Build a simple Actor-based shopping-basket REST API with Scala, Akka and Play 2

1. Choice of toolbox:

Scala http://www.scala-lang.org is a programming language that mixes object-oriented and functional programming. It's often seen as an alternative language to Java for programmers who want functional features in a statically typed programming language that runs on the JVM while keeping a Java feel. It provides compile time constraints that could help to avoid certain erroneous scenarios.

Scala's primary concurrency construct is *Actors*. *Actors* are basically concurrent processes that communicate by exchanging messages. Actors can also be seen as a form of active objects where invoking a method corresponds to sending a message. The Scala Actors library provides both asynchronous and synchronous message sends, however the news is that starting with Scala 2.11.0, the Scala Actors library is deprecated. Already in Scala 2.10.0 the default actor library is Akka.

Akka http://akka.io Any system with the need for high-throughput and low latency is a good candidate for using Akka. Akka Actors let you manage service failures (Supervisors), load management (back-off strategies, timeouts and processing-isolation), as well as both horizontal and vertical scalability (add more cores and/or add more machines).

REST (Representational State Transfer) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. There are several libraries and frameworks available for building systems that produce and consume RESTful services in the Scala ecosystem. Here is an overview of the main libraries:

Play https://www.playframework.com
One of the two main Scala REST frameworks, Play is based on the Model-view-controller pattern. Each entry point, paired with an HTTP verb, maps to a Controller function. The controller enables views to be web pages, JSON, XML, or just about anything else.
Play's stateless architecture enables horizontal scaling, ideal for serving many incoming requests without having to share resources (such as a session) between them. It's at the forefront of the Reactive programming trend, in which servers are event-based and parallel processing is used to cater to the ever-increasing demands of modern websites.
If you're a Java developer, Play REST support would be closest to compare with Spring's approach to build RESTful web services via REST Controllers. Play is a full-fledged web framework, so it can be used to develop a client as well for the shopping-basket server, as an alternative to testing via **Postman** http://www.getpostman.com.

Spray http://spray.io
Is a lightweight and widely used toolkit for building REST/HTTP-based integration layers on top of Scala and Akka. It comes with a small, embedded, fast HTTP server, called spray-can, that is a great option when your use-case does not require a fully-fledged servlet container like Tomcat, Jetty etc. It was released in 2011 and designed to work with the Akka actor model and its documentation states is highly testable, which is great news since the shopping-basket API must be supported with a tests-suite.

Akka HTTP http://doc.akka.io/docs/akka-stream-and-http-experimental/1.0-M2/scala/http/
It was recently announced that Spray would be replaced with Akka HTTP, thus cementing Akka HTTP as the successor of Spray. It's maintained by Lightbend and it's been recommended that Spray users migrate to it soon. One major drawback of Akka HTTP is that it's not as mature as Spray, and it's performance is not optimized yet. Users may also notice a lack of documentation for some cases.
Akka HTTP is built on top of Akka Streams. It provides a DSL-based approach based on Spray.

There are much more frameworks available if you want to develop RESTful services in Scala, however I am not going to assess all.

2. Getting started:

As my first choice to try out is Spray, I'd figured it out that the best way to start learning Akka and Spray – without reading a book, which by the way can obviously be as well a great choice - is to download Lightbend Activator and study code of Akka and Spray template projects.

```
Cristinas-iMac:dev cristina$ activator new scala-akka-spray activator-akka-spray
```

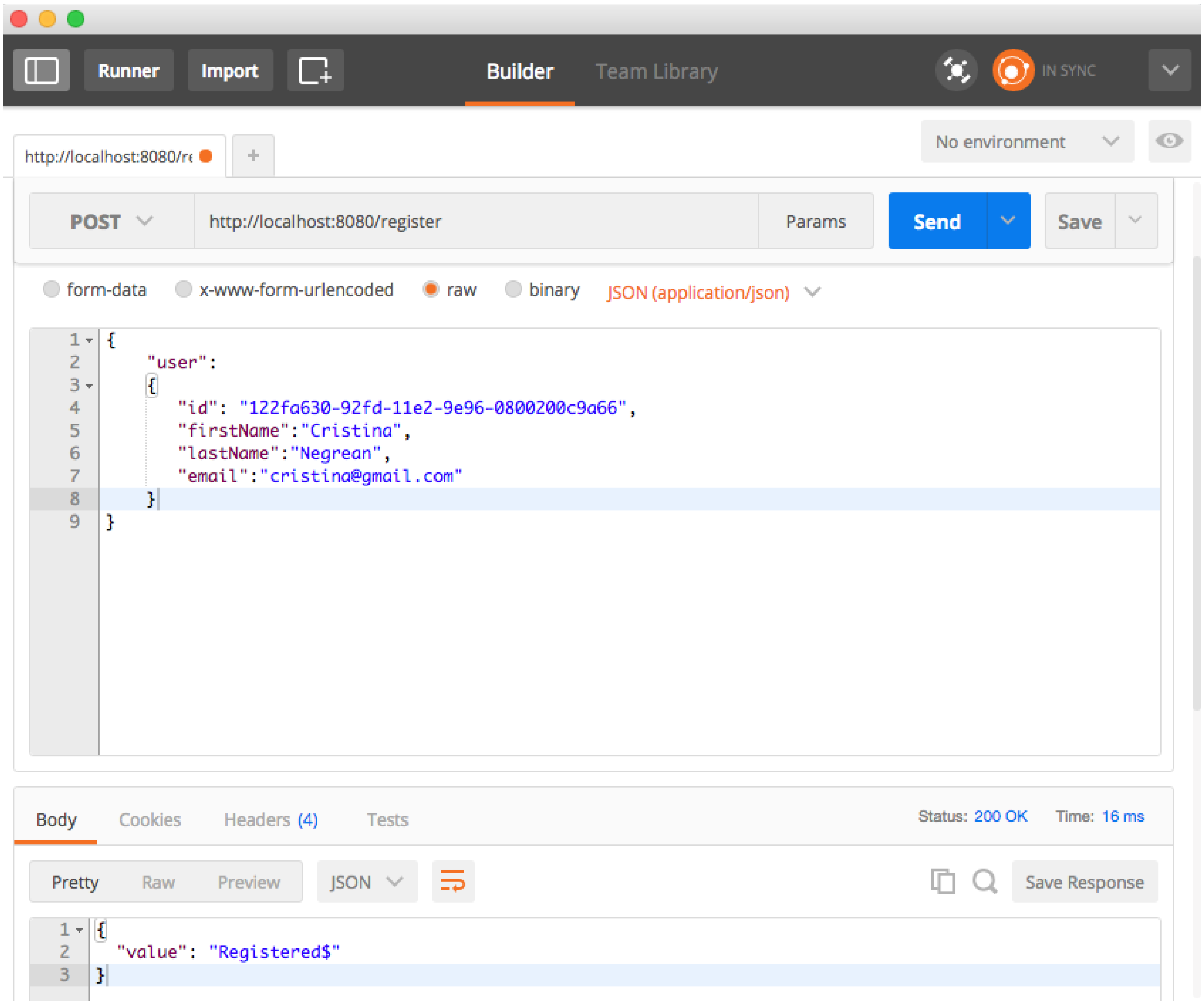
Thus one simple command and I have already got lots of work done and I can go have some beer :)
Ok, not so fast, if you're a Java developer like me and you are familiar with Spring Boot CLI, then the Lightbend Activator gives you a similar developer experience. Activator template projects resemble to Spring Boot's opinionated 'starter' POMs to simplify Maven or Gradle configuration of dependent libraries and get you started up fast. At first glance, I am pleasantly surprised of how interactive is the Lightbend Activator tool, by reading the output in the terminal:

```
Fetching the latest list of templates...
OK, application "scala-akka-spray" is being created using the "activator-akka-spray" template.

To run "scala-akka-spray" from the command line, "cd scala-akka-spray" then:
/Users/cristina/dev/scala-akka-spray/activator run

To run the test for "scala-akka-spray" from the command line, "cd scala-akka-spray" then:
/Users/cristina/dev/scala-akka-spray/activator test

To run the Activator UI for "scala-akka-spray" from the command line, "cd scala-akka-spray" then:
/Users/cristina/dev/scala-akka-spray/activator ui
```

```
Cristinas-iMac:dev cristina$ cd scala-akka-spray/
Cristinas-iMac:scala-akka-spray cristina$ activator run
```

```
Getting org.scala-sbt sbt 0.13.0 ...
downloading file:/Users/cristina/install/activator-dist-1.3.10/repository/org.scala-sbt/sbt/0.13.0/jars/sbt.jar ...
```

At this point one can already guess: **SBT** is the most used build tool for Scala projects, just as Maven or Gradle are for Java. SBT will download all the template project dependencies and start the sample REST server. As a curious human being, I have fired up a Postman test collection against the user registration API.



POST http://localhost:8080/register

form-data x-www-form-urlencoded [✓ raw] binary JSON (application/json)

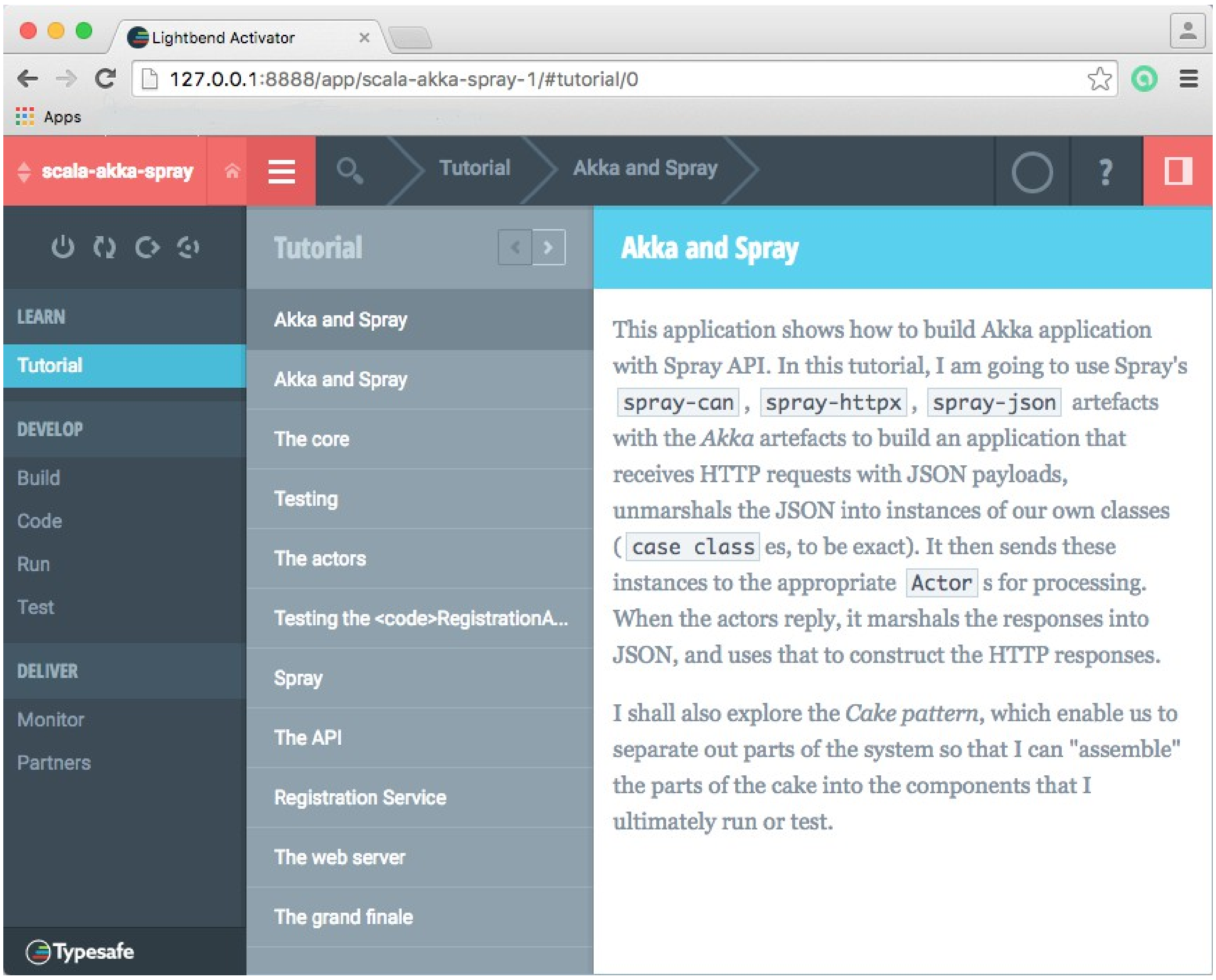
```
{
    "user":
    {
      "id": "122fa630-92fd-11e2-9e96-0800200c9a66",
      "firstName":"Cristina",
      "lastName":"Negrean",
      "email":"cristina@gmail.com"
    }
}
```

Status: 200 OK Time: 16 ms

```
{
  "value": "Registered$"
}
```

Before adjusting sails towards the shopping-basket REST server, it is worth having a look at the Lightbend Activator UI. It is a web browser based IDE that allows you to:

- learn & document via an integrated tutorial feature;
- develop via browsing and adjusting the code or creating Eclipse and/or IntelliJ IDEA project files so that you can import the code in the IDE of your choice;
- monitor as Activator can be used to configure and run your application locally with New Relic gathering monitoring information that will be later pushed to New Relic server in a separate dashboard.



Akka and Spray

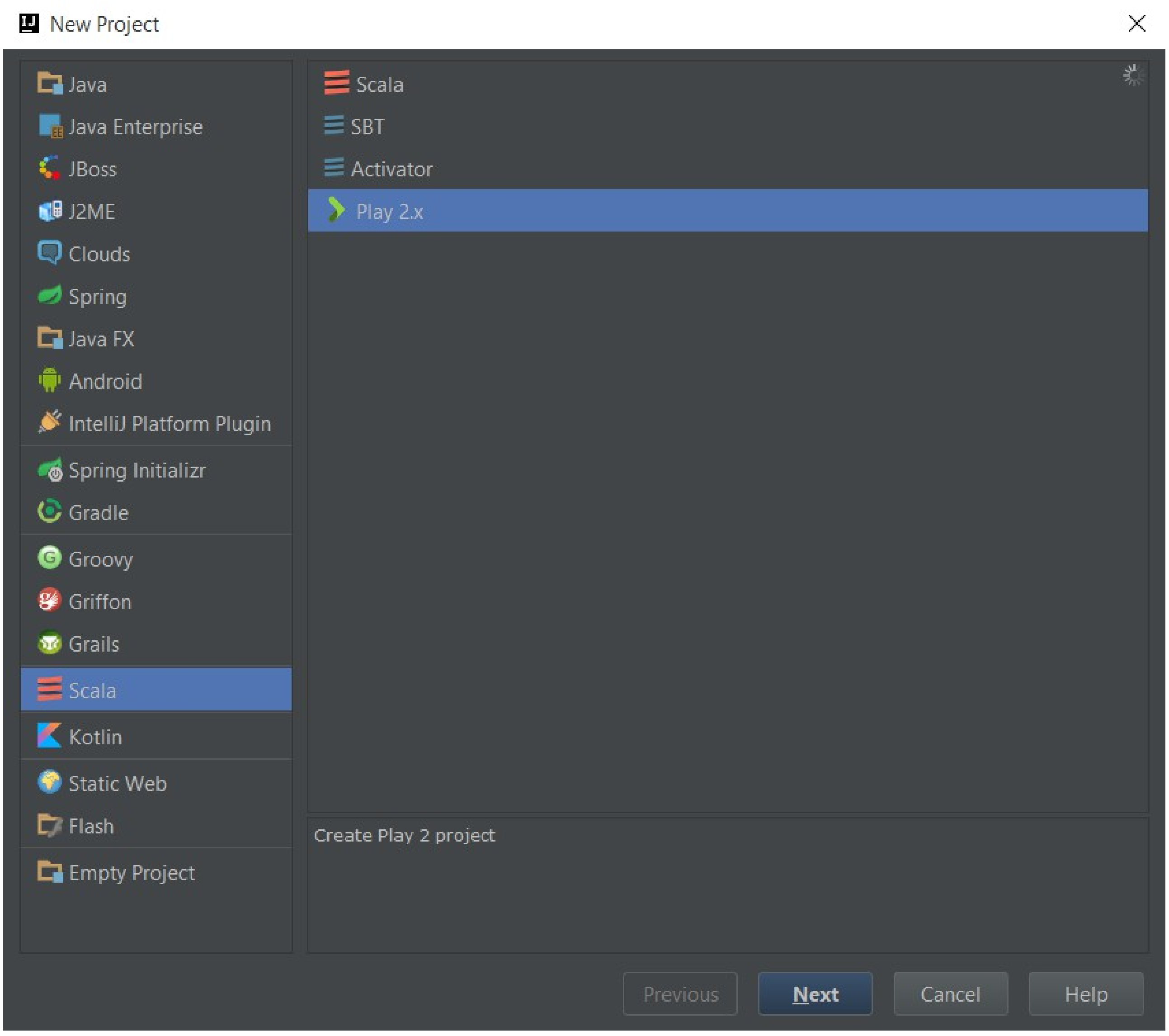
This application shows how to build Akka application with Spray API. In this tutorial, I am going to use Spray's `spray-can`, `spray-httpx`, `spray-json` artefacts with the *Akka* artefacts to build an application that receives HTTP requests with JSON payloads, unmarshals the JSON into instances of our own classes (`case class`es, to be exact). It then sends these instances to the appropriate `Actor`s for processing. When the actors reply, it marshals the responses into JSON, and uses that to construct the HTTP responses.

I shall also explore the *Cake pattern*, which enable us to separate out parts of the system so that I can "assemble" the parts of the cake into the components that I ultimately run or test.

I noticed that Activator template projects for Akka and Spray do not always use latest versions of Scala, Akka and Spray. One can adjust these in build.sbt file.

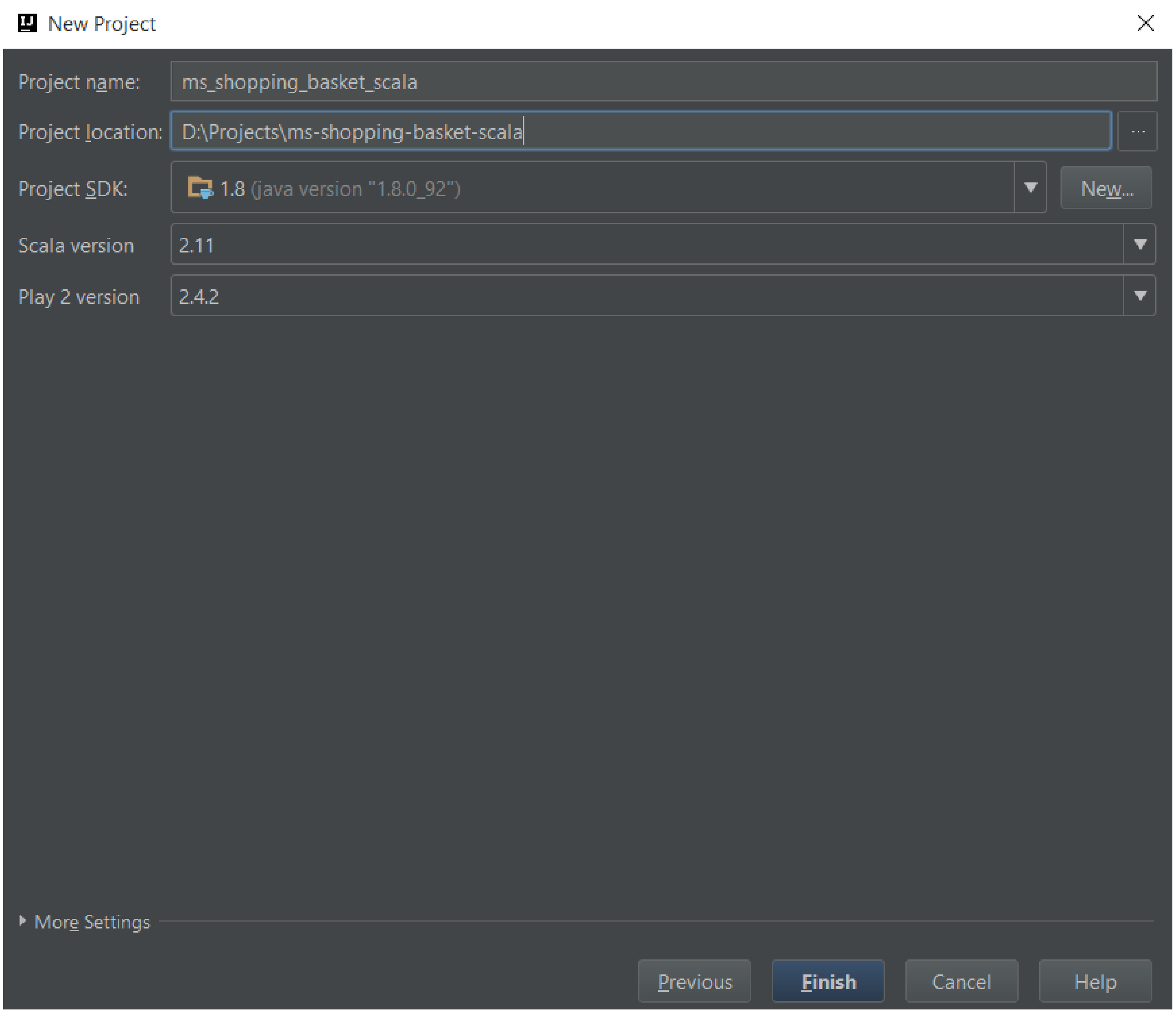
I will switch to IntelliJ IDEA, for my next stop: Play 2.

When using IntelliJ IDEA, I need to install Scala SDK and SBT build tool separately. To install Scala just go to http://www.scala-lang.org/download and download the binaries for your system. To install SBT you can follow the instructions at: http://www.scala-sbt.org/download.html
Also you'll need to install plugins for Scala, SBT and Play in IntelliJ IDEA, so you can work fully from within IDE instead of CLI (Command Line Interface)



IntelliJ IDEA new Scala project wizard allows selecting one of the above project templates. I will select Play 2.x.

On next wizard screen, see below, you will be asked to provide project details, project SDK – I am using hereby my default Java SDK installation, Scala and Play library versions.


New Project

Project name: ms_shopping_basket_scala

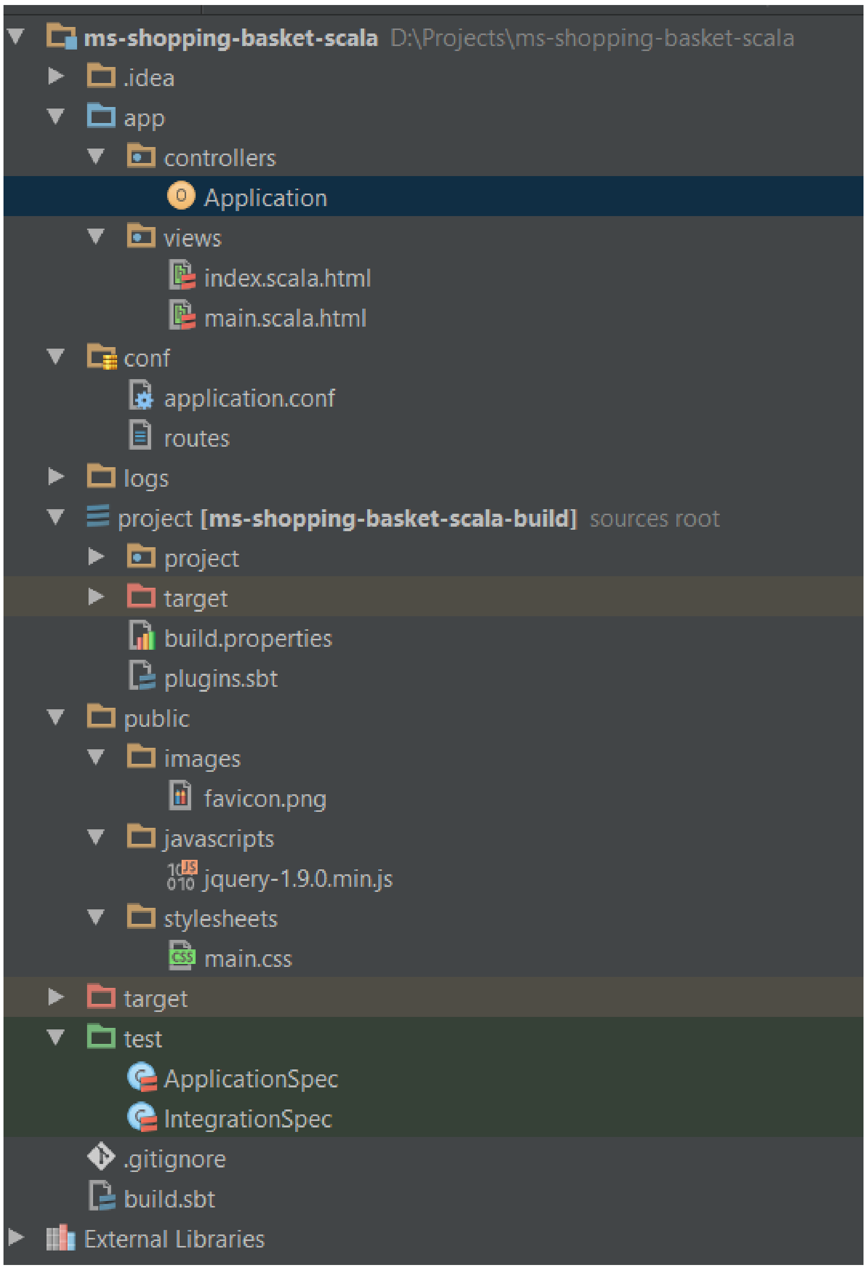
Project location: D:\Projects\ms-shopping-basket-scala

Project SDK: 1.8 (java version "1.8.0_92")

Scala version: 2.11

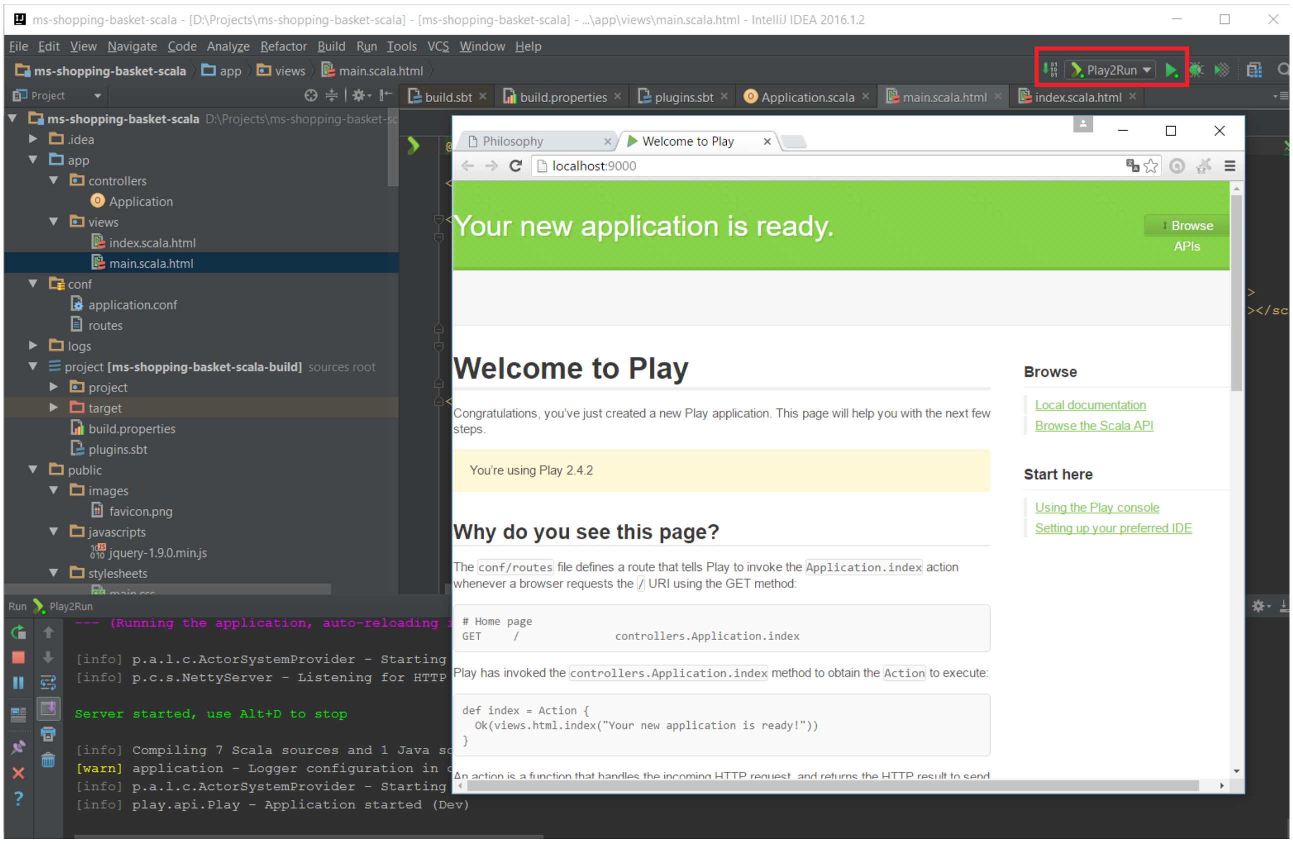
Play 2 version: 2.4.2

After clicking 'Finish' you will get below basic project structure.



```
▼ ms-shopping-basket-scala  D:\Projects\ms-shopping-basket-scala
  ▶ .idea
  ▼ app
    ▼ controllers
        Application
    ▼ views
        index.scala.html
        main.scala.html
  ▼ conf
      application.conf
      routes
  ▶ logs
  ▼ project [ms-shopping-basket-scala-build]  sources root
    ▶ project
    ▶ target
      build.properties
      plugins.sbt
  ▼ public
    ▼ images
        favicon.png
    ▼ javascripts
        jquery-1.9.0.min.js
    ▼ stylesheets
        main.css
  ▶ target
  ▼ test
      ApplicationSpec
      IntegrationSpec
    .gitignore
    build.sbt
▶ External Libraries
```

By clicking the green play button along “Play2Run” task, the application will be started and by accessing http://localhost:9000 in the browser, you'll get a basic explanation on how the application works. I won't detail that here, so feel free to read that on your own.


ms-shopping-basket-scala - [D:\Projects\ms-shopping-basket-scala] - [ms-shopping-basket-scala] - ...\app\views\main.scala.html - IntelliJ IDEA 2016.1.2
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
ms-shopping-basket-scala app views main.scala.html
Play2Run
build.sbt build.properties plugins.sbt Application.scala main.scala.html index.scala.html
Project
ms-shopping-basket-scala D:\Projects\ms-shopping-basket-sc
.idea
app
controllers
Application
views
index.scala.html
main.scala.html
conf
application.conf
routes
logs
project [ms-shopping-basket-scala-build] sources root
project
target
build.properties
plugins.sbt
public
images
favicon.png
javascripts
jquery-1.9.0.min.js
stylesheets
Philosophy
Welcome to Play
localhost:9000
Browse APIs
Run Play2Run

Your new application is ready.

Welcome to Play

Congratulations, you've just created a new Play application. This page will help you with the next few steps.

You're using Play 2.4.2

Why do you see this page?

The `conf/routes` file defines a route that tells Play to invoke the `Application.index` action whenever a browser requests the `/` URI using the GET method:

```
# Home page
GET     /                   controllers.Application.index
```

Play has invoked the `controllers.Application.index` method to obtain the `Action` to execute:

```
def index = Action {
  Ok(views.html.index("Your new application is ready!"))
}
```

An action is a function that handles the incoming HTTP request, and returns the HTTP result to send

Browse

Local documentation

Browse the Scala API

Start here

Using the Play console

Setting up your preferred IDE

```
--- (Running the application, auto-reloading i

[info] p.a.l.c.ActorSystemProvider - Starting
[info] p.c.s.NettyServer - Listening for HTTP

Server started, use Alt+D to stop

[info] Compiling 7 Scala sources and 1 Java sc
[warn] application - Logger configuration in c
[info] p.a.l.c.ActorSystemProvider - Starting
[info] play.api.Play - Application started (Dev)
```

3. Coding the Shopping Basket

3.1 Build configuration

Let's start with build configuration for application. The file called **build.sbt** with the following content should be placed to the root directory of application example.

```
name := "ms-shopping-basket-scala"
version := "1.0"
lazy val `ms-shopping-basket-scala` = (project in
file(".")).enablePlugins(PlayScala)
scalaVersion := "2.11.8"
libraryDependencies ++= Seq(
  jdbc , cache , ws   , specs2 % Test,
  "org.mockito"              %  "mockito-all"               % "1.10.19"   %
Test,
  "com.typesafe.play"        %% "play-specs2"               % "2.5.3"   % Test,
  "com.typesafe.play"        %% "play-test"                 % "2.5.3"   % Test,
  "com.typesafe.akka"        %% "akka-testkit"              % "2.4.2"  % Test
)
unmanagedResourceDirectories in Test <+=  baseDirectory
( _ /"target/web/public/test" )
// https://www.playframework.com/documentation/2.5.x/Migration24#routing
routesGenerator := InjectedRoutesGenerator
resolvers ++= Seq(
  "scalaz-bintray" at "https://dl.bintray.com/scalaz/releases",
  "Typesafe repository" at "http://repo.typesafe.com/typesafe/releases/",
  "Typesafe Maven repository" at "https://repo.typesafe.com/typesafe/maven-
releases/",
  "Typesafe Snapshots" at
"https://oss.sonatype.org/content/repositories/snapshots/"
)
```

Application name, version and target version of Scala are specified at the top of the file. Managed dependencies can be added by simply listing them in the *libraryDependencies* setting. Dependency declaration can look like this:

libraryDependencies += "groupID" % "artifactID" % "revision"

It is allowed to add a list of dependencies at once, like in build.sbt example above.

SBT uses the standard Maven2 repository by default. However, additional repositories could be added using the following pattern:

resolvers += "name" at "location"

Note that by default I end up with versions: Scala 2.11.7, Play 2.4.2 and SBT 0.13.5. If you want to upgrade to latest & greatest - https://www.playframework.com/documentation/2.5.x/Highlights25 – change the following:

- replace 2.4.2 with **2.5.3** in file **project/plugins.sbt** to use current latest release of Play
- replace 0.13.5 with **0.13.11** in file **project/build.properties** to use latest SBT release
- replace 2.11.7 with **2.11.8** in file **build.sbt** in project root directory for `scalaVersion :=`

3.2 Application configuration

All required configuration settings is placed in **/conf/application.conf** file.

This file can be used to override default Play framework settings – like thread pool, logging – and add new application level configuration.

As far as the shopping-basket application concerned, most important settings are listed below, where the Play Module is registering a custom class `ShoppingBasketModuleRegister`

which is basically binding the `shopping-basket-actor` This will cause the actor to be instantiated by Guice – DI framework for Play and Akka - allowing it to be dependency injected itself in Play Controller components. See documentation for details.

```
# ~~~~~
# The secret key is used to secure cryptographics functions.
# If you deploy your application to several instances be sure to use the same key!
application.secret="%APPLICATION_SECRET%"
# The application languages
# ~~~~~
play.i18n.langs=[ "en" ]

play.modules.enabled += "shoppingbasket.modules.ShoppingBasketModuleRegister"
```

Listing of ***shoppingbasket.modules.ShoppingBasketModuleRegister***

```
class ShoppingBasketModuleRegister extends AbstractModule with AkkaGuiceSupport {
 def configure = {
    bindActor[ShoppingBasketActor]
(ShoppingBasketModuleRegister.ShoppingBasketRegisterName)
  }
}
object ShoppingBasketModuleRegister {
  val ShoppingBasketRegisterName = "shopping-basket-actor"
}
```

One can configure database connection settings and application logging as well in application.conf. I will use for now in-memory data store for demo purpose and configure logging in a separate **logback.xml** file:

```
<configuration>
<conversionRule conversionWord="coloredLevel"
```

```
converterClass="play.api.Logger$ColoredLevel" />
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%coloredLevel - %logger - %message%n%xException</pattern>
    </encoder>
  </appender>
  <!--
    The logger name is typically the Java/Scala package name.
    This configures the log level to log at for a package and its children
packages.
  -->
  <logger name="play" level="INFO" />
  <logger name="application" level="DEBUG" />
  <logger name="akka" level="INFO"/>
  <root level="ERROR">
    <appender-ref ref="STDOUT" />
  </root>
  <root level="DEBUG">
    <appender-ref ref="STDOUT"/>
  </root>
</configuration>
```

3.3 Domain model

Listing of ***shoppingbasket.dao.entities.Product -*** Scala case class that should contain all fields. A product has at least a name, a description and a price. Each product is uniquely identified by an id. As I am using syntax: **id = java.util.UUID.*randomUUID*().toString** to create a random unique identifier, the id is a String, but when you would use a database to store Product entities, best practice is to define it as **id: Option[Long]**

Case class constructor parameters are val by default, so accessor methods are generated for the parameters, but mutator methods are not generated**.** As every product has a certain stock which is updated when products are added of deleted in the shopping-basket.. So I will go and define the stock parameter as a var, so both accessor and mutator methods are generated.

```
case class Product(id: String, name: String, description: String, price:
BigDecimal, currency: String, var stock: Long) {

   def isAvailable = stock > 0
}
```

Listing of ***shoppingbasket.dao.entities.gen.ProductGen -*** The product catalog is implemented as an in-memory list. The product catalog life-cycle spans until will application restart or a HTTP GET request is initiated for URL path **/api/products**

```
object ProductGen {

  val intRandomNumber = new Random(2)
  def getNewProduct: Product = Product(
    id = UUID.randomUUID().toString,
    name = Random.alphanumeric.take(10).mkString(""),
    description = Random.alphanumeric.take(50).mkString(""),
    price = BigDecimal(getRandom(2, 80)),
    currency = "EUR",
    stock = getRandom(1, 10)
  )
  var getProducts: scala.collection.immutable.List[Product]= {
    (1 to getRandom(10, 50)).map(i => getNewProduct)(collection.breakOut)
  }
  private def getRandom(from: Int, to: Int): Int = {
    if (from < to)
```

```
      from + new Random().nextInt(Math.abs(to - from))
    else from - new Random().nextInt(Math.abs(to - from))
  }
}
```

3.4 Data Access Layer

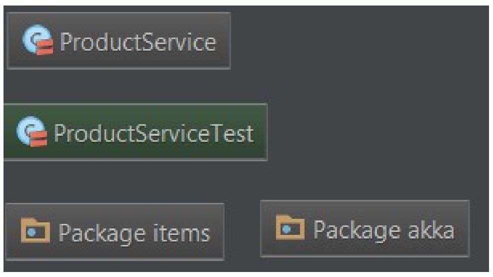
Listing of ***shoppingbasket.dao.ProductRepository***

```
@Singleton
class ProductRepository {

  def collection = Future.successful(ProductGen.getProducts)
  /** Retrieve a product from the list */
  def item(id: String): Future[Option[Product]] =
Future.successful(ProductGen.getProducts.find(_.id == id))
}
```

A common use case within **Akka** is to have some computation performed concurrently without needing the extra utility of **an Actor**. If you find yourself creating a **pool of Actors** for the sole reason of performing a calculation in parallel, there is an easier (and faster) way by using scala.concurrent.Future companion. In general, if the API you are using returns `Future`s, it is non-blocking, otherwise it is blocking.

3.5 Service Layer

Listing of ***shoppingbasket.service*** package



Listing of ***shoppingbasket.service.ProductService* –** read-only access to ProductRepository and enriches the returned list of products with pagination info

```

@Singleton
class ProductService @Inject()(productRepo: ProductRepository) {
  /**
     * Get all products from the system
     *
     * @param page      who needs to retrieved, default 1
     * @param pageSize  dimension of the extract data, default 25
     * @param available if the requested data needs to be available
     * @return a [[Item]] of [[shoppingbasket.dao.entities.Product]] as
[[scala.concurrent.Future]]
     *          with the extract data
     */
  def products(
      page: Int = 1,
      pageSize: Int = 25,
      available: Option[Boolean] = None)(implicit ec: ExecutionContext):
Future[Item[Product]] = {
    Logger.debug(s"entry data: $page - $pageSize - $available")
    // get all data based on the given parameter `available`
    val productsFuture = productRepo.collection.map {
      _.filter(product => available.fold(true)(_ == product.isAvailable))
    }
    productsFuture.map {
      products =>
        Logger.debug(s"retrieved data - $products")
        val sliceStartPosition = (page - 1) * pageSize
        // get just a slice of those elements
        val returnedElements = products.slice(sliceStartPosition,
sliceStartPosition + pageSize)

        Item[Product](PaginationItem(products.size, page, returnedElements.size),
returnedElements)
    }
  }
  /** Get a product from the system */
  def product(id: String)(implicit ec: ExecutionContext): Future[Option[Product]] =
    productRepo.item(id)
}
```

Listing of ***shoppingbasket.service.akka.ShoppingBasketActor* –** receives messages regarding operations on a ShoppingBasket

```
class ShoppingBasketActor @Inject()(productRepo: ProductRepository)(implicit ec:
ExecutionContext) extends Actor {
  def receive = {
    case request: ShoppingBasketCreate =>
      handleRequest(handleShoppingBasketCreateRequest, request)
    case request: ShoppingBasketView =>
      handleRequest(handleBasketView, request)
    case request: ShoppingBasketItemView =>
      handleRequest(handleBasketItems, request)
    case request: ShoppingBasketItemDelete =>
      handleRequest(handleBasketItemDelete, request)
    case createItemInfo: ShoppingBasketCreateItemSingle =>
      handleRequest(handleShoppingBasketCreateItemSimple, createItemInfo)
    case _ => ???
  }
  private def handleRequest[T](f: T => Future[_], p: T) : Unit = {
    val realSender = sender()
    f(p) onComplete {
      case Success(response) => realSender ! response
      case Failure(ex) => realSender ! Left(ServiceErrors("", ex.toString()))
    }
  }

private[akka] var shoppingBasketSeq: scala.collection.mutable.Seq[ShoppingBasket] =
scala.collection.mutable.Seq.empty[ShoppingBasket]

private[akka] def handleShoppingBasketCreateRequest(request: ShoppingBasketCreate):
Future[Either[ServiceErrors, String]] =

  productRepo.collection.map {
    products =>
      val itemsIdsSeq = request.items.map(_.product.id)
      val retrievedProducts = products.filter(p => itemsIdsSeq.contains(p.id))
      if (retrievedProducts.size != request.items.size) {
        Left(ServiceErrors("/items", "elements not discovered"))
      } else {
        // continue to second type of validation //availability of the item
        val unavailableErrorProducts = retrievedProducts
          .filterNot(p => checkProductAvailability(p, requestedProductById(p.id,
request.items)))
          .flatMap(p => ServiceErrors("/items", s"there are not sufficient $
{p.name}").errors)
        if (unavailableErrorProducts.nonEmpty) {
          // it looks like we have a few products which are not available
          Left(ServiceErrors(unavailableErrorProducts))
        } else {
          // this thing usually should be in a transaction
          updateItems(request.items) match {
            case Success(_) =>
              // we can store the new list
              val newShoppingBasket = ShoppingBasket(
                UUID.randomUUID().toString,
                request.items.map(i => ShoppingBasketItem(UUID.randomUUID(), i))
              )
              shoppingBasketSeq = shoppingBasketSeq :+ newShoppingBasket
              Right(newShoppingBasket.id)
            case Failure(errors) =>
              Left(ServiceErrors("/items", errors.toString()))
          }
        }
      }
  }

...

}
```

The **shoppingbasket.service.items** contains data transfer objects used passed by service layer to web/http layer and Play Controller Components with regards to Pagination and a bunch of Scala case classes for handling operations on shopping basket items.

3.6 HTTP Layer and Play Controller Components

The **router** is the component in charge of translating incoming HTTP Requests into action calls (a static, public method of a **Controller**).

An HTTP request is seen as an event by the Play MVC framework. The event contains two major pieces of information:

- The Request path (such as /api/shoppingbaskets/6f13ea77-e658-43c3-9f31-c7e16a6c76c4, /api/products) including the query string.
- The HTTP method (GET, POST, PUT, DELETE)

Listing of **conf/routes**:

```
# Product
GET         /api/products
shoppingbasket.controllers.ProductController.products(page: Int ?= 1, pageSize: Int
?= 50, available: Option[Boolean])
GET         /api/products/:id
shoppingbasket.controllers.ProductController.product(id: String)
# Shopping Basket
POST        /api/shoppingbaskets
shoppingbasket.controllers.BasketController.post()
GET         /api/shoppingbaskets/:id
shoppingbasket.controllers.BasketController.get(id: String)
# Shopping Basket Item Operations
GET         /api/shoppingbaskets/:basketId/items/:itemId
shoppingbasket.controllers.BasketItemController.itemByBasket(basketId: String,
itemId: String)
DELETE      /api/shoppingbaskets/:basketId/items/:itemId
shoppingbasket.controllers.BasketItemController.deleteItemFromBasket(basketId:
String, itemId: String)
POST        /api/shoppingbaskets/:basketId/items
shoppingbasket.controllers.BasketItemController.addItemToBasket(basketId)
```

Listing of **shoppingbasket.controller.BasketController**:

```
class BasketController @Inject()(val messagesApi: MessagesApi,
                                   @Named("shopping-basket-actor")
shoppingBasketActor: ActorRef)
                                  (implicit ec: ExecutionContext) extends Controller
with I18nSupport {
  import scala.concurrent.duration._
  implicit val timeout: Timeout = 5.seconds
  /** Handles the creation of a new shopping basket */
  def post() = Action.async { implicit request =>
    request.body.asJson.map {
      json =>
        json.validate[PostBasketForms].map {
          form =>
            (shoppingBasketActor ?
BasketCreationMapper.toServiceObj(form)).mapTo[Either[ServiceErrors, String]].map {
              case Right(id) => Created.withHeaders((LOCATION,
basketLocationURl(id)))
              case Left(errors) => BadRequest(errors.toString)
            }
        }.recoverTotal(e => Future.successful(BadRequest("Detected error: " +
JsError.toJson(e))))
    }.getOrElse {
      Future.successful(BadRequest("Expecting Json Data"))
    }
  }
  /** Retrieve a basket by id */
  def get(id: String) = Action.async { implicit request =>
    (shoppingBasketActor ?
ShoppingBasketView(id)).mapTo[Option[ShoppingBasketDisplay]].map {
      case Some(display) =>
        import
shoppingbasket.controllers.views.formatters.ShoppingBasketDisplayFormatter._
        Ok(Json.toJson(new
ShoppingBasketDisplayViewMapper(request).toView(display)))
      case None => NotFound
    }
  }
  private def basketLocationURl(id: String)(implicit req: Request[_]) =
    shoppingbasket.controllers.routes.BasketController.get(id).absoluteURL()
(req).stripSuffix("/").trim
}
```

3.7 Unit tests are located under test/shoppingbasket.

Example listing of the ***shoppingbasket.controllers.BasketControllerTest***

```
@RunWith(classOf[JUnitRunner])
class BasketControllerTest extends PlaySpecification {
  private def getResponse(jsonString: String): Option[Future[Result]] =
    route(
      FakeRequest(
        POST,
        "/api/shoppingbaskets",
        FakeHeaders(Seq(("Content-Type", "application/json"))),
        Json.parse(jsonString))
    )
  "BasketController#post" should {
    "return not create a shopping basket for no json body with product item" in new
WithApplication {
      val response = route(FakeRequest(POST, "/api/shoppingbaskets"))
      response must beSome.which(status(_) == BAD_REQUEST)
    }
    "return create the shopping basket request with a single available item" in new
WithApplication {
      val firstProduct = ProductGen.getProducts.head
      // make sure that this product is still available
      ProductGen.getProducts.head.copy(stock = 10)
      val jsonString =
        s""" {
           | "items": [
           |        {
           |            "product": {
           |                "id": "${firstProduct.id}"
           |            },
           |            "capacity": 2
           |        }
           |    ]
           | }
        """.stripMargin
      val response = getResponse(jsonString)
      response must beSome.which(status(_) == CREATED)
    }
    "return an error if the product is not available" in new WithApplication {
      val firstProduct = ProductGen.getProducts.head
      // make sure that this product is not available
      ProductGen.getProducts.head.stock = 0
      val jsonString =
        s""" {
             | "items": [
             |        {
             |            "product": {
             |                "id": "${firstProduct.id}"
             |            },
             |            "capacity": 2
             |        }
             |    ]
             | }
        """.stripMargin
      val response = getResponse(jsonString)
      response must beSome.which(status(_) == BAD_REQUEST)
    }
    "return an error if no product is available in the request" in new
WithApplication {
      val jsonString =
        """ {
           | "items": [
           |        {
           |            "capacity": 2
           |        }
           |    ]
           | }
        """.stripMargin
      val response = getResponse(jsonString)
      response must beSome.which(status(_) == BAD_REQUEST)
    }
    "return an error if the product does not exists" in new WithApplication {
      val jsonString =
        """ {
           | "items": [
           |        {
           |            "product": {
           |                "id": "ps4"
           |            },
           |            "capacity": 2
           |        }
```

```
               |     ]
               | }
            """.stripMargin
         val response = getResponse(jsonString)
         response must beSome.which(status(_) == BAD_REQUEST)
      }
   }
}
```

4. Running and testing the Service

There are 2 options: from CLI located in project root directory: **sbt run** or from IntelliJ IDEA using the Play2Run button.

After booting up, one can fire requests to the API via Postman. Test collections are located in project source directory.

The comple source code can be found in GitHub: https://github.com/cristinanegrean/ms-shopping-basket-scala

What's next? Defining the ShoppingBasket as well as an Entity, Using MySQL/PostgreSQL to store Product and ShoppingBasket entities and developing a client for the ShoppingBasket REST API.