

Scala Akka Spray RESTful “ATM”application

***Marta Doberschuetz***

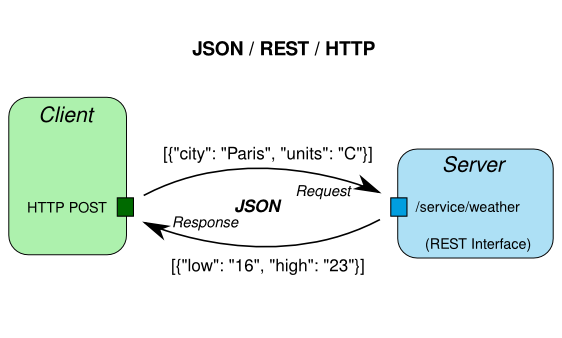
**T00171021**

Industry mentor: Billy Stack, Continuum Commerce

IT Tralee supervisor: Catherine Woods

# Introduction

The main objective of the project was to design an application using Scala programming language and a range of Scala-related technologies.

The application is a web service hosted on a local server (it could also be hosted on the cloud). A client calls to the web service by sending a HTTP request in a JSON format and the web service sends a HTTP request in a JSON format back. The call is made either with cURL or with Play! application. 

*Figure 1.1. Example of JSON/REST/HTTP model.*

*Source: http://safehammad.com/tag/json/*

Project goals were:

* to get familiar with Scala programming language
* to get familiar with web service concept and a broad range of technologies
* to get familiar with Actor paradigm.

The important thing about the project was that its goal was not to learn the language or be an expert in any of the mentioned technologies. The purpose of the project was to get to know the concepts of web services and technologies used to develop web services.

The technologies used in the development of the project were: REST, Spray, Akka, Slick, curl, Play!, Heroku, Github.

# Scala

### 2.1. An object-oriented and functional language

In the past few years the world of technology has been developing rapidly. Programmers have been facing new challenges “for high-level domain modelling, rapid development, and, more recently, parallelism and concurrency.” [[1]](#footnote-1) Martin Odersky, a German computer scientist, began to work on a language that would meet those challenges. He experimented with design of a language that would combine the concepts of the traditional object-oriented model and functional programming. The result of the research was Scala, created in 2003.[[2]](#footnote-2)

The name “Scala” stands for “scalability”. The term scalability means for the network, process or system to be able to manage when the demands of the users grow and effectively grow with those demands.[[3]](#footnote-3)

Scala is a combination of two programming paradigms: object-oriented and functional.

Object-oriented programming uses objects to describe the current world state. Objects are described by data fields and the behaviour of the objects is described by methods. Objects then, as instances of classes, interact with one another to create applications and programs. In Scala, the value of a function is an object. [[4]](#footnote-4)

In object-oriented programming, objects structure programs. They are containers for both data and operations performed on that data. Objects are also values themselves and can be stored in other objects or passed as parameters in operations. [[5]](#footnote-5) Although object-oriented languages are based on objects, they also have such thing as primitive data types (like in Java), that are not objects and are used to store values, or some methods do not belong to any objects. All of that seems to limit scalability. Whereas in Scala there is no such limitation as Scala is pure object-oriented language in the sense that in Scala everything is an object: every value, every operation. [[6]](#footnote-6) Let’s analyse the following example:

**object** Example2 {

**def** main(args: Array[String]): Unit = {

println(1+2)

//is same as:

println((1).+(2))

}

}

The above is not just a simple operation of addition. In fact, when you perform a calculation like: 1+2 in Scala, what you actually do is you invoke a method called “+” on an object which value is “1” and you pass a parameter of value “2” to that method. [[7]](#footnote-7)

Functional programming uses functions to perform calculations. It does not use mutable variables, assignments, loops and other imperative control structures. In functional programming functions can act like values and those values can be produced, consumed and composed. Additionally functions can be defined in other functions, they can be passed as parameters and returned as results. Scala treats all functions as first-class values, which means they have the same status as primitive data types or String objects. Functions can be passed as arguments to other functions, functions can be returned as a result or stored in variables. A function can be defined inside another function same way an integer variable can be defined inside a function. [[8]](#footnote-8) “Functions that are first-class values provide a convenient means for abstracting over operations and creating new control structures. This generalization of functions provides great expressiveness, which often leads to very legible and concise programs. It also plays an important role for scalability.” [[9]](#footnote-9)

“Immutable data structures are one of cornerstones of functional programming.” [[10]](#footnote-10) In Java strings are objects of class String, they are immutable, which means that when you create an object String and then you want to change that object, what you in fact do, is you create another object. If we just look at strings in Java we can say that Java is a functional language. In this case Scala is like Java – it treats strings in a mathematical sense and not as an array of characters – Scala does not allow mutability. Scala also introduces immutable lists, tuples, maps and sets.

Another characteristic of a functional language is the fact that its methods should have no side effects. If a function or expression has a side effect it means that not only does it return a value, it also modifies state or it interacts with the outside world. Functional languages encourage methods with no side effect, Scala gives programmers a choice: they can either write in an imperative style (with mutable data and side effects) or they can avoid it as Scala makes it easy to do so.

Methods with no side effects are called: referentially transparent, which means that a method call can be replaced by its result and the program’s semantics will not be affected. [[11]](#footnote-11) In fact, if an expression in Scala has no side effects it can be evaluated using a method called a “substitution model”, which idea is to reduce an expression to a value.

There is one more advantage to the fact that Scala combines functional and object-oriented concepts. The combination of the two programming styles also makes the code more concise. Let’s look at the two code snippets to visualise it:

Class Point in Java:

//Java:

**public** **class** Point {

**private** **final** **double** x, y;

**public** Point(**final** **double** X, **final** **double** Y) {

x = X;

y = Y;

}

**public** **double** getX() {

**return** x;

}

**public** **double** getY() {

**return** y;

}

**public** Point(

**final** **double** X, **final** **double** Y,

**final** **boolean** ADD2GRID

) {

**this**(X, Y);

**if** (ADD2GRID)

*grid*.add(**this**);

}

**public** Point() {

**this**(0.0, 0.0);

}

**double** distanceToPoint(**final** Point OTHER) {

**return** *distanceBetweenPoints*(x, y,

OTHER.x, OTHER.y);

}

**private** **static** Grid *grid* = **new** Grid();

**static** **double** distanceBetweenPoints(

**final** **double** X1, **final** **double** Y1,

**final** **double** X2, **final** **double** Y2

) {

**double** xDist = X1 - X2;

**double** yDist = Y1 - Y2;

**return** Math.*sqrt*(xDist\*xDist + yDist\*yDist);

}

}[[12]](#footnote-12)

Same class in Scala:

//Scala

**class** Point(

**val** x: Double, **val** y: Double,

addToGrid: Boolean = **false**

) {

**import** Point.\_

**if** (addToGrid)

grid.add(**this**)

**def** **this**() {

**this**(0.0, 0.0)

}

**def** distanceToPoint(other: Point) =

distanceBetweenPoints(x, y, other.x, other.y)

}

**object** Point {

**private** **val** grid = **new** Grid()

**def** distanceBetweenPoints(x1: Double, y1: Double,

x2: Double, y2: Double) = {

**val** xDist = x1 - x2

**val** yDist = y1 - y2

math.sqrt(xDist\*xDist + yDist\*yDist)

}

}[[13]](#footnote-13)

## 2.2. Comparison of Scala and Java

Scala runs on a Java platform and uses Java libraries. “Scala programs compile to JVM bytecodes.”[[14]](#footnote-14) When writing programs in Scala one can avail of the richness that Java provides: libraries, methods, classes, fields, interfaces. In fact, Scala has “borrowed” a lot of the Java types: an integer in Java is an Integer in Scala, and on top of that, Scala has introduced some additional methods connected to those types. For example, there is a method called: toInt in Scala that parses a string to an integer. So instead of:

String someNumber = "12";

**int** number = Integer.*parseInt*(someNumber);

is what we would have written in Java, in Scala we simply write:

**def** someNumber = "12"

someNumber.toInt.

Another advantage to Scala programming language is that its code doesn’t take up as many lines. On average Scala program contains half the amount of lines of what the same Java program would. And it is not just about less typing, which means less reading and trying to understand code, but also less errors. As an example the code in Java would look like that:

**public** **class** MyClass {

**private** **int** index;

**private** String name;

**public** MyClass(**int** index, String name){

**this**.index = index;

**this**.name = name;

}

}

,[[15]](#footnote-15) whereas the code in Scala would look like that:

**class** MyClass(index: Integer, name: String) {

}

The code in Scala is obviously more concise, therefore easier to read and understand. Also it gives less space for potential errors.

Semicolons are optional in Scala. Many experienced Java programmers would say they would prefer to know exactly where the statement ends. The rules for semicolon inference are actually simple. There is no need to put a semicolon at the end of the line. The end of the line is the end of a statement. If the statement consists of several lines, you can either wrap them in braces:

(x

+ )

or leave the + sign at the end of the line and the compiler will know the next line also belongs to the same statement:

x +

y.

Scala is statistically typed. A computer program consists of variables, expressions, functions that are of a certain type. There are interfaces defined between different parts of the computer program and the main idea behind a types system is to eliminate bugs. The parts of the computer program are connected and the compiler can check if those parts are connected correctly at either compile time or run time. When the checking for bugs is done at compile time we say it happens statically, when it happens at run time we say it happens dynamically.[[16]](#footnote-16)

We talk about compile time when we are in the process of building our program and we talk about run time when we actually execute our program. So in static type system the compiler will inform us of an error just when we are building a program, so since Java is statically typed, when we enter something like this:

String s = 123;

the error message will read: Type mismatch, cannot convert from int to String.

In dynamic programs, the programmer does not indicate type, like in Javascript:

var s = 123

s = "abc".

And when we perform the following calculation:

var x = s \* 10

an error will occur at run time, not at compile time.

Scala is statically typed, like Java, but Scala has a system of type that are “inferred”, so Scala has the ability to “guess” what type the variable is without the programmer having to indicate the type. So when we type:

**val** s = "abc"

Scala will figure out that s is in fact a String.

Of course it is correct to type in:

**val** s:String = "abc",

it may even increase readability or when it comes to numbers, sometimes we define a number that is of floating type, although we don’t actually give any floating point.

In the case below an error will occur:

**val** z:Int = "abc".

So because Scala has this inferred type it behaves a bit like a dynamic language and dynamic typed system enthusiasts claim the speed of writing code to be one of its big advantages. So Scala is statically typed programming language with dynamic elements.

Scala differentiate between two types of variables: val and var. A val is similar to final in Java:

**val** s:String = "abc".

Once initialized it cannot be reassigned:

,

the error reads:



and the code will not compile,

whereas var is like a non-final variable in Java and can be reassigned:

.

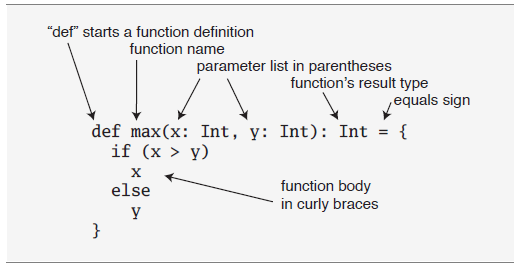
Types in Scala can be inferred. It means that you don’t always have to specify what type the variable is and the compiler will infer it, guess it. The following statement:

**var** bla:String = "bla"

is the same as:

**var** bla = "bla".

Functions in Scala have the following structure:

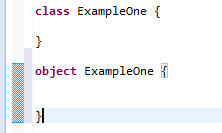


*Figure 2.1. Function structure in Scala.*

*Source: Odersky, Spoon, Venners, 2010*

The return type is optional and if not indicated a function will return the last value computed by the compiler. The only place where the type has to be specified is the parameters. It is also worth mentioning that the default access modifier in Scala is public.

Scala cannot have static members. Scala has singleton objects instead of static members. The object definition looks the same as class definition:

.

If the name of the object has the same name as the class, it is called a companion object. They both have to be defined in the same source file. The advantage here is that companion object and its companion class can access their private members. “If you are a Java programmer, one way to think of singleton objects is as the home for any static methods you might have written in Java.”[[17]](#footnote-17) Those static methods can be invoked the similar way we invoke them in Java – using the name of the object followed by the dot and the name of the method. Singleton objects cannot be instantiated, therefore they cannot take parameters. Companion classes can take parameters.

If an object does not have the same name as the class it is called a standalone object. To run a Scala application a standalone object has to have a main method, the same way to run a Java application you have to a main method defined. The signature of a main method is the following:

**object** Summer {

**def** main(args: Array[String]): Unit = {

}

}.

There is another way of running a singleton object. You can use the Application trait. What you do is you simply put: extends Application, after the name of the object and you don’t have to type the main method definition after that:

**object** Summer **extends** Application{

}

and then you put the body of the application as normal. Saves typing.

In Java the file name has to be exactly the same as the class name. In Scala it doesn’t.

In Java we can define a class that has a method and then we can write another class that extends the first class and therefore inherits all the methods of that first class. In Scala we have traits. A trait can be “mixed in” to a class. We say that we “mix in” a trait, we do not inherit it. [[18]](#footnote-18)

# Building a RESTful web service with Scala

## 3.1. Actor paradigm

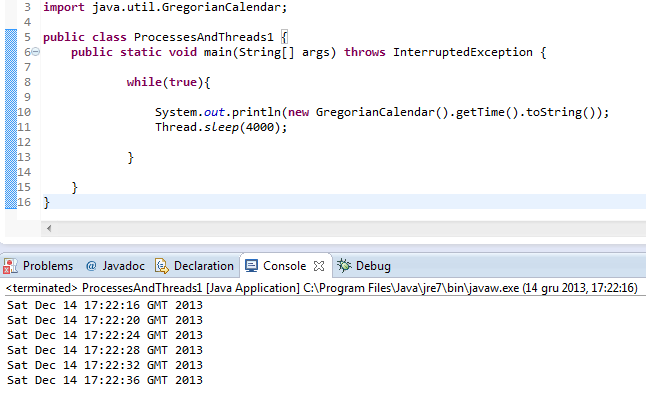
### Multithreaded applications in Java

When we work on our computer we take for granted that our computer runs a few programs at the same time: we can type in a document while our computer is downloading a movie from the Internet or we watch a clip on youtube and at the same time our computer has to make sure that the time on the clock on the taskbar changes accordingly every minute. We talk about concurrency when multiple threads or processes are being executed independently, in parallel.

Java provides a support for multithreaded applications.

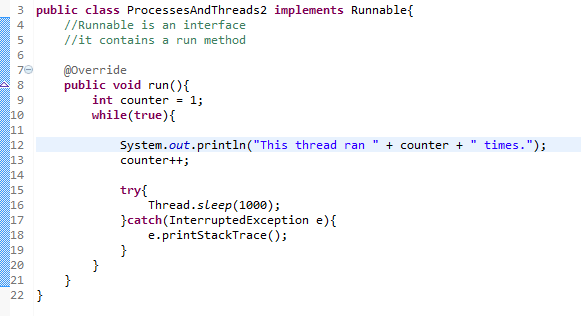
A process is “an instance of a computer program that is being executed”. [[19]](#footnote-19) A process starts when the program is initiated. For example, when you click on the Eclipse icon, a process that runs Eclipse starts. A program, such as Eclipse, contains several instructions, so the process executes those instructions, but when a single instruction is being executed we refer to it as a thread. So a thread is “an execution of the smallest sequence of programmed instructions”. One process can contain multiple threads. The execution of those threads is managed by an operating system scheduler. When multiple threads are contained in one process they share resources, processes do not share resources (resources like memory). Threads also share the code of the process they are in. They also share the process’s values that the process’s variables reference. Multithreading is a model in computer science that allows for multiple threads within one process to be executed independently at the same time. The concept is crucial in modern applications. Imagine an application that has to wait to finish one calculation before performing another one, it would appear to the user as frozen.

Single-threaded Java example:

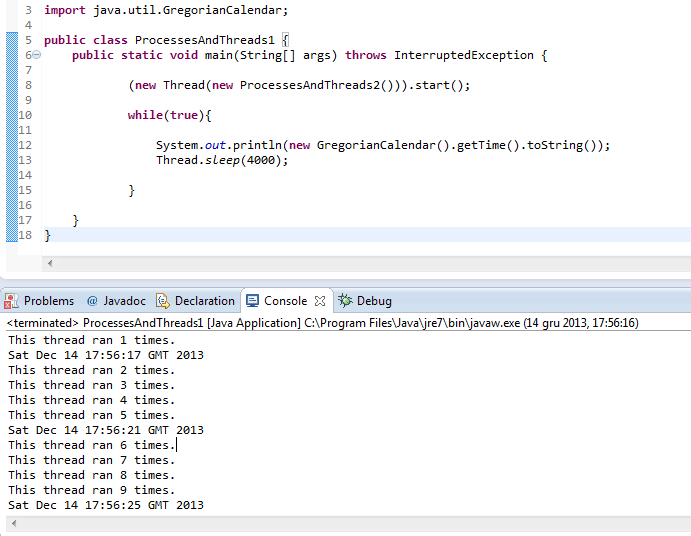


The above application executes an infinite while loop. The body of the loop consists of a print statement that prints date and time, but since the thread is stalled, it prints the date and time every 4 seconds. It is a single threaded application as only one instruction is being executed at a time.

Two-threaded Java application example:



We have created another class called ProcessesAndThreads2 and we created another thread within that class. In order to do that we had to implement an interface called Runnable, which contains a method called run. That method is called whenever we run that thread.



In the above screenshot we can see that we added a call to class ProcessesAndThreads2 in order to be able to access the thread in that class and then we start the thread with start() method. The thread from the ProcessesAndThreads2 class is called a background thread and the thread in the main class is called the foreground thread. Note that the background thread sleeps every 1000 milliseconds so 1 second. Thanks to the while loop created in class ProcessesAndThreads2 we can see with what frequency the threads are being executed.

Java’s built-in multi-threaded model introduces locks. Without locks shared data would be accessed by multiple threads and therefore create chaos: data would be changed by one thread while being read by another at the same time, there would be no way of keeping track of the changes. A situation where two different threads access the same data and update the value of that data without locking is called a race condition. You can imagine a bank account system where account balance is accessed by two separate threads at a time and made changes. Locks enable for only one thread to access shared data at any given time so there is no fear of invalid data value. The problem with Java’s support for concurrency is that programmers coding large application often have to think what data will be accessed and modified where by certain threads and what locks will have to be held on data. With each method call there is a possibility of deadlock. “A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.”[[20]](#footnote-20) In the case of Java concurrency the “actions” mentioned above are simply threads. So the big problem with Java’s shared data and locks are race conditions and deadlocks. Scala seems to be able to deal with that problem way better.

### Multithreaded applications in Scala

Scala has gone a step further. Scala uses actors. Actors provide concurrency model. You could compare actors to threads in Java. Actors in Scala have mailboxes that receive messages.

Here is an example of an application that uses an actor:

**import** scala.actors.\_

**object** SillyActor **extends** Actor{

**def** act(){

**for** (i <- 1 to 5){

println("I am an actor")

Thread.sleep(1000)

}

}

SillyActor.start

}[[21]](#footnote-21)

Note that a package scala.actors has been imported and that the method act() was overridden. In the above case the actor only prints a message, it doesn’t use any mailboxes. Here is another example of an actor:

**object** SeriousActor **extends** Actor **with** Application {

**def** act(){

**for** (i <- 1 to 5){

println("I am also and actor!")

Thread.sleep(1000)

}

}

SeriousActor.start

}[[22]](#footnote-22)

The two actors above run independently from each other.

There is another way of defining actors – by using a utility method called “actor” in object scala.actors.Actor:

**val** seriousActor2 = actor {

**for** (i <1 to 5)

println("That is the question.")

Thread.sleep(1000)

}[[23]](#footnote-23)

Actors can communicate without using shared memory and locks. They send messages to each other by using the “!” method. In the example below the actor waits for a message in its mailbox. The actor then prints out whatever message it receives, he does so by calling “receive”:

**val** echoActor = actor {

**while** (**true**) {

receive {

**case** msg =>

println("received message: "+ msg)

}

}

}[[24]](#footnote-24)

Sending or receiving messages does not stop the actor. “The sent message waits in the receiving actor’s mailbox until the actor calls receive.” [[25]](#footnote-25)

The actor will only respond to messages that match one of the case statements. For example:

**val** intActor = actor {

receive {

**case** x: Int => // I only want Ints

println("Got an Int: "+ x)

}

}[[26]](#footnote-26)

In this case the actor will only accept integer values so if a message is sent and it contains anything but an integer the actor will silently ignore it and will only respond to integers.

## 3.2. Technologies

While developing the application several technologies were used.

### Akka

Akka is a toolkit that allows to create concurrent applications. Akka guarantees thread safety as it uses the Actor paradigm.

### REST

A web service is “a method of communications between two electronic devices over the World Wide Web”. [[27]](#footnote-27) In layman’s terms “a web service is a function that can be accessed by other programs over the web (HTTP). (…) A web service is not targeted at humans but rather at other programs.” If we were to create a web service that adds two numbers, it would be given a URL address and it would have to be in a format (like XML or JSON) that would be understood by other programs. [[28]](#footnote-28)

There are two types of web services:

* “big” web services that follow the SOAP standard
* RESTful web services[[29]](#footnote-29).

SOAP (Simple Object Access Protocol) is a type of a web service that brings its own protocol specification. The purpose of SOAP protocol is to exchange structured information when implementing web services in computer networks. SOAP uses XML as message format and HTTP or SMTP for message transmission. [[30]](#footnote-30)

REST (Representational state transfer) is “an architectural style for distributed hypermedia systems.”[[31]](#footnote-31). “REST is almost always the right answer”. [[32]](#footnote-32) REST is said to be easier when it comes to creating clients, developing APIs, understanding documentation. SOAP only allows XML, whereas REST allows other data formats (like JSON). Some experts argue that SOAP is more secure as it supports not only SSL, but also WS-Security, WS-AtomicTransaction and WS-ReliableMessaging so SOAP is unreplaceable when it we deal with programs that demand high-security standards such as a banking system. [[33]](#footnote-33)

### Spray

Spray is a Scala framework that allows to build the REST/HTTP layer that is responsible for handling HTTP requests. REST service is running inside an Akka actor.

### Slick

Slick is a Functional Relational Mapping for Scala that provides a way of working with Scala’s relational databases.

### cURL

cURL is a command-line tool for transferring data using different protocols, one of them is HTTP.

### GitHub

GitHub is a web-based hosting service that provides Git revision control system for programmers.

### Play! framework

Play! is a web application framework.

## 3.3. Building the application

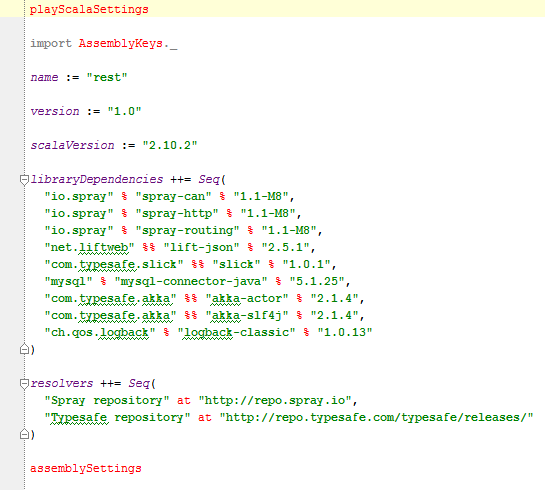
While building the web service I have used Oleg Yermolaiev’s code available on his GitHub account: https://github.com/oermolaev/simple-scala-rest-example. I have modified his application by adding two new functions: withdraw and deposit to mimic a behaviour of an ATM machine.

The application built is performed in several steps.

### 3.3.1. Application configuration files

#### build.sbt

You start building the application by creating the build configuration files. You create a file called build.sbt and you place it in the root directory of your project. The file specifies application name, its version, target version of Scala. It also defines dependencies (libraries like Akka, Spray, Slick).

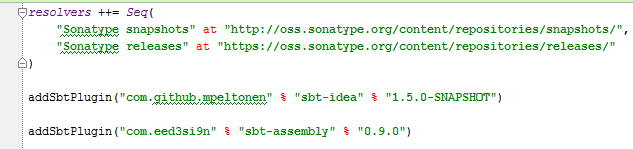


*Figure 3.1. build.sbt file*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

#### plugins.sbt

Project build definition can be extended by using plugins. Example of a plugin can be: *sbt-idea* – it allows to build IntelliJ IDEA project files.

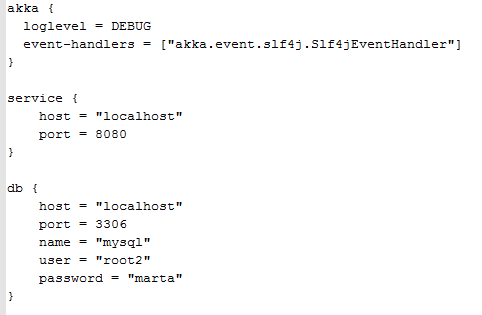


*Figure 3.2. plugins.sbt*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

#### application.conf

File called application.conf should be placed in */src/main/resources*. It contains application settings. They are grouped into service-related settings and database settings. Database setting are database host, port number, name of the database, username and password.

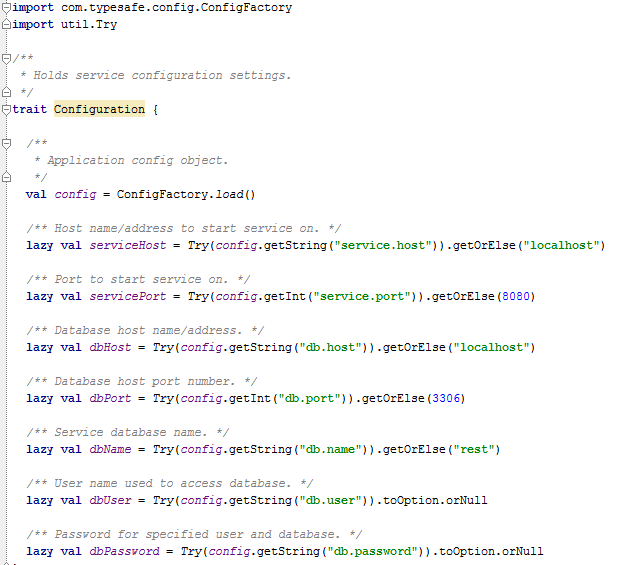


*Figure 3.3. application.conf*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

#### Configuration.scala

When the application is run all configuration settings are retrieved from configuration.conf file and loaded into the Configuration.scala trait.

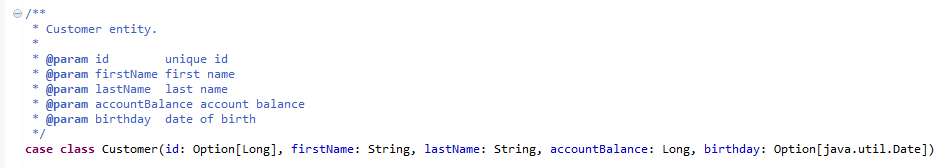
****

*Figure 3.4. Configuration.scala*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

#### Customer.scala

File customer.scala defines Customer entity :



*Figure 3.5.Customer.scala: case class Customer*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

Customer has an id, firstName, lastName, accountBalance and birthday. The accountBalance attribute has been added so that the application could serve as an ATM program where Customer can make withdraws and deposits.

File Customer.scala also creates table customers by using Slick’s Table object. Table is of type Customer:

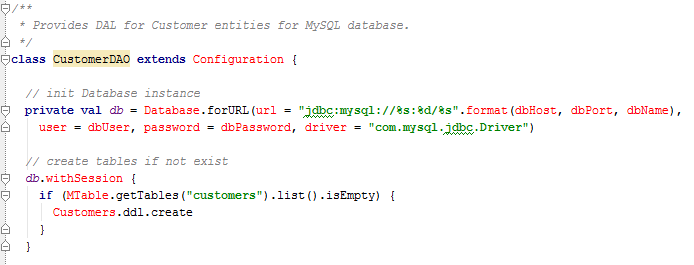


*Figure 3.6.Customer.scala: object Customers*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

#### CustomerDAO.scala

Customer Data Access Object file specifies how to connect to the database, it creates table customers.

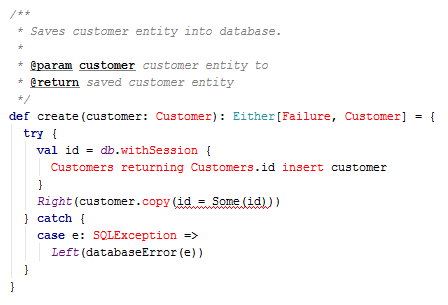


*Figure 3.7.CustomerDAO.scala: database connection string, method for customers table creation*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

The file also defines several other functions responsible for interaction with the database: create, update, delete, get, withdraw and deposit.

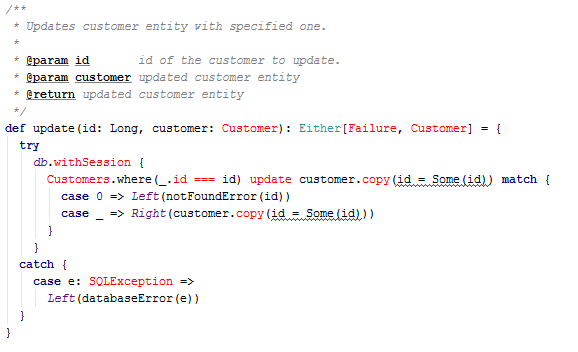
Function *create* saves customer’s entity into the database. It takes customer’s entity as a parameter and returns saved customer’s entity.



*Figure 3.9.CustomerDAO.scala: function create*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

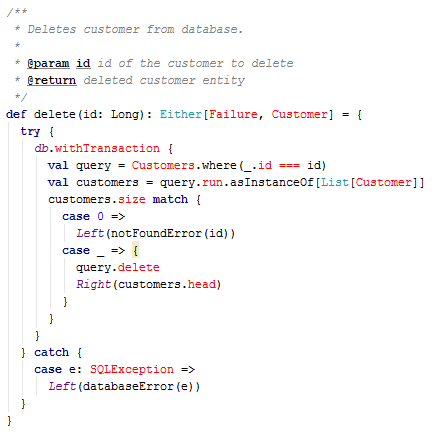
Function *update* updates customer’s entity with specified values, it takes customer’s id and updated customer’s entity as parameters and it returns updated customer’s entity.



*Figure 3.10. CustomerDAO.scala: function update*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

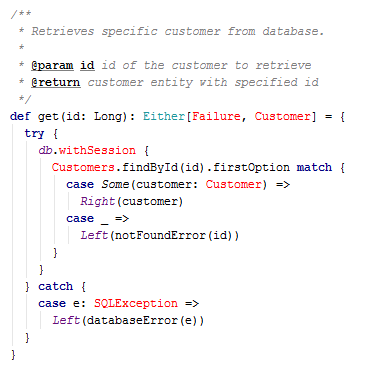
Function *delete* deletes customer from the database. It takes customer’s id as a parameter and it returns deleted customer’s entity.



*Figure 3.11. CustomerDAO.scala: function delete*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

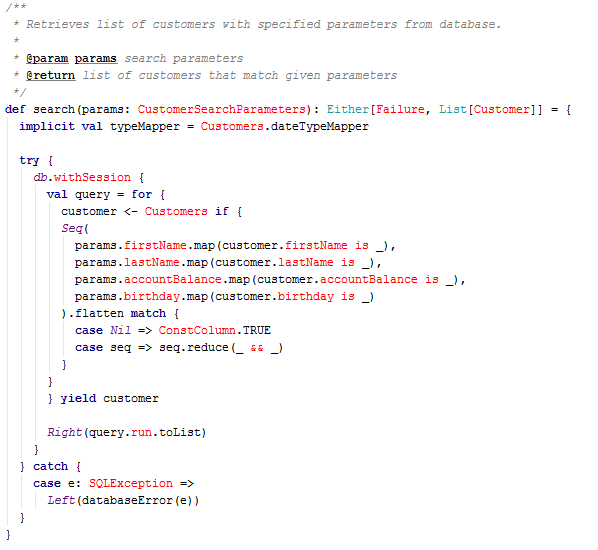
Function *get* retrieves a specified cutomer from the database. The customer is specified by id number. The function takes customer’s id number as a parameter and returns customer’s entity with specified id.



*Figure 3.12. CustomerDAO.scala: function get*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

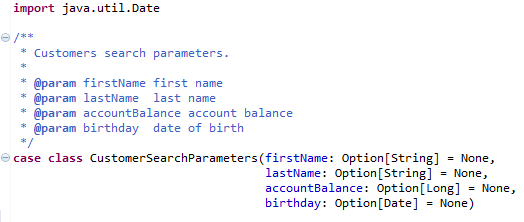
Function *search* retrieves a list of customers with specified parameters from the database.



*Figure 3.13. CustomerDAO.scala: function search*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

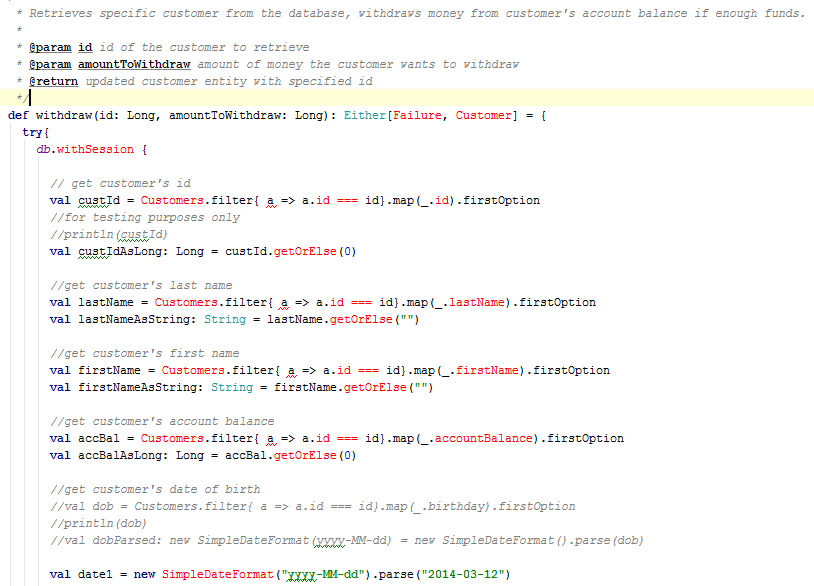
Function *search* takes *params* as a parameter. *Params* is of type *CustomerSearchParameters*. *CustomerSearchParameters.scala* file specifies parameters with which a customer entity can be searched with.



*Figure 3.14. CustomerSearchParameters.scala*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

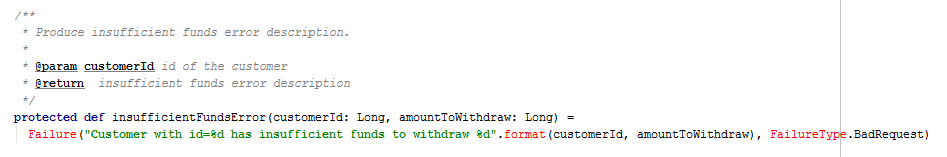
Function *withdraw* allows to withdraw money from the account of a specified customer retrieved from the database. The function takes two parameters: customer’s id number and the amount that the customer wants to withdraw. The return type is Either which in Scala “represents a value of one of two possible types (a disjoint union.) Instances of Either are either an instance of [Left](http://www.scala-lang.org/api/2.9.3/scala/Left.html) or [Right](http://www.scala-lang.org/api/2.9.3/scala/Right.html).” By convention Left indicates an error and Right indicates success. In the case of the *withdraw* function Left returns a Failure, which is a specific error like “bad request” or “database error”. In the body of the function we retrieve all the fields for the specified customer. We retrieve the id number, customer’s first name, customer’s last name, customer’s account balance and birthday.



*Figure 3.15.Function withdraw: part 1*

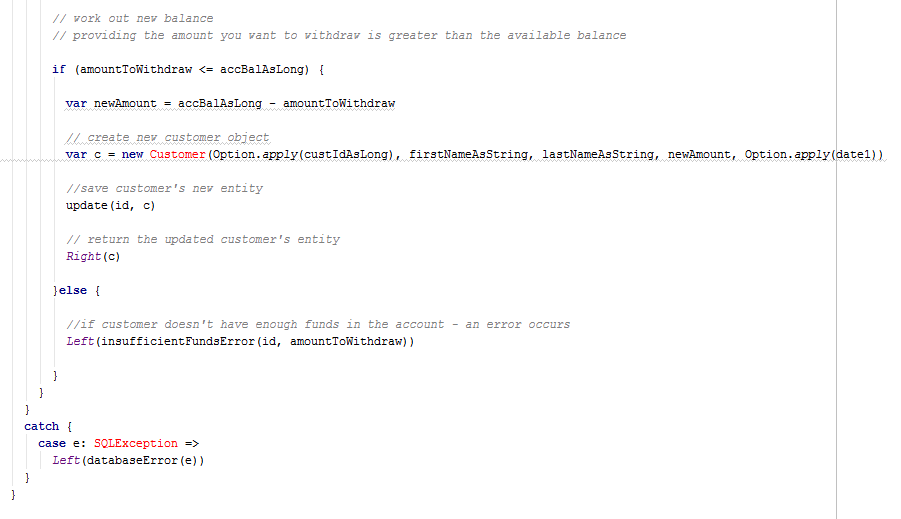
*Source:*

Next we work out the new balance in customer’s account. If the amount that the customer wants to withdraw is less than the funds available an “insufficient funds error” with be thrown, otherwise the new balance is updated and new updated customer’s entity is returned.



*Figure 3.16.Insufficient funds error*

*Source:*



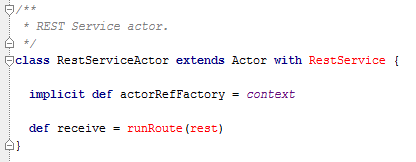
*Figure 3.17.Function withdraw: part 2*

*Source:*

Function *deposit* looks identical to the *withdraw* function, except it doesn’t have the if statement as the customer can deposit any amount they want, and of course, the new balance is an addition of the old balance and the amount that the customer wants to deposit.

#### RestServiceActor.scala

RestServiceActor.scala defines the REST/HTTP layer that is responsible for handling HTTP requests. It is built with Spray framework. The REST service is running inside an Akka Actor:



*Figure 3.18.RestServiceActor.scala*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

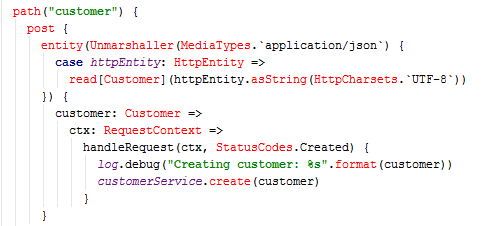
Route is also defined in the same file:



*Figure 3.19.Route definition in RestServiceActor.scala*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

Route definition sets the respond media type to JSON. Route for POST looks like that:

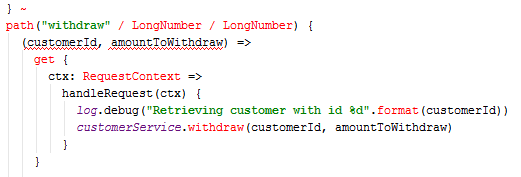


*Figure 3.20.Route for POST in RestServiceActor.scala*

*Source: https://github.com/oermolaev/simple-scala-rest-example*

We notice that the route calls *create* function defined in CustomerDAO.scala file, which simply creates a new customer entity.

The route for GET /withdraw/<id>/<amountToWithdraw> looks like that:



*Figure 3.21.Route for GET* /withdraw/<id>/<amountToWithdraw> *in RestServiceActor.scala*

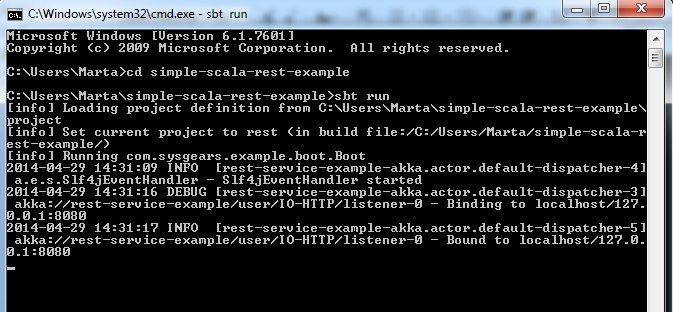
*Source: https://github.com/oermolaev/simple-scala-rest-example*

We notice that the route calls *withdraw* function defined in CustomerDAO.scala file.

# Calling to the application

## 4.1. Running the application

We run the application by navigating to the route directory of the application and executing the *sbt run* command:



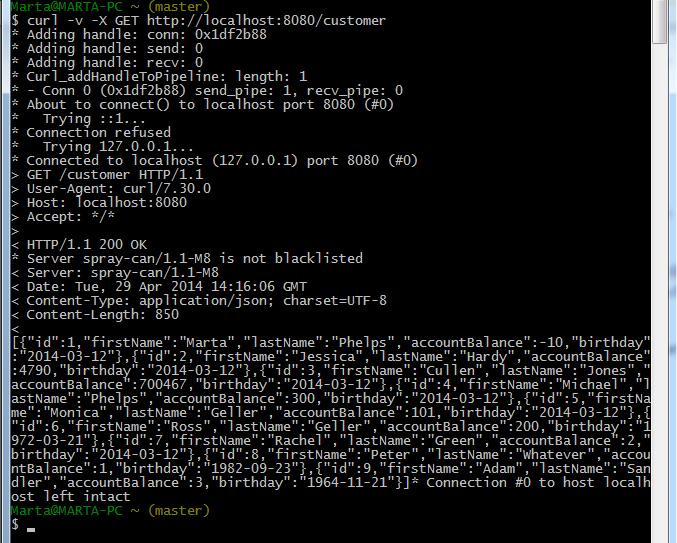
*Figure 4.1.Running the application*

The application runs on port 8080 on localhost.

## 4.2. Calling to the application with cURL

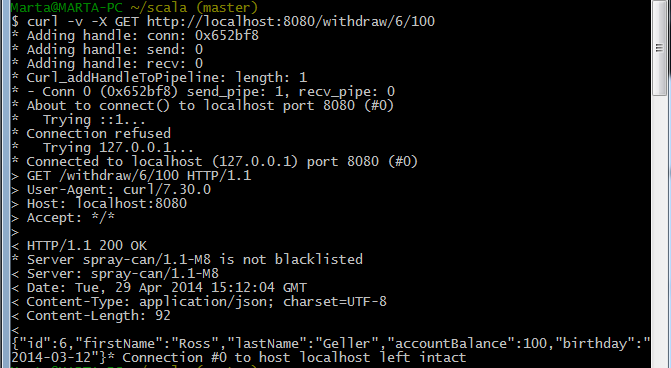
Once the application is running we can call to it with cURL since we don’t have the UI built yet. We do so by executing a cURL command inside bash. The following command will list all available customers:

curl -v -X GET http://localhost:8080/customer



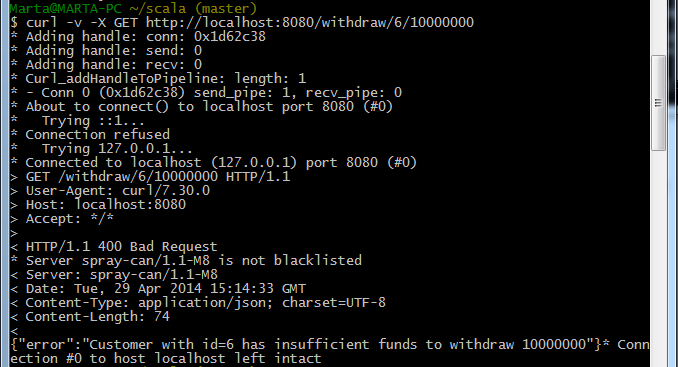
*Figure 4.2.Calling to the application with a cURL command.*

The following command will withdraw 100 from the account of customer with id 6:



*Figure 4.3. Calling to the application with a cURL command.*

The following command will return an error as customer with id 6 has only 100 in his account.



*Figure 4.4. Calling to the application with a cURL command.*

We notice that the HTTP request in in JSON format and the response is also in JSON format. The response also returns response status like the previous example “400 bad request”:

 or in the example before that “200 ok”:

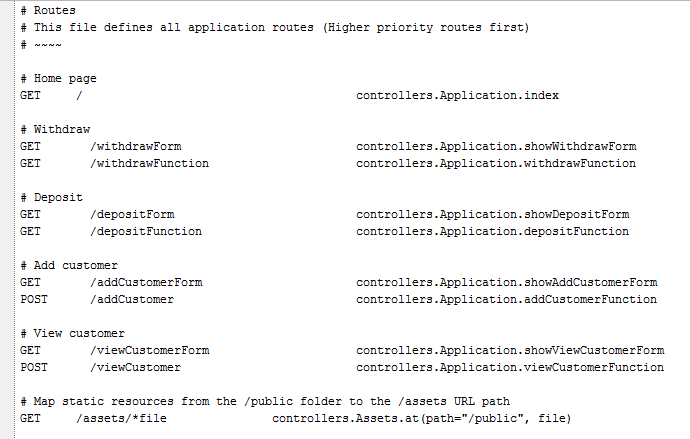
.

## 4.3. Calling to the application with Play!

cURL can be “replaced” with a UI designed with Play! framework. Play provides templates for designing applications. The main entry point for the application is file conf/routes. The file defines all accessible URLs. In the case of my Play application the route:

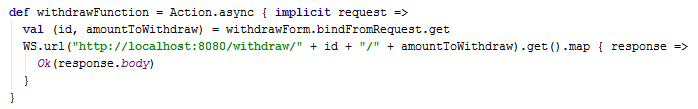


means that when the web server receives a GET request for the /withdrawFunction path it is to retrieve the Action to execute from the controllers.Application.withdrawFunction.



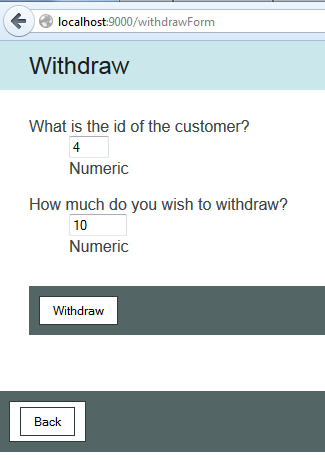
*Figure 4.5. conf/routes file in the Play application.*

The Action for withdrawFunction is specified in the controllers/Application file.

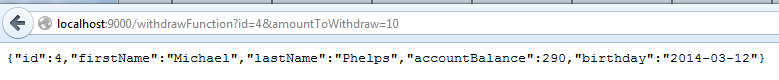


*Figure 4.6.withdrawFunction in the controllers/Application file .*

The withdrawFunction returns an Action that handles form submission. It takes values entered in the textfields and passes them as id and amountToWithdraw to the path. It sends a GET request with the specified path and the parameters entered by the user and sends a request back and displays the body of the request in the browser.

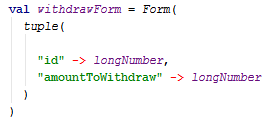


*Figure 4.7.Withdraw form submission.*



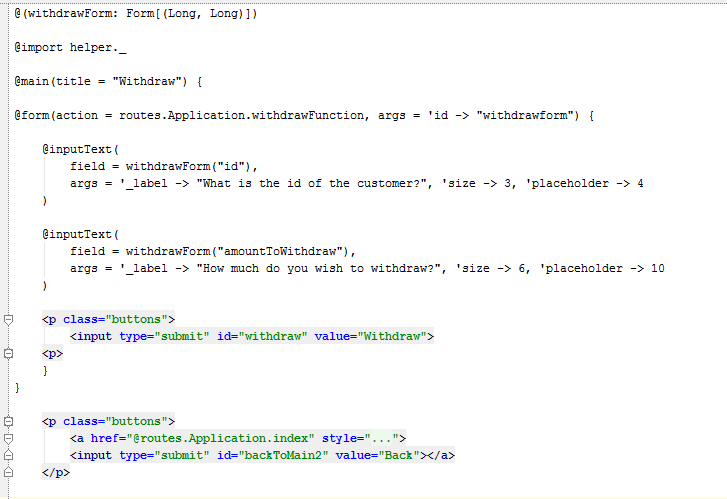
*Figure 4.8.withdrawFunction response.*

All forms are also defines in the controllers/Application file. To define a form you use Play’s *Form* object.



*Figure 4.9.withdrawForm definition.*

This Play application uses several web pages to show different forms.



*Figure 4.10.withdrawForm Scala HTML page.*

In the case of withdrawForm we define two text fields: one for id number and one for the amount to withdraw. We can define their sizes, and give them placeholders. We also specify a button that will execute form submission. We notice that the form has an action assigned. The action is withdrawFunction. We can see how elements of the Play application are all connected.

# Deploying a Scala application to Heroku

My last task in implementing the Scala Spray Akka RESTful “ATM” application was to deploy the Spray application to Heroku and then get the Play application to call to it. Unfortunately I have not been successful. I assume the database setting of the Spray application differ from the ones required by Heroku. I have been, tough, able to deploy a sample application available in Heroku’s tutorial “Getting started with Scala on Heroku” (Source: https://devcenter.heroku.com/articles/getting-started-with-scala).

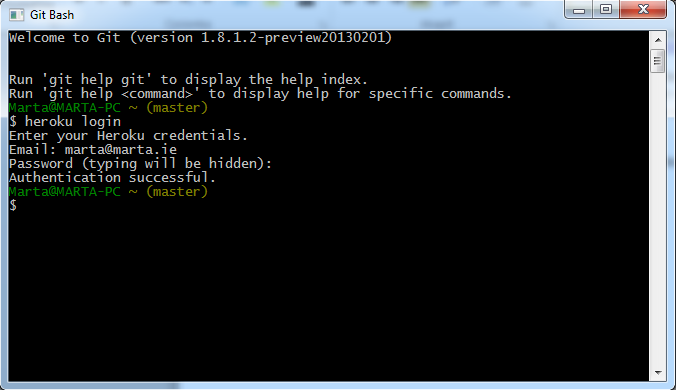
Heroku is a PaaS – platform as a service. It means that it provides a computing platform and a solution stack as a service. “A solution stack is a set of [software](http://en.wikipedia.org/wiki/Software) subsystems or components needed to perform a task without further external dependencies.” (Wikipedia). An example could be: an operating system, web server, database and a programming language when developing a web application.

Heroku allows to build, deploy and manage applications. It supports several programming languages, Scala being one of them.

This is how to I deployed a Scala application to Heroku:

There are several steps to complete. First thing I had to do was to install a Heroku Toolbelt. Heroku Toolbelt allows to start using Heroku. It consists of Heroku client – Heroku command-line interface tool for creating and managing Heroku apps, Foreman – a tool for running apps locally, Git – revision control system that allows to push apps to Heroku.

Once I installed Heroku toolbelt I created a Heroku account to be able to log in to Heroku in the Git bash shell by typing: **heroku login** command:



Next I created an application source folder, where I put all the files I needed to deploy my application to Heroku. I called the folder “hello” and I created several other folders and files inside it. I created a Web.scala file in: hello/src/main/scala/Web.scala. It is very important to create those files exactly as Heroku tutorial instructs as when you deploy your application Heroku searches for those files in the specified locations. Web.scala is a sample application provided by Heroku and its code looks like that:

**import** org.jboss.netty.handler.codec.http.{HttpRequest, HttpResponse}

**import** com.twitter.finagle.builder.ServerBuilder

**import** com.twitter.finagle.http.{Http, Response}

**import** com.twitter.finagle.Service

**import** com.twitter.util.Future

**import** java.net.InetSocketAddress

**import** util.Properties

**object** Web {

**def** main(args: Array[String]) {

**val** port = Properties.envOrElse("PORT", "8080").toInt

println("Starting on port:"+port)

ServerBuilder()

.codec(Http())

.name("hello-server")

.bindTo(**new** InetSocketAddress(port))

.build(**new** Hello)

println("Started.")

}

}

**class** Hello **extends** Service[HttpRequest, HttpResponse] {

**def** apply(req: HttpRequest): Future[HttpResponse] = {

**val** response = Response()

response.setStatusCode(200)

response.setContentString("Hello World")

Future(response)

}

}

In order for Heroku to recognise the Web.scala application as a Scala application a file called build.properties has to be created. You have to put in a folder called: project. The file has to contain:

sbt.version=0.12.0

Next you have to declare dependencies in another file called: build.sbt that you put in the root directory of the project folder. A dependency defines what libraries the application needs in order to run. The build.sbt file has to contain:

import com.typesafe.startscript.StartScriptPlugin

seq(StartScriptPlugin.startScriptForClassesSettings: \_\*)

name := "hello"

version := "1.0"

scalaVersion := "2.9.2"

resolvers += "twitter-repo" at "http://maven.twttr.com"

libraryDependencies ++= Seq("com.twitter" % "finagle-core" % "1.9.0", "com.twitter" % "finagle-http" % "1.9.0")

The libraries that are declared in the folder do not have to be installed on your computer unless you use Eclipse to test the Web.scala application.

There is one more file that you have to create before attempting to run your application. It is called build.sbt and it has to be put in the project folder and it has to contain:

resolvers += Classpaths.typesafeResolver

addSbtPlugin("com.typesafe.startscript" % "xsbt-start-script-plugin" % "0.5.3")

//here I don’t get it?! ASK BILLY

## [Add the start script plugin](https://devcenter.heroku.com/articles/getting-started-with-scala#add-the-start-script-plugin)

At deploy time, Heroku runs sbt clean compile stage to build your Scala app. [Typesafe](http://typesafe.com/)’s [xbst-start-script-plugin](http://github.com/typesafehub/xsbt-start-script-plugin) adds a stage task to sbt that generates start scripts for your application.

To use the plugin, create this file:

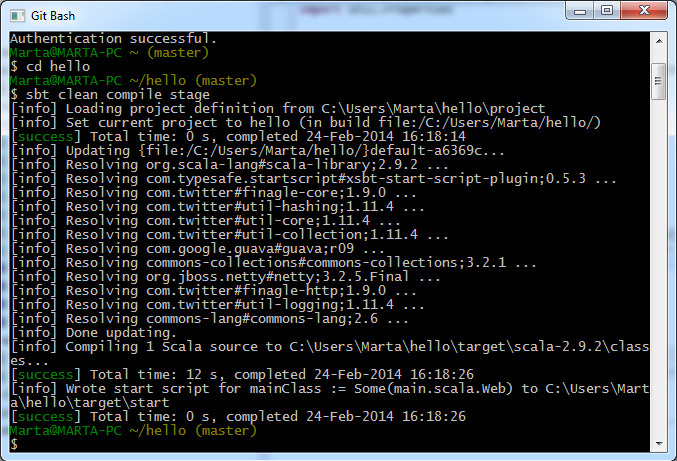
#### [project/build.sbt](https://devcenter.heroku.com/articles/getting-started-with-scala#project-build-sbt)

resolvers += Classpaths.typesafeResolver

addSbtPlugin("com.typesafe.startscript" % "xsbt-start-script-plugin" % "0.5.3")

The stage task, by convention, performs any tasks needed to prepare an app to be run in-place. Other plugins that use a different approach to prepare an app to run could define stage as well.

Next step is to build the application locally by typing: sbt clean compile stage:



Once the application build successfully we can run it. There are two ways of running the application locally. First one is to run the application using Foreman. In order to do that you have to create a file called: Procfile, which contains a list of commands that shall be executed to start a web dyno. Although the tutorial states that Procfile is a text file, you have to make sure that the file doesn’t have the .txt extention. In our case, Procfile should contain the following command:

web: target/start Web.

The word “web” is a single process type and it indicates that, as the tutorial reads, “this process will be attached to the HTTP routing stack of Heroku, and receive web traffic when deployed.” The HTTP routing is

//ASK BILLY what is the HTTP routing

//also: what is a cedar stack?

The next step is to run the application using Foreman by typing the command:

foreman start.

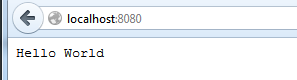
Unfortunately when I tried to execute that command, Foreman could not find the Procfile. Only when I actually typed:

target/start Web

I managed to run the Web.scala application locally:



The result in the browser:

.

Next step is to store the application on Heroku using git. I had to create a file called .gitignore that contained:

target

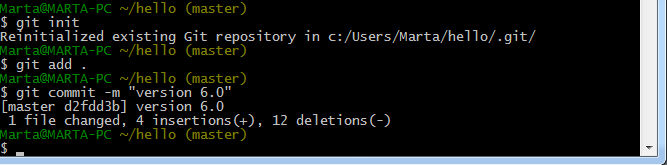
project/boot

project/target

project/plugins/target

The file .gitignore indicates which files/folders are to be ignored when pushing an application to Heroku.

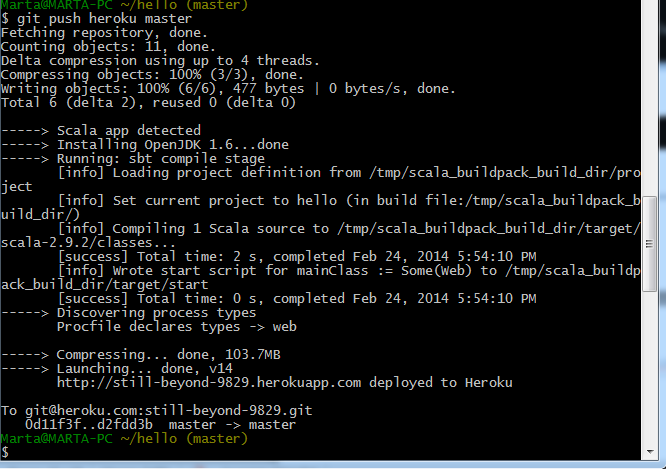
The next step is to push all the files to Heroku. We do that by typing:



In order to create the app we simply type:



Next step is to deploy the application:

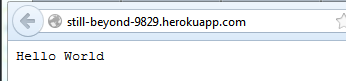


Once the code, the dependencies and the process types are successfully deployed to Heroku, we can visit the application. We tell Heroku to execute a process type, which is done by running the command associated with the process type in a dyno. I mentioned a dyno before. A dyno is a lightweight container – a basic composition of Heroku. A dyno, a lightweight container, runs a single user-specified command. (Paraphrased).

To visit the application, this time, on Heroku, and not on our local computer, we type: heroku open:



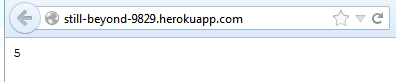
The result in the browser:

.

We note that the URL for the web service is: <http://still-beyond-9829.herokuapp.com/>.

In this way I have successfully deployed a Scala application to Heroku.

I have modified the Web.scala code so that the application would call a method called add() that adds two integers and displays the result in the browser. For the time being the application is “hard coded” and does not take input from the user.

.

# Spray

# Play! framework

# References:

* <http://www.drdobbs.com/architecture-and-design/interview-with-scalas-martin-odersky/231001802>
* Wikipedia
* https://devcenter.heroku.com/articles/getting-started-with-scala

1. https://typesafe.com/blog/why-scala [↑](#footnote-ref-1)
2. https://typesafe.com/blog/why-scala [↑](#footnote-ref-2)
3. Book, page 45 [↑](#footnote-ref-3)
4. Book, page 45 [↑](#footnote-ref-4)
5. (book, page 45). [↑](#footnote-ref-5)
6. (Book, page 46). [↑](#footnote-ref-6)
7. book page 46 [↑](#footnote-ref-7)
8. (Book, page 47). [↑](#footnote-ref-8)
9. (Book, page 47). [↑](#footnote-ref-9)
10. (Book, page 48). [↑](#footnote-ref-10)
11. (Book, page 48). [↑](#footnote-ref-11)
12. http://en.wikipedia.org/wiki/Scala\_%28programming\_language%29 [↑](#footnote-ref-12)
13. http://en.wikipedia.org/wiki/Scala\_%28programming\_language%29 [↑](#footnote-ref-13)
14. (Book, page 48). [↑](#footnote-ref-14)
15. (code from the Book, page 50) [↑](#footnote-ref-15)
16. (Wikipedia). [↑](#footnote-ref-16)
17. (Odersky M., Spoon L., Venners B., 2010, p.67) [↑](#footnote-ref-17)
18. page 218 in the book [↑](#footnote-ref-18)
19. (Wikipedia) [↑](#footnote-ref-19)
20. http://en.wikipedia.org/wiki/Deadlock [↑](#footnote-ref-20)
21. book page 610 [↑](#footnote-ref-21)
22. book page 611 [↑](#footnote-ref-22)
23. book page 611 [↑](#footnote-ref-23)
24. book page 61 [↑](#footnote-ref-24)
25. (book, page 611). [↑](#footnote-ref-25)
26. Source code: Book page 612 [↑](#footnote-ref-26)
27. http://en.wikipedia.org/wiki/Web\_service [↑](#footnote-ref-27)
28. http://stackoverflow.com/questions/226108/what-is-a-web-service-in-plain-english [↑](#footnote-ref-28)
29. http://docs.oracle.com/javaee/6/tutorial/doc/giqsx.html [↑](#footnote-ref-29)
30. http://en.wikipedia.org/wiki/SOAP\_%28protocol%29 [↑](#footnote-ref-30)
31. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\_arch\_style.htm [↑](#footnote-ref-31)
32. http://www.counsellingbyabhi.com/2013/03/difference-between-rest-and-soap.html [↑](#footnote-ref-32)
33. http://www.counsellingbyabhi.com/2013/03/difference-between-rest-and-soap.html [↑](#footnote-ref-33)