their configuration through a `<configuration>` element. In this example, the `maven-jar-plugin` modifies its `archive` attribute and specifically the archive's `manifest` attribute, which controls the contents of the MANIFEST.MF file. It includes three elements:

- `addClassPath`: Setting this element to *true* tells the `maven-jar-plugin` to add a `Class-Path` element to the MANIFEST.MF file, and to include all dependencies in that `Class-Path` element
- `classpathPrefix`: If you plan on including all of your dependencies in the same directory as the JAR you are building, then you can omit this element; otherwise, use `classpathPrefix` to specify the prefixes of all dependent JAR files. In Listing 1, `classpathPrefix` specifies that all dependencies should be located in a "`lib`" folder relative to the archive.
- `mainClass`: Use this element to define the name of the class to execute when the user executes the JAR file with a `java -jar` command.

maven-dependency-plugin

Once you've configured the MANIFEST.MF file with these three elements, your next step is to actually copy all of the dependencies to the `lib` folder. For this, you use the `maven-dependency-plugin`, as shown in Listing 2:

Listing 2. Using maven-dependency-plugin to copy dependencies to lib

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <executions>
                    <execution>
                        <id>copy</id>
                        <phase>install</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>
                              ${project.build.directory}/lib
                            </outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
```

The `maven-dependency-plugin` has a `copy-dependencies` goal that will copy your dependencies to the directory of your choosing. In this example, I copied dependencies to the `lib` directory under the `build` directory (`project-home/target/lib`).

With your dependencies and modified MANIFEST.MF in place, you can launch the application with a simple command:

```
java -jar jarfilename.jar
```

2. Customizing MANIFEST.MF

While the `maven-jar-plugin` allows you to modify common portions of a `MANIFEST.MF` file, there are times when you need a more customized MANIFEST.MF. The solution to this is two-fold:

1. Define all of your custom configurations in a "template" MANIFEST.MF file.
2. Configure the `maven-jar-plugin` to use your MANIFEST.MF file and augment it with any Maven customizations.

As an example, consider a JAR file that contains a Java agent. For a Java agent to run, it needs to define a `Premain-Class` and permissions. Listing 3 shows the contents of such a MANIFEST.MF file:

Listing 3. Premain-Class definition in a custom MANIFEST.MF file

```
Manifest-Version: 1.0
Premain-Class: com.geekcap.openapm.jvm.agent.Agent
Can-Redefine-Classes: true
Can-Retransform-Classes: true
Can-Set-Native-Method-Prefix: true
```

In Listing 3, I've specified that the `Premain-Classcom.geekcap.openapm.jvm.agent.Agent` will be granted permission to redefine and retransform classes. Next, I update the `maven-jar-plugin` to include the MANIFEST.MF file, as shown in Listing 4:

Listing 4. Including Premain-Class

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifestFile>
                          src/main/resources/META-INF/MANIFEST.MF
                        </manifestFile>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <classpathPrefix>lib/</classpathPrefix>
                            <mainClass>
                              com.geekcap.openapm.ui.PerformanceAnalyzer
                            </mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
```

Maven 3

Maven 2 has established its place as one of the most popular and well-used open source Java life-cycle management tools. Maven 3, promoted to alpha 5 in September 2010, brings some eagerly awaited changes to Maven. See the Related topics section to find out what's new in Maven 3.

This is an interesting example because it both defines a `Premain-Class` that allows the JAR file to work as a Java agent and has a `mainClass` that allows it to run as an executable JAR file. In this particular example, I've used `OpenAPM` (a code tracing tool that I built) to define code tracing that will be recorded by the Java agent and a user interface, which will facilitate analysis of recorded traces. In short, the example shows the power of combining an explicit manifest file with dynamic modifications.

3. Dependency trees

One of the most useful features of Maven is its support for dependency management: you simply define the libraries your application depends on, and Maven locates them (either in your local or a central repository), downloads them, and uses them to compile your code.

On occasion, you might need to know the origin of a particular dependency — such as if you were to find different and incompatible versions of the same JAR file in your build. In this case, you would need to prevent one version of the JAR file from being included in your build, but first you would need to locate the dependency holding the JAR.

Locating dependencies turns out to be surprisingly easy once you know the following command:

```
mvn dependency:tree
```

The `dependency:tree` argument displays all of your direct dependencies and then shows all sub-dependencies (and their sub-dependencies, and so on). For example, the Listing 5 is an excerpt from a client library required by one of my dependencies:

Listing 5. A Maven dependency tree

```
[INFO] ------------------------------------------------------------------------
[INFO] Building Client library for communicating with the LDE
[INFO]    task-segment: [dependency:tree]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:tree {execution: default-cli}]
[INFO] com.lmt.pos:sis-client:jar:2.1.14
[INFO] +- org.codehaus.woodstox:woodstox-core-lgpl:jar:4.0.7:compile
[INFO] |  \- org.codehaus.woodstox:stax2-api:jar:3.0.1:compile
[INFO] +- org.easymock:easymockclassextension:jar:2.5.2:test
[INFO] |  +- cglib:cglib-nodep:jar:2.2:test
[INFO] |  \- org.objenesis:objenesis:jar:1.2:test
```

You can see in Listing 5 that the `sis-client` project requires the `woodstox-core-lgpl` and the `easymockclassextension` libraries. The `easymockclassextension` library, in turn, requires the `cglib-nodep` library and the `objenesis` library. If I were having problems with `objenesis`, such as two versions, 1.2 and 1.3, then this dependency tree would show me that the 1.2 artifact was being imported indirectly by the `easymockclassextension` library.

The `dependency:tree` argument has saved me many hours of debugging a broken build; I hope it will do the same for you.

4. Building for different environments

Most substantial projects have at least a core group of environments consisting of tasks related to development, quality assurance (QA), integration, and production. The challenge of managing all of those environments is in configuring your build, which has to connect to the correct database, execute the correct set of scripts, and deploy all the right artifacts to each environment. Using Maven profiles lets you do all this without having to build explicit instructions for each environment individually.

The key is in combining environmental profiles with task-oriented ones. Each environmental profile defines its specific locations, scripts, and servers. So, in my pom.xml file, I would define the task-oriented build as shown in Listing 6:

Listing 6. A deployment build

```
 <build>
    <plugins>
        <plugin>
            <groupId>net.fpic</groupId>
            <artifactId>tomcat-deployer-plugin</artifactId>
            <version>1.0-SNAPSHOT</version>
            <executions>
                <execution>
                    <id>pos</id>
                    <phase>install</phase>
                    <goals>
                        <goal>deploy</goal>
                    </goals>
                    <configuration>
                        <host>${deploymentManagerRestHost}</host>
                        <port>${deploymentManagerRestPort}</port>
                        <username>${deploymentManagerRestUsername}</username>
                        <password>${deploymentManagerRestPassword}</password>
                        <artifactSource>
                            address/target/addressservice.war
                        </artifactSource>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

This build executes the `tomcat-deployer-plugin`, which is configured to connect to a specific host, port, and to specific username and password credentials. All of this information is defined using variables, such as `${deploymentmanagerRestHost}`. These variables are defined in my profiles on a per-environment basis, as shown in Listing 7:

Listing 7. Environment profiles

```
<profiles>
    <profile>
        <id>dev</id>
        <properties>
            <deploymentManagerRestHost>10.50.50.52</deploymentManagerRestHost>
            <deploymentManagerRestPort>58090</deploymentManagerRestPort>
            <deploymentManagerRestUsername>myusername</deploymentManagerRestUsername>
            <deploymentManagerRestPassword>mypassword</deploymentManagerRestPassword>
        </properties>
    </profile>
    <profile>
        <id>qa</id>
        <properties>
            <deploymentManagerRestHost>10.50.50.50</deploymentManagerRestHost>
            <deploymentManagerRestPort>58090</deploymentManagerRestPort>
            <deploymentManagerRestUsername>myotherusername</deploymentManagerRestUsername>
            <deploymentManagerRestPassword>myotherpassword</deploymentManagerRestPassword>
        </properties>
    </profile>
</profiles>
```

Deploying Maven profiles

In the profiles in Listing 7, I defined two profiles and activated them based on the value in the profile name. If the profile selected were `dev` then the development deployment information would be used. If the profile selected were `qa`, then the QA deployment information would be used, and so on.

Here's the command to deploy to development:

```
mvn -Pdev clean install
```

The `–Pdev` flag tells Maven to activate the development configuration and passing `-Pqa` would activate the QA configuration.

5. Custom Maven plug-ins

Maven puts dozens of prebuilt plug-ins at your disposal, but at some point you might find yourself in need of a custom plug-in. Building a custom Maven plug-in is straightforward:

1. Create a new project with the POM packaging set to "`maven-plugin`."
2. Include an invocation of the `maven-plugin-plugin` that defines your exposed plug-in goals.
3. Create a Maven plug-in "`mojo`" class (a class that extends `AbstractMojo`).
4. Annotate the class to define goals and variables that will be exposed as configuration parameters. The annotation `@Mojo` is required, and controls how and when the mojo is executed.
5. Implement an `execute()` method that will be invoked when your plug-in is invoked.

As an example, Listing 8 shows relevant portions of a custom plug-in designed to deploy Tomcat:

Listing 8. TomcatDeployerMojo.java

```
package net.fpic.maven.plugins;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugins.annotations.LifecyclePhase;
import org.apache.maven.plugins.annotations.Mojo;
import org.apache.maven.plugins.annotations.Parameter;

/**
 * Goal that deploys a web application to Tomcat
 */
@Mojo(name = "deploy", defaultPhase = LifecyclePhase.INSTALL)
public class TomcatDeployerMojo extends AbstractMojo {
    /**
     * The host name or IP address of the deployment server
     */
    @Parameter(alias = "host", property = "deploy.host", required = true)
    private String serverHost;

    /**
     * The port of the deployment server
     */
    @Parameter(alias = "port", property = "deploy.port", defaultValue = "58020", required = true)
    private String serverPort;

    /**
     * The username to connect to the deployment manager (if omitted then the plugin
     * attempts to deploy the application to the server without credentials)
     */
    @Parameter(alias = "username", property = "deploy.username")
    private String username;

    /**
     * The password for the specified username
     */
    @Parameter(alias = "password", property = "deploy.password")

    private String password;

    /**
     * The name of the source artifact to deploy, such as target/pos.war
     */
    @Parameter(alias = "artifactSource", property = "deploy.artifactSource", required = true)
    private String artifactSource;

    /**
     * The destination name of the artifact to deploy, such as ROOT.war.
     * If not present then the
     * artifact source name is used (without pathing information)
     */
    @Parameter(alias = "artifactDestination", property = "deploy.artifactDestination")
    private String artifactDestination;

    public void execute() throws MojoExecutionException {
        getLog().info("Server Host: " + serverHost +
                ", Server Port: " + serverPort +
                ", Artifact Source: " + artifactSource +
                ", Artifact Destination: " + artifactDestination);

        // Validate our fields
        if (serverHost == null) {
            throw new MojoExecutionException(
                    "No deployment host specified, deployment is not possible");
        }
        if (artifactSource == null) {
            throw new MojoExecutionException(
                    "No source artifact is specified, deployment is not possible");
        }

...
    }
}
```

At the class level, the `@Mojo` annotation value `name` specifies the goal that this MOJO executes and the `lifecyclePhase` specifies the phase in which the goal executes. Each exposed property has a `@Parameter` annotation that specifies the alias through which the parameter will be executed, in addition to an expression that maps to a system property containing the actual value. If the property has a `required=true`, value set then it is required. If it has a `defaultValue`, then that value will be used if one is not specified. In the `execute()` method, you can invoke `getLog()` to gain access to the Maven logger, which, depending on the logging level, will output the specified message to the standard output device. If the plug-in fails, throwing a `MojoExecutionException` will cause the build to fail.

Conclusion

You can use Maven just for builds, but at its best Maven is a project life-cycle management tool. This article has presented five lesser known features that can help you become more effective at using Maven. See the Related topics section to learn more about Maven.

Related topics

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- "Introduction to Apache Maven 2" (Sing Li, developerWorks, December 2006): Familiarize yourself with the fundamental skills required to work on projects built using Maven 2, starting with this developerWorks tutorial.
- "Managing Java Build Lifecycles with Maven" (Steven Haines, InformIT.com, July 2009): This tutorial describes the basis for Maven, the architecture behind it, and how to use it to manage the build life cycles of your Java projects.
- *Maven: The Complete Reference, Edition 0.7* (Tim O'Brien, et al.; Sonatype 2010): This free online book is an excellent resource for learning Maven, including tips for using the forthcoming Maven 3. Publisher SonaType is a software vendor that develops one of the most popular Maven repository projects, Nexus.
- Maven 3 release notes: Learn about changes to Maven in version 3, including improvements to its usability and performance.
- Maven home page: Download Maven and learn more about it from the Apache Software Foundation.