

Chapter 43

Scala Build Tools



43.1 Introduction

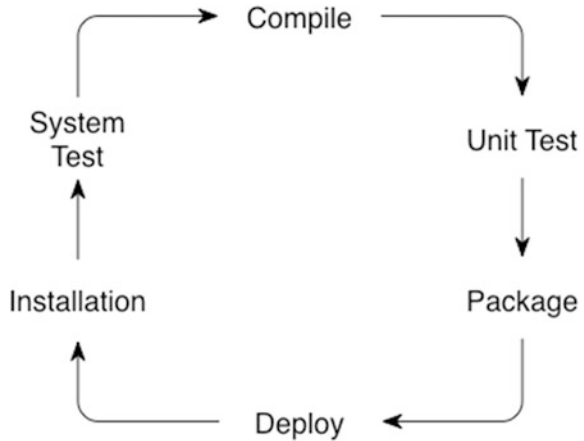
There are many ways in which a Scala application can be *built*. These include the REPL loop and automated compilation within an IDE such as the Eclipse-based Scala IDE. However, neither of these is suitable for centrally building large applications as might be found within many commercial organisations. The most popular build environments for Scala are Maven and SBT. In this chapter, we will briefly examine both so that you have a flavour of both tools and the potential benefits and drawbacks of each.

43.2 Why We Need a Build Tool

The first question to consider is why we need a build tool in the first place. In this book, we have been compiling our Scala applications using the IntelliJ IDE although we could have compiled them from the command line or used the Scala REPL interpreter. However, many applications are comprised of many parts, all of which need to be processed in the appropriate way and package as required. For example, many Web applications are made up of multiple components:

- HTML files and image files
- PHP scripts
- Java or Scala services
- The libraries used by Scala and Java
- Configuration files
- Database scripts
- Property or metadata data files

Fig. 43.1 Build cycle for applications



Multiple people may develop all of these elements at different times and on different machines. In addition, the process of building a project may require several steps or involve different phases, such as compiling the code, testing, the code, packaging the system up as required by the target platform, installing it on that platform and deploying it into the target runtime environment. This is illustrated in Fig. 43.1.

These steps must be repeatable and must bring together a diverse range of elements.

As mentioned before we could, of course, use our favourite IDEs or write our own build scripts; however:

- You can build your projects manually but this is tedious and error-prone.
- You can use IDEs like Scala Eclipse. This approach is easy, but not very portable to server environments, and requires each person to build their part independently of others.
- You can write scripts to automate the process using tools such as Ant. This approach does have benefits to commend it; however, a great deal of time can be spent on designing, testing and maintaining the scripts.

Another approach is to use a dedicated build tool such as Maven or SBT. Both these tools come with useful internal or default knowledge about different types of projects, how to build them (for Java and Scala) and what constitutes the normal build cycle for different types of applications.

43.3 Maven

Maven is an industry standard project build tool that understands project lifecycle as well as the steps that make up such a lifecycle (see <http://maven.apache.org>).

Maven was originally designed for the Java programming language but is equally applicable to Scala and thus is very widely used within industry.

Maven is a convention over configuration-based system. This means that if you follow the standard conventions then you do not need to explicitly specify additional information. For example, if you are creating a Web application then as long as you follow the conventions, Maven will know where to find the elements that make up a Web application and can package them appropriately when asked to build the system. This greatly reduces the amount of project set-up and management required. However, the defaults are primarily oriented towards Java applications and thus for Scala we need to indicate that we are working with Scala and thus there are additional configurations required (although in practice these can be provided by a Maven archetype—a type of template that provides defaults for different types of projects).

The other major feature of Maven is its ability to handle the dependencies that applications have to libraries (and additionally the transitive dependencies that these libraries themselves have). Maven does this using dependency information associated with the type of the project and the specific libraries used by the developer. To access the definitions for these libraries it uses a dependency framework that can download libraries from a central repository. This idea is illustrated in Fig. 43.2. When a developer runs a Maven command, Maven checks to see if it is necessary to access a particular library. If that is required it will look in a number of predefined repository locations and access the first that it finds.

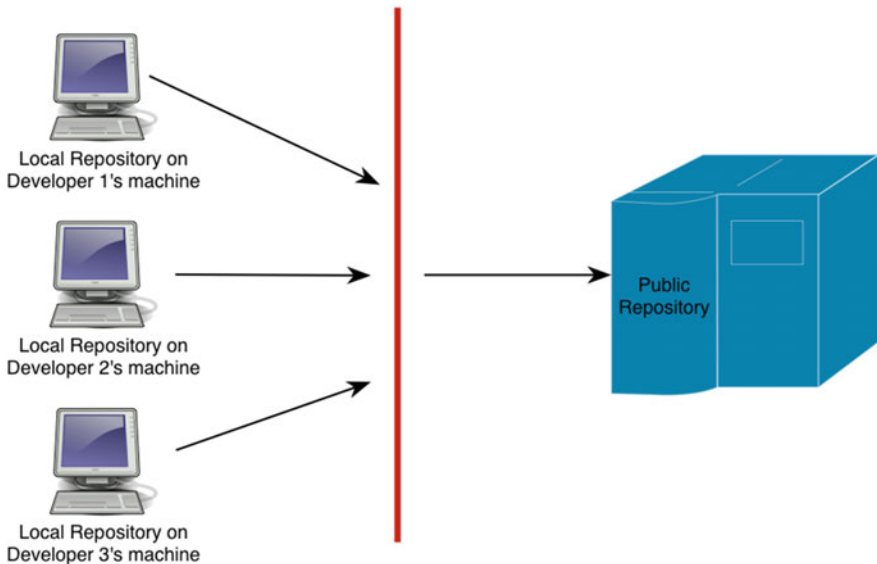


Fig. 43.2 Local versus remote repository structure

Maven also understands the concept of versioning of libraries and thus it can distinguish between the latest release of a library as well as previous releases of that library (such as release 1, 2 or 3) and access an appropriate version.

43.3.1 *Maven Repositories*

The concept of a repository is very important within Maven, and there are two flavours of repository as shown in Fig. 43.2. These are remote and local repositories:

- **Remote repository.** A remote repository is accessed over a network and may be hosted internally to an organisation and/or externally on the Internet. The central Maven repository can be viewed via a browser at <http://mvnrepository.com>. In many cases an organisation will have their own version either to control the library versions used or to improve performance.
- **Local repository.** A local cache of downloaded artefacts, libraries and latest builds is maintained on each developer's machine. Maven first checks locally before trying to download a library—thus reducing the overhead of library access.

The use of repositories and library version information is one of Maven's biggest benefits.

43.3.2 *The Maven POM*

The core concept within Maven is the Project Object Model or POM (and example is shown in Fig. 43.3). The POM is actually an XML file that is used to tell Maven what type of project is being created and to provide Maven with any additional configuration information that cannot be deduced using the convention over configuration model.

The POM contains detailed metadata information about the project, including

- Organisational information (such as your group Id which is often your organisation's domain in reverse) and project-specific information such as the name of the project and the version of the project being built.
- Dependencies such as libraries being used within the project. For example, ScalaTest is a common library to specify.
- The type of project being constructed such as a stand-alone application, a Web application, a service.
- Application and testing resources such as data or configuration files.

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4     http://maven.apache.org/maven-v4_0_0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6   <groupId>com.jjh.helloworld</groupId>
7   <artifactId>helloworld</artifactId>
8   <packaging>jar</packaging>
9   <version>1.0-SNAPSHOT</version>
10  <name>helloworld</name>
11  <url>http://maven.apache.org</url>
12  <dependencies>
13    <dependency>
14      <groupId>junit</groupId>
15      <artifactId>junit</artifactId>
16      <version>3.8.1</version>
17      <scope>test</scope>
18    </dependency>
19  </dependencies>
20 </project>

```

Fig. 43.3 Simple Java POM

The POM file can also be used to override any of the default assumptions made by Maven but this is often not required.

The Maven conventions not specified in Fig. 43.3 include the location of the source code and the test code, the locations of the repositories containing the libraries, the steps involved in constructing a stand-alone application, etc. These are all defaulted. For example, Maven assumes that all source code is found in the following locations:

- src/main/java
- src/main/test

This is why it is common to find such structures in many other (non-Maven) projects.

Of course, looking at the above directories you will note that they specify *java* in the path; we are working with Scala and thus this is one of the things that must be changed (or at least added) if we are to use Maven to build a Scala application. This is considered in the next section.

43.3.3 Scala and Maven

To use Maven with Scala we need to add some additional information to the project POM file. We need to indicate that we are using Scala and that location of our Scala code will not be under Java. In Fig. 43.4 we add a dependency specifying that we are using Scala and specify that the Scala source code can be found in:

- src/main/scala
- src/test/scala

Note that this does not stop us having multiple source directories under src/main and src/test, and it is not uncommon in projects that use multiple languages to have a structure such as:

- src/main/java
- src/main/scala
- src/main/javascript

```

1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4          http://maven.apache.org/maven-v4_0_0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6      <groupId>com.jeh</groupId>
7      <artifactId>HelloWorld</artifactId>
8      <version>0.0.1-SNAPSHOT</version>
9      <inceptionYear>2008</inceptionYear>
10     <properties>
11         <scala.version>2.10.3</scala.version>
12     </properties>
13
14     <repositories>[]
21
22     <pluginRepositories>[]
29
30     <dependencies>
31         <dependency>
32             <groupId>org.scala-lang</groupId>
33             <artifactId>scala-library</artifactId>
34             <version>${scala.version}</version>
35         </dependency>
36     </dependencies>
37
38     <build>
39         <sourceDirectory>src/main/scala</sourceDirectory>
40         <testSourceDirectory>src/test/scala</testSourceDirectory>
41         <plugins>[]
78     </build>
79     <reporting>[]
90 </project>

```

Fig. 43.4 A Scala configuration for a Maven POM file

```

project/
  pom.xml - Defines the project
  src/
    main/
      java/ - Contains all java code that will go in your final artifact.
              See maven-compiler-plugin for details
      scala/ - Contains all scala code that will go in your final artifact.
              See maven-scala-plugin for details
      resources/ - Contains all static files that should be available on the classpath
                  in the final artifact. See maven-resources-plugin for details
      webapp/ - Contains all content for a web application (jspx, css, images, etc.)
              See maven-war-plugin for details
      site/ - Contains all apt or xdoc files used to create a project website.
            See maven-site-plugin for details
    test/
      java/ - Contains all java code used for testing.
              See maven-compiler-plugin for details
      scala/ - Contains all scala code used for testing.
              See maven-scala-plugin for details
      resources/ - Contains all static content that should be available on the
                  classpath during testing. See maven-resources-plugin for details

```

Fig. 43.5 Scala Maven project structure

- src/test/java
- src/test/scala
- src/test/javascript

Thus the final default project structure for such an application may be modified to include both a Java and a Scala root directory within the *main* and *test* paths. This is illustrated in Fig. 43.5.

The dependency entry that specifies that we are using Scala indicates the *groupId* for Scala (essentially the Scala organisation’s domain) and the *artifactId* (the name of the library) followed by the version. In the POM file the version is indicated by a Maven property *scala.version*, which is set at the top of the file. This means that it is easy to find and change the version of Scala being used—this is a common idiom in Maven files. The end result is that the actual dependency is as shown below:

```

<dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala-library</artifactId>
  <version>2.10.3</version>
</dependency>

```

We must also indicate where to find the Scala libraries if they are not available within the main Maven repository. For a Scala project, Maven must be told where to find the Scala libraries. This may occur if you wish to use a non-standard or milestone version of Scala. In our case we are also adding the Scala tools repository as this provides some bridging tools between Maven and Scala:

```

<repositories>
  <repository>
    <id>scala-tools.org</id>
    <name>Scala-Tools Maven2 Repository</name>
    <url>http://scala-tools.org/repo-releases</url>
  </repository>
</repositories>

```

43.3.4 *Maven Lifecycle Commands*

The same build lifecycle commands can be used whatever the project is (as Maven understands what they mean relative to that type of project), and thus we have the following available:

- **validate**—validate the project; check it complies with the rules for that type of project.
- **compile**—compile the source code into .class files.
- **test**—test the compiled source code; run the tests defined within the src/test directory.
- **package**—package in distributable format, e.g. jar
- **install**—install the package into the local repository
- **deploy**—copies the final package to the remote repository for sharing with other developers and projects.

If you are using an IDE, such as the Scala IDE, then you can use a Maven plugin that will help with creating projects, issuing Maven commands, finding dependencies, etc. For example, using the New Maven Project wizard with the Scala IDE displays a Maven archetype selection dialog. A Maven archetype is essentially a definition or template for a particular type of project. There are archetypes for Web-based applications, archetypes for particular frameworks and a Scala archetype (as shown in Fig. 43.6).

The end result of creating a new project in this way is that the default Scala Maven project structure is created as shown in Fig. 43.7.

43.4 SBT

Although Maven was intended to simplify the definition and construction of applications, it can seem somewhat complex for very simple applications. As a consequence there have been initiatives to further simplify the project definition and build process. One such initiative is the Simple Build Tool, known as SBT for short.

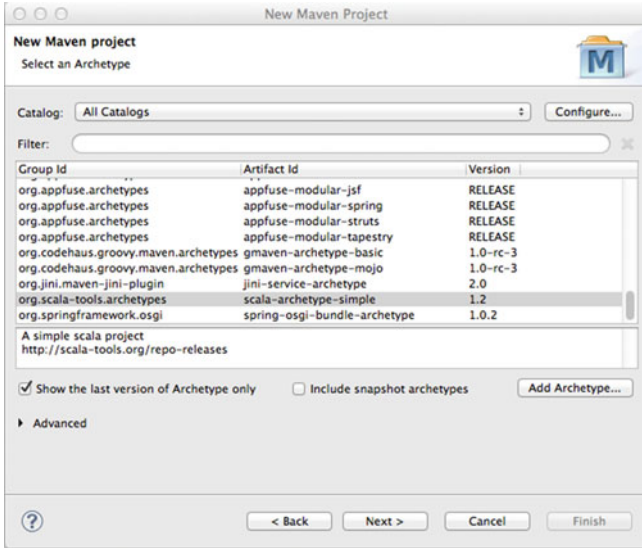


Fig. 43.6 Selecting the Scala Maven archetype

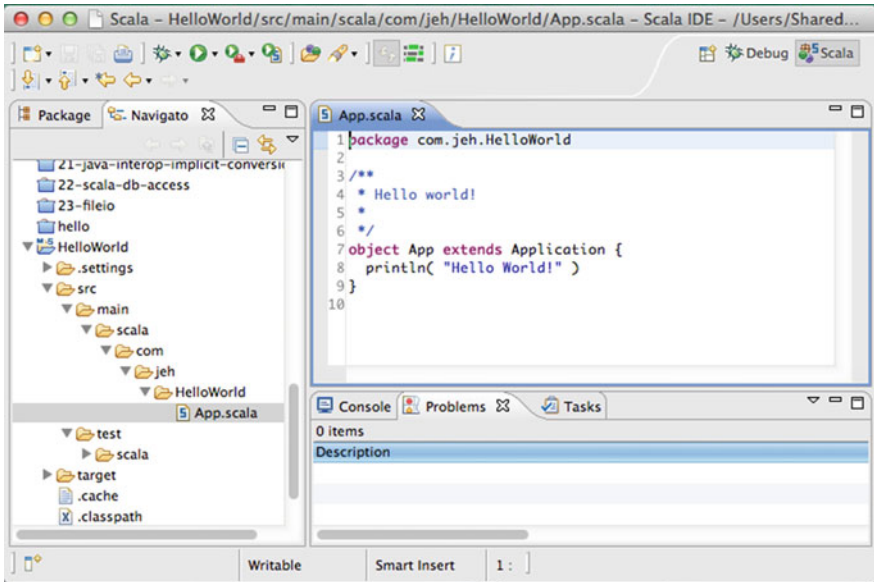


Fig. 43.7 A Scala Maven project in the Scala IDE

SBT is an Open Source build system for Scala (and for Java although it was originally designed for, and is implemented in, Scala). The key features of the SBT are:

- Minimal configuration for simple projects,
- Support for Scala and (many) Scala Test frameworks,
- Build tasks written in Scala DSL,
- Dependency support via Ivy. Ivy is a dependencies management system that handles the version of, and dependencies between, libraries,
- Integration with Scala interpreter.

It is used in many Scala projects including in the construction of Scala itself. It aims to simplify the build process to allow easy creation, compilation and deployment of Scala-based applications.

43.4.1 *Creating an SBT Project*

To create a project using SBT you need to take the following steps:

- Install SBT and create a script to launch it.
- Create a project directory with source files in it.
- Create your build definition.

SBT can be downloaded from the main SBT home page (<http://www.scala-sbt.org>)—there are a number of ways in which it is distributed but the simplest in many cases will be to download the `sbt.zip` file. Once you extract the SBT content into an appropriate location you will need to configure it for your environment—see the guidance on the SBT download page for your platform.

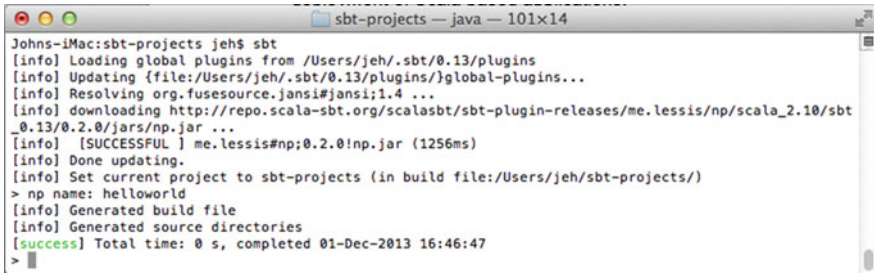
By following the SBT conventions we can get started by ensuring that:

- Sources in the base directory
- Sources in `src/main/scala` or `src/main/java`
- Tests in `src/test/scala` or `src/test/java`
- Data files in `src/main/resources` or `src/test/resources`
- jars in `lib` directory

By default, SBT will build projects with the same version of Scala used to run SBT itself.

SBT provides an interactive mode (or console) in which you can use the SBT console to issue a series of commands and control the build process. Using the `sbt` command without any options allows the user to enter the interactive SBT console.

We will use SBT to create a new project via the SBT console. To do this we can use the `np` (new project) SBT plugin; prior to version 0.13 of SBT you could create a default project directly using SBT; however, it is now necessary to use the `np` plugin (see the `np` site for directions on how to do this). This is indicated in



```
Johns-iMac:sbt-projects jeh$ sbt
[info] Loading global plugins from /Users/jeh/.sbt/0.13/plugins
[info] Updating {file:/Users/jeh/.sbt/0.13/plugins/}global-plugins...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] downloading http://repo.scala-sbt.org/scalasbt/sbt-plugin-releases/me.lessis/np/scala_2.10/sbt_0.13/0.2.0/jars/np.jar ...
[info] [SUCCESSFUL ] me.lessis#np;0.2.0!np.jar (1256ms)
[info] Done updating.
[info] Set current project to sbt-projects (in build file:/Users/jeh/sbt-projects/)
> np name: helloworld
[info] Generated build file
[info] Generated source directories
[success] Total time: 0 s, completed 01-Dec-2013 16:46:47
>
```

Fig. 43.8 Creating a project using SBT

Fig. 43.8. When issuing the *np* command from within the SBT console, it is necessary to provide the information for the name of the project and your organisation, etc. The actual template structure follows that of Maven described earlier.

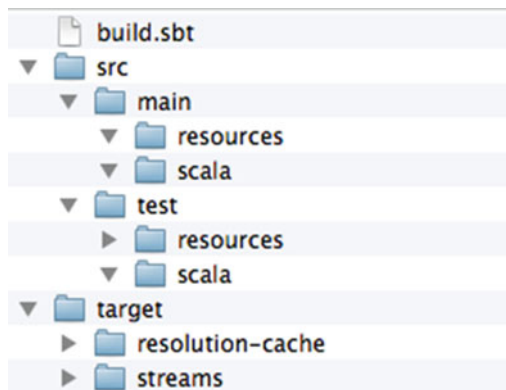
Once we have created a new project, we can now place our application code under the *src/main/scala* directory structure and place our tests under the *src/test/scala* structure (as shown in Fig. 43.9). The resources directory is for data files, property files, etc., that might be used with our application.

Although the aim of SBT is that there is little or no configuration required for simple projects, there are two ways of configuring SBT. The two approaches are the *light* configuration and the *full* configuration.

The light configuration approach is more akin to Ant (another Java build tool), which is an imperative approach to the definition of a build. That is, the light approach allows you to specify what constitutes the elements that comprise the version of the system to build explicitly but using a lightweight syntax. Figure 43.10 illustrates the SBT configuration file created for our new project earlier.

The *build.sbt* file is also where we would place any library dependency information for our project. For example,

Fig. 43.9 Project structure



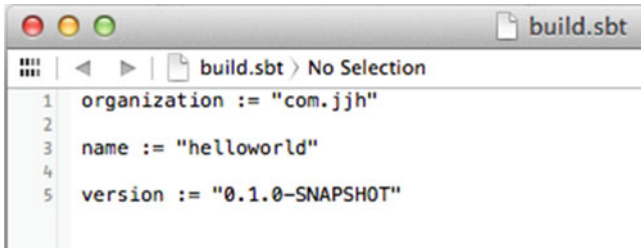


Fig. 43.10 SBT build.sbt configuration file

```
// Set the project name and version
name := "my-project"
version := "1.0.0"
// Add a single dependency, for tests.
libraryDependencies += "junit" % "junit" % "4.8" % "test"
// Add multiple dependencies.
libraryDependencies ++= Seq(
  "net.databinder" %% "dispatch-google" % "0.7.8",
  "net.databinder" %% "dispatch-meetup" % "0.7.8" )
```

This specifies ‘my-project’ as the name of the project; that it is version 1.0.0; and that it is dependent on the JUnit library as well as two additional libraries.

The full configuration approach is essentially a Scala program implemented using the SBT DSL for defining the build process. As such you can write almost anything in the full configuration. However, the DSL makes it easier to specify that common tasks are to be performed. An example of part of a full configuration file is shown in Fig. 43.11. As can be seen from this example, it captures similar information to the simple *build.sbt* file but does it in terms of Scala objects.

43.4.2 SBT Lifecycle Commands

In a similar manner to Maven, SBT can be used to issue a series of build commands such as

- **clean** Deletes all generated files (in the target directory).
- **compile** Compiles the main sources (in `src/main/scala` and `src/main/java` directories).
- **test** Compiles and runs all tests.

```

import sbt._
import Keys._

object BuildSettings {
  val buildOrganization = "odp"
  val buildVersion      = "2.0.29"
  val buildScalaVersion = "2.9.0-1"

  val buildSettings = Defaults.defaultSettings ++ Seq (
    organization := buildOrganization,
    version      := buildVersion,
    scalaVersion := buildScalaVersion,
    shellPrompt  := ShellPrompt.buildShellPrompt
  )
}

```

Fig. 43.11 Part of a Scala definition for a full configuration file for SBT

- **console** Starts the Scala interpreter with a classpath including the compiled sources and all dependencies. To return to sbt, type: quit, Ctrl+D (Unix), or Ctrl+Z (Windows).
- **run** <argument>* Runs the main class for the project in the same virtual machine as sbt.
- **package** Creates a jar file containing the files in src/main/resources and the classes compiled from src/main/scala and src/main/java.
- **help** <command> Displays detailed help for the specified command. If no command is provided, displays brief descriptions of all commands.

However, SBT can be used with Eclipse as an underlying tool. This requires the installation of the SBT plugin in a similar manner to the installation of a Maven plugin.

Online References

Ant home page <http://ant.apache.org/>

Maven home page <http://maven.apache.org/>

Maven and Scala <http://www.scala-lang.org/node/347>

Maven Repository <http://mvnrepository.com/>

Simple Build Tool (SBT) <http://www.scala-sbt.org>

SBT and Eclipse <https://confluence.dev.bbc.co.uk/display/linkeddata/Cheat+sheet+for+using+Eclipse+to+develop+Scala+applications+on+the+sandbox>

SBT np plugin (for new projects)—<https://github.com/softprops/np>