REPORT ON THE PLAYFRAMEWORK

It's the second report, it will evolve as researshes go further.

Play ! Framework have all the necesary tools to make a modern and modulary web application. It allow us to link the different web devlopment methods such as : HTTP protocol, REST (Representational State Transfer), HTML, JavaScript, JSON.

Play has the same architecture than REST :

- Interface gestion and data gestion are separated due to the MVC (Model-View-Controller) developing system. Also support the compilation of assets operated in the browser to use it as a development platform.
- He doesnt keep states between queries(stateless) wich avoid bad gestion habits from the beginning of web devlopment (memory filling, losing your session after a certain time of use). Once again, the browser is put to use by exploiting its local storage capabilities.
- Application ressources are identified in a unique way and predefined because of the route gestion system, which promote the natural habits of these to be hidden by the browser and proxies to facilitate scalability.

Installation and configuration of environment variables

To run the Play framework, you need Java 5 or later. If you wish to build Play from source, you will need the Git source control client to fetch the source code and Antto build it.

Be sure to have Java in the current path

Download the latest Play binary package and extract the archive. For convenience, you should add the framework installation directory to your system PATH.

TypeSafeActivator

It is directly given by the official website who help to use Play more efficiency.

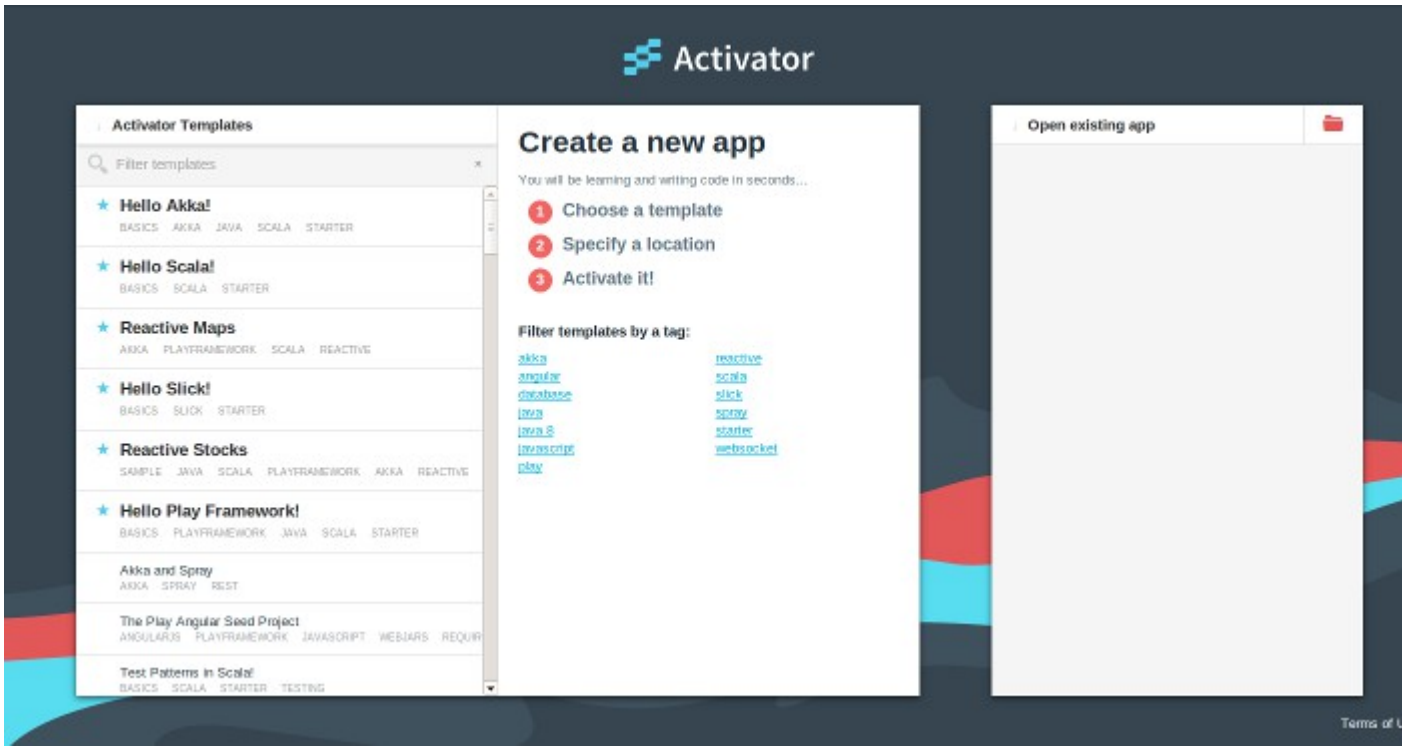
Here's how it works :

Create An Application Using the Web Interface.

Once installed you can run Typesafe Activator's web interface either from a file browser or from a command line. From a file browser, double-click on either the activator (for Mac & Linux) or activator.bat (for Windows) script. From a command line, run the script with a ui argument:

```
activator ui
```

This will start a local web server that can be reached at: http://localhost:8888



Inside your browser you can now create a new application based on one of the 281 templates. Both Typesafe and the community have contributed templates to Activator.Learn more about how to contribute your own template.

Once you have selected a template and optionally entered a name and location, select the "Create" button to have your new application created.

Create An Application Using the Command Line

If you'd prefer to create a new application from a command line, run:

```
activator new
```

You will then be prompted to enter an application name and the template name to start with. The output will look similar to this:

```
Enter an application name
> hello-scala

The new application will be created in /home/typesafe/Desktop/hello-scala

Enter a template name, or hit tab to see a list of possible templates
>
hello-akka      hello-play      hello-scala      reactive-stocks
> hello-scala
OK, application "hello-scala" is being created using the "hello-scala" template.

To run "hello-scala" from the command-line, run:
/home/typesafe/Desktop/hello-scala/activator run
```

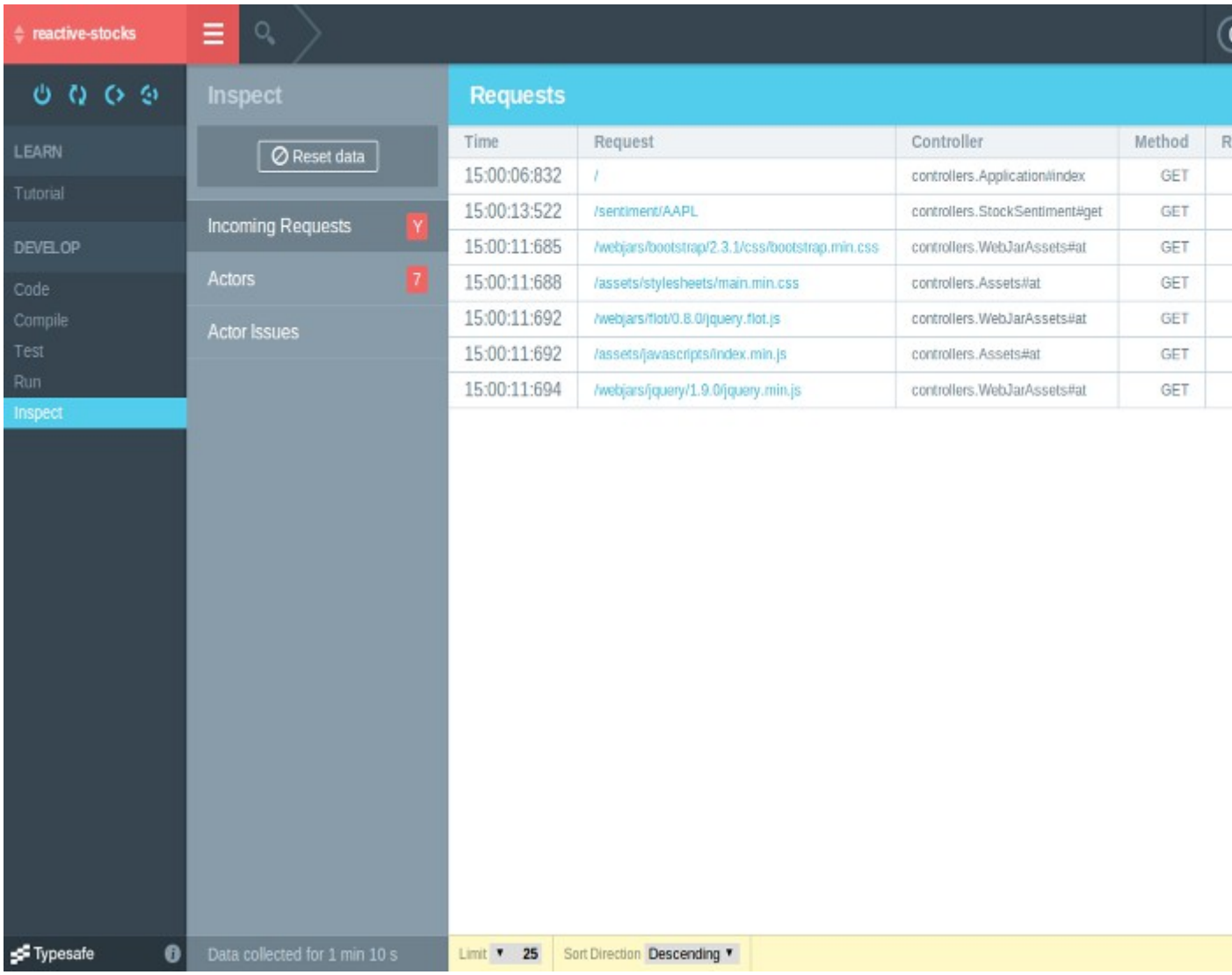
```
To run the test for "hello-scala" from the command-line, run:
/home/typesafe/Desktop/hello-scala/activator test

To run the Activator UI for "hello-scala" from the command-line, run:
/home/typesafe/Desktop/hello-scala/activator ui
```

Open an Existing Application Using the Web Interface

Existing applications can be opened by running `activator` or `activator.bat` from a project's root directory. If the Activator UI is already running, then openhttp://localhost:8888/home in your browser and either select a known existing app, or folder icon next to **Open existing app** to browse to an existing app.

Working with Applications in the Activator UI



Once you have created or opened an application in the Activator UI you can:

- Read the Tutorial
- Browse & edit the code *(select **Code**)*
- Open the code in IntelliJ IDEA or Eclipse *(select **Code** then the gear icon)*
- See the compile output *(select **Compile**)*
- Test the application *(select **Test**)*
- Run the application *(select **Run**)*
- Inspect the application *(select **Inspect**)*
- Create a new application or open an existing one *(select the application's name in the top left then select **Manage Applications**)*

Don't forget that when you open your app an RUNING_PID file appears in the activator folder where console is. You have tout delete it every time otherwise you cannot open later an app. It will fail.

Application presentation:

Yoann and I like to cook so we decided to make an application for recipes called « smart recipes ». https://github.com/mkiam/AvansSoftwareOfficial

The functionality of our application that will allow us to take hand on play while respecting the tasks that have been set are:
An authentication login+ password, facebook, twitter, google+ , system
An account creation systemloggout system and secured authentificatio,
A dynamic profile according to each user
A favorite recipes system
Unit tests
The ability to add recipes
Part of intelligent search from such simple ingredients and region around the world.

First step with hello world :

We made an application Hello World based on this app :
https://www.playframework.com/documentation/1.2.x/firstapp

Here's what we discovered :

Route file

The main entry point of your application is the conf/routes file. This file defines all of the application's accessible URLs. If you open the generated routes file you will see this first 'route':

```
GET                     /                           Application.index
```

That simply tells Play that when the web server receives a **GET** request for the / path, it must call the Application.index Java method. In this case, Application.indexis a shortcut for controllers.Application.index, because the controllers package is implicit.

When you create standalone Java applications you generally use a single entry point defined by a method such as:

```
public static void main(String[] args) {
  ...
}
```

Controlers

A Play application has several entry points, one for each URL. We call these methods**action** methods. Action methods are defined in special classes that we call **controllers**.

The controllers.Application.java controller looks like :

```
package controllers;

import play.mvc.*;

public class Application extends Controller {
```

```
        public static void index() {
                render();
        }

}
```

You see that controller classes extend the play.mvc.Controller class. This class provides all useful methods for controllers, like the render() method we use in the index action.

The index action is defined as a public static void method. This is how action methods are defined. You can see that action methods are static, because the controller classes are never instantiated. They are marked public to authorize the framework to call them in response of a URL. They always return void.
The default index action is simple: it calls the render() method which tells Play to render a template. Using a template is the most common way (but not the only one) to generate the HTTP response.

Templates are simple text files that live in the /app/views directory. Because we didn't specify a template, the default one for this action will be used:Application/index.html

Models

All the entity you need for you application should be hear with an particular instace and attributes with annotations sometimes. For exemple our appliication need Person who will connect and use it. So we have one model name : Person.

Here is what our class looks like :

```
package models;
```

```
import play.db.ebean.Model;


import javax.persistence.Entity;
import javax.persistence.Id;


@Entity
public class Person extends Model {
	@Id
	public Long id;
	public String login;
	public String email;
	public String name;
	public String surname;
	public String password;
	public Person(String login, String email, String name, String surname, String password) {
		this.login=login;
		this.surname = surname;
		this.email = email;
		this.name = name;
		this.password = password;
	}
	public Person(String name) {
		this.name= name;
```

```
                // TODO Auto-generated constructor stub
        }
        public static Finder<String,Person> find = new Finder<String,Person>(
                        String.class, Person.class
                        );
        public static Person authenticate(String login, String password) {
                return find.where().eq("login",login)
                                .eq("password", password).findUnique();
        }
}
```

Views

the helloworld/app/views/main.html template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>#{get 'title' /}</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <link rel="stylesheet" type="text/css" media="screen"
      href="@{'/public/stylesheets/main.css'}" />
    <link rel="shortcut icon" type="image/png"
      href="@{'/public/images/favicon.png'}" />
  </head>
  <body>
    #{doLayout /}
  </body>
</html>
```

Can you see the #{doLayout /} tag? This is where the content ofApplication/index.html will be inserted.

Application building :

using the activator website we chose the "create app" on JAVA.

Then you have to choose play-java-intro remembering to specify the destination directory.

Activator does the rest. Your application will now mentioned explicitly in the selected directory.

It was delivered with a database application to add people you could of course change in the future. Thus we also began the construction of the application

File « application.conf » and database

Play comes with an integrated database. It was therefore no need to write the sql code inside classes. You just have to uncomment the lines in file Application .conf :

db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""

and the database can now be used.

Authentication section in details and account establishment.

We must first start by adding the routes file located in the conf directory of the application line : POST /person controllers.Application.addPerson()

Indeed, as you understand it is to add a new person to the users of our application. This is the objective of the creation account.

We have not yet defined what a person in our case and we need to adapt avon method add Person our need.

For that we give to our favorite text editor if you want Eclipse (you simply go with a console in the directory created for our application and tap it the "activator eclipse" command and then import the application as project existing in eclipse) Otherwise a simple text editor like notepad ++ Just seen as Play compiles and executes itself when you save anything. He will tell you for sure if it encounters errors.

For this purpose in the package of the application models must each time define entities when they are needed. Here we have Person we will use it. So Edit this class by defining the atributes you deem useful for creating accounts. We for our part chose a login, an email, a name, a first name and a password. I suggest you add an automatic id to order the insertion in your database. Donate We need an id attribute that will be a long précéderez youID annotation of doing the necessary imports.

Also add in your class :

```
public  static Finder<String,Person> find = new Finder<String,Person>(
                        String.class, Person.class
                        );
```

it is to identify a person from his login is only a string.

Then in the application class that is in our package controller must change the addPerson method:

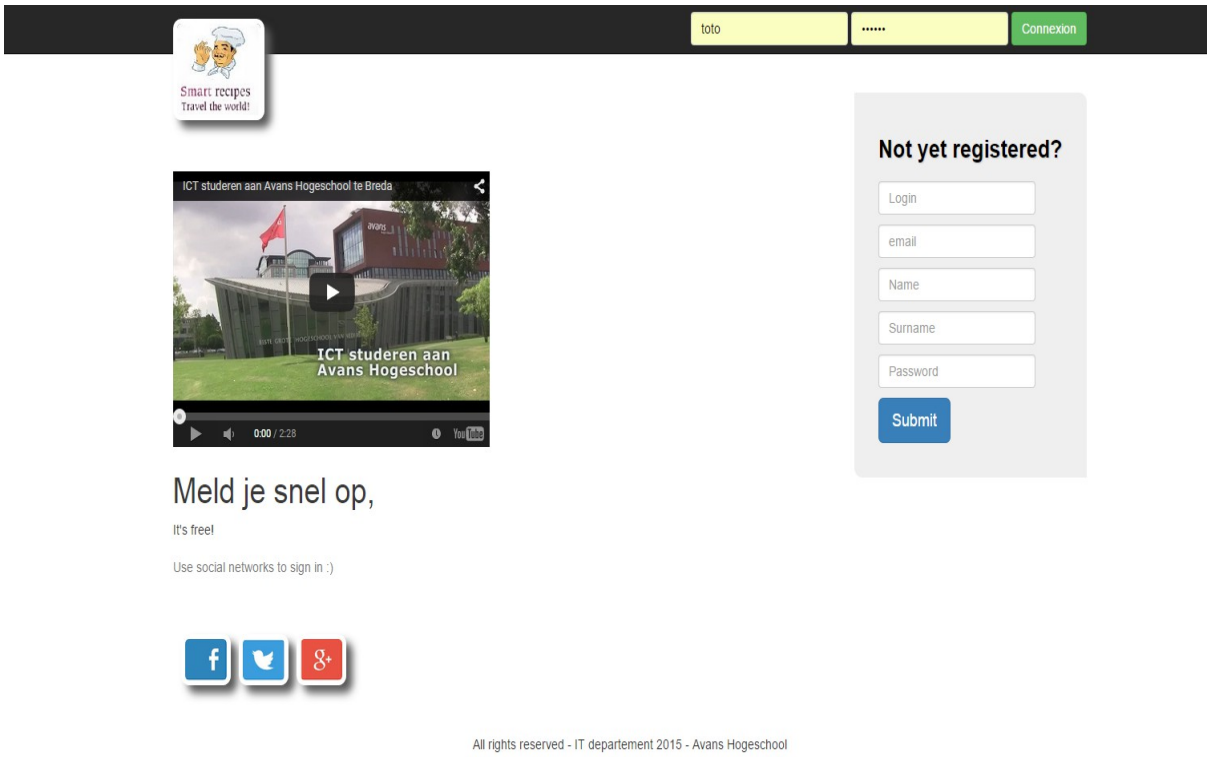
```
public  static Result addPerson() throws SQLException {

        Person person = Form.form(Person.class).bindFromRequest().get();
        person.save();
         session().clear();
       session("login", person.login);

         return redirect(
             routes.Application.menu()
                 );
   }
```

the last thing we have left to do is create a form in the index page. Scala. Html package views

This is what ours looks like with our own css:



You will need to create your form and put a specific action to realize when it is validated:

```
<form method="POST" action="@routes.Application.addPerson()">
```

Let authentication:

First authentication requires querying the database on the identity due the user. Ie whether the single login and pasword combination exists in it

As before you must add a new line routes file

```
POST    /login                  controllers.Application.authenticate()
```

then define this method: / this is to get our form in our html page

```
Form<Login> loginForm = Form.form(Login.class).bindFromRequest();
```

//to take care of errors that can occurs

```
        if (loginForm.hasErrors()) {
            return badRequest(index.render(loginForm));
        } else {
            //one session to store user login to know if he is connected and to get his login
if we need
            session().clear();
            session("login", loginForm.get().login);
//you can redirect user if he create his account to an menu page if you created one
            return redirect(
              routes.Application.menu()
            );
        }
    }
```

An inner class for authentication information retrieved

```
 // -- Authentication


    public static class Login {


        public String login;
        public String password;

        public String validate() {
            if (Person.authenticate(login, password) == null) {
```

```
            return "Invalid user or password";
        }
        return null;
    }

}
```

in the Person class we must also add the authenticate method that queries the db

```
public static Person authenticate(String login, String password) {
            return find.where().eq("login",login)
                .eq("password", password).findUnique();
        }
```

to end up as just now we have a form; There is a bit different: already added before the html tag of your login page:

```
@(form: Form[Application.Login])
```

ensuite pour le formuliare il faut procéder comme suit :

```
@helper.form(routes.Application.authenticate) {
                                    <div class="form-group">

                                        <input type="text" required id ="login"
name="login" class="form-control login"
                                               placeholder="Login">
                                    </div>
                                    <div class="form-group">
                                        <input type="password" required
id="password" name="password" class="form-control password"
```

placeholder="Password">

}

normally if you add a new user and / or you try to connect it should redirect you to your menu.

Secured authentification

Le but ici est de s'assurer que l'utilisateur est bien authentifié quand il s'enfonce en profondeur sur le site

Pour cela il faut crer une nouvelle classe :

avec les imports voici ce que ça donne :

```
package controllers;

import play.*;
import play.mvc.*;
import play.mvc.Http.*;
import models.*;

public class Secured extends Security.Authenticator {

    @Override
//to see if the loggin is coorectly in the session : it should be if the user //successfuly authenticate
```

```
    public String getUsername(Context ctx) {

return ctx.session().get("login");
    }

    @Override
//if he we don't find the login in the session we send him to the login page to /try again
    public Result onUnauthorized(Context ctx) {
        return redirect(routes.Application.index());
    }
}
```

So to render any page one the website you should add one special annotation :

```
@Security.Authenticated(Secured.class)
```

as example menu method :

```
@Security.Authenticated(Secured.class)
    public static Result menu() {
        return ok(menu.render());
    }
```

Loggout

The user can logout if he finish to use the site to be sure that nobody can use his session after him.

For this step we should make an loggout method in our Application class

the flash notion appears wich consists of notify the user that loggout is done.

Look at the method :

```
public static Result logout() {
    //we empty the session to oblige an next user to connect and to not acess
//to last user data
session().clear();
//flash method
    flash("success", "You've been logged out");
    return redirect(
        routes.Application.index()
    );
  }
```

comment an recipes

add and search an recipes

add an favorites recipes

account preferences

webservices

Strenghts of Playframework

-database is provided

-play compile automaticly and detect errors as soon as you save

-Play and unit tests

-you can run the serveur directly to access to your web pages

-You can use a simple text editor

-You can add evolutions if you want like another database, JPA system, plugin that you need and that play doesn't provide.

Difficulties encountered :

-Class with same method from another didn't work because of a class problem certainly.

Play limits :

The fact that play is mostly developped in scala make the research in the source code difficult.