UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA, ELETTRONICA E SISTEMISTICA
DIMES

Introduction to SBT

Antonio Caliò[1]

[1]DIMES Dept., University of Calabria
Rende (CS), IT
a.calio@unical.it

Big Data Analytics - Computer Engineering for the IoT

Outline

Presentation agenda

Why?

Why a Build System?

- As your project grows in complexity, compiling your source code "by hand" will soon become a nightmare! For this reason you need a smart system to build your entire project.

Why SBT?

- It is specifically built for java and scala projects. It represents the build tool of choice for more than 90% of all Scala projects
- It is typesafe and parallel
- The compilation process is incremental
- You can easily declare *watches* over source file, so that they are compiled as soon as SBT detects a change in the code
- It can be extended with a number of community-driven plugins

Presentation agenda

sbt by example

- First you need to download and install SBT following this link
- Create the root directory for your project

  ```
  mkdir proot
  cd proot # move inside the folder
  touch build.sbt # create the build file
  ```

- One way of building the project is by opening an interactive folder inside the project main folder

  ```
  sbt
  sbt:proot> compile
  ```

  The above command will build your project accordingly to what it is specified inside the file `build.sbt`

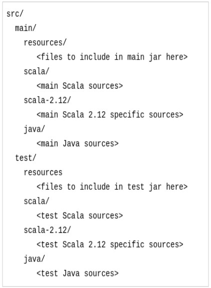
- You can create *watches* over the files composing your project as follows:

  ```
  sbt:proot> ~compile
  ```

  With the above instruction SBT will re-compile the project if anything change on disk

Create a source file

- The tree structure of your SBT project must adhere to the following structure:



```
src/
  main/
    resources/
       <files to include in main jar here>
    scala/
       <main Scala sources>
    scala-2.12/
       <main Scala 2.12 specific sources>
    java/
       <main Java sources>
  test/
    resources
       <files to include in test jar here>
    scala/
       <test Scala sources>
    scala-2.12/
       <test Scala 2.12 specific sources>
    java/
       <test Java sources>
```

Figure: sbt project dir structure

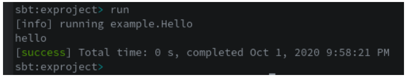
- All the scala source file goes into the src/main/scala subdirectory
- Add the following class to that directory

```
package example

object Hello extends App {
  println("Hello")
}
```

Run your App

- first run `compile` once again
- then issue the command `run` and you should be able to see the output of your program



```
sbt:exproject> run
[info] running example.Hello
hello
[success] Total time: 0 s, completed Oct 1, 2020 9:58:21 PM
sbt:exproject>
```

Figure: sbt run

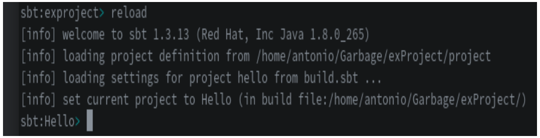
Setting the build.sbt file

- First, you assign a name to your project.
- A good start for the build sbt file would be the following:

```
ThisBuild / scalaVersion := "2.12.7"
ThisBuild / organization := "com.example"

lazy val hello = (project in file("."))
  .settings(
    name := "Hello"
  )
```

- Every time you update the sbt file you should call `reload` on the interactive shell in order the updates to take effect



```
sbt:exproject> reload
[info] welcome to sbt 1.3.13 (Red Hat, Inc Java 1.8.0_265)
[info] loading project definition from /home/antonio/Garbage/exProject/project
[info] loading settings for project hello from build.sbt ...
[info] set current project to Hello (in build file:/home/antonio/Garbage/exProject/)
sbt:Hello>
```

Figure: sbt reload

Enable testing

- Change the build.sbt as follows

```
ThisBuild / scalaVersion := "2.12.7"
ThisBuild / organization := "com.example"

lazy val hello = (project in file("."))
  .settings(
    name := "Hello",
    libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.5" % Test,
 )
```

- then for running the tests:

```
sbt:Hello> reload
sbt:Hello> test
sbt:Hello> ~testQuick
```

Writing a test

- Under the *src* folder create a test folder and save the following file:

```
import org.scalatest._

class HelloSpec extends FunSuite with DiagrammedAssertions {
  test("Hello should start with H") {
    assert("hello".startsWith("H"))
  }
}
```

- Then you can reload the project an run the tests once again

Add a library dependency

- Dependencies are defined in the `build.sbt` file.

```
ThisBuild / scalaVersion := "2.12.7"
ThisBuild / organization := "com.example"

lazy val hello = (project in file("."))
  .settings(
    name := "Hello",
    libraryDependencies += "com.typesafe.play" %% "play-json" % "2.6.9",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.3.1",
    libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.5" % Test,
  )
```

Make a subproject

- You can include subproject inside your main one.

```
ThisBuild / scalaVersion := "2.12.7"
ThisBuild / organization := "com.example"

lazy val hello = (project in file("."))
  .settings(
    name := "Hello",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.3.1",
    libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.5" % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
  )
```

Add ScalaTest to the subproject

- Change the build.sbt as follows:

```
ThisBuild / scalaVersion := "2.12.7"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.0.5"

lazy val hello = (project in file("."))
  .settings(
    name := "Hello",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.3.1",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies += scalaTest % Test,
  )
```

Broadcast Commands

- If you want any command sent to `hello` to be broadcaster to the `hellocore` project you can use the `aggregate` function.

```
ThisBuild / scalaVersion := "2.12.7"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.0.5"

lazy val hello = (project in file("."))
  .aggregate(helloCore)
  .settings(
    name := "Hello",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.3.1",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies += scalaTest % Test,
  )
```

Define dependencies

- if you want to define a dependency for a project you must use the `.dependsOn` function as follows:

```
ThisBuild / scalaVersion := "2.12.7"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.0.5"

lazy val hello = (project in file("."))
  .aggregate(helloCore)
  .dependsOn(helloCore)
  .settings(
    name := "Hello",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies += "com.eed3si9n" %% "gigahorse-okhttp" % "0.3.1",
    libraryDependencies += scalaTest % Test,
  )
```

Add the Play-Json dependency

```
ThisBuild / scalaVersion := "2.12.7"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.0.5"
val gigahorse = "com.eed3si9n" %% "gigahorse-okhttp" % "0.3.1"
val playJson  = "com.typesafe.play" %% "play-json" % "2.6.9"

lazy val hello = (project in file("."))
  .aggregate(helloCore)
  .dependsOn(helloCore)
  .settings(
    name := "Hello",
    libraryDependencies += scalaTest % Test,
  )

lazy val helloCore = (project in file("core"))
  .settings(
    name := "Hello Core",
    libraryDependencies ++= Seq(gigahorse, playJson),
    libraryDependencies += scalaTest % Test,
  )
```

Add another source file

- Add a new source file under the folder *core/src/main/scala/example*, name it `Wheater.scala`

```
package example.core

import gigahorse._, support.okhttp.Gigahorse
import scala.concurrent._, duration._
import play.api.libs.json._

object Weather {
  lazy val http = Gigahorse.http(Gigahorse.config)

  def weather: Future[String] = {
    val baseUrl = "https://www.metaweather.com/api/location"
    val locUrl = baseUrl + "/search/"
    val weatherUrl = baseUrl + "/%s/"
    val rLoc = Gigahorse.url(locUrl).get.
      addQueryString("query" -> "New York")
    import ExecutionContext.Implicits.global
    for {
      loc <- http.run(rLoc, parse)
      woeid = (loc \ 0  \ "woeid").get
      rWeather = Gigahorse.url(weatherUrl format woeid).get
      weather <- http.run(rWeather, parse)
    } yield (weather \\ "weather_state_name")(0).as[String].toLowerCase
  }

  private def parse = Gigahorse.asString andThen Json.parse
}
```

Update the main class

- Create the file `Hello.scala` under `src/main/scala/example/Hello.scala`, then run the app.

```
package example

import scala.concurrent._, duration._
import core.Weather

object Hello extends App {
  val w = Await.result(Weather.weather, 10.seconds)
  println(s"Hello! The weather in New York is $w.")
  Weather.http.close()
}
```

Add a plugin

- To add a plugin you must add a file `plugins.sbt` under the `project` folder. Here we add the `sbt-native-packager` plugin. It is very useful as it

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.3.4")
```

Enable a plugin

- Change the `build.sbt` file as follows

```
ThisBuild / scalaVersion := "2.12.7"
ThisBuild / organization := "com.example"

val scalaTest = "org.scalatest" %% "scalatest" % "3.0.5"
val gigahorse = "com.eed3si9n" %% "gigahorse-okhttp" % "0.3.1"
val playJson  = "com.typesafe.play" %% "play-json" % "2.6.9"

lazy val hello = (project in file("."))
  .aggregate(helloCore)
  .dependsOn(helloCore)
  .enablePlugins(JavaAppPackaging)
  .settings(
    name := "Hello",
    libraryDependencies += scalaTest % Test,
  )
```

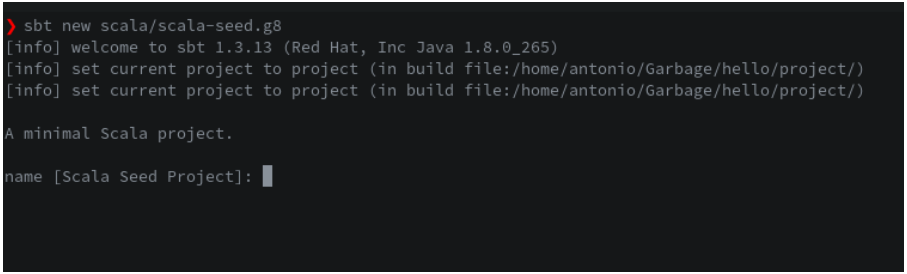
Distribute your project

- once you enabled the packager plugin, you can create:
  1. a .zip distribution. You just need to run: dist inside the console
  2. a docker image. You just need to tun Docker/publishLocal inside the console

Presentation agenda

Create a new project

- You can create a new project with the `sbt new` command (you need at least the version 0.13.13 of sbt).



```
❯ sbt new scala/scala-seed.g8
[info] welcome to sbt 1.3.13 (Red Hat, Inc Java 1.8.0_265)
[info] set current project to project (in build file:/home/antonio/Garbage/hello/project/)
[info] set current project to project (in build file:/home/antonio/Garbage/hello/project/)

A minimal Scala project.

name [Scala Seed Project]: 
```

Figure: Sbt-new

- `socala/scala-seed.g8` is a template that will initialize the directory structure of you project. There are several templates, but this is always a good starting point for most of the projects

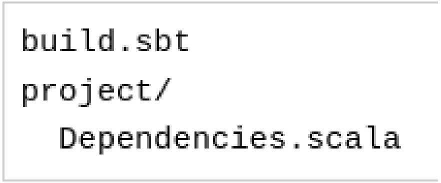
Understanding the directory structure

- In sbt's terminology, the base directory is where the build.sbt is located. This can be regarded as root folder of your project
- Source Code directory. The source code structure resembles the one of Maven. Every path is relative to the source base directory

```
src/
  main/
    resources/
       <files to include in main jar here>
    scala/
       <main Scala sources>
    scala-2.12/
       <main Scala 2.12 specific sources>
    java/
       <main Java sources>
  test/
    resources
       <files to include in test jar here>
    scala/
       <test Scala sources>
    scala-2.12/
       <test Scala 2.12 specific sources>
    java/
       <test Java sources>
```

Build definition files

- The main file is the `build.sbt`
- There are other support files located in other sub-directories of the base directory. For instance we can have the `Dependencies.scala` under the project subdirectory.
- Under the project directory you can also have a `plugins.sbt` file where you define the plugin involved in the process of building your project



```
build.sbt
project/
  Dependencies.scala
```

Figure: Dependencies file

Presentation agenda

What is a build definition?

A build definition is defined in the `build.sbt` file and it consists of a collection of subprojects. A subproject is declared as follows:

```
lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    scalaVersion := "2.12.7"
  )
```

A subproject is represented by a series of key/value pairs, listed under the `.settings()` method.

How to defines settings

- a key-value pair is called *setting expression*. A *setting expression* has the following structure:

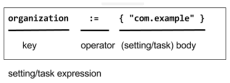


Figure: Setting Expression

- There are three parts:
  1. Left-hand side: key
  2. Operator
  3. Right-hand side: body

Presentation agenda

Adding library dependencies

- To depend on third-party libraries there are two options.
  1. Drop the jars into `lib/` folder – so you would have an *unmanaged* dependency
  2. Express the dependency in the `build.sbt` file – so you would have a *managed* dependency

```
val derby = "org.apache.derby" % "derby" % "10.4.1.3"

ThisBuild / organization := "com.example"
ThisBuild / scalaVersion := "2.12.10"
ThisBuild / version      := "0.1.0-SNAPSHOT"

lazy val root = (project in file("."))
  .settings(
    name := "Hello",
    libraryDependencies += derby
  )
```

Types of dependency

There are two type of dependencies:

- *unmanaged* - if you just drop a jar file inside the `lib` folder
- *managed* - if you download the dependency from a respository

Unmmanaged dependency

- Unmanaged dependencies work like this: add jars to lib and they will be placed on the project classpath
- Dependencies in lib go on all the classpaths (for compile, test, run, and console). If you wanted to change the classpath for just one of those, you would adjust Compile / dependencyClasspath or Runtime / dependencyClasspath for example.
- You do not need to change anything in the `build.sbt` file in order to use unmanaged dependencies, unless you want to override some configuration, for instance changing the base directory for the unmanaged dependencies:

  ```
  unmanagedBased := baseDirectory.value / "custom_lib_direcotry"
  ```

Managed dependency

- You define a managed dependency via the `libraryDependencies` key
- A new dependency looks like this:

```
libraryDependencies += groupId % artifactId % revision [% configurataion]
```

- You can also define a sequence of dependencies and add them with the ++= operator, like this:

```
libraryDependencies ++= Seq(
  groupID % artifactID % revision,
  groupID % otherID % otherRevision
)
```

Getting the right Scala version with %%

- When you use the definition: "organization %% moduleName % version" as opposed to "organization % moduleName % version", sbt will add the Scala version to the artifact name.
- Therefore the following definition:

```
libraryDependencies += "org.scala-tools" % "scala-stm_2.11" % "0.3"
```

is equivalent to:

```
libraryDependencies += "org.scala-tools" %% "scala-stm" % "0.3"
```

- The %% operator is very useful as many Scala libraries are compiled for multiple Scala versions. In this way you can select the version that better fits your project

Resolvers

- Not all the packages live on the same server. sbt uses Maven2 repository by default. But if your dependency is on another repository you need to add a *resolver* inside your build.sbt file, so that Ivy can find the dependency.
- To add a resolver, here is the syntax:

```
// resolvers += name at location
resolvers += "Sonatype OSS Snapshots" at "https://oss.sonatype.org/content/repositories/snapshots"
// if you have a local maven repository
resolvers += "Local Maven Repository" at "file://"+Path.userHome.absolutePath+"/.m2/repository"
```