

# Scalatron Player Setup

## About Scalatron

Scalatron is an educational resource for groups of programmers that want to learn more about the Scala programming language or want to hone their Scala programming skills. It is based on Scalatron BotWar, a competitive multi-player programming game in which coders pit bot programs (written in Scala) against each other.

The documentation, tutorial and source code are intended as a community resource and are in the public domain. Feel free to use, copy, and improve them!

## Table of Contents

<b>Overview</b>	<b>3</b>
The "Casual" Path	3
The "Serious" Path	3
<b>The "Casual" Path: Using the Scalatron Browser IDE</b>	<b>4</b>
<b>The "Serious" Path: Building Locally</b>	<b>5</b>
Set up Java	5
Set up Scala	5
Set up a Development Environment	5
<b>Set up IntelliJ IDEA</b>	<b>6</b>
Installation & Configuration	6
Set up a Project for Your Bot	11
Write and Build Your Bot	16
Debug Your Bot	17
<b>Working with the Command Line</b>	<b>18</b>
Creating new Scalatron project	18
Compiling Scala and Scalatron	18
Publishing the bot to the server	19
VIM and Scala	19
<b>Working with SBT (Simple Build Tool)</b>	<b>19</b>
Background	19
A Basic SBT Build File	20
Scalatron Dependency and Tests	20
Starting the Scalatron Simulation	21
Conclusion	21
A CLI-Friendly Scalatron Template	22



---

## Overview

Scalatron is a game server application running on a central computer. This server hosts the bot tournament by continuously simulating short rounds of the Scalatron BotWar game. Players participate in the tournament by publishing their bots into the server.

To publish a bot, a player has to provide the server with a `.jar` archive file that contains the bot's control function, which was implemented in Scala and then compiled. There are two approaches to doing this, which in the Scalatron documentation are referred to as the "serious" and "casual" paths, respectively.

### The "Casual" Path

The "casual" path is intended for less experienced programmers or for shorter bot coding sessions (2-3 hours). On this path, players write, build, debug and publish their bots in a web browser. The browser-based development environment is provided by an embedded web server which is hosted inside the game server and requires no setup by the user.

A subsequent chapter of this document describe how to code and build your bot using the browser-based development environment that is provided by the Scalatron server.

### The "Serious" Path

The "serious" path is intended for experienced programmers planning for a longer bot coding session (5-7 hours). On this path, bots are built locally by each player, using an IDE or command line tools. The bots are then published into the tournament by copying them into the plug-in directory on the central computer from which the game server loads them at the start of each round.

If you intend to go down the "serious" path, you need to complete the following steps in order to publish a bot program into the Scalatron game server:

1. write a Scala program consisting of one or more `.scala` source files
2. compile the Scala source code into Java `.class` files
3. bundle those class files into a Java `.jar` archive file
4. publish your `.jar` archive file into the plug-in directory used by the game server

Although all of these steps can be performed with a simple text editor and by using the command line, many people prefer to use an Integrated Development Environment (IDE) that supports Scala, such as Eclipse or IntelliJ IDEA. Unfortunately, the setup process is pretty involved and may take up to 30 minutes. Ideally someone in your group should already familiar with the development environment you intend to use.

---

---

## The “Casual” Path: Using the Scalatron Browser IDE

The Scalatron server provides a built-in, browser-based development environment, the Scalatron IDE. To use this environment, you need to have access to a running Scalatron server somewhere on the network, or on your own computer. If you are participating in a workshop with multiple users, ask the organizer for the address of that machine.

If you want to run your own server on your local machine, follow the simple steps outlined in the Scalatron [Server Setup](#) guide. Basically:

1. download the Scalatron distribution, e.g. from <http://scalatron.wordpress.com>
2. unzip the compressed file to a local directory, e.g. `/Scalatron`
3. launch the game server by double-clicking the application, e.g. at `Scalatron/bin/Scalatron.jar`
4. open a browser and point it to the server, usually at <http://servername:8080>
5. log in as Administrator and create a user account for yourself

Once you have access to a running Scalatron server and have an account on it, perform the following steps:

1. open a browser and point it to the server (e.g. using ``http://servername:8080``, where `servername` is the name of the computing running the Scalatron server; if you are running it locally, use `localhost`)
  2. This will display a log-in screen that should show your name as one of the available user accounts. Log in.
  3. This will take you into a browser-based code editor, with a bare-bones bot pre-loaded. Edit the bot, then click **Run in Sandbox** to build your bot and run it in a private sandbox game.
  4. Once you have something that works, click **Publish into Tournament** to build your bot and publish it into the tournament.
  5. Woohoo!
-

---

## The “Serious” Path: Building Locally

### Set up Java

- Scala source code compiles into Java byte code, which can be executed with a [Java Virtual Machine](#) (JVM). In order to execute applications written in Scala, you therefore need such a JVM on your computer. The JVM is part of the Java Runtime Environment (JRE), which is generally pre-installed on most operating systems. If it is not, you can download it for free at <http://www.java.com>.
- To compile Scala code and to properly work with the Java library (e.g., to see its source code), however, you also need the the Java Development Kit (JDK). If you have already been developing for Java, you should obviously already have this, too. If not, download a JDK [from Oracle](#) or from the web site of your OS provider.
- Please refer to the documentation of the IDE you will be using on how to configure it to use the appropriate JDK. Maybe some day we'll be able to also add the steps to this guide...

### Set up Scala

- Compiled Scala code relies on functionality defined in the Scala standard library, which is therefore required to be present on your computer (as a Java archive .jar file). There are several ways to obtain and install this package, but the most convenient way is to get the entire “stack” of Scala technologies available from Typesafe, a company that supports a range of key Scala projects. So:
- Download the [Typesafe Stack](#) from <http://typesafe.com>
- Install the Typesafe Stack
  - on OSX, the default directory is `/Applications/typesafe-stack`
  - on Windows, the default directory is `C:/Program Files/typesafe-stack`

### Set up a Development Environment

Subsequent chapters of this document describe how to build your bot on the "serious" path via the following alternatives:

- Using [IntelliJ IDEA](#), a sophisticated development environment for Java that has pretty good Scala support (this is a commercial tool, but a free "Community Edition" is available)
  - Using the command line
  - Using [SBT](#), the [Simple Build Tool](#) for Scala
-

## Set up IntelliJ IDEA

**IntelliJ IDEA** is a commercial development environment for Java and many other languages. It now (April 2012) has quite adequate support for Scala development via a Scala plug-in. But it is just one of several options you can use. Another one is the **Scala IDE** based on **Eclipse**. Unfortunately, this tutorial is not (yet) able to cover both environments, so the following steps are, for the moment, specific to IntelliJ IDEA. They were tested with IDEA version 11.2 on MacOS X.

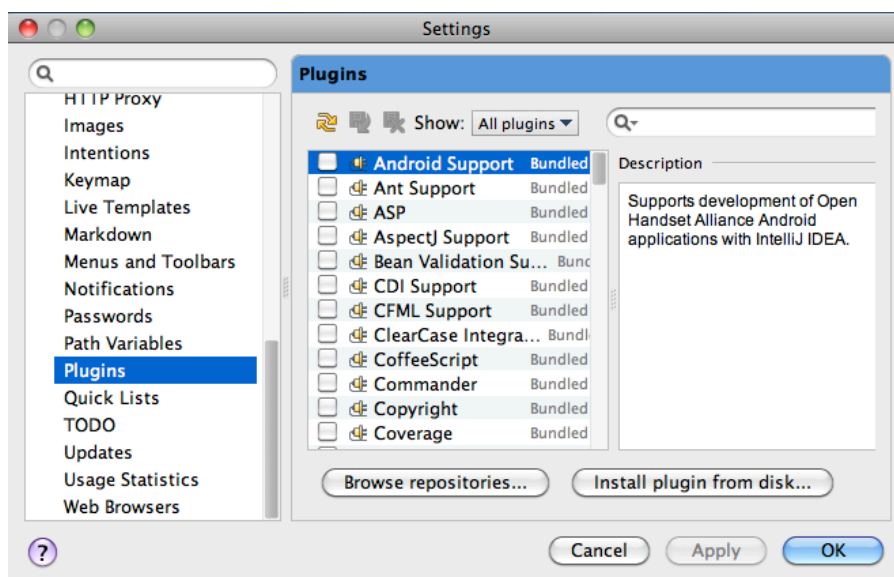
### Installation & Configuration

#### Download and Install the IDE

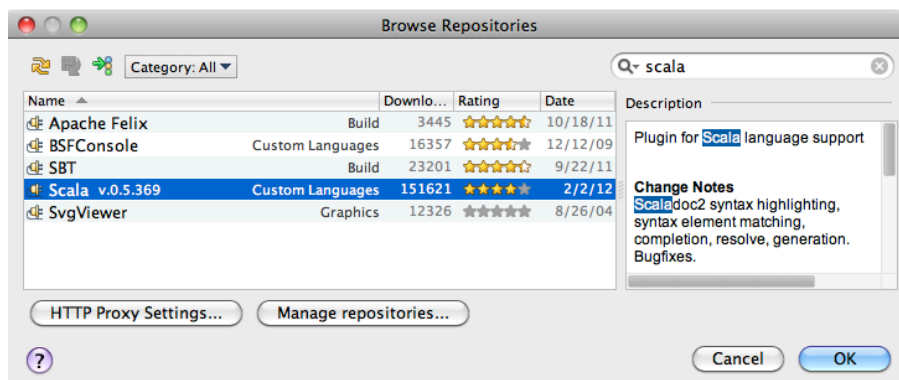
- Download IntelliJ IDEA from <http://jetbrains.com>
- You can use either of the following versions:
  - **Ultimate Edition** (Evaluation Version)
  - **Community Edition**

#### Install the IDEA Scala plug-in

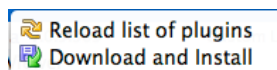
- In the main menu, select **IntelliJ IDEA** > **Preferences**
- In the list on the left, scroll down to the entry **Plugins**



- At the bottom of the pane that appears, choose **Browse Repositories**
- Via the search box at the top right, search for “**Scala**”
- In the list that appears on the left, scroll down to the entry called **Scala**



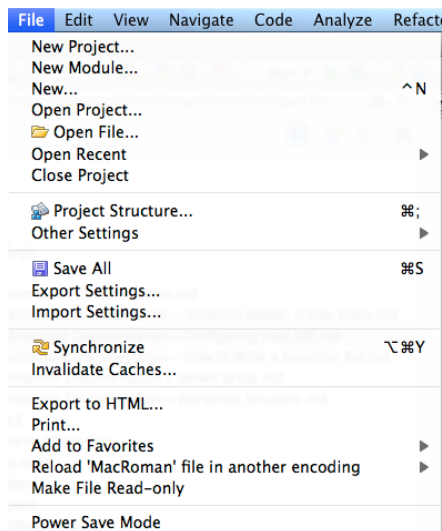
- Control-click on that entry and select **Download and Install...**



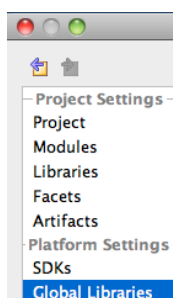
- After the plug-in was downloaded and installed, you will have to re-start IDEA

### Configure the Scala Compiler

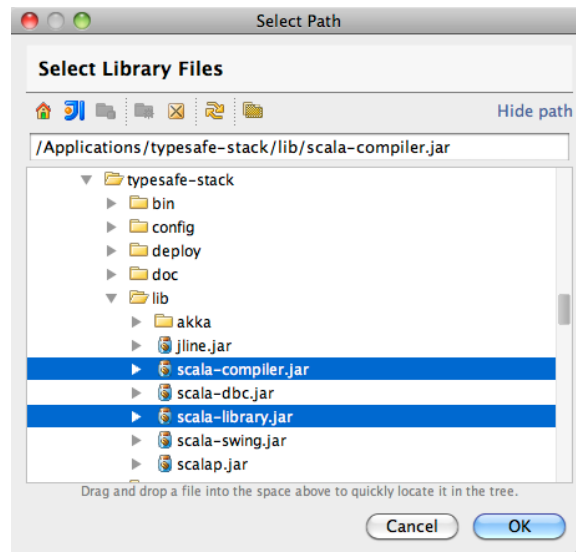
- In order to compile Scala code in IDEA, we need to tell IDEA (or rather, the Scala plug-in within IDEA) where it can find the Scala compiler. The Scala compiler comes as part of the Typesafe Stack which we already installed, and resides in a Java archive file called `scala-compiler.jar`.
- We now need to create a kind of “library bundle” containing both the Scala compiler and the Scala library, which we’ll then tell the plug-in to use when it wants to compile our code.
- In the main menu, select **File > Project Structure...**



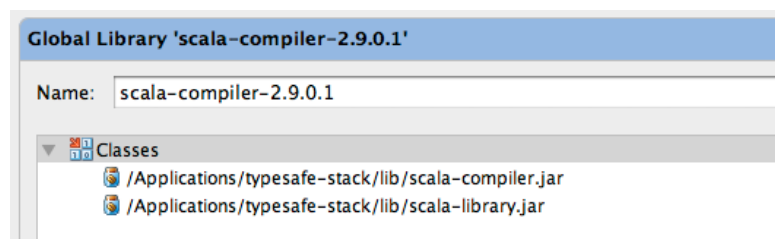
- In list at upper left, select **Global Libraries**



- We want to add an entry called “**scala-compiler**” to the list of “library packages” in the center. To do that, we...
- Click on the ‘+’ icon above the list in the center
- In the **Select Library Files** dialog that appears...



- Browse for and select the following libraries on disk:
  - /Applications/typesafe-stack/lib/scala-compiler.jar
  - /Applications/typesafe-stack/lib/scala-library.jar
- This will add an entry "scala-compiler" to the global library list; you will want to end up with the following structure:

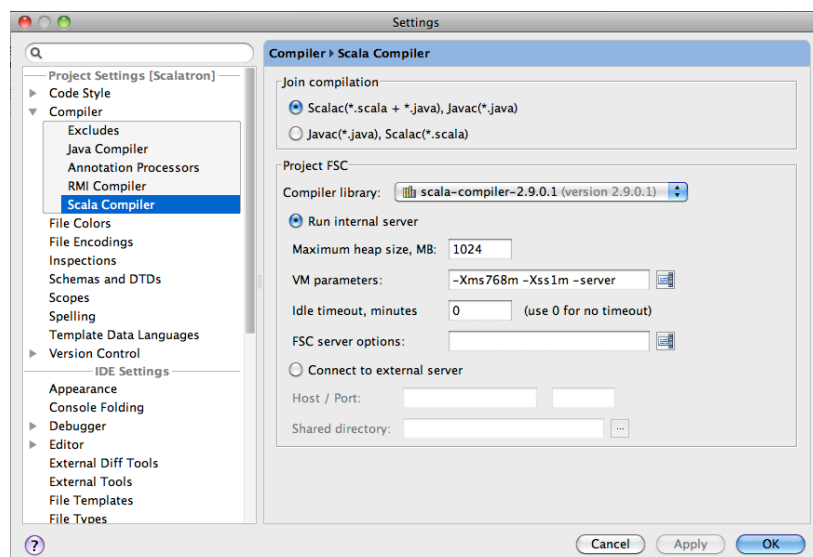


- We can now configure the IDEA Scala plug-in to use this “library package” in order to access the compiler and to compile our Scala code.

### Configure the Fast Scala Compiler option

- In the main menu, select **IntelliJ IDEA > Preferences** (on MacOS X) or **File > Settings** (on Windows)
- This will pop up a dialog with the title **Settings**:

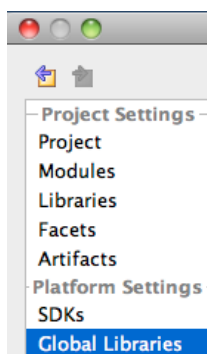




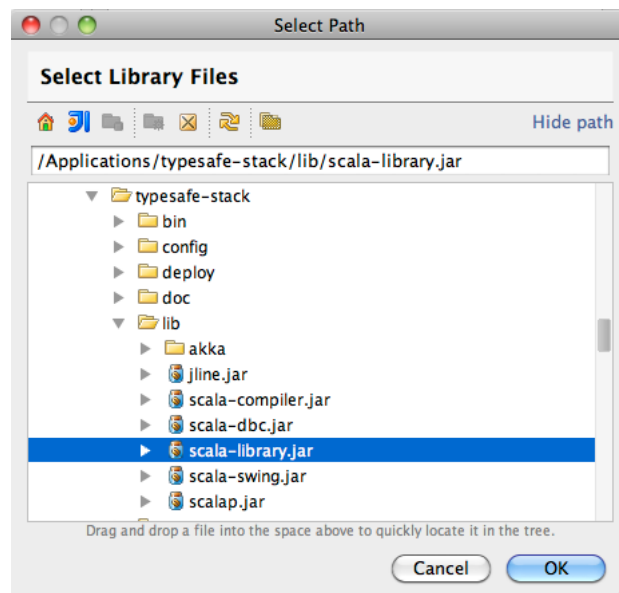
- In this dialog, in the tree at left navigate to **Compiler > Scala Compiler**
- In the right hand panel, find the box titled **Project FSC**
- In the drop-down list **Compiler library**, select the Scala compiler “library package” we created earlier: `scala-compiler.jar` (or something similar, whatever you’d entered as the name).
- This will allow IDEA to do fast incremental compilations using an internal compilation server.


### Configure the Scala Library

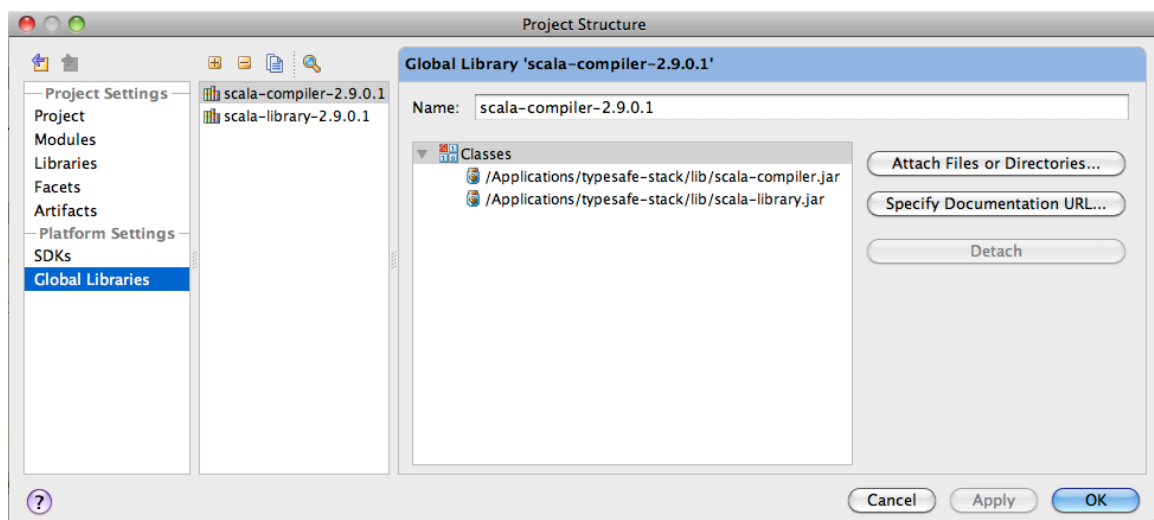
- **Note: this step is NOT a duplicate of the preceding step! You really need to do this, too!**
- Before, we created a “library package” for the IDEA Scala plug-in that contained the Scala compiler and the Scala standard library. For our own projects, we’ll also need to include the Scala standard library as a dependency - but we don’t need the compiler. We’ll therefore create another “library package”, but this time one that contains **ONLY** the Scala standard library (`scala-library.jar`).
- In the main menu, select **File > Project Structure...**
- In list at upper left, select **Global Libraries**



- We want to add an entry called “**scala-library**” to the list of “library packages” in the center. To do that, we...
- Click on the ‘+’ icon above the list in the center
- In the **Select Library Files** dialog that appears...



- click the '+' icon (  ) to add one library and select the following libraries on disk:
  - /Applications/typesafe-stack/lib/scala-library.jar
- this will add an entry "scala-library" to the global library list. Your project/module will later depend on this library.
- just to verify, the setup you want to end up with after adding "library packages" in both steps should look like this:

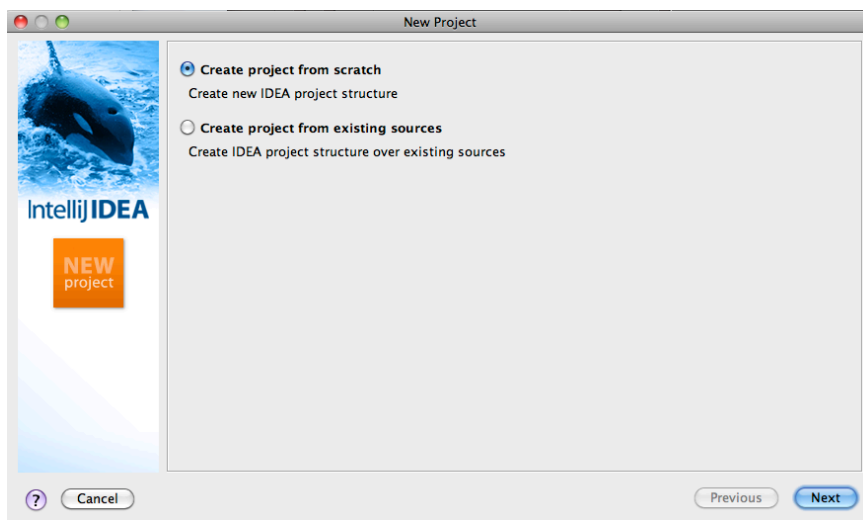


## Set up a Project for Your Bot

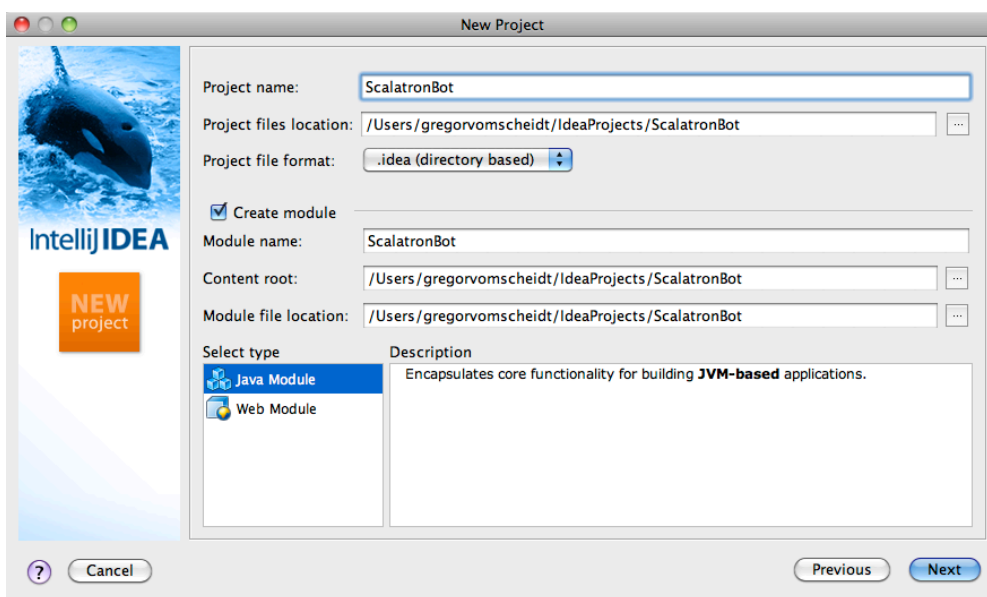
The following steps are specific to IntelliJ IDEA and were tested with IDEA version 11.2.

### Create a Scala Bot project

- This is the project you will use to write and compile your bot source code
- In the main menu, select **File > New Project...**



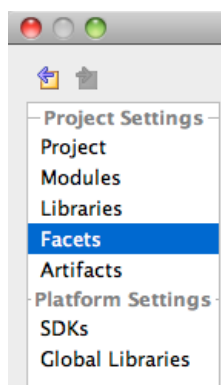
- enter a project name ("**ScalatronBot**")




- Accept all defaults
- Do NOT yet configure Scala - we will do this manually. As of this writing (March 2012), the process of configuring the Scala Facet in IDEA still seems a bit flaky.

### Add the Scala Facet to you Project

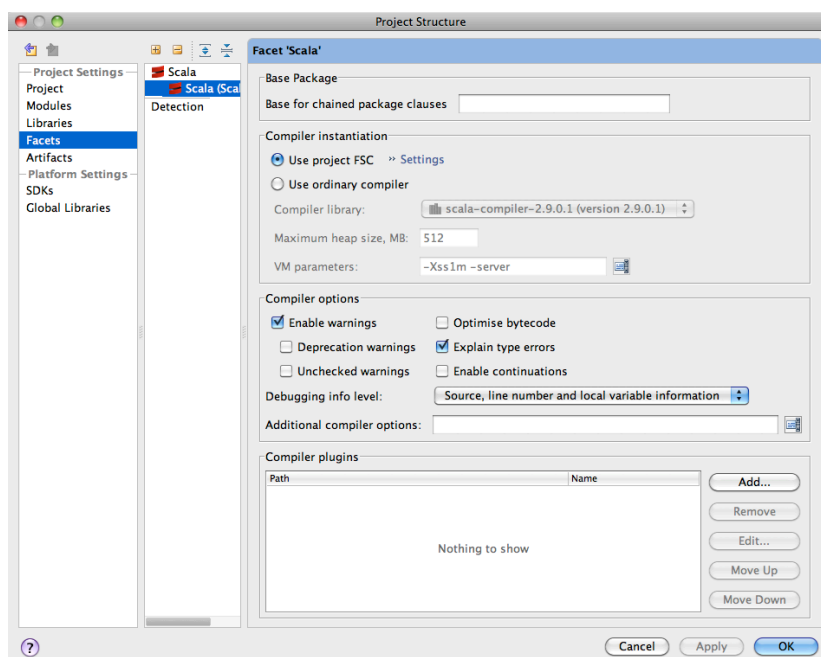
- A “Facet” is IDEA’s way of knowing what technologies your project uses and to store configuration information for these technologies. We need to add the “Scala” facet to your project. In the next step, we’ll configure it.
- In the main menu, select **File > Project Structure...**
- In list at upper left, select **Facets**



- Above the middle list, click the '+' icon (  ) to add a new facet
- Add the Scala facet to your project/module

### Configure the Scala Facet

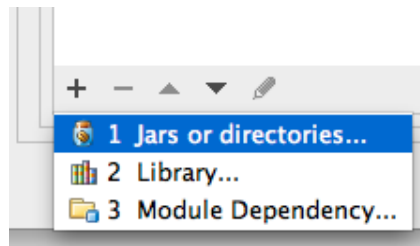
- First we'll configure your project to use the Fast Scala Compiler. The Fast Scala Compiler is an incremental compile server managed by the Scala plug-in within IDEA. Using it dramatically accelerates your build times because symbols do not need to be re-loaded for every build.
- From the preceding step, you should still have the **Project Structure** dialog visible. In the list at the left, you should have selected **Facet** and in the list in the middle you should have selected the **Scala** facet.



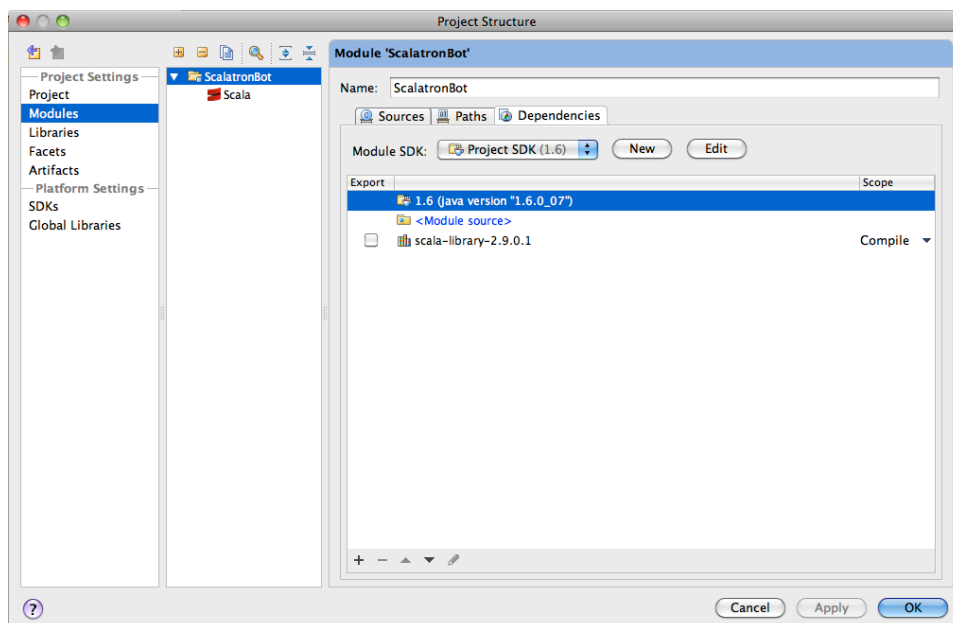
- Now in the panel on the right titled **Compiler instantiation**, check the box **use project FSC** ("FSC" is the "Fast Scala Compiler").
- This tells IDEA that to compile your Scala source code, it should use the Fast Scala Compiler. We configured this compiler earlier on to use the appropriate Scala compiler library.
- In the same dialog, check the box "**Explain type errors**" further down. This tells the compiler to generate more verbose error messages when it encounters type errors.

## Configure Scala Library as Module Dependency

- We now need to tell IDEA that your Scala “ScalatronBot” project depends on the Scala “library package” containing `scala-library.jar` which we created earlier.
- From the preceding step, you should still have the **Project Structure** dialog visible.
- In the list at the left, you should now selected **Modules**
- In the list in the middle, you should select your module ("ScalatronBot")
- In the panel on right, traverse to the tab **Dependencies**
- Click the '+' icon at the bottom of panel to add a dependency (Note: in some IDEA versions this is to the right of the list; and in some there may be a button “Add...”).
- From pop-up menu, select **Library...**



- From the list, select `scala-library.jar`
- The result should look something like this:

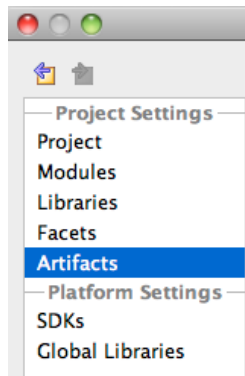


- Close the dialog with **OK** to apply the settings

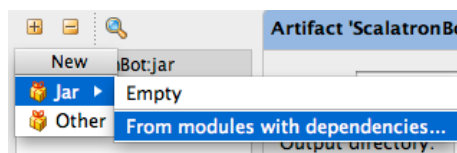
## Configure the .jar Artifact

- An “artifact” is something of (hopefully) permanent value generated by the build process. In our case it will be a Java archive (`.jar`) file that contains the compiled bot. The server will attempt to locate and open this `.jar` file and to extract a factory class from it which it can then use to instantiate a control function for our bot.
- In the main menu, select **File > Project Structure...**
- This re-opens the **Project Structure** dialog

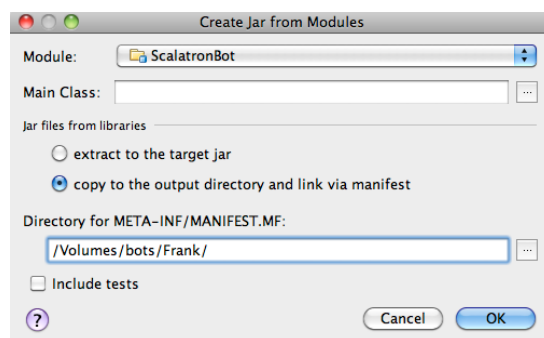
- In list at upper left, select **Artifacts**



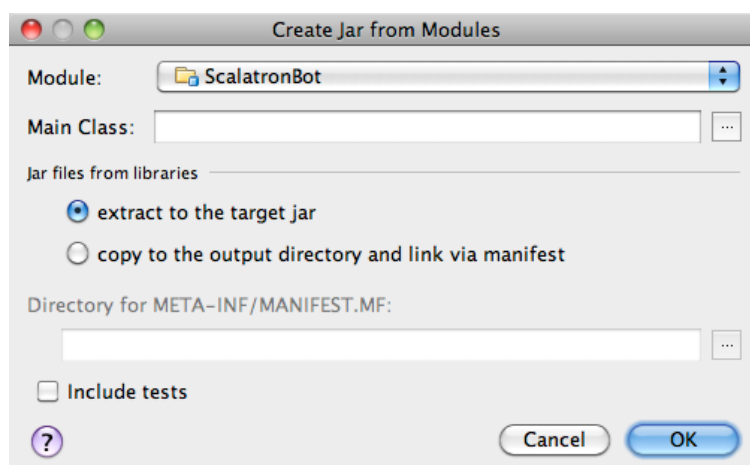
- Above the list in the middle of the dialog, click the '+' icon (📁) to add a new artifact
- From the pop-up menu, select **'Jar'**
- In the options that appear, select **from Modules with dependencies...**



- This will pop up a dialog with the title **Create Jar from Modules**:



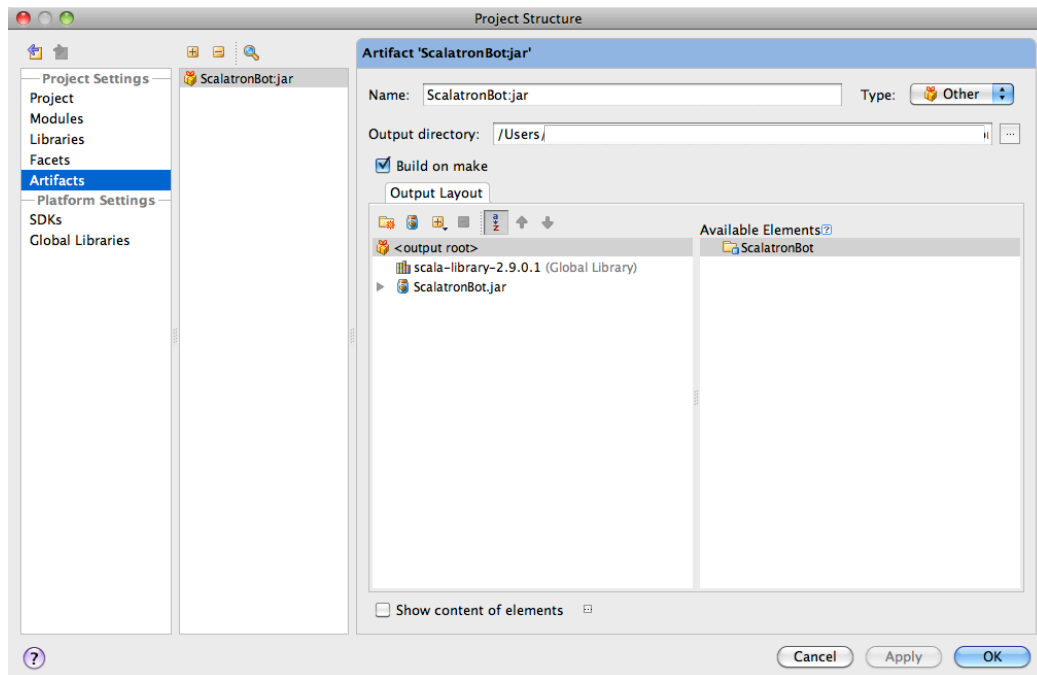
- Leave the field **Main class** empty
- In the section **Jar files from libraries** select the option **copy to the output directory and link via manifest**:



- The reason we do this is because we do not need or want the Scala standard library (the `scala-library.jar` file, which takes up about 9MB) to be bundled into your bot, since the game

server has that code anyway. We only want the `.class` files generated from your own `.scala` source files in the plug-in (these will be closer to 25KB). This also saves a bit of copy time each time you build.

- Confirm with **OK**
- In the main dialog, in the list in the middle, an entry called “ScalatronBot.jar” should have appeared and be selected. In the associated panel on the right, check the box **Build on make**. This will tell IDEA to automatically create the plug-in `.jar` file we just configured as an artifact each time you build your code.



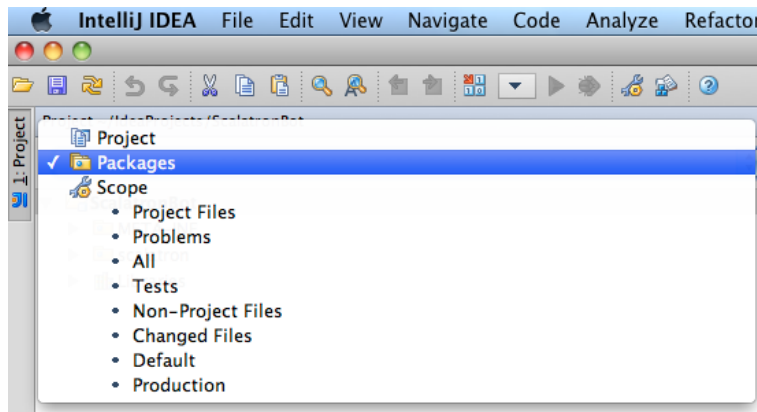
- Optionally, you can remove the scala-library entry from the list titled **Output Layout**
- Now change the output directory to your plug-in directory on the network share where the game server will be looking for bot plug-ins, e.g. (if your name is Frank), to:
  - /Volume/bots/Frank/

## Write and Build Your Bot

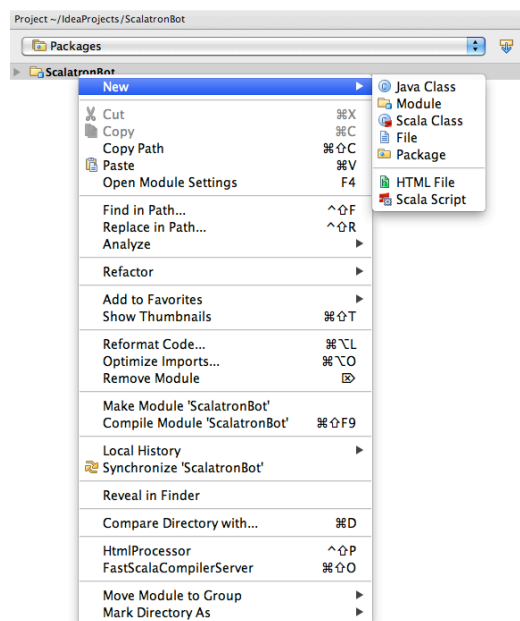
The following steps are specific to IntelliJ IDEA and were tested with IDEA version 11.2 on MacOS X.

### Create a Bot Code File

- You should now be in the main window of IDEA, with a panel containing a tree structure showing the contents of your project on the left.
- Optionally, in the drop-down at the top of that panel at the left (called **Project**), change the view setting from **Project** to **Packages**. This simplifies the display a bit and only shows packages relevant to your project.



- Control-click on your Module ("ScalatronBot") and select **New > Scala Class**:



- Enter the class name **"Bot"** and click **OK**
- in the file, enter the following Scala source code:

```
class ControlFunctionFactory {
    def create = (input: String) => "Status(text=Hello World)"
}
```



## Build Your Bot

- In the main menu, select **Build > Make Project**. You'll soon use this a lot. Make a mental note of the keyboard shortcut...
- This should compile the Scala code in your `Bot.scala` file into the `.jar` artifact file we configured earlier.
- If you configured your artifact's destination path to be your personal directory below the shared plug-in directory on the game server, the `.jar` file will be copied into that directory and the next time the server starts a game round and re-scans the plug-in directories, your plug-in will be picked up and your bot should appear in the game arena, saying "Hello World".
- Now read the [Game Rules](#), the [Protocol](#) and then get started with the [Tutorial](#)! Have fun!

## Debug Your Bot

The IntelliJ IDEA development environment provides a very convenient mechanism for debugging your bot by connecting to a running Scalatron server application, generally on your local machine.

### Setting up Remote Debugging

To use the remote debugging mechanism, follow these steps:

- In IDEA, select **Run > Edit Configurations** in the main menu.
- In the dialog **Run/Debug Configurations** that appears, add a new configuration by clicking the **+** at the top left.
- In the pop-up menu **Add New Configuration** that appears, select **Remote** as the configuration type.
- In the tab that appears on the right, choose a name for your configuration, such as **Scalatron Remote**.
- The text in the table also explains the option that you need to add to the command line invocation of the Scalatron server:  
`-xdebug -xrunjdp:transport=dt_socket,server=y,suspend=n,address=5005`
- Choose the option at the bottom **Build on Make** to automatically build your `.jar` file before you connect to the server.
- Close the dialog with **OK**

### Initiating a Remote Debugging Session

To debug your bot, you now need to follow a two-step process:

1. Start the Scalatron Server with the required JVM options.
2. Build your bot and connect the debugger to the running server

To start your server, you have again two options:

- open a terminal window, then launch it from the command line
- within the IDE, configure the Scalatron server as a tool and run it from there

In both cases, you will need to specify the command line options you want. For details, please refer to the Scalatron [Server Setup](#) guide. Here is an example command that integrates the remote debugging options with some Scalatron options:

```
java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005
-jar Scalatron.jar -headless no -x 50 -y 50
```

To build the bot and connect to the debugger, simply select the **Scalatron Remote** configuration you created earlier and choose **Run > Run...** from the main menu.

---

## Working with the Command Line

The following guide was kindly provided by Lauri Lehmijoki on his blog at <http://lauri.lehmijoki.net/en/nerd/scalatron-on-cli.html>

### Creating new Scalatron project

Create a new project directory for your bot:

```
mkdir MorningLightMountain && cd MorningLightMountain
```

(MorningLightMountain is a nefarious alien in Peter Hamilton's Commonwealth Saga, a science fiction book series.)

Then create the required sub-directories:

```
mkdir bin src
```

Edit the file `src/Bot.scala` and add the code for the simplest bot:

```
// Contents of src/Bot.scala
class ControlFunctionFactory {
  def create =
    (input: String) =>
      "Status(text=MorningLightMountain will conquer the whole universe!)"
}
```

Next we'll compile the Scalatron bot, and then we'll deploy it on the server. Will humans prevail, or will MorningLightMountain take over and enslave us?

### Compiling Scala and Scalatron

Add an empty `MANIFEST.MF` file into your project's root:

```
touch MANIFEST.MF
```

Jar needs it as a header file.

With the following script (available as a gist on github at <https://gist.github.com/2276912>) you can compile your Scalatron bot:

```
#!/bin/bash
# For compiling a Scalatron bot.
# Assumes that the MANIFEST.MF file is located in the current directory.
# Author: Lauri Lehmijoki - http://lauri.lehmijoki.net
set -e

BUILD_DIR=.build
rm -rf $BUILD_DIR && mkdir $BUILD_DIR

echo "Building the Scalatron bot..."
```

```
scalac -sourcepath src -d $BUILD_DIR `find src -name *.scala`

cd $BUILD_DIR

# MANIFEST.MF can be an empty file, but it has to exist
jar -cfm ../bin/ScalatronBot.jar ../MANIFEST.MF *.*

cd ..

echo "...ready!"
echo "Produced bin/ScalatronBot.jar"
```

Save the script with name `build` into your project's root and make it executable with

```
chmod u+x build.
```

Now you can compile the Scalatron bot with

```
./build.
```

## Publishing the bot to the server

This step is easy. All we need to do is to make the `Scalatron.jar` available for the server. Symbolic links will help us. Let's assume that you've installed the Scalatron server in `~/Scala/Scalatron/server` and the MorningLightMountain project is in `~/Scala/Scalatron/MorningLightMountain`. Run the following commands to create the symbolic link:

```
mkdir ~/Scala/Scalatron/server/bots/MorningLightMountain
ln -s ~/Scala/Scalatron/MorningLightMountain/bin/ScalatronBot.jar ~/Scala/Scalatron/server/bots/MorningLightMountain/ScalatronBot.jar
```

Start the Scalatron server, and MorningLightMountain is there!

## VIM and Scala

VIM fits well into Scala development. However, it doesn't have built-in support for Scala indentation and syntax highlighting. Lorenzo Danielsson has written a [blog post](http://lorenzod8n.wordpress.com/2008/01/11/getting-scala-syntax-highlighting-to-work-in-vim) on how to get them working (<http://lorenzod8n.wordpress.com/2008/01/11/getting-scala-syntax-highlighting-to-work-in-vim>).

## Working with SBT (Simple Build Tool)

The following guide was adapted from an article written by J. M. Hofer, which he kindly made available on his blog at <http://jmhofer.johoop.de/?p=292> and gave permission to include here.

### Background

J. M. Hofer says: "Unfortunately, it's not yet well documented or obvious how to set the Scalatron environment up, except for a huuuge instructions document for IntelliJ IDEA. I mean, 13 pages?! - That just has to be doable in an easier way! As I'm an Eclipse user and somewhat fluent with SBT, I thought I'd try to setup Scalatron with an SBT build. With the sbteclipse (and sbt-idea) plugins, I'll then get the IDE setup for free."

## A Basic SBT Build File

So, let's create a basic SBT build file. As I already suspect that I'll need a build that's a bit more complicated than a few lines of configuring standard settings, let's start with a Build.scala file:

```
import sbt._
import Keys._

object Build extends Build {
  val bot = Project(
    id = "mybot",
    base = file("."),
    settings = Project.defaultSettings ++ botSettings)

  val botSettings = Seq[Setting[_]](
    organization := "de.johoop",
    name := "my-scalatron-bot",
    version := "1.0.0-SNAPSHOT",

    scalaVersion := "2.9.1",
    scalacOptions ++= Seq("-deprecation", "-unchecked"))
}
```

This simply initializes my bot project and adds a few standard Scala options I usually like to use.

## Scalatron Dependency and Tests

Next, I want to add the dependency to the Scalatron jar file to the project. As Scalatron isn't published anywhere, I'll just create a lib directory and put the jar file from the Scalatron distribution in there.

Another basic thing that's still missing from the build is the configuration of some kind of tests. I absolutely want to write tests against my bot's functionality, as it's very difficult to debug a bot during a running bot war. I'll use Specs2 for that, of course, because it's all kinds of genius. We'll add the following lines to our botSettings:

```
libraryDependencies ++= Seq(
  "org.specs2" %% "specs2" % "1.8.2" % "test",
  "org.pegdown" % "pegdown" % "1.0.2" % "test",
  "junit" % "junit" % "4.7" % "test"),

testOptions := Seq(
  Tests.Filter(_ == "de.johoop.scalatron.BotSpec"),
  Tests.Argument("html", "console")),

testOptions <+= crossTarget map { ct =>
  Tests.Setup { () =>
    System.setProperty("specs2.outDir",
      new File(ct, "specs2").getAbsolutePath)
  }
},
```

The first few lines add all required dependencies: pegdown for the nice HTML reports, junit for running the tests from within Eclipse using JUnit.

Then I tell SBT to just execute my main test specification called BotSpec and ignore any sub specifications. And I tell it to create HTML reports, too, and where to put them.

## Starting the Scalatron Simulation

I can now already hack and test my bot to my heart's content. However, I can't yet try it out in the Scalatron environment. Let's do something about this. I need a way to start the Scalatron simulation, and I probably want to configure the directory where all the bots are kept (the enemy bots as well as my own).

For this, I first create two simple SBT keys and add them to the `Build` class:

```
val botDirectory = SettingKey[File]("bot-directory")
val play = TaskKey[Unit]("play")
```

The `botDirectory` setting will simply be initialized to where I want to keep the bots (in the `botSettings` sequence):

```
botDirectory := file("bots"),
```

The `play` task will have the job to take my bot jar and deploy it into the bot directory, and then to start a (forked) Scalatron simulation. In order to be able to do this, it will require quite a few dependencies as inputs:

- `botDirectory`: where the bots go,
- `name`: in order to name my bot in the bot directory,
- `javaOptions`: so that I can configure memory settings and the like for the simulation,
- `unmanagedClasspath in Compile`: to retrieve the `scalatron.jar` from and finally
- `Keys.package in Compile`: to retrieve my bot's jar from.

So, here we go (again, add this to the `botSettings`):

```
play <:= (botDirectory, name, javaOptions,
          unmanagedClasspath in Compile,
          Keys.`package` in Compile) map {
          (bots, name, javaOptions, ucp, botJar) =>

            IO createDirectory (bots / name)
            IO copyFile (botJar, bots / name / "ScalatronBot.jar")

            val cmd = "java %s -cp %s scalatron.main.Main -plugins %s" format (
              javaOptions mkString " ",
              Seq(ucp.files.head, botJar).absString,
              bots.absolutePath)
            cmd run
          }
}
```

## Conclusion

And that's it!

I can now deploy my bot to the bot directory and run the simulation from within SBT just by typing `play` into the SBT console.

Except... except that I noticed that the simulation seems to use a lot of memory, so I added a java option for this:

```
javaOptions += "-Xmx1g"
```

And also, the Eclipse project I generate via the `sbteclipse` plugin unfortunately warns me about two Scala runtimes (one generated by the plugin, one inside the Scalatron jar). I won't do anything

against this, hoping that we'll soon get a clean published Scalatron jar that doesn't automatically include the Scala runtime...

Here's my complete build file as a gist: <https://gist.github.com/2260897>

Now go and create your own Scalatron bot!

**Note from the (Scalatron) Editor:** there is no need to bundle the Scala runtime (`scala-library.jar`) into your bot; dropping it from the `ScalatronBot.jar` should get rid of the sbteclipse project warning.

## A CLI-Friendly Scalatron Template

Independently, David Winslow (@godwinsgo) was so kind as to post a command-line-friendly SBT template for Scalatron on github at <https://github.com/dwins/scalatron-template>. Thanks!