LEARNING sbt

#sbt

Table of Contents

About

Chapter 1: Getting started with sbt

Remarks

The Simple Build Tool (SBT for short) can be used to build Scala (or Java) project code. This includes managing code, dependencies, and resources that must be built, tested, and/or compiled to a `.jar` or other artifact. Custom tasks can be created to manage all of these processes.

A note on the name; SBT is sometimes referred to as the 'Scala Build Tool'. While this was not the original intent, it has come to be commonly used as well. SBT may be used to build any project on the JVM.

`.sbt` files, or 'SBT build definitions' are specially interpreted files, written in Scala, that are used by SBT to define a build. `.scala` build definitions may also be written and imported into an `.sbt` file.

Versions prior to `13.6` required that any `.sbt` file has each statement separated by a blank line. Without the blank line, the `.sbt` file will break.

A universal package exists in ZIP and TGZ formats.

Versions

Version	State	Release Date
0.13.12	Stable	2016-07-17

Examples

Install SBT on Linux

Full instructions can be found here.

1. Install the JDK.
2. Set the Java Environment variable.

```
export JAVA_HOME=/usr/local/java/jdk1.8.0_102
echo $JAVA_HOME
/usr/local/java/jdk1.8.0_102
export PATH=$PATH:$JAVA_HOME/bin/
echo $PATH
...:/usr/local/java/jdk1.8.0_102/bin/
```

3. Install Scala.

```
sudo wget http://www.scala-lang.org/files/archive/scala-2.11.8.deb
sudo dpkg -i scala-2.11.8.deb
sudo apt-get update
sudo apt-get install scala
```

4. Install SBT.

```
wget https://bintray.com/artifact/download/sbt/debian/sbt-0.13.9.deb
sudo dpkg -i sbt-0.13.9.deb
sudo apt-get update
sudo apt-get install sbt
```

RPM-based Linux Distributions

- Download SBT repository definitions and add it to YUM:

```
curl https://bintray.com/sbt/rpm/rpm | sudo tee /etc/yum.repos.d/bintray-sbt-rpm.repo
```

- Install SBT according to the definitions previously added to YUM:

```
sudo yum install sbt
```

Install SBT on Windows

Install

MSI installers can be found here. This is the latest stable version. Download and execute to install.

Verify Installation

- Use the `WindowsKey + R`, type `cmd`.
- Alternatively, navigate to the `.sbt` (for example, in `C:\Users\Hopper`) and type `cmd` in the address bar.
- Type `sbt about` to get version information, verifying it is installed. You should see something like this:

```
Java HotSpot(TM) 64-But Server VM warning: ignoring option MaxPermSize=256m; support was
removed in 8.0
[info] Set current project to root--sbt (in build file:/C:/Users/Hopper/.sbt/)
[info] This is sbt 0.13.8
...
```

Install on Mac OSX

Full official instructions can be found here.

MacPorts

Install MacPorts. Then, in the terminal execute:

```
port install sbt
```

Homebrew

Install Homebrew. Then, in the terminal execute:

```
brew install sbt
```

Sources

Download sbt *All platforms* (tgz) installation from SBT.

```
sudo su
cd /opt
mkdir sbt
cd sbt
curl https://dl.bintray.com/sbt/native-packages/sbt/0.13.13/sbt-0.13.13.tgz -o sbt-0.13.13.tgz
```

Then, execute following

```
tar zxf sbt-0.13.13.tgz
ln -s sbt-0.13.13 latest
```

Inside your $HOME make sure to update ~/.profile - by adding following lines

```
export SBT_HOME=/opt/sbt/latest
export PATH=$PATH:$SBT_HOME/bin
```

Verification

In the terminal execute:

```
which sbt
```

You should expect output similar to:

```
/opt/local/bin/sbt
```

If you get no output sbt is not installed.

Import SBT Project into Eclipse

This assumes you have installed both Eclipse and SBT.

- Install the SBT plugin for Eclipse from the Eclipse marketplace.
- In the command line switch directory to the root directory of the project.

```
$ cd ~/home/sample/project
```

- Execute sbt, which will load the project.

```
$ sbt
```

- Compile the project to ensure dependencies are obtainable.

```
> compile
```

- Run the `eclipse` task:

```
> eclipse
```

- Go into Eclipse and select the menu option:

```
File > New > Project From Existing Sources
```

- In the wizard, navigate to your project directory and select it. Eclipse will handle the rest.

Read Getting started with sbt online: https://riptutorial.com/sbt/topic/2351/getting-started-with-sbt

Chapter 2: Build Overview

Remarks

Official documentation is at www.scala-sbt.org.

Examples

Directory Structure

The standard structure for a project built by SBT is:

```
projectName/
    build.sbt
    project/
      <SBT sub-build information>
    src/
      main/
        scala/
           <Scala source files>
        java/
           <Java source files>
        resources/
           <Resource files>
      test/
        scala/
           <Scala test files>
        java/
           <Java test files>
        resources/
           <Resource files>
```

Other directories may exist, but the build deals primarily with these. In the base directory `build.sbt` is placed, whose contents at a minimum are:

- `name := <name of build>`: This is the name of the project.
- `version := <version number>`: This is the version of the project for downstream code to reference.
- `scalaVersion := <version of Scala>`: This is the version of Scala that the project's bytecode is built against.

The `project` directory is where the `meta-build` (as opposed to the `proper-build`) files are placed. This directory can have it's own `build.sbt` file that executes in exactly the same manner, creating an environment for the `proper-build` SBT build to execute. This is recursive, so the `project` directory can have it's own `project` directory where a `meta-meta-build` occurs, and so on.

Upon building, SBT will create a `target` directory in which class files and other components are placed.

Cheat Sheet

This sheet assumes that you are in the root directory of the project, containing the `build.sbt`. `$` indicates a command prompt and `>` indicates commands run inside the SBT console.

Compile a project

```
$ sbt compile
```

Test a project

```
$ sbt test
```

Enter SBT REPL:

```
$ sbt
```

Enter Scala Console with Built Project Available

```
$ sbt
> console
```

Generate Scaladoc

This is an example of executing an SBT 'Task'. The SBT site has more information on generating Scaladoc documentation.

```
$ sbt doc
```

or:

```
$ sbt
> doc
```

Read Build Overview online: https://riptutorial.com/sbt/topic/6761/build-overview

Chapter 3: Dependencies

Examples

Add a Managed Library Dependency

`libraryDependency` is the `SettingKey` that handles 'managed' library dependencies, which are dependencies that are automatically downloaded, matching the supplied versions. To add a single dependency:

```
libraryDependencies += "com.typesafe.slick" %% "slick" % "3.2.0-M1"
```

The first part, `"com.typesafe.slick"`, indicates the library package. The second part, `"slick"`, is the library in question. The final part, `"3.2.0-M1"`, is the version. Because the library is joined by `%%` the version of Scala supplied by the `scalaVersion` setting key will be utilized.

You can add multiple libraries at once using `++=`:

```
libraryDependencies ++= Seq(
  "com.typesafe.slick" %% "slick" % "3.2.0-M1" % "compile",
  "com.typesafe.slick" %% "slick-hikaricp" % "3.2.0-M1",
  "mysql" % "mysql-connector-java" % "latest.release"
)
```

Remember Scala's functional nature, allowing you to compute dependencies. Just remember to return a `Seq`:

```
libraryDependencies ++= {
  lazy val liftVersion = "3.0-RC3" //Version of a library being used
  lazy val liftEdition = liftVersion.substring(0,3) //Compute a value
  Seq(
    "net.liftweb" %% "lift-webkit" % liftVersion % "compile",  // Use var in Seq
    "net.liftmodules" %% ("ng_" + liftEdition) % "0.9.2" % "compile",  // Use computed var in
Seq
  )  // Because this is the last statement, the Seq is returned and appended to
libraryDependencies
}
```

Add a Repository

A repository is a place that SBT looks for `libraryDependencies`. If the build complains about not finding a dependency, it can be lacking the correct repository. Within SBT, the repositories are listed in the `resolvers` SettingKey:

```
resolvers += "Flyway" at "https://flywaydb.org/repo"
```

This follows the syntax of 'Repository name' at 'url location'.

Pin Library to Project Version of Scala

If your project has this:

```
scalaVersion := 2.11  // Replace '2.11' with the version of Scala your project is running on
```

Then you can use `%%` to automatically get the version of the library compiled against the version of Scala the project is using:

```
libraryDependencies += "com.typesafe.slick" %% "slick" % "3.2.0-M1"
```

Note that having the above two lines is equivalent to having this one line:

```
libraryDependencies += "com.typesafe.slick" % "slick_2.11" % "3.2.0-M1"
```

Pin Library to Specific Version of Scala

A library can be 'pinned' to a specific version of Scala using the `%` operator between the `groupId` and the `artifactId` (the first two strings in a library dependency). In this example, we pin the library with the `artifactId` of `slick` to Scala version `2.10`:

```
 libraryDependencies += "com.typesafe.slick" % "slick_2.10" % "3.2.0-M1"
```

Read Dependencies online: https://riptutorial.com/sbt/topic/6760/dependencies

Chapter 4: Getting started with daily development

Examples

Daily continuous development example with Scala

```
# install sbt with homebrew (if you didn't)
brew install sbt

# - create a new scala project
# - name your project name when asked like: hello-world
sbt new sbt/scala-seed.g8

# go to the new project directory that you named
cd hello-world

# run sbt to open the sbt shell
sbt

# run your project in continuous running mode
~ run

# to continuously see the test outputs
# open up a new terminal tab, run sbt, type:
~ test

# to continuously compile
# open up a new terminal tab, run sbt, type:
~ compile
```

~ used for continuous operations in sbt as seen above.

Read Getting started with daily development online: https://riptutorial.com/sbt/topic/9842/getting-started-with-daily-development

Chapter 5: Projects

Examples

Multiple Projects in the same Build (Subprojects)

Sometimes a build combines multiple source directories, each of which is their own 'project'. For instance, you might have a build structure like this:

projectName/ build.sbt project/ src/ main/ ... test/ ... core/ src/ main/ ... test/ ... webapp/ src/ main/ ... test/ ...

In the above project, the code in `projectName/src` is considered the `root` project. There are two other modules, or 'subprojects', `core` and `webapp`.

Configuring a subproject is similar to configuring the root project, except that the subdirectory is specified in the project. This example shows a root project that aggregates a `core` and `webapp` project.

```
lazy val root = (project in file(".")).aggregate(core,webapp).dependsOn(core, webapp)

lazy val core = (project in file("core"))

lazy val webapp = (project in file("webapp")).dependsOn(core)
```

The values passed to `file()` are the directories relative to the project root.

The `webapp` project depends on the `core` project, which is indicated by the `dependsOn` clause, which takes the `core` value specified on the line above. `dependsOn` and `lazy` evaluation ensure that dependencies are available before projects utilize them. In this case, `webapp` depends on `core`, so `core` will be compiled before the build attempts to compile `webapp`.

`aggregate` makes tasks defined in one project available to the project that aggregates it. For instance, executing `compile` in the `root` project will also execute `compile` in `core` and `webapp`.

Configure Macros in a Project

In the `build.sbt` file (or where the project is defined if it is in another location), add the following setting:

```
scalacOptions += "-language:experimental.macros"
```

For instance, a project might be defined like this:

```
lazy val main = project.in(file("."))  // root project
  .settings(scalacOptions += "-language:experimental.macros",
            addCompilerPlugin("org.scalamacros" % "paradise" % "2.1.0" cross
CrossVersion.full))
```

In the above example, the `paradise` plugin is included in order to provide complete support of Scala `2.10.x`.

Display Settings

When in the SBT console, to list all definable settings for a project:

```
settings
```

Or, to get a subproject's (for example, named `webapp`) settings:

```
project webapp
settings
```

The first line above navigates into the specific subproject.

To show the value of a specific setting (for instance, `organization`):

```
show organization
```

This will display the value of that setting.

Read Projects online: https://riptutorial.com/sbt/topic/6790/projects

Chapter 6: Tasks

Examples

Create a Simple Task

All that is needed to define a task is a declaration of it's type and a description:

```
lazy val exampleTask = taskKey[Unit]("An example task that will return no value.")
```

Because `Unit` is the type, this task is composed entirely of side-effects. Once defined, to implement actions:

```
exampleTask := {
  val s: TaskStreams = streams.value
  s.log.info("The example task was executed.")
}
```

If these are defined in `build.sbt`, you can load the project and execute it:

```
> exampleTask
[info] The example task was executed.
```

Read Tasks online: https://riptutorial.com/sbt/topic/7542/tasks

Credits

S. No	Chapters	Contributors
1	Getting started with sbt	Altius, andriosr, Ani Menon, Community, Eugene Yokota, James, karel, kn_pavan, mko, Nathaniel Ford
2	Build Overview	Nathaniel Ford
3	Dependencies	Nathaniel Ford
4	Getting started with daily development	Inanc Gumus
5	Projects	Nathaniel Ford
6	Tasks	Nathaniel Ford