journal: Computer

author: Roxana Tesileanu, roxana.te@web.de

affiliation: INCDS, Brasov Station

date: October 2017

Using Linux as a Development Platform for Scala Projects

Abstract

Scala has already proven its advantages, and many big projects are implemented using this language. Though, newcomers and researchers from other branches than computer science might experience anxiety facing this functional programming language. With no reason though, because there already are many useful learning resources available. This paper offers an introduction into the software development process of Scala projects pointing to some of the available learning resources.

1. Introduction

In order to implement software development projects, one should consider which tools to choose for the different components of the development system, which according to Rehman and Paul (2003) are the following: the hardware platform, the operating system, editors, compilers and assemblers, debuggers, version control system and bug tracking (Tesileanu 2017). For implementing Scala projects, working with the Simple Build Tool (SBT) ([http://www.scala-sbt.org](http://www.scala-sbt.org/)) condenses the list of the components of a development system, as it unites different steps in one tool (compiling, building, testing and debugging) (see Suereth and Farwell, 2016). Another advantage of using SBT is the opportunity of working interactively in Scala REPL, a tool for evaluating expressions in Scala (<https://docs.scala-lang.org/overviews/repl/overview.html>) within a SBT session with all dependencies and project classes on the classpath, to develop bits of code which can then be inserted in the editor of choice.

It is also important to understand that software development "is not just writing code", but rather a more comprehensive process (Rehman and Paul, 2003). Rehman and Paul (2003) describe it as comprising the following steps: requirement gathering, writing functional specifications, creating architecture and design documents, implementation and coding, testing and quality assurance, software release, documentation, and, support and, development and release of new features. Each project starts with a requirement analysis which investigates the real-world need of the final product. Which functions should the new software carry in real-world problem solving within the specified domain? Further, functional specifications are declared to state the functionality of a software product at an abstract level "defining its input/output behavior". On the basis of the functional specifications, an architecture of the product is created. The architecture "defines the different components of the product and how they interact with each other", without providing the explicit details on how they should be implemented to reach the desired functionality. This happens at the design stage, when design documents are created which define each individual component to the level of functions and procedures. Using the design documents and development tools (SBT and editor) the code is then implemented and tested. There are more types of testing: unit testing (testing one part or one component of the product using test cases to test functionality of this part of the software), sanity testing to check if all components compile, regression or stress testing to check the long-term behavior of the product when used continuously over a period of time, and functional testing using test cases built on functional specifications. If a bug (an anomaly) is found it must be reported and fixed. The documentation includes: technical documentation developed during the development process, technical documentation prepared for technical support staff, and end-user manuals and guides. Finally, the last stage of the life cycle of a software development project is the support and release of new versions depending on requirements.

The present article aims at introducing the reader to the tools needed to put in motion the development system of Scala projects on Linux, with emphasis on the Ubuntu distribution. The following sections will cover brief introductions in using the VIM editor as an editor for Scala code and using SBT to create and manage Scala projects, also addressing the issue of dependency management.

2. Using VIM as an editor for Scala code

VIM is an editor which can be used from within the Linux command line and can be adapted to support the editing of Scala code by installing the vim-scala package of Derek Wyatt found at <https://github.com/derekwyatt/vim-scala>.

Under Linux you can install VIM using the package tools at hand (like e.g. dpkg, apt-get, aptitude, rpm or yum) depending on the installed distribution. In Ubuntu the command for installing VIM is: “$ sudo apt install vim”. Afterwards, you open the command line and launch VIM by typing “$ vim <ENTER>”. The following instructions will make the first VIM session easy to pass through:

- because VIM starts in command mode you can change to insert mode with the "i" key

- to exit insert mode and return to command mode, press the <ESC> key

- to write the open file to the hard drive use the command ":w filename" (in command mode) or just ":w" if the file already has a name

- to save the changes type ":" in command mode

- to exit VIM type ":wq" to quit and save changes, or ":q!" to discard changes (both in command mode)

- pressing <ESC> will not just place you in command mode but also cancel an unwanted and partially completed command in command mode.

The following paragraphs will help you navigate through the VIM file, and, process text (append, delete, copy, paste, search and replace, edit multiple files and get help).

To move the cursor you use: "l" (or right arrow), "h" (or left arrow), "j" (meaning going down one line), or "k" (meaning going up one line). Moving inside the current line is possible with "0" (to go to the beginning of the current line) and "$" (to go to the end of the current line). To go to the end of the file, i.e. to the last line of the file you use "G" and to go to the beginning of the file, i.e. to the first line you use "gg". To go to a specific line you use "numberG" (for example: 2G to go the the second line).

When you are in command mode, you can start inserting text by using the "i" key to place you in insert mode, but there are also other commands which can place you in insert mode so that you can start appending text. One of them is the "a" key, which lets you append text right at the spot where the cursor is placed, or the "A" key which lets you append text directly at the end of the line. You can also use the "o" key to open a line below the current line or "O" to open a line above the current line.

You can undo changes by using the "u" key and redo changes the using "<CTRL>r".

Deleting text in command mode goes with (but not only) the following commands:

- the "x" key deleted the current character; "3x" will delete the current character and the next two characters

- the "dd" command deletes the current line; "6dd" will delete the current line and the next five lines

- the "dG" command will deleted to the end of the file

- the "d$" command will delete to the end of the line

- the "d0" command will delete to the beginning of the line.

The commands based on "d" nut just delete text but also copy it to a paste buffer, which can be later recalled with the "p" command to paste the contents of the buffer after the cursor or the "P" command to paste the contents of the buffer before the cursor.

Cutting, copying and pasting text is done more traditionally with the "y" command (which stands for "yank", i.e. copy). To copy the current line you use "yy" ("6yy" means copy the current line and the next five lines). To copy to the end of the line use "y$" and to the beginning of the line use "y0". You can join lines with "J".

Searching and replacing is done within the line and within the whole file. Searching within the line is done with "f" from "find" (for example "fa" will move the cursor to the next occurence of "a"). You can also use the substitute command for a line (for example: while in command mode "s:caar/car<ENTER>" means substitute "caar" with "car", and replaces the first occurence of the searched word). The ":#,#s/big/small/g" replaces "big" with "small" within a range of lines (":" starts the command, "#" stands for a line number and "g" stands for "global").

To search the entire file use "/" followed by the searched word and the <ENTER> key. You can go to the next occurrence of the searched word with the "n" (from "next") command. To search and replace over the entire file, i.e. globally, you can use the command ":%s/you/YOU/g" (":" starts the command, "%s" find and substitute, "g" globally).

The search can also be done with options. For example entering ":set ic" will allow finding also combinations with capitals. To disable ignoring case enter ":set noic". You can also enable the "is" option (with the ":set is" command), which can be disabled with ":set nois".

To edit multiple files you can open them together from the beginning (for example "$ vim file1, file2, file3) or you open additional files after you started VIM with the command ":e additional\_file". The command "buffers" displays a list of files under editing. You can switch between files with the command ": buffer 1" for example to go to the first file listed by the ":buffers" command, or ":buffer 2" to go to the second file listed by the ":buffers" command. To copy content from one file to another you copy with yank, switch with buffer and paste. This works only if the files were opened together or with the ":e" command. Opening different VIM windows will not allow copying between files in this way. Instead, you have to use <SHIFT><CTRL><C> to copy from one file and <SHIFT><CTRL><V> to paste into another file. The <SHIFT><CTRL><V> command lets you paste text from all kind of sources (from documents, web sites, etc.). To insert an entire file into another you can use the ":r other\_file\_name\_to\_be\_inserted\_into\_the\_current" command, where "r" stands for "retrieving".

To select text to paste you can also use the visual mode by typing "v" in command mode and then move the cursor to the end of your selection. The selected text will be highlighted. Then press "y" to yank it and then "p" to paste it.

A useful debugging command is the matching parentheses command. You can find out if the corresponding parenthesis or bracket is missing with "%" by moving the cursor on the first parenthesis or bracket and typing "%".

To excute and external command, i.e. a command as if you were in the command line, with the ":! your\_command" command. For example the ":! ls <ENTER>" command will list the objects of the working directory.

Finally for getting help in VIM you type ":help <ENTER>". To close the help window use ":q". To find help on a specific command use ":help some\_command", for example type ":help user-manual" to get to the user manual. For an interactive tutorial for VIM you can also use the "VIM Tutor" initially created by Pierce and Ware, which can be started in the command line with the command "$ vimtutor <ENTER>". A short introduction in VIM can be also found in Shotts (2009), a lecture which gently introduces readers to the Linux command line in general.

3. Using SBT to create and manage Scala projects

In this section the reader will find out how to install SBT and use it to create and manage Scala projects. Furthermore, the issue of integrating external (managed) and internal (unmanaged) dependencies and of forking the Java Virtual Machine (JVM) will be discussed.

3.1 Installing SBT

You can use the Scala REPL either directly from within the command-line, initiating a REPL sesssion with the command "$ scala", or, by launching a SBT-session with the command "$ sbt" and then from within SBT launch the Scala REPL using the command "> console". On Ubuntu you can install the Scala language using the apt command ("$ sudo apt install scala") with no additional commands. Check the package management tool you use to get the same result. You can open the Scala REPL and type in some expressions; to exit type ":q" (VIM is useful after all). For guidance, check the book of Jason Swartz "Learning Scala" and the book of Mark Lewis "Introduction to Programming and Problem Solving Using Scala" for a great introduction to the Scala language.

Installing SBT requires a look at the SBT homepage (<http://www.scala-sbt.org/>) in order to get the four commands needed to install it using the command line. At the time the present document is written the commands for Ubuntu are:

$ echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt.list

$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823

$ sudo apt-get update

$ sudo apt-get install sbt

If you are using rpm distributions visit the SBT download page (<http://www.scala-sbt.org/download.html>) and look for the appropriate installing commands.

Next, open a Scala REPL session in SBT using the commands introduced at the beginning of this section and type in some expressions. You exit the console with ":q" and the command "> exit" gets you out of the SBT and back to the command-line.

3.2 Creating a Scala project

In this section we will create a Scala project called Scala\_Playground in order to continue the playground series of Linux introductions started by William Shotts (2009) in his book "The Linux Command Line".

Open the terminal with <CTRL-ALT-t>. It starts directly in your home directory. Next, create a directory called Scala\_Playground ("$ mkdir Scala\_Playground <ENTER>"). Check if the directory was created with the command "$ls <ENTER>" which lists the contents of the directory in which you are placed. Further, switch to the Scala\_Playground directory ("$ cd Scala\_Playground <ENTER>") and create a part of the inner structure of the current directory by adding another directory called src with its subdirectory main, with its subdirectory scala. This is done with the command "$ mkdir -p src/main/scala" which creates the entire chain of directories. Check it with "$ls" and "$cd" commands, return to the Scala\_Playground directory with "cd - ".

Next, in the Scala\_Playground directory, create a second directory called project. The ls command should print now for the Scala\_Playground two contents: project and src.

The final step in creating the Scala project is to create two files: one directly in the Scala\_Playground directory called build.sbt and the second in the Scala\_Playground/project directory called build.properties. Open VIM when you are in the Scala\_playground/project directory and type in:

sbt.version = 0.13.15

Go in the command mode with <ESC> and save the changes and give the open file a name: ":w build.properties". Exit vim with ":q".

The build.properties file sets the version of SBT.

Next, when you are in the Scala\_Playground directory open VIM and edit the basic settings of the Scala project: project name, Scala version used, and the dependencies you need. For the moment we will ignore dependencies and create the basic Scala project. Type in the file opened in VIM the following two lines:

name := "Scala\_Playground"

scalaVersion := "2.11.10"

The above two lines are not in Scala, they are in SBT's own language.

Now, get into the command line mode with <ESC> and save the changes and give the file the name build.sbt (":w build.sbt"). Exit VIM with ":q".

You should now have three objects in the Scala\_Playground directory: the build.sbt file, the project directory and the src directory.

You can now launch SBT while you are in the Scala\_Playground directory using the "$ sbt" command. You can see the SBT's output as it uses the information from the two edited files and creates the desired Scala project. When the output is finished you get to see the SBT prompt ">". The project was created successfully. Now, launch the Scala REPL with the command "> console". When the REPL is open you get to see the Scala prompt "scala>". You can type in some expressions to see if it works.

Type ":q" to get back to SBT and "exit" to get back to the command-line.

You can see that the Scala\_Playground directory has gained some additional directories: the Scala\_Playground/target directory and the Scala\_Playground/project/target directory. These are created by SBT when it creates the Scala project, and serve as locations for the output of SBT tasks (e.g. output following compilations or packaging actions).

In the src/main/scala directory you should place your Scala codes in form of scala files.

3.3 Some basic SBT tasks

Open SBT. When you see the SBT prompt ">" type in your first SBT command: "help". This generates an output which you can use to get further information about SBT and about the configuration of your project (command "about", command "settings"). The command "reload" reloads the project in the current directory, which is useful after you've changed the project settings (for example after adding dependencies) or after you've changed or added source files. It does the sanity check as it compiles the source files and outputs the errors encountered in the compilation process, accompanied by messages indicating possible reasons. It is very often helpful to try to read them in order to debug the code.

The command "tasks" is also listed in the help output. The SBT tasks show much of the functions SBT can take over. Type in "tasks" and check the default list of SBT tasks:

- starting the Scala REPL with the command "console"

- compiling source files with the command "compile"

- running an application with the command "run"

- testing the code executing all tests

- and many more.

For a detailed introduction in SBT you should check the SBT documentation at http://www.scala-sbt.org/documentation.html and read the book of Joshua Suereth and Matthew Farwell (2016) "SBT in action: the Simple Scala Build Tool".

3.4 Managing dependencies

Depending on the field in which a software developer is active, it is often necessary to use field-specific third-party Scala libraries which offer additional functionality. For data analysis, for example, the best known libraries are Spark (<http://spark.apache.org/>) which is a general engine for large-scale data processing (see Karau et al. 2015 for an introduction in Spark), ScalaNLP (<http://www.scalanlp.org/>) which is a suite of Machine Learning and numerical computing libraries, or Figaro (<https://www.cra.com/work/case-studies/figaro>) which is a probabilistic programming language for probabilistic modeling based on Scala (see Pfeffer 2016 for an introduction in Figaro).

For GUI programming, the best known libraries are Scala.swing (<http://www.scala-lang.org/api/2.9.1/scala/swing/package.html>) which is actually a Scala package but needs to be added to the dependencies list in order to be imported and used in Scala projects, and ScalaFX (<http://www.scalafx.org/>) which is based on Scala and sits on top of JavaFX (see Lewis 2015 for an introduction in GUI programming using Scala).

For many software developers, the main reason of using SBT is that it offers an easy management of dependencies from repositories. Also the packaging and integration of libraries in form of .jar files into the project is possible. Internal libraries in form of .jar files are called unmanaged dependencies and you should place them into the project's lib directory, which should be created inside the project's directory if you intend to use libraries in form of .jar files for a Scala project. External libraries are found on repositories and are called managed dependencies, as they are configured in the build definition and downloaded automatically from repositories (SBT Documentation 2017).

3.4.1 Using external Scala libraries

The first step towards integrating the functionality of these external libraries into our Scala code is to find out their repository coordinates for SBT use. One way to find out this piece of information is to search for them in the Maven Repository (<https://mvnrepository.com/>). The next step is to add the Maven dependency of the external library to the Scala project (Karau et al. 2015). This is done by including an additional setting called "libraryDependencies" in the project's buid.sbt file. This setting can take the form of a Sequence for including more than one external library dependency and in general looks like the following:

libraryDependencies ++= Seq(

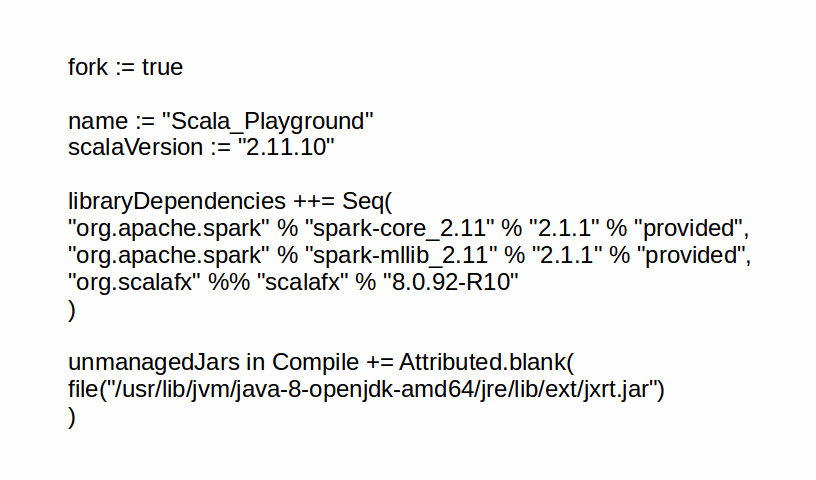
groupID %% artifactID % revision,

groupID %% artifactID % revision

)

where the exact library coordinates are found in the Maven Repository.

Figure 1 presents the contents of the build.sbt file of a Scala project called Scala\_Playground (created in section 3.2) which includes the managed dependencies for enabling the use of Spark's MLlib (Spark's library for Machine Learning) and of scalaFX (library for GUI programming) and, the unmanaged dependecy for enabling the use of javaFX package needed for GUI programming using scalaFX.

  
Fig 1: The contents of the build.sbt file of a Scala project with three external dependencies and one internal dependency (if more than one .jar file is needed, then they should be placed in the project's lib directory instead of specifying them in the build.sbt file).

3.4.2 Using internal Scala libraries

Sometimes it is necessary to use internal libraries in form of .jar files. Jar files are automatically created from compiled source files by setting the variable "exportJars" of the build.sbt file to true (exportJars := true). The process of .jar creation is called packaging. The newly created .jar file is placed in the project's target/scala-2.11 directory. If you want to make use of an internal library in form of a .jar file in your Scala code, it has to be added to classpath so that it becomes available during compilation, testing, running, and when using Scala REPL (SBT Documentation 2017). This can be achieved by overriding the settings variable called "unmanaged Jars" and indicating the path to the .jar file you want to use:

unmanagedJars in Compile := Attributed.blank(file("mypath/\*.jar")).

The above setting is equivalent to the use of "-cp" when starting the Scala interpreter from outside SBT ("$ scala - cp mypath/\*.jar ").

If more than one .jar file is used, then you should create a new directory called lib inside the Scala project's directory in which you can place the necessary jars. The lib directory will serve as common location and will make the jars available to the compiler and interpreter, all the jars added to it being placed on the project classpath (SBT Documentation 2017).

3.4.3 Using a forked Java Virtual Machine (JVM)

Running applications with the run task or using the REPL in SBT with the console task disposes the code to use the same JVM instance as SBT (SBT Documentation 2017). There are some situations though, when running user code in the same JVM as SBT causes problems, and demand that the code is run in a different instance of JVM. In SBT this is called forking the JVM (Suereth and Farwell 2016). You can enable forking by setting the variable called "fork" in your build.sbt file to true ("fork := true"). The situations requiring JVM forking appear either when the user code employs System.exit (which shuts down the JVM), or when it starts multiple threads (some of which may not terminate until JVM terminates) like in the case of creating a GUI (SBT Documentation 2017). You can read more on forking in Suereth and Farwell (2016) and in the SBT documentation (<http://www.scala-sbt.org/1.x/docs/Forking.html>). The build.sbt file in fig. 1 enables forking in order to ensure the save close of GUIs created by user code.

4. Conclusion

The advantages of learning Scala are manifold (Swartz 2016) and it is not the scope of this paper to cover them all. Though, from my personal point of view, the biggest advantage of Scala is that it allows you, through its functional nature, to express your ideas in a natural manner, and this is very important in developing code for statistical computation. This is why I believe Scala will allow researchers to advance their statistical skills by paying more attention to the pseudocode, and not to the details of how to implement it. This paper should offer a starting point for those willing to learn this functional programming language.

Biography:

Roxana Tesileanu graduated Forestry and Environmental Sciences (Dipl. Forstwirtin, the German equivalent to BSc plus MSc in Forestry) in 2008 at Albert-Ludwigs-Universitaet in Freiburg i. Br., Germany, worked as project team member in more international projects (at WSL and Agroscope in Switzerland), and since 2014 as research assistant at INCDS Romania. She is specialized in soil chemistry, environmental statistics and bioinformatics.

Bibliography:

1. Rafeeq Ur Rehman and Christopher Paul. The Linux Development Platform. Pearson, 2003.

2. Roxana Tesileanu. Introduction to Statistical Computing in Scala - an Implementation of the K-Nearest Neighbors Classifier. 2017.

https://www.researchgate.net/publication/319505548\_Introduction\_to\_Statistical\_Computing\_in\_Scala-an\_Implementation\_of\_the\_K-Nearest\_Neighbors\_classifier

3. Joshua Suereth and Matthew Farwell. SBT in action: the Simple Scala Build Tool. Manning Publications Co., Shelter Island. 2016

4. Michael C. Pierce and Robert K. Ware. VIM Tutor 1.7.

5. William E. Shotts. The Linux Command Line. 2009

6. Jason Swartz. Learning Scala. O'Reilly. 2009

7. Mark C. Lewis. Introduction to Programming and Problem Solving Using Scala. Trinity University. Texas. 2015

http://www.programmingusingscala.net/home/introduction-to-programming-and-problem-solving-using-scala

8. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. Learning Spark. O’Reilly,

Sebastopol, 2015.

9. Avi Pfeffer. Practical Probabilistic Programming. Manning Publications Co., Shelter Island. 2016

10. SBT Documentation. http://www.scala-sbt.org/documentation.html. 2017