

|  |
| --- |
|  |

**SBT (Scala Build Tool)**

**Guidelines and Best Practices**

# Table of Contents

[Table of Contents 2](#_Toc425435424)

[1. Introduction 3](#_Toc425435425)

[2. Installation 3](#_Toc425435426)

[3. Configuration Files 3](#_Toc425435427)

[4. CoMMON COMMANDS 4](#_Toc425435428)

[5. case study : hellosBT 5](#_Toc425435429)

[6. more features 11](#_Toc425435430)

[7. INTEGRATION WITH OTHER TOOLS/TECHNOLOGIES 13](#_Toc425435431)

[8. Multiproject build 19](#_Toc425435432)

# 

# Introduction

SBT stands for 'Scala Build Tool' is an open source tool for building pure Scala and/or Scala and Java mix projects, similar to Java's Maven or Ant.

Its main features are:

* Native support for compiling Scala code and integrating with Scala test/code-coverage/static-code analysis etc. frameworks
* Dependency management (SBT internally using Apache Ivy for dependency management)
* Continuous compilation, testing and deployment
* Integration with the Scala interpreter for rapid iteration and debugging
* Support for mixed Java/Scala projects

When continuous compilation mode is entered, the Scala compiler is instantiated only once which eliminates subsequent startup costs, and source file changes are tracked so that only affected dependencies are recompiled.

The interactive console allows modifying build settings on the fly and entering the Scala REPL along with all class files of the project.

# Installation

Please refer SBT official site (<http://www.scala-sbt.org/0.13/tutorial/Setup.html>) for details about how to install SBT on Windows/Mac/Linux etc. OS.

# Configuration Files

Before we jump into bulding process, We need to understand key configuration files which are the heart of the SBT build process. This section lists these key files with sample configuration.

Key configuration files are :

* build.sbt
* plugins.sbt
* user.properties

**build.sbt** : In SBT, project-specific properties, such as library dependencies, Scala version and so on, which are required for a successful build are declared in the build definition. The .sbt file should be located in the base directory and is generally named build.sbt. Like pom.xml for Maven, build.sbt is the key configuration file for SBT.

Each key-value pair is a build property. So, we could say that a build definition is a list of properties.



**Plugins.sbt** : plugins.sbt : A plugin extends the build definition, most commonly by adding new settings. The new settings could be new tasks. For example, a plugin could add a codeCoverage task which would generate a test coverage report. There are various plugins are configured in following plugins.sbt to enable the integration with different test, test, database migration, automation testing etc. frameworks.



**build.propeties** : You can force a particular version of sbt by creating a file build.properties.



# CoMMON COMMANDS

These are the list of basic commands you’ll come across often in order to create, compile and package your SBT build.

|  |  |
| --- | --- |
| COMMANDS | DESCRIPTION |
| clean | Deletes all generated files (in the target directory). |
| compile | Compiles the main sources (in src/main/scala andsrc/main/java directories). |
| test | Compiles and runs all tests. |
| run | 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. |
| publish | It will publish your generated artifacts to configured repository e.g. Ivy or Maven. |

# case study : hellosBT

This section guides you how to create SBT build of simple Scala and Java mix project. We’ll also see the use of various commands described in above section e.g. clean, compile, test, run, package, publish etc.

**HelloSBT Project** : As part of understanding SBT build process, We’ll create a simple HelloSBT project from scratch that will contain simple scala and java source code, simple scala/java test cases and other SBT related configuration file. We’ll build this project using SBT and execute various commands to clean, compile and test our build.

**Prerequisites :**

* It is assumed that you’ve installed Scala, Please refer (<http://www.scala-lang.org/download/>) to download and install latest version of Scala.
* Please make sure you have SCALA\_HOME correctly configured in user variable section of System’s Environment Variable entries.
* We assume you have installed SBT on your windows system. Please refer ( <http://www.scala-sbt.org/0.13/tutorial/Setup.html> ) for more details about how to install SBT on Windows.
* Please make sure you’ve SBT\_HOME is created automatically inside ‘system variable’ of your system’s Environment Variables after installation of SBT.
* Upon successful installation of SBT, You should have two directories i.e. ‘.sbt’ and ‘.Ivy’ created automatically under C:\Users\<USER\_HOME>. These are the directories where SBT will download and store all the dependencies you may have mentioned in your build.sbt file.
* You should have Scala Eclipse IDE for writing Scala and Java sourcecode.

**Assumptions :**

This case study assumes that your system has following version of SBT and Scala installed.

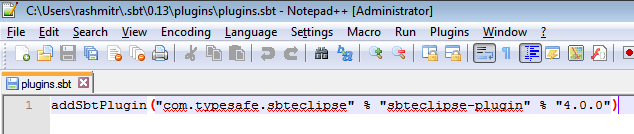
* Scala (version 2.11.4)
* SBT (version 0.13.8)

Let’s start preparing the required configuration and project structure for HelloSBT project.

1. **Eclipse plugin :** Let’s begin our case study by adding Eclipse plugin for sbt. This plugin is used for converting our sbt project into eclipse project so we can import and work with our project using Eclipse IDE.

Create plugins.sbt under C:\Users\<USER\_HOME>\.sbt\0.13\plugins directory. Copy and paste following content into plugins.sbt

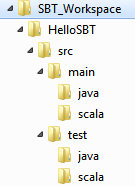
*addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")*



The plugins.sbt which we created above is called a global plugins file because the plugins registered in this file is available to all the project. You can name it as global.sbt as well. Making Eclipse IDE compatible project might be the requirement for all our SBT project and thus we have decided to register this plugins in global plugins file.

You however can also create local plugins file with the same name i.e. plugins.sbt (you can name it as local.sbt as well) and save it under the /project folder. Plugins registered in this local plugins file is available exclusively to that project only and no other project has access to those plugins. You’ll get more idea when we create local plugins file for our HelloSBT project.

1. **Project structure creation :** 
   1. Create following project structure in your C:\



* 1. Create a file build.sbt in C:\SBT\_Workspace\HelloSBT. Copy and paste following content into it.

*//Organization*

*organization := "com.cybage.hellosbt"*

*//Version of your project*

*version := "0.1.0"*

*//Scala version*

*scalaVersion := "2.11.6"*

*//Name of Project*

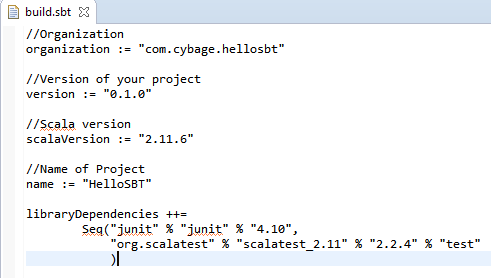
*name := "HelloSBT"*

*libraryDependencies ++=*

*Seq("junit" % "junit" % "4.10",*

*"org.scalatest" % "scalatest\_2.11" % "2.2.4" % "test"*

*)*



* 1. Create plubins.sbt inside project folder (i.e. C:\SBT\_Workspace\HelloSBT\project\) with below content.

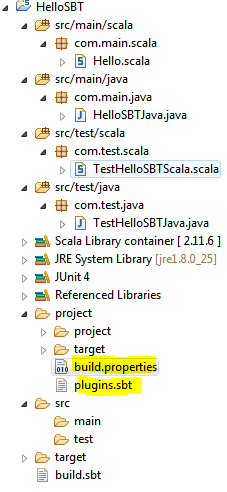
*//Group, ArtifactId, Version*

*addSbtPlugin("org.scalastyle" %% "scalastyle-sbt-plugin" % "0.7.0")*

* 1. Create plugins.sbt inside project folder (i.e. C:\SBT\_Workspace\HelloSBT\project\) with below content.

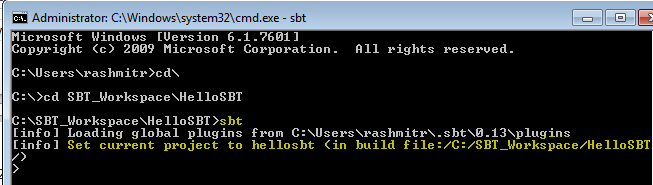
*sbt.version=0.13.8*

Refer below screen for understanding.

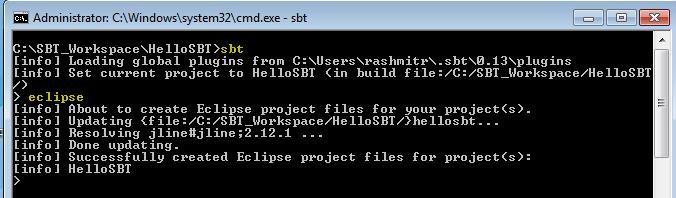


* 1. Open command prompt, Change to HelloSBT directory and type command ‘sbt’ as shown below. Execution of this ‘sbt’ command will make HelloSBT as SBT project and gives you a shell prompt (i.e. > ). This shell prompt is called SBT Interactive shell because SBT allows you to type & execute various basic build commands (e.g. clean, compile, test etc) and interactively shows the result of each command. Henceforth we’ll refer this prompt as interactive shell for rest of the documentation.

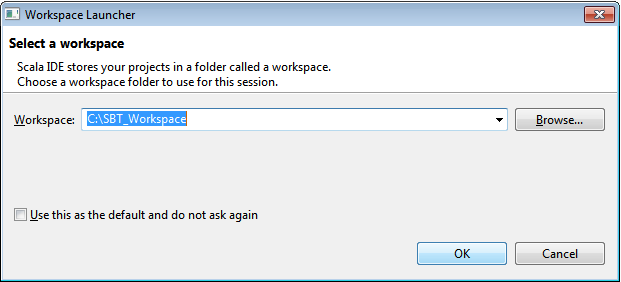
By making sbt project allows you to start interactive command execution, incremental compilation and continuous compilation etc features for your project which we’ll be looking into subsequent sections.

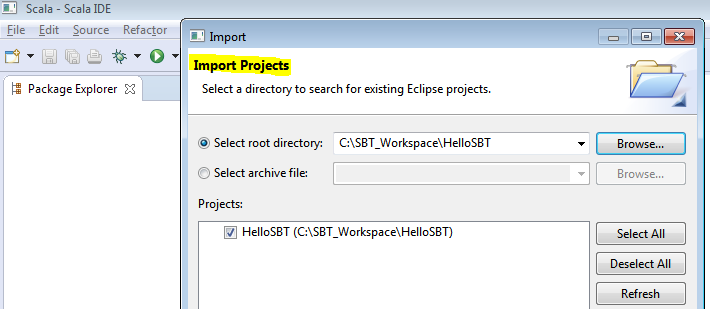


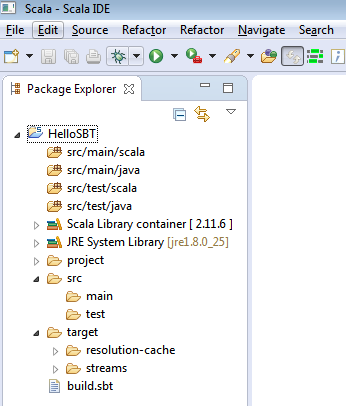
* 1. Type *eclipse* on command prompt as show in image. This will make your SBT project compatible to be imported in Eclipse IDE. Hope you remember that the *eclipse* command is not SBT’s default command but it comes through the plugins which we registered in our global plugins file.



* 1. Now, Import the HelloSBT project into Scala Eclipse IDE as shown below.





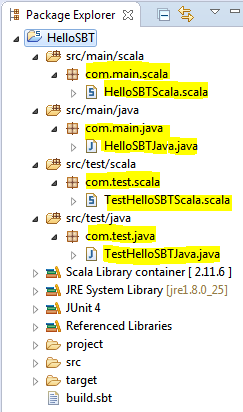


At the time of writing this document, Unlike Maven/Ant, there is no in-built sbt plugin available in Eclipse. However sbt allowed us to build Eclipse project structure which we’ve just built and imported here. Scala developer can now do the development of source code using Scala Eclipse IDE.

Let’s move our focus on writing some sample java and scala programs as well as java and scala test cases which we’ll be using for building our sbt project.

**Package Creation :**

Using Eclipse, create the packages e.g. com.main.scala, com.main.java, com.test.scala, com.test.scala under their respective src and test folder. Please refer following screen to understand the location of each package(highlighted). At the end of package creation, Your HelloSBT project structure should look exactly the same as in below screen.



**Source Code (Scala, Java and their test cases) :**

Now let’s create following list of scala and copy/paste java source codes and test cases (highlighted) and put it in the respective folder as visible in above screen.

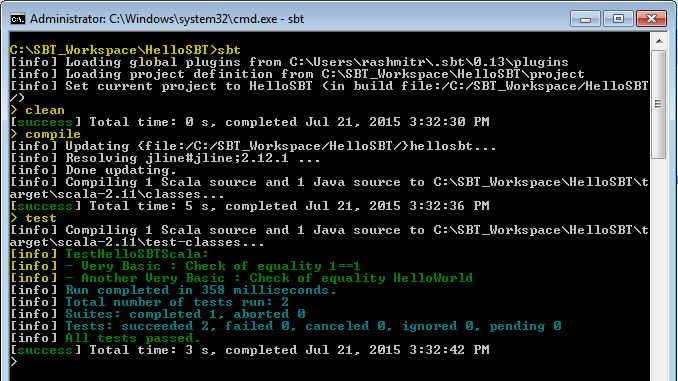
|  |  |  |
| --- | --- | --- |
| **ClassType** | **ClassName** | **SourceCode** |
| Scala main class | HelloSBTScala.scala  Location:  C:\SBT\_Workspace\HelloSBT\src\main\scala\com\main\scala | **package** com.main.scala  **class** HelloSBTScala(**val** name:*String*){  **def** hello() : *String* = "Hello "+name  }  **object** HelloSBTScala {  **def** main(args: Array[*String*]): Unit = {  **val** obj = **new** HelloSBTScala("World")  println(obj.hello())  }  } |
| Java main class | HelloSBTJava.java  Location : C:\SBT\_Workspace\HelloSBT\src\main\java\com\main\java | **package** com.main.java;  **public** **class** HelloSBTJava {  String name;  **public** HelloSBTJava() {  }  **public** HelloSBTJava(String name){  **this**.name =name;  }  **public** String hello(){  **return** "Hello "+name;  }  **public** **static** **void** main(String args[]){  HelloSBTJava obj = **new** HelloSBTJava("Cybage");  String msg = obj.hello();  System.***out***.println(msg);  }  } |
| Scala test case | TestHelloSBTScala.scala  Location : C:\SBT\_Workspace\HelloSBT\src\test\scala\com\test\scala | **package** scalatest  **import** org.scalatest.FunSuite  **class** TestHelloSBTScala **extends** FunSuite {  test("Very Basic : Check of equality 1==1") {  assert(1 == 1)  }  test("Another Very Basic : Check of equality HelloWorld") {  assert("Hello World" == "Hello World")  }  } |
| Java test case | TestHelloSBTJava.java  Location: C:\SBT\_Workspace\HelloSBT\src\test\java\com\test\java | **package** com.test.java;  **import** **static** org.junit.Assert.\*;  **import** org.junit.Test;  **import** com.main.java.HelloSBTJava;  **public** **class** TestHelloSBTJava {  @Test  **public** **void** testHello() {  HelloSBTJava obj = **new** HelloSBTJava("Cybage");  *assertEquals*("Hello Cybage", obj.hello());  }  } |

**Building the Project**

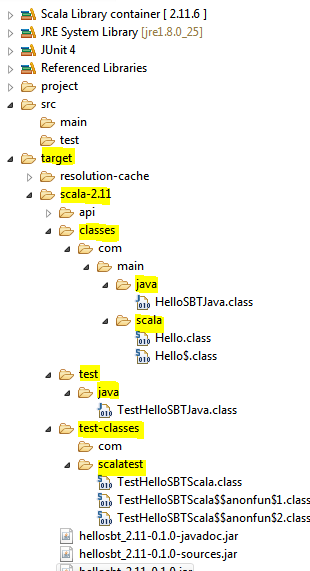
We are done with all the required configuration, source code and test cases creation. It’s time to build our project and test it.

* Open command prompt
* Change to directory C:\SBT\_Workspace\HelloSBT
* Type *sbt,* It’ll open interactive shell prompt which will allow you to execute various build commands.
* Clean the project by executing *clean* command on interactive shell. (i.e. *>clean*)
* Compile the project by executing *compile* command (i.e. *>compile*)
* Run the test-cases by executing *test* command (i.e. *>test*)

Refer below screen where we’ve executed the same commands as mentioned above.



Please pay attention to compile and test commands. The *compile* command has executed all your Java and Scala source code and generated respective class files in the target folder in your project structure. Similarly, *test* command has executed all the scala and java test cases and their respective class files available inside target folder. Revisit your Eclipse IDE project structure and verify the generated class files under the target folder. Refer below screen.



**Packaging and Publishing the artifact**

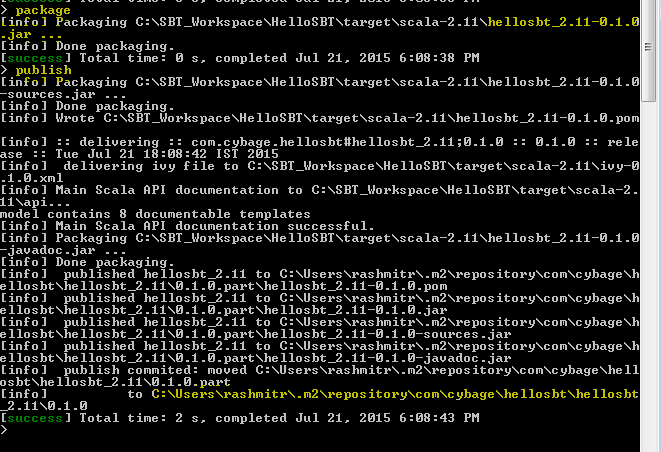
If require, You can publish project artifact into local maven repository by adding following entry to build.sbt:

*publishTo := Some(Resolver.file("file", new File(Path.userHome.absolutePath+"/.m2/repository")))*

After updating the build.sbt, Open command prompt and start interactive shell by typing *sbt* command on base project’s directory. Execute *clean, compile, test* commands in order on interactive shell to make sure the build is error free and ready to package and publish.

Type *package* to package the project into jar file. Check the name of artifact (i.e. jar file) created on the successful execution of command.

Type *publish* to publish the generated artifact to your local maven repository. It’ll read the path of local maven repository you mentioned in build.sbt (i.e. publishTo key/value pair)



This is the end of our HelloSBT case study. Hope you should have fair idea about how to build a project using SBT and understanding of various basic commands e.g. sbt, clean, compile, test, package, publish etc. You should have now fair amount of understanding of various configuration files e.g. build.sbt, global/local plugins.sbt file, user.properties and interactive sbt shell.

So far so good. We have now enough understanding of SBT to explore more features of it.

We’ll now focus on integrating various test, code coverage, code analysis, test automation, Database migration etc. open source tools with SBT.

# more features

There are different mode of running sbt build commands and it allows to revisit the commands you’ve already executed on your interactive shell. This section describes how to use sbt once you have set up your project and how to get list of recently executed commands on interactive shell. Here we assumes you’ve installed sbt and completed HelloSBT mentioned in above section.

**Interactive Mode :**

Running sbt with no command line arguments starts it in interactive mode.

To start interactive Mode, Run *sbt* command in your project directory with no arguments:

C:\SBT\_Workspace\HelloSBT>sbt

**> (**It’s an interactive shell**)**

You could type compile the scala/java sourcecode at the sbt prompt using compile command:

*> compile*

To compile again, press up arrow and then enter.

To run your program, type *run*.

To leave interactive mode, type exit or use *Ctrl+Z (Windows)*.

Interactive mode has a command prompt where you can type various build commands and it’ll get executed interactively. We’ve already worked with interactive mode in our HelloSBT case study.

**Batch Mode**

You can also run sbt in batch mode, specifying a space-separated list of sbt commands as arguments. For sbt commands that take arguments, pass the command and arguments as one argument to sbt by enclosing them in quotes.

For example,

*C:\SBT\_Workspace\HelloSBT>sbt clean compile test*

**Triggered Mode**

To speed up your edit-compile-test cycle, you can ask sbt to automatically recompile or run tests whenever you save a source file. That is, if opted by the user, specific tasks will be run whenever the user makes a change in any of the source code. This was originally meant for continuous compilation, but has now been extended for various other tasks. It is also possible to run multiple commands

in this mode.

For example, If you want to compile one or more source files as and when any change made to it, you can type it as below by prefixing the compile command with ~.

*> ~ compile*

Press enter to stop watching for changes. You can use the ~ prefix with either interactive mode or batch mode.

**Tab completion**

Interactive mode has tab completion, including at an empty prompt. A special sbt convention is that pressing tab once may show only a subset of most likely completions, while pressing it more times shows more verbose choices.

**History command**

Interactive mode remembers history, even if you exit sbt and restart it. The simplest way to access history is with the up arrow key. The following commands are also supported:

|  |  |
| --- | --- |
| Command | Description |
| ! | Show history command help. |
| !! | Execute the previous command again. |
| !: | Show all previous commands. |
| !:n | Show the last n commands. |
| !n | Execute the command with index n, as shown by the !: command. |
| !-n | Execute the nth command before this one. |
| !string | Execute the most recent command starting with ‘string.’ |
| !?string | Execute the most recent command containing ‘string.’ |

# INTEGRATION WITH OTHER TOOLS/TECHNOLOGIES

This section covers integration of SBT with various open source tools/technologies which are useful to improve the code quality, automation testing and database migration which often require during software development lifecycle.

We have categorized the list of tools/technologies based on their respective usage :

* Testing
  1. Junit
  2. ScalaTest
* Continous Integration
  1. Jenkins
* Code Coverage

1. Jacoco
2. Scoverage

* Static code analysis
  1. Scalastyle
* Automation testing
  1. Gatling
  2. Selenium
* Database Migration
  1. Flyway
* Sonarqube and Sonar-runner

**Testing**

Testing gives you an insight into working of an application. It helps in understanding the extent to which the application does its job and the cases where it fails to do so. SBT provides support for running tests in continuous mode so that you can see the impact the moment a change is made.

All the major Scala testing framework can be used within SBT project just by adding them as a project dependency for the test configuration.

* **JUnit :** JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks.

Integrating JUnit in SBT project will help us to compile and run JUnit test cases written in Java.

To integrate JUnit with SBT, Append following configuration in *build.sbt.*

|  |  |
| --- | --- |
| build.sbt | *libraryDependencies += "junit" % "junit" % "4.10" % “test”* |

* **ScalaTest :** ScalaTest primarily written in Scala language which can be used to test either Scala or Java code.

To integrate ScalaTest with SBT, Append following configuration in *build.sbt.*

|  |  |
| --- | --- |
| build.sbt | *libraryDependencies += "org.scalatest" % "scalatest\_2.11" % "2.2.4" % "test"* |

**Continuous Integration**

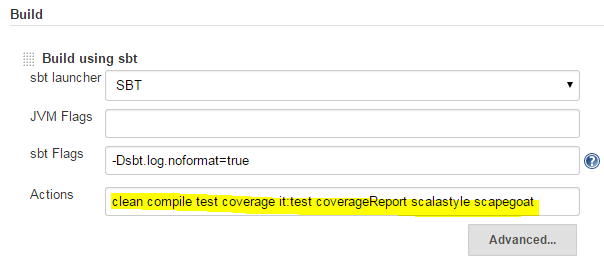
* **Jenkins :** Jenkins is an open-source continuous integration server with 300+ plugins to support all kinds of software development.

You can configure Jenkins to build your SBT project. Please refer <https://wiki.jenkins-ci.org/display/JENKINS/sbt+plugin> to configure SBT plugin into Jenkins.

Download and install sbt-plugin using Manage Jenkins -> Manage Plugins link in Jenkins dashboard.

Using Manage Jenkins -> Configure System link you can link your SBT installation by providing path of sbt-launch jar.

You can configure the SBT which you just configured in above steps for the project which you would like to get managed by Jenkins for CI. You can supply sbt build command here (highlighted in below image).



**Code coverage**

* **Jacoco :** JaCoCo is a free code coverage library for Java

To integrate Jacoco with SBT, Below are the configuration required in respective configuration file.

|  |  |
| --- | --- |
| *build.sbt* | *jacoco.settings* |
| *plugins.sbt* | *addSbtPlugin("de.johoop" % "jacoco4sbt" % "2.1.6")* |
| *Command to execute on Interactive shell* | *sbt jacoco:cover* |

* **Scoverage :** Scoverage is a free Apache licensed code coverage tool for scala.

To integrate Scoverage with SBT, Below are the configuration required in plugins.sbt.

|  |  |
| --- | --- |
| *plugins.sbt* | *addSbtPlugin("org.scoverage" % "sbt-scoverage" % "1.1.0")* |

**Usage :**

You can use following command to enable the code coverage :

*>sbt clean coverage test*

You can also enable coverage directly in your build by typing following in build.sbt

*coverageEnabled := true*

After the tests have finished you should then run

*>sbt coverageReport*

to generate the reports. You will find the coverage reports inside target/scoverage-report. There are HTML and XML reports. The XML is useful if you need to programmatically use the results, or if you're writing a tool.

If you're running the coverage reports from within an sbt console session (as opposed to one command per sbt launch), then the coverage command is sticky.

To turn it back off when you're done running reports, use the *coverageOff* command on interactive shell or reset coverageEnabled with set *coverageEnabled := false* in build.sbt

**Static code analysis**

* **ScalaStyle :** Scalastyle examines the Scala code and indicates potential problems with it. It is similar to what Checkstyle does for Java except Scalastyle is for Scala language.

To integrate Scalastyle with SBT, Below are the configuration required in plugins.sbt.

|  |  |
| --- | --- |
| *plugins.sbt* | *addSbtPlugin("org.scalastyle" %% "scalastyle-sbt-plugin" % "0.7.0")*  *resolvers += "sonatype-releases" at "https://oss.sonatype.org/content/repositories/releases/"* |

**Usage :**

You will need a configuration file. The easiest way to get one is to use the scalastyleGenerateConfig command:

*sbt scalastyleGenerateConfig*

This will create a scalastyle-config.xml in the current directory, with the default settings. Then, you can check your code with the scalastyle command

*sbt scalastyle*

This produces a list of errors on the console, as well as an XML result file target/scalastyle-result.xml.

Please visit <http://www.scalastyle.org/sbt.html> for more information.

**Automation Testing**

* **Gatling :** Gatling is an open-source load testing framework based on Scala, Akka and Netty.

The software is designed to be used as a load testing tool for analyzing and measuring the performance of a variety of services, with a focus on web applications.

To integrate Gatling with SBT, Below are the configuration required in build.sbt and plugins.sbt.

|  |  |
| --- | --- |
| *build.sbt* | *libraryDependencies +="io.gatling.highcharts" % "gatling-charts-highcharts" % "2.1.6" % "test"*  *libraryDependencies +="io.gatling" % "gatling-test-framework" % "2.1.6" % "test"*  *enablePlugins(GatlingPlugin)* |
| *plugins.sbt* | *addSbtPlugin("io.gatling" % "gatling-sbt" % "2.1.5")* |

Please visit <http://gatling.io/docs/2.1.6/extensions/sbt_plugin.html> for more information.

* **Salenium :** Selenium is a portable software testing framework for web applications. Selenium provides a record/playback tool for authoring tests without learning a test scripting language.

To integrate Selenium with SBT, Below are the configuration required in plugins.sbt.

|  |  |
| --- | --- |
| *build.sbt* | *libraryDependencies += "org.seleniumhq.selenium" % "selenium-java" % "2.35.0" % "test"* |

Please visit <http://scalatest.org/user_guide/using_selenium> for more information

**Database Migration**

* **Flyway :** Flyway is an open source database migration tool. It strongly favors simplicity and convention over configuration.It is based around 6 basic commands: Migrate, Clean, Info, Validate, Baseline and Repair. Migrations can be written in SQL (database-specific syntax (such as PL/SQL, T-SQL, ...) is supported) or Java.

|  |  |
| --- | --- |
| *build.sbt* | *seq(flywaySettings:\_\*)*  *libraryDependencies += "mysql" % "mysql-connector-java" % "5.1.22"*  *flywayUrl := "jdbc:mysql://[hostname]:[port]/[schema]"*  *flywayUser := "[db\_username]"*  *flywayPassword := "[db\_password]"* |
| *plugins.sbt* | *addSbtPlugin("org.flywaydb" % "flyway-sbt" % "3.2.1")*  *resolvers += "Flyway" at "http://flywaydb.org/repo"* |

Please visit <http://flywaydb.org/getstarted/firststeps/sbt.html> for more information.

**Sonarqube and sonar-runner**

* SonarQube (formerly known as Sonar) is an open source platform for continuous inspection of code quality. Sonarqube supports more than 20+ languages, However as on writing this document there is no direct support for Scala language.

Sonarqube supports Scoverage report through Scoverage plug-in. Before we begin integration, We are expecting that your system should have following list of softwares installed in your system.

* You should have Sonarqube server installed in your system.
* You should have sonar-runner installed in your system
* You should have enabled SBT-Scoverage integration for your project. If not, Please refer prior section that details the SBT-Scoverage integration.

Please follow below steps in order to achieve SBT-Sonarqube integration.

1. **Scoverage plug-in Installation** :

This plug-in makes sonar reading the statement coverage report (xml) generated by sbt scoverage.

* Download the sonar-scoverage-plugin-5.1.1.jar from <https://github.com/RadoBuransky/sonar-scoverage-plugin/releases> and copy to C:\<sonarqube\_home>\extensions\plugins directory.

1. **Sonar-runner configuration :** The Sonar-runner is recommended as the default launcher to analyze a project with SonarQube.

* Open *sonar-runner.properties* file available under C:\<sonar-runner\_HOME>\conf , Provide correct values for following list of keys :

*sonar.host.url = <url of sonarqube server>*

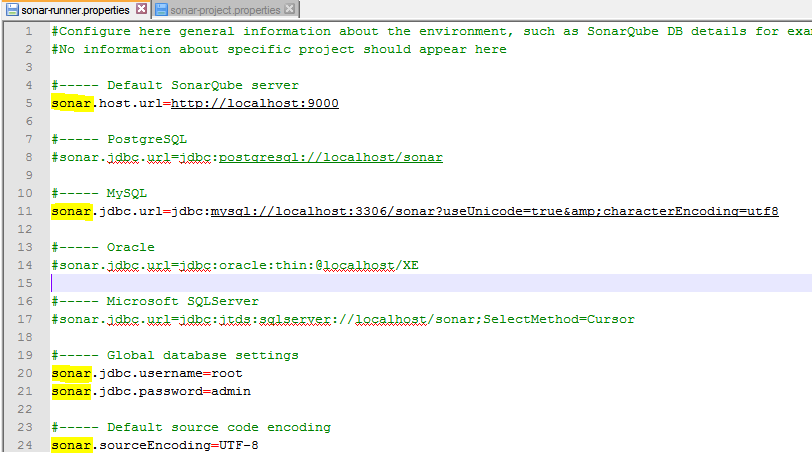
*sonar.jdbc.url =<db\_url>*

*sonar.jdbc.username=<username>*

*sonar.jdbc.password=<password>*

*sonar.sourceEncoding=UTF-8*

Here is the sample file, you should change value that suits your system detail.



1. **sonar-project.properties**

* Create *sonar-project.properties* file and save it under the root directory of your SBT project.

*sonar.projectKey=<project\_key>*

*sonar.projectName=<project\_name>*

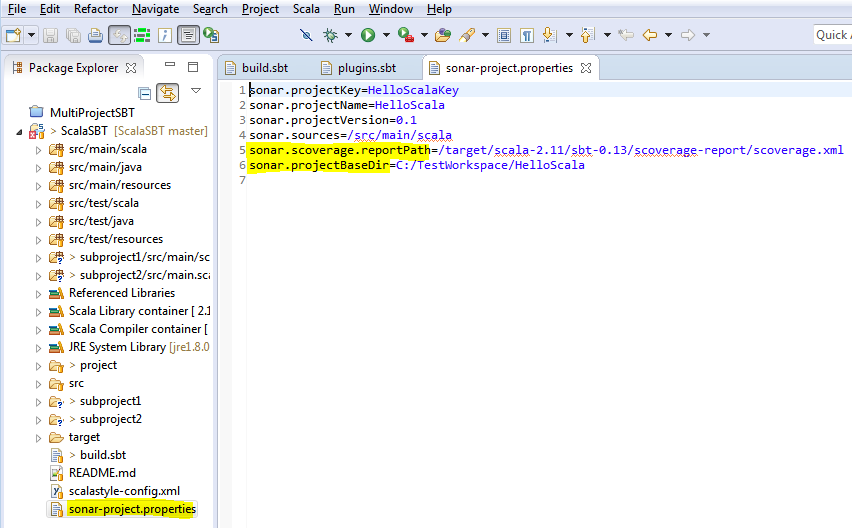
*sonar.projectVersion=<project\_version>*

*sonar.sources=/src/main/scala*

*sonar.scoverage.reportPath=<path of scoverage.xml in project>*

*sonar.projectBaseDir=<path of your project>*

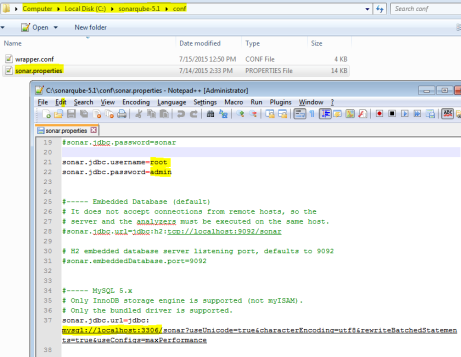
Here is the sample file for your understanding.



1. **sonar.properties**

* Open *sonar.properties* file available under your C:\<Sonarqube\_HOME>\conf. Make sure you have configured Database properties correctly for your sonarqube server.

Here is the sample file for your understanding.



We are done with all the required configuration, Its time to generate scoverage report through SBT and get it published on Sonarqube runner. Follow below instruction to get it done.

* Run *>sbt coverage test* on SBT interactive shell to generate scoverage XML report.
* Go to the page http://localhost:9000/ sonar dashboard and log in to sonar server with default username “sonar” and password : “sonar”.
* Click on “Quality Profiles” on the dashboard and create a quality Profile for Scala
* Run sonar-runner to launch the report on SonarQube web server.Configura

# Multiproject build

A build definition that consists of multiple project configurations is termed as a multiproject build. These are extremely useful when your project is a combination of two or more modules. If they depend on one another, you could also specify the dependencies so that whenever a change is made to one project, it is reflected in the projects that depend on it.

Let’s understand how to configure and work with multiproject build.

Create following project structure in your Eclipse:



At the end of project structure creation you’ll have following list of directory and package structure in your IDE :

MultiProjectSBT – Name of Base Project

subproject1 – Name of sub-module 1

subproject2 – Name of sub-module 2

**Source packages :**

src/main/scala (under MultiProjectSBT)

subproject1/src/main/scala (under subproject1)

subproject2/src/main/scala (under subproject2)

**Packages :**

com.main.test (under MultiProjectSBT)

com.sp1.test (under subproject1)

com.sp2.test (under subproject2)

Base project’s build.sbt is available directly under MultiProjectSBT directory whereas every submodule have their independent build.sbt.

**Source code :**



Copy HelloMain.scala into com.main.test package of MultiProjectSBT

Copy Human.scala, Friend.scala, Animal.scala into com.sp1.test package of subproject1

Copy FriendTraitClient.scala into com.sp2.test package of subproject2



**SBT configuration files :**

Copy and paste following build.sbt as seen in below image:





We have configured two module i.e. sp1 and sp2 into our parent build.sbt (i.e. the one under MultiProjectSBT directory) . Here ‘dependsOn’ configuration describe that sp2 module is dependent on sp1, It means you can’t build sp2 without building sp1 module.



Finally we should have following build.properties into MultiProject’s project folder:





All the source codes and configuration files are now at it place, Let’s begin building the project.

**General build procedure**

Open command prompt and change to directory (e.g C:\Scala\_SBT\MultiProjectSBT), Type following commands :

*C:\Scala\_SBT\MultiProjectSBT> sbt*

*>clean*

*>compile*

*>run*



To list all the projects in given build, You can type following command on sbt interactive shell

*>projects*



If you can able to execute above command without any error that means your project is configured properly.

Let’s package the build by typing clean, compile, package in sequence at the sbt interactive shell as can be seen in following image.



After successful packaging, Check the project structure in the eclipse IDE(see below screen). You should have three project artifacts (jar files) created under your project structure.



subproject1\_2.11-1.0.jar (under subproject1 directory)

subproject2\_2.11-1.0.jar (under subproject2 directory)

multiprojectsbt\_2.10-0.1-SNAPSHOT.jar (under project base directory)

In order to understand the dependency of sp2 module on sp1, Delete ‘dependsOn(sp1)’ and try compiling the project, You should get error as seen below.



It shows, sp2 must have an access to all the scala class dependency in order to build correctly.