ALL GUIDES

# Building Java Projects with Maven

This guide walks you through using Maven to build a simple Java project.

## What you'll build

You'll create an application that provides the time of day and then build it with Maven.

## What you'll need

- About 15 minutes
- A favorite text editor or IDE
- JDK 8 or later

## How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Set up the project.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-maven.git`
- cd into `gs-maven/initial`
- Jump ahead to [initial].

**When you finish**, you can check your results against the code in `gs-maven/complete`.

## Set up the project

First you'll need to setup a Java project for Maven to build. To keep the focus on Maven, make the project as simple as possible for now. Create this structure in a project folder of your choosing.

### Create the directory structure

In a project directory of your choosing, create the following subdirectory structure; for example, with `mkdir -p src/main/java/hello` on *nix systems:

```
└── src
    └── main
        └── java
            └── hello
```

Within the `src/main/java/hello` directory, you can create any Java classes you want. To maintain consistency with the rest of this guide, create these two classes: `HelloWorld.java` and `Greeter.java`.

`src/main/java/hello/HelloWorld.java`

```
package hello;

public class HelloWorld {
  public static void main(String[] args) {
    Greeter greeter = new Greeter();
    System.out.println(greeter.sayHello());
  }
}
```

COPY

`src/main/java/hello/Greeter.java`

```
package hello;

public class Greeter {
  public String sayHello() {
    return "Hello world!";
  }
}
```

COPY

Now that you have a project that is ready to be built with Maven, the next step is to install Maven.

Maven is downloadable as a zip file at https://maven.apache.org/download.cgi. Only the binaries are required, so look for the link to apache-maven-*{version}*-bin.zip or apache-maven-*{version}*-bin.tar.gz.

Once you have downloaded the zip file, unzip it to your computer. Then add the *bin* folder to your path.

To test the Maven installation, run `mvn` from the command-line:

```
mvn -v
```

If all goes well, you should be presented with some information about the Maven installation. It will look similar to (although perhaps slightly different from) the following:

```
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T16:41:47+00:00)
Maven home: /home/dsyer/Programs/apache-maven
Java version: 1.8.0_152, vendor: Azul Systems, Inc.
Java home: /home/dsyer/.sdkman/candidates/java/8u152-zulu/jre
Default locale: en_GB, platform encoding: UTF-8
OS name: "linux", version: "4.15.0-36-generic", arch: "amd64", family: "unix"
```

Congratulations! You now have Maven installed.

INFO: You might like to consider using the Maven wrapper to insulate your developers against having the correct version of Maven, or having to install it at all. Projects downloaded from Spring Initializr have the wrapper included. It shows up as a script `mvnw` in the top level of your project which you run in place of `mvn`.

## Define a simple Maven build

Now that Maven is installed, you need to create a Maven project definition. Maven projects are defined with an XML file named *pom.xml*. Among other things, this file gives the project's name, version, and dependencies that it has on external libraries.

Create a file named *pom.xml* at the root of the project (i.e. put it next to the `src` folder) and give it the following contents:

`pom.xml`

COPY

```
<?xml version="1.0" encoding="UTF-8"?>
<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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.springframework</groupId>
    <artifactId>gs-maven</artifactId>
    <packaging>jar</packaging>
    <version>0.1.0</version>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
```

```
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                    <mainClass>hello.HelloWorld</mainClass>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

With the exception of the optional `<packaging>` element, this is the simplest possible *pom.xml* file necessary to build a Java project. It includes the following details of the project configuration:

- `<modelVersion>`. POM model version (always 4.0.0).
- `<groupId>`. Group or organization that the project belongs to. Often expressed as an inverted domain name.
- `<artifactId>`. Name to be given to the project's library artifact (for example, the name of its JAR or WAR file).
- `<version>`. Version of the project that is being built.
- `<packaging>` - How the project should be packaged. Defaults to "jar" for JAR file packaging. Use "war" for WAR file packaging.

> When it comes to choosing a versioning scheme, Spring recommends the semantic versioning approach.

At this point you have a minimal, yet capable Maven project defined.

## Build Java code

Maven is now ready to build the project. You can execute several build lifecycle goals with Maven now, including goals to compile the project's code, create a library package (such as a JAR file), and install the library in the local Maven dependency repository.

To try out the build, issue the following at the command line:

```
mvn compile
```

This will run Maven, telling it to execute the *compile* goal. When it's finished, you should find the compiled *.class* files in the *target/classes* directory.

Since it's unlikely that you'll want to distribute or work with *.class* files directly, you'll probably want to run the *package* goal instead:

```
mvn package
```

The *package* goal will compile your Java code, run any tests, and finish by packaging the code up in a JAR file within the *target* directory. The name of the JAR file will be based on the project's `<artifactId>` and `<version>`. For example, given the minimal *pom.xml* file from before, the JAR file will be named *gs-maven-0.1.0.jar*.

To execute the JAR file run:

```
java -jar target/gs-maven-0.1.0.jar
```

> If you've changed the value of `<packaging>` from "jar" to "war", the result will be a WAR file within the *target* directory instead of a JAR file.

Maven also maintains a repository of dependencies on your local machine (usually in a *.m2/repository* directory in your home directory) for quick access to project dependencies. If you'd like to install your project's JAR file to that local repository, then you should invoke the `install` goal:

```
mvn install
```

The *install* goal will compile, test, and package your project's code and then copy it into the local dependency repository, ready for another project to reference it as a dependency.

Speaking of dependencies, now it's time to declare dependencies in the Maven build.

## Declare Dependencies

The simple Hello World sample is completely self-contained and does not depend on any additional libraries. Most applications, however, depend on external libraries to handle common and complex functionality.

For example, suppose that in addition to saying "Hello World!", you want the application to print the current date and time. While you could use the date and time facilities in the native Java libraries, you can make things more interesting by using the Joda Time libraries.

First, change HelloWorld.java to look like this:

`src/main/java/hello/HelloWorld.java`

```
package hello;

import org.joda.time.LocalTime;

public class HelloWorld {
  public static void main(String[] args) {
    LocalTime currentTime = new LocalTime();
    System.out.println("The current local time is: " + currentTime);
    Greeter greeter = new Greeter();
    System.out.println(greeter.sayHello());
  }
}
```

COPY

Here `HelloWorld` uses Joda Time's `LocalTime` class to get and print the current time.

If you were to run `mvn compile` to build the project now, the build would fail because you've not declared Joda Time as a compile dependency in the build. You can fix that by adding the following lines to *pom.xml* (within the `<project>` element):

```
<dependencies>
                <dependency>
                        <groupId>joda-time</groupId>
                        <artifactId>joda-time</artifactId>
                        <version>2.9.2</version>
                </dependency>
</dependencies>
```

COPY

This block of XML declares a list of dependencies for the project. Specifically, it declares a single dependency for the Joda Time library. Within the `<dependency>` element, the dependency coordinates are defined by three sub-elements:

- `<groupId>` - The group or organization that the dependency belongs to.
- `<artifactId>` - The library that is required.
- `<version>` - The specific version of the library that is required.

By default, all dependencies are scoped as `compile` dependencies. That is, they should be available at compile-time (and if you were building a WAR file, including in the */WEB-INF/libs* folder of the WAR). Additionally, you may specify a `<scope>` element to specify one of the following scopes:

- `provided` - Dependencies that are required for compiling the project code, but that will be provided at runtime by a container running the code (e.g., the Java Servlet API).
- `test` - Dependencies that are used for compiling and running tests, but not required for building or running the project's runtime code.

Now if you run `mvn compile` or `mvn package`, Maven should resolve the Joda Time dependency from the Maven Central repository and the build will be successful.

## Write a Test

First add JUnit as a dependency to your pom.xml, in the test scope:

COPY

```
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
</dependency>
```

Then create a test case like this:

`src/test/java/hello/GreeterTest.java`

COPY

```
package hello;

import static org.hamcrest.CoreMatchers.containsString;
import static org.junit.Assert.*;

import org.junit.Test;

public class GreeterTest {

  private Greeter greeter = new Greeter();

  @Test
  public void greeterSaysHello() {
    assertThat(greeter.sayHello(), containsString("Hello"));
  }

}
```

Maven uses a plugin called "surefire" to run unit tests. The default configuration of this plugin compiles and runs all classes in `src/test/java` with a name matching `*Test`. You can run the tests on the command line like this

```
mvn test
```

or just use `mvn install` step as we already showed above (there is a lifecycle definition where "test" is included as a stage in "install").

Here's the completed `pom.xml` file:

`pom.xml`

COPY

```
<?xml version="1.0" encoding="UTF-8"?>
<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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>

        <groupId>org.springframework</groupId>
        <artifactId>gs-maven</artifactId>
        <packaging>jar</packaging>
        <version>0.1.0</version>

        <properties>
                <maven.compiler.source>1.8</maven.compiler.source>
                <maven.compiler.target>1.8</maven.compiler.target>
        </properties>

        <dependencies>
                <!-- tag::joda[] -->
                <dependency>
                        <groupId>joda-time</groupId>
                        <artifactId>joda-time</artifactId>
                        <version>2.9.2</version>
                </dependency>
                <!-- end::joda[] -->
                <!-- tag::junit[] -->
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>4.12</version>
                        <scope>test</scope>
                </dependency>
                <!-- end::junit[] -->
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.apache.maven.plugins</groupId>
                                <artifactId>maven-shade-plugin</artifactId>
                                <version>3.2.4</version>
                                <executions>
                                        <execution>
                                                <phase>package</phase>
                                                <goals>
                                                        <goal>shade</goal>
                                                </goals>
                                                <configuration>
                                                        <transformers>
                                                                <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
```

```
<mainClass>hello.HelloWorld</mainClass>
                                                                    </transformer>
                                                                </transformers>
                                                            </configuration>
                                                        </execution>
                                                    </executions>
                                                </plugin>
                                            </plugins>
                                        </build>

</project>
```

The completed **pom.xml** file is using the Maven Shade Plugin for the simple convenience of making the JAR file executable. The focus of this guide is getting started with Maven, not using this particular plugin.

## Summary

Congratulations! You've created a simple yet effective Maven project definition for building Java projects.

## See Also

The following guides may also be helpful:

- Building Java Projects with Gradle

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.