

Play.Rules Return

Contents

1	Remerciements	5
2	Préface	6
2.1	Objectifs	6
2.2	Qui sommes-nous ?	6
2.3	Dédicaces	6
2.4	Avant-propos	6
2.5	Remarques	7
3	Introduction - Pourquoi Play!>	7
4	Installation	8
4.1	Prérequis	8
4.2	Modification sous OSX	9
4.3	Modification sous Linux	9
4.4	Modification sous Windows	10
4.5	Vérification	10
5	1er contact	11
5.1	Génération du squelette de l'application	11
6	Paramétrage de l'IDE	17
6.1	Paramétrage d'IntelliJ	17

7	On code !!!	20
7.1	Très important !!!	21
7.2	Construisons les bases de notre application : paramétrages	21
7.2.1	Il nous faut une base de données	21
7.2.2	Il nous faudra des modèles	21
7.3	Les modèles	22
7.3.1	Création d'un 1er modèle : Category	22
7.3.2	Création d'un 2ème modèle : Bookmark	23
7.4	Les contrôleurs	25
7.4.1	Création du contrôleur Bookmarks	25
7.4.2	Création du contrôleur Categories	26
7.4.3	Modification du contrôleur Application	27
7.5	Les vues	30
7.5.1	Modification de la vue principale : index.scala.html	30
8	Un peu de cosmétique	34
8.1	Prérequis	34
8.2	Utilisation	35
8.2.1	Modifions le code de <code>main.scala.html</code>	35
9	Charger des données au démarrage	38
10	Validation des données	40
10.1	Enrichissement du modèle et vérification des données	40
10.2	Validation côté client	43
10.2.1	Installation de Play2-HTML5Tags	43
10.2.2	Déclaration de Play2-HTML5Tags	44
10.2.3	Utilisation de Play2-HTML5Tags	45

11 Gestion de l'authentification	49
11.1 Complétons la classe User	50
11.2 Création d'un contrôleur Secured	51
11.3 Allons modifier le contrôleur Application	51
11.4 Création d'un contrôleur Authentication	52
11.5 Allons modifier les routes	53
11.6 Allons modifier main.scala.html et index.scala.html	54
11.6.1 main.scala.html	54
11.6.2 Il faut aussi modifier index.scala.html	56
11.7 Allons créer un formulaire de login / Vue	56
11.8 Allons ajouter des users dans initial-data.yml	57
11.8.1 Ajoutons des utilisateurs :	57
11.8.2 Modifions Global.java	57
12 Services (JSON)	61
12.1 Primo :	61
12.2 Création de notre service JSON	62
12.2.1 Allons modifier le contrôleur Bookmarks	62
12.3 Utilisation du service JSON	64
12.3.1 Directement avec l'url	64
12.3.2 Plus utile : via une requête ajax	65
12.4 Sécurisation	68
12.4.1 SecuredJson.java	69
12.4.2 Modification de l'annotation dans Bookmarks.java	70
12.4.3 Testons	70
12.4.4 Mais comment puis m'authentifier via une requête ajax ???	71
12.5 Utilisons tout ça	76
12.5.1 Nouveau Contrôleur : SingleApp.java	77
12.5.2 routes	77
12.5.3 Nouvelle vue : mainPage.scala.html	77
12.6 Testons	82
12.7 Conclusion	87

13 Les assets dans Play	88
13.1 Assets ???	88
13.2 Assets “compilés”	88
13.3 Mise en oeuvre	89
13.3.1 Préparation	89
13.3.2 Coffeescript	89
13.3.3 Less	94
14 Créer des templates Play	98
14.1 Pourquoi ?	98
14.2 Prérequis : Installation de conscript	99
14.3 Installation de giter8	99
14.4 Création/Préparation de votre template projet	99
14.4.1 Dans /project/plugin.sbt :	100
14.4.2 Dans /src/main/g8/default.properties :	100
14.4.3 Dans /src/main/g8/\$application_names\$/ :	100
14.4.4 Vérifier que /src/main/g8/\$application_names\$/project/Build.scala a bien le contenu suivant:	100
14.4.5 Dans /src/main/g8/\$application_names\$/conf/application.conf :	101
14.4.6 Vérifier que dans /src/main/g8/\$application_names\$/project/plugin.sbt :	101
14.4.7 Vérifier que dans /src/main/g8/\$application_names\$/project/build.properties :	101
14.4.8 Poussez moi tout ça sous github	102
14.5 Installation d’un template :	102
15 Coder son application Play!> en Scala	102
15.1 Intro	102
15.2 Liens entre Stateless et programmation fonctionnelle	102
15.3 Iteratee	104
15.3.1 Un exemple avec Comet	104
15.3.2 Controleur	105

15.3.3 D'autres exemples intéressants	106
---	-----

PLAY.RULESRETURN

September 6, 2012



Initiation & Recettes
Play2! ► en douceur
Par @k33g_org & @loic_d

1 Remerciements

//TODO: à compléter

- @kraco_fr : pour ses “tips Play”
- @LyonJUG : pour son équipe qui “maintient l’envie de continuer”
- @ndeverge : le 1er à avoir fait une pull request sur le projet :)

2 Préface

Ce livre est conçu et écrit par @k33g_org et @loic_d (nos petits noms sur Twitter). Il est complètement open source. Faites en ce que vous voulez. Vous pouvez participer par le biais des pull requests et issues du repository GitHub : <https://github.com/3monkeys/play.rules> (répertoire : `livre.play.deux`), si vous détectez des fautes, avez des idées, des remarques; etc. ...

Si cet e-book vous plaît, n’hésitez pas à nous le faire savoir (nous faisons ça sur notre temps personnel de façon purement bénévole).

2.1 Objectifs

- Démontrer que Play2!>; est facile à apprendre
- Que faire des Webapp en Java (et Scala), ce n’est pas si difficile (même si vous ne venez pas de Java)
- ... et ce, quel que soit votre niveau

2.2 Qui sommes-nous ?

//TODO

2.3 Dédicaces

//TODO

2.4 Avant-propos

A l’heure actuelle, ce livre est encore incomplet. Cependant nous essayons de lui conserver une certaine structure et de ne publier que des choses utilisables : vous n’aurez pas de chapitres qui ne servent à rien (l’e-book est régénéré uniquement dans le cas d’un ajout d’un chapitre complet ou de correction).

Donc, beaucoup de TODO mais quand même du contenu pour “s’y mettre”.

- Pendant le “chantier”, la structure et le contenu peuvent changer.
- Les images doivent être retaillées

En espérant que cet e-book vous serve & vous fasse aimer Play2!>.

2.5 Remarques

Les premiers chapitres traitent de la version **Java** de Play2!>;, mais la version **Scala** sera elle aussi abordée.

3 Introduction - Pourquoi Play!>

Play Framework dans sa version 1 a apporté un vent de révolution au monde des frameworks Web Java.

Les principaux points d’innovation étaient à l’époque

- Une productivité grandement améliorée (notamment via déploiement à chaud des modifications de code)
- Une architecture simplifiée et centrée sur les modèles (tout en facilitant l’évolutivité et la maintenabilité)
- Une “coeur” modulaire et extensible
- Un framework basé sur REST
- Un framework permettant de monter très facilement en charge (scalabilité)
- Et enfin une architecture entièrement stateless côté serveur

L’architecture stateless consiste à ne garder aucun état sur le serveur. Une telle architecture propose de nombreux avantages. Si notre application se trouve derrière un load balancer, on peut ajouter un serveur, en couper un pour le mettre à jour... sans aucune conséquence pour le client qui ne risque pas de perdre sa session pendant l’opération... un utilisateur pourra être dirigé aléatoirement vers un serveur ou un autre à chaque requête, pas besoin de mettre en place une session HTTP distribuée (usine à gaz...)

Cela va de paire avec l’aspect RESTful : Prenez n’importe quelle page dont l’URL d’accès est définie par une méthode HTTP `Get`. Vous pourrez toujours mettre cette page en bookmark dans votre navigateur et y revenir plus tard. Le serveur ne conservant pas de session, vous n’aurez pas de surprise, vous obtiendrez toujours le même résultat. Si vous avez déjà utilisé des frameworks stateful comme JSF, vous avez sûrement remarqué que ce n’était pas simple

avec une telle architecture. Note : une page bookmarkable sera aussi plus facile à mettre en cache et sera compatible avec le bouton “back” du navigateur (là encore je vous renvoie à JSF...).

Note : On peut bien sûr gérer des informations sur la session utilisateur côté client, via un cookie (signé et crypté) ou via le stockage local HTML5.

Play!> 2 conserve tous ces principes et en apporte de nouveaux...

- Toute requête est potentiellement asynchrone
- Tout est typé et vérifié à la compilation, même les templates de vues et les fichiers de définitions des routes HTTP (nous verrons plus loin de quoi il s’agit).

L’aspect asynchrone est particulièrement intéressant. En ce moment on parle beaucoup de “real time web applications”. Certains n’y voient qu’un buzzword, mais le concept est vraiment intéressant et Play!> 2 permet de faire ce genre de choses facilement. On peut par exemple, depuis une requête HTTP, récupérer une multitude d’informations en appelant différents services externes (twitter, linkedin ...), mixer ces infos et les pousser vers le client, tout ça en mode non bloquant, c’est à dire que le navigateur ne reste pas en attente entre la demande d’informations et la réponse. Après la requête, la connexion est libérée, puis une autre connexion sera créée dans le sens inverse (server -> client) une fois que le résultat sera calculé. Nous verrons ces concepts plus en détail tout à la fin de cet ebook.

4 Installation

version de Play!> utilisée : 2.0.1

Qu’allons nous voir ? ... Installation de Play2!>

- sous OSX
- sous Linux
- sous Windows

4.1 Prérequis

- Vous devez avoir Java sur votre machine : attention, vous devez avoir le JDK 6 minimum (Eh oui, vous avez le 7 aussi) : donc attention à ne pas proposer Play2!> à des clients étant encore en JDK 5 (si, ça existe encore !)

- Téléchargez Play!> sur <http://www.playframework.org/>
- Dézippez l'archive dans un répertoire

Ensuite, il faut modifier votre path.

4.2 Modification sous OSX

Dans une console (Terminal), tapez la commande suivante :

```
sudo pico ~/.bash_profile
```

Puis ajoutez la ligne suivante dans votre fichier de configuration :

```
export PATH=$PATH:/endroitOuvousAvezDezippePlay/play-2.0.1
```

Sauvegardez (sous pico, c'est **Ctrl+o**) et quittez l'éditeur, fermez votre Terminal.

où **endroitOuvousAvezDezippePlay** est le chemin vers Play!> et **play-2.0.1** le nom du répertoire dans lequel il y a les éléments constitutifs du framework (je laisse le numéro de version car il m'arrive de travailler sur plusieurs versions).

4.3 Modification sous Linux

Dans une console (Terminal), tapez la commande suivante :

```
vi ~/.profile
```

Puis ajoutez les lignes suivantes à la fin de ce fichier :

```
export PLAY_HOME=/endroitOuvousAvezDezippePlay/play-2.0.1
export PATH=$PLAY_HOME:$PATH
```

Sauvegardez et quittez l'éditeur (sous vi, c'est **ESCAPE**, **:**, **wq**), fermez votre console.

où **endroitOuvousAvezDezippePlay** est le chemin vers Play!> et **play-2.0.1** le nom du répertoire dans lequel il y a les éléments constitutifs du framework (je laisse le numéro de version car il m'arrive de travailler sur plusieurs versions).

4.4 Modification sous Windows

Modifier les variables d'environnement de Windows, via **Panneau de configuration\Système et sécurité\Système**, puis **Paramètres systèmes avancés** sur la gauche. Dans la boîte de dialogue qui s'affiche, cliquer le bouton **Variables d'environnement...** en bas.

Ajouter une nouvelle **Variable système** :

- Nom de la variable = `PLAY_HOME`
- Valeur de la variable = `endroitOuvrezVousAvezDezippePlay\play-2.0.1`

Puis modifier la valeur de la variable `Path` en ajoutant `%PLAY_HOME%;` au début (ne pas oublier le `'` !).

Cliquer sur tous les boutons **OK** pour fermer les différentes boîtes de dialogue.

4.5 Vérification

Nous allons vérifier la bonne installation du framework. Ouvrez une nouvelle fois votre Console ou Terminal (il faut que cela soit une nouvelle session pour la prise en compte de la modification du `path`), et tapez la commande suivante :

```
play help
```

Cela “mouline” un peu car Play2!> télécharge quelques dépendances. Vous devriez obtenir ceci :

Play!> vous propose un nom par défaut pour votre application : acceptez

```

k33g-orgs-MacBook-Air:w k33g_org$ play new bookmarks

  _ _ _ _ _
 | ' _ \ | / _ ' | | | _ |
 | _ \ | / _ \ | \ ( )
 | _ | _ | _ | _ |
 | _ | _ | _ | _ |

play! 2.0.1, http://www.playframework.org

The new application will be created in /Users/k33g_org/Dropbox/Public/play.rules.tmp/w/bookmarks

What is the application name?
> bookmarks

```

Play!> vous demande quel type de projet vous souhaitez générer, choisissez la version Java (deuxième choix donc) et validez :

```
1. Default (sh)
The new application will be created in /Users/k33g_org/Dropbox/Public/play.rules.tmp/w/bookmarks

What is the application name?
> bookmarks

Which template do you want to use for this new application?

1 - Create a simple Scala application
2 - Create a simple Java application
3 - Create an empty project

> 
```

C'est terminé :

```
1. Default (bash)

The new application will be created in /Users/k33g_org/Dropbox/Public/play.rules.tmp/w/bookmarks

What is the application name?
> bookmarks

Which template do you want to use for this new application?

1 - Create a simple Scala application
2 - Create a simple Java application
3 - Create an empty project

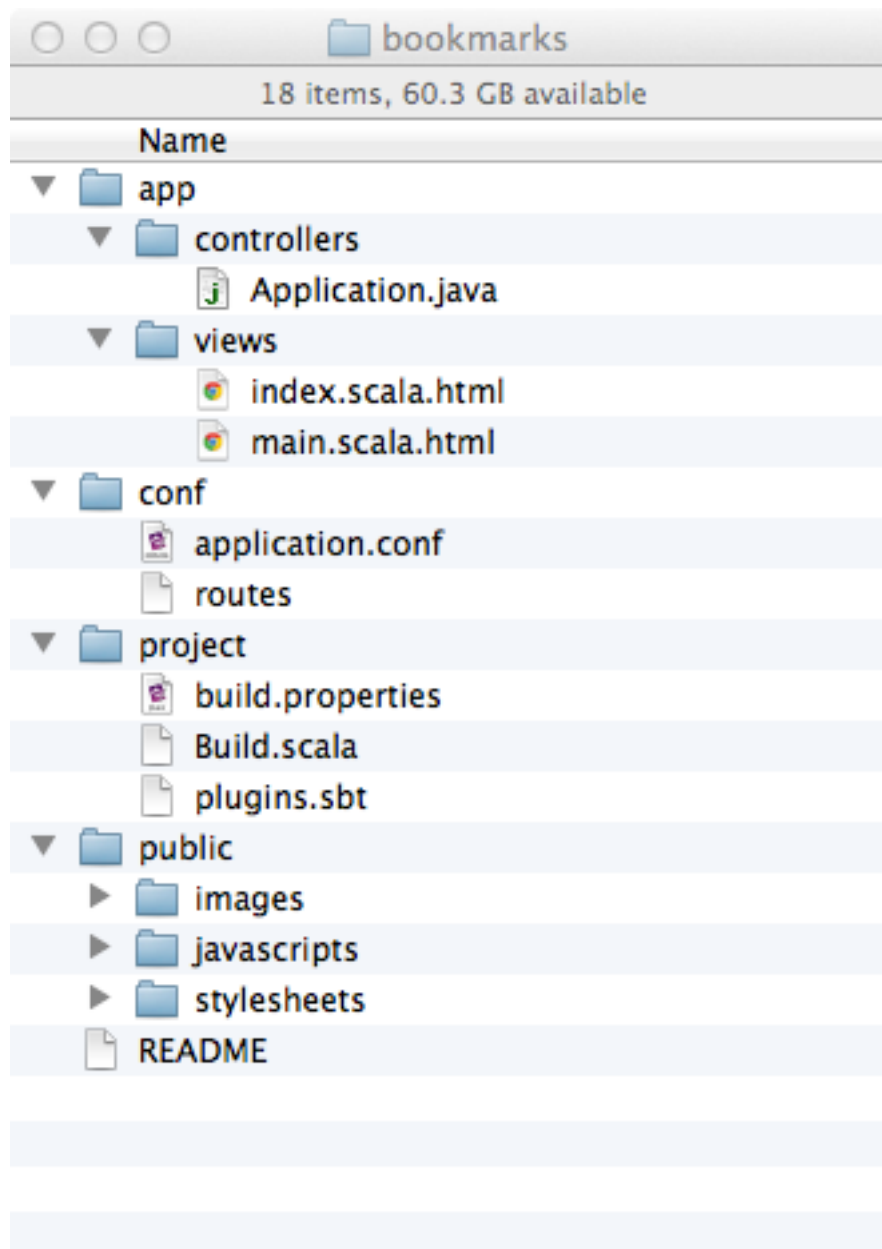
> 2

OK, application bookmarks is created.

Have fun!

k33g-orgs-MacBook-Air:w k33g_org$
```

Si vous aller jeter un coup d'oeil dans votre répertoire, vous pourrez vérifier que Play!> a généré toute l'arborescence applicative nécessaire :



Pour plus de détail sur l'anatomie d'une application Play!>, allez faire un tour par là : <http://www.playframework.org/documentation/2.0.1/Anatomy>

Lançons donc notre application pour être réellement sûr que nous avons tout ce qu'il faut. pour cela, tapez les commandes :

```
cd bookmarks
```

play

La première fois, cela risque de prendre du temps, car Play!> télécharge divers éléments dont il a besoin pour fonctionner. Patientez un peu. Vous arrivez ensuite sur un “prompt” qui prend le nom de votre application [bookmarks] :

```

1. Default (sh)

k33g-orgs-MacBook-Air:w k33g_org$ cd bookmarks/
k33g-orgs-MacBook-Air:bookmarks k33g_org$ play
Getting org.scala-tools.sbt sbt_2.9.1 0.11.2 ...
:: retrieving :: org.scala-tools.sbt#boot-app
  confs: [default]
    37 artifacts copied, 0 already retrieved (7324kB/93ms)
[info] Loading project definition from /Users/k33g_org/Dropbox/Public/play.rules.tmp/w/bookmarks/
project
[info] Set current project to bookmarks (in build file:/Users/k33g_org/Dropbox/Public/play.rules.
tmp/w/bookmarks/)

  _ _ | _ _ _ _ | _
 | ' _ \ | / _ ' | | | _ |
 | _ \ | / _ \ | _ \ ( )
 | _ | _ | _ | _ | _ |

play! 2.0.1, http://www.playframework.org

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[bookmarks] $ 

```

Tapez **run** et validez :

```

1. Default (sh)

  _ _ | _ _ _ _ |
  | ' \ | / _ ' | | |
  | _ / | \ _ \ | \ ( )
  | _ | _ _ | _ \
  | _ | _ _ | _ \

play! 2.0.1, http://www.playframework.org

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[bookmarks] $ run

[info] Updating {file:/Users/k33g_org/Dropbox/Public/play.rules.tmp/w/bookmarks/}bookmarks...
[info] Done updating.
--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on port 9000...

(Server started, use Ctrl+D to stop and go back to the console...)

```

Vous pouvez lire que Play!> a démarré une application web sur laquelle vous pouvez vous connecter via <http://localhost:9000>. Allons-y. En parallèle, côté serveur, ça compile :

```
1. Default (sh)

play! 2.0.1, http://www.playframework.org

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[bookmarks] $ run

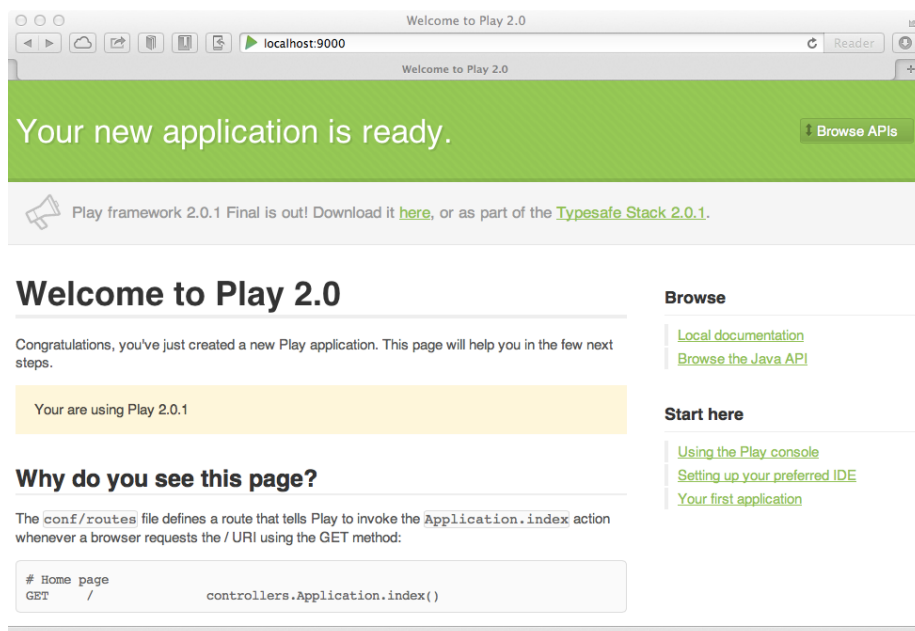
[info] Updating {file:/Users/k33g_org/Dropbox/Public/play.rules.tmp/w/bookmarks/}bookmarks...
[info] Done updating.
--- (Running the application from SBT, auto-reloading is enabled) ---

[info] play - Listening for HTTP on port 9000...

(Server started, use Ctrl+D to stop and go back to the console...)

[info] Compiling 4 Scala sources and 2 Java sources to /Users/k33g_org/Dropbox/Public/play.rules.
tmp/w/bookmarks/target/scala-2.9.1/classes...
[info] play - Application started (Dev)
[]
```

Et au bout de quelques instants, si tout va bien, vous obtenez cette page dans votre navigateur :



6 Paramétrage de l’IDE

Qu’allons nous voir ?

- *Comment paramétrer un IDE pour “bosser” facilement avec Play2!>*

Pour le moment je ne parle que d’IntelliJ. Sachez cependant qu’avec un peu d’habitude, il est possible de “faire du Play” avec un bon éditeur de texte comme SublimeText, UltraEdit, Notepad++, TextMate, ...

Nous avons donc une installation de Play2!> et un squelette d’application opérationnels. Avant d’aller plus loin, nous allons paramétrer un IDE pour nous faciliter le développement (il est aussi possible d’utiliser un simple éditeur de texte). Play!> peut fonctionner avec plusieurs IDE :

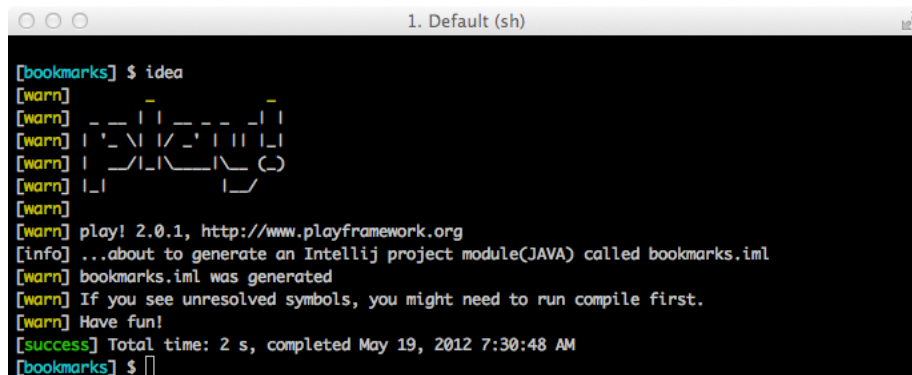
- IntelliJ
- NetBeans (il reste à ce jour encore quelques réglages/développements à réaliser)
- Eclipse

Je vous propose d’utiliser la version Community d’IntelliJ (qui semble faite pour Play!>), qui a l’avantage d’être gratuite et puissante à la fois. Pour les autres IDE, allez faire un tour sur le site de Play!>, tout est expliqué.

//TODO : liens etc ...

6.1 Paramétrage d’IntelliJ

Pour cela nous devons transformer notre arborescence projet en “module IDEA”. Tout d’abord, arrêtez votre application : faites un **Ctrl+c**, puis relancez Play!> : **play** (vous êtes toujours dans le répertoire de votre application). Une fois que vous êtes revenu au prompt **[bookmarks]**, tapez la commande **idea** et validez. Vous obtenez ceci :



```
1. Default (sh)

[bookmarks] $ idea
[warn]
[warn]  _ _ | | _ _ _ _ _ | |
[warn] | ' _ \ | / _ ' | | | |
[warn] | _ / | _ \ _ \ _ \ ( )
[warn] | _ | _ | _ \
[warn]
[warn] play! 2.0.1, http://www.playframework.org
[info] ...about to generate an IntelliJ project module(JAVA) called bookmarks.iml
[warn] bookmarks.iml was generated
[warn] If you see unresolved symbols, you might need to run compile first.
[warn] Have fun!
[success] Total time: 2 s, completed May 19, 2012 7:30:48 AM
[bookmarks] $
```

Play!> a généré dans le répertoire de l'application un fichier bookmarks.iml.

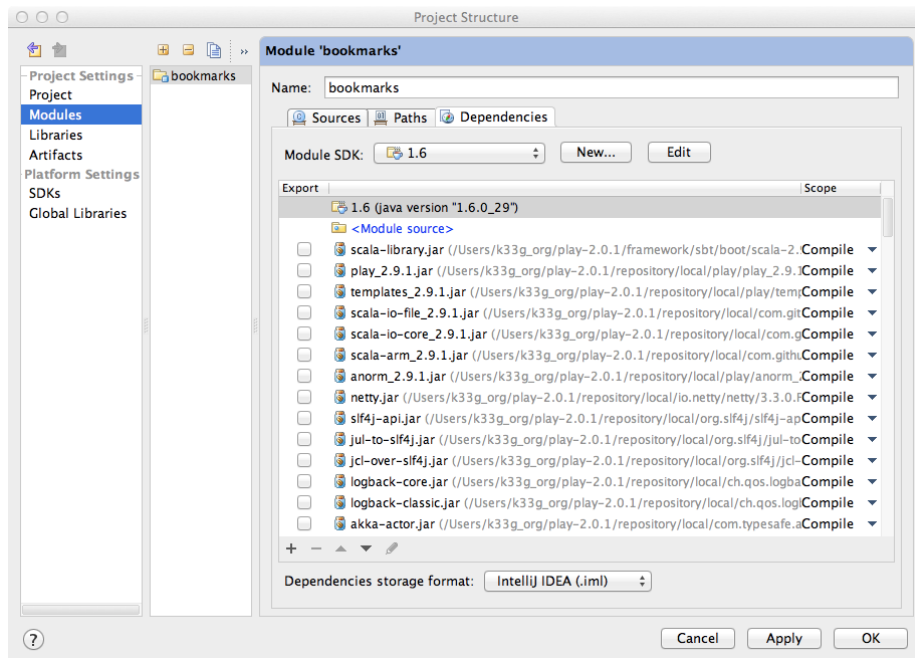
Passons au paramétrage du projet :

- Démarrez IntelliJ
- Créez un nouveau projet
- Choisir “Create project from scratch”
- Donnez un nom au projet (j’ai choisi de lui donner le même nom que mon application : bookmarks)
- Faite pointer “Project files location” sur le répertoire de votre application
- Décochez l’option “Create module”
- Cliquez sur “Finish”

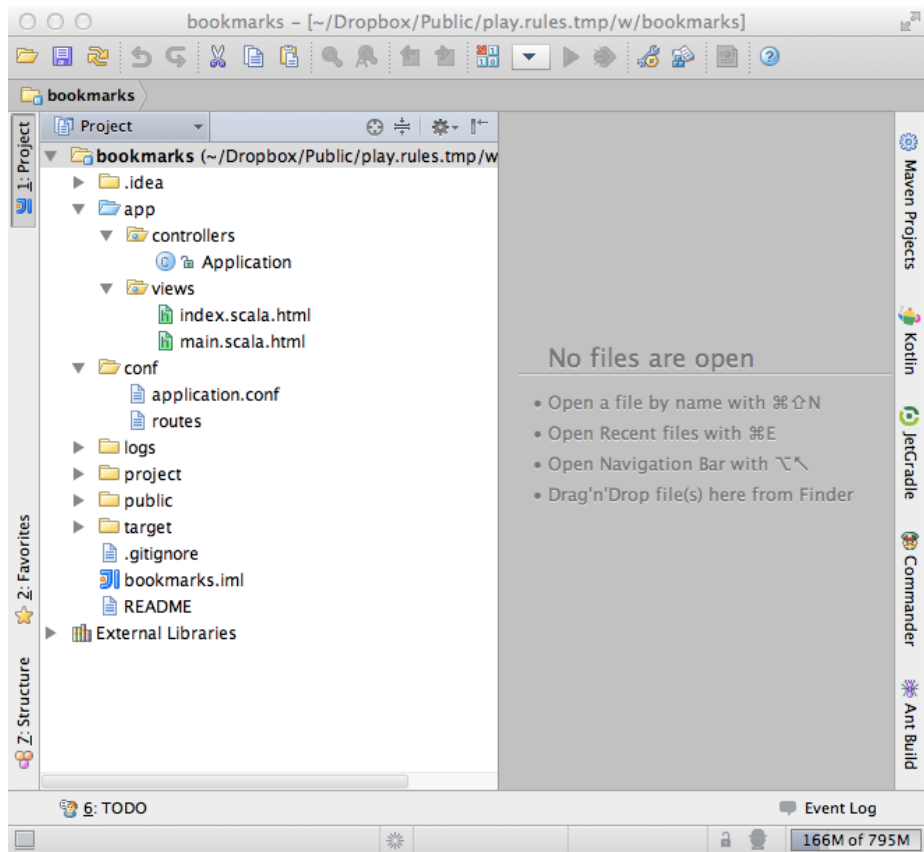
IntelliJ va vous afficher une fenêtre “Project Structure” :

- Dans la rubrique “Modules” de “Projects Settings”, ajoutez un module (utilisez l’icône “+”)
- Sélectionnez le choix “import existing module”
- “Pointez” vers le fichier bookmarks.iml
- Cliquez sur “Finish”

Vous devriez obtenir l’écran suivant (si vous avez tout fait comme il faut) :



Cliquez sur “OK”. Votre projet est prêt :



Nous sommes enfin prêts à commencer.

Si vraiment vous souhaitez utiliser un autre IDE, c'est expliqué ici : <https://github.com/playframework/Play20/wiki/IDE>

7 On code !!!

Qu'allons nous voir ? ... Comment paramétrer notre application

- déclarer la base de données embarquée proposée par Play!>
- créer nos premiers modèles
- créer nos premiers contrôleurs, les modifier
- paramétrer "les routes"
- créer notre première vue

... faites vous un bon café, et soyez attentif, cela devient sérieux.

7.1 Très important !!!

Plutôt que d'utiliser la commande `run` vous pouvez utiliser `~run` et à ce moment là Play!> compile à la volée dès qu'il détecte un changement dans le code. Cela va accélérer grandement votre travail (merci à [@kraco_fr](#) pour ça, j'étais prêt à mettre Play!> en pause tellement la compilation était devenue lente et là c'est que du bonheur.)

7.2 Construisons les bases de notre application : paramétrages

```
//TODO : expliquer ce que va faire l'application
```

7.2.1 Il nous faut une base de données

Tout d'abord, nous avons besoin d'une base de données. Nous allons aller au plus simple : Play!> “embarque” une base de donnée “H2 Database” (<http://www.h2database.com/html/main.html>) qui est une base de donnée rapide, légère, qui peut même fonctionner en mémoire et qui respecte les standards JDBC. Cela signifie que vous pouvez facilement prototyper vos applications avec cette base de données pour ensuite changer de base de manière transparente.

Pour définir notre base, allons faire un tour dans le fichier `conf/application.conf` et au niveau de la section `# Database configuration` ajoutons ceci :

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:file:play"
```

Nous avons donc expliqué à Play!>, que nous souhaitons utiliser la base de données “H2 Database” en mode fichier. Comme cela toutes nos modifications seront sauvegardées.

7.2.2 Il nous faudra des modèles

Dans un premier temps :

- Créez un répertoire `models` dans `app` car cela n'est pas fait automatiquement (pour le moment ?) (dans IntelliJ, utilisez la fonction “New Package” sur le répertoire `app` et appelez le `models`)
- Dans `conf/application.conf` modifiez la partie `# Ebean configuration` : Décommentez la ligne : `ebean.default=models.*`

Remarque : Play!> 2 utilise **Ebean** comme framework de persistance au lieu de Hibernate+JPA (comme le faisait Play!> v1). Ebean est un ORM Java qui continue à utiliser les annotations JPA (`@Entity`, `@OneToMany`, ...) pour le mapping et qui propose une API plus simple (en tous les cas plus moderne) et qui a la particularité d’être “*sessionless*”.

```
//TODO : expliquer sessionless
```

7.3 Les modèles

Nous allons étudier ça par l’exemple.

7.3.1 Création d’un 1er modèle : Category

Créez une classe `Category.java` dans `app/models` :

```
package models;

import play.db.ebean.Model;
import javax.persistence.*;

@Entity
public class Category extends Model{

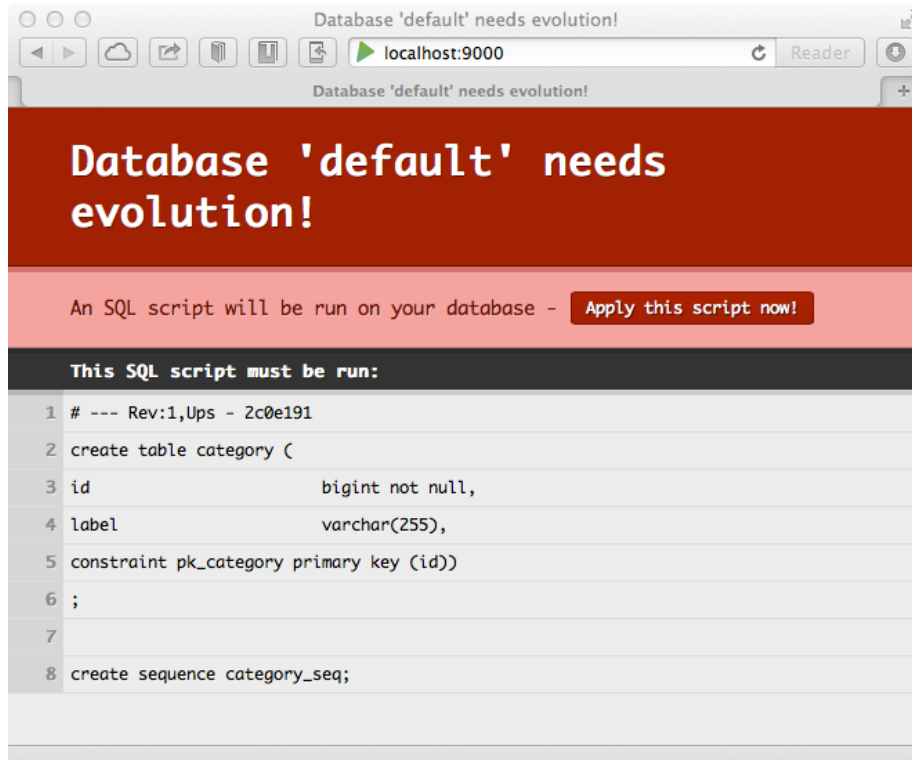
    @Id
    public Long id;
    public String label;

    public static Finder<Long, Category> find =
        new Finder<Long, Category>(Long.class, Category.class);
}
```

Remarque : Nous avons eu besoin d’importer `play.db.ebean.Model` pour pouvoir persister nos modèles, `javax.persistence.*` pour pouvoir utiliser les annotations. Nous avons fait précéder notre classe de l’annotation `@Entity` et hériter de `Model` pour pouvoir bénéficier des fonctionnalités de persistance. Et l’utilisation de l’annotation `@Id` nous permet de définir que la clé du modèle en base de données est `id`. Notez le membre `find` de type `Finder`, `Finder` est un type “apporté” par Ebean, il nous permettra d’interroger nos modèles.

Redémarrez dès maintenant l'application et connectez vous à <http://localhost:9000/> avec votre ordinateur.

Play!> détecte (c'est un peu long, je vous l'accorde) que vous avez créé un modèle et vous propose donc de créer le modèle de données (la table category dans notre cas) :



Cliquez sur “**Apply this script now !**” pour créer la structure de données dans la base. ... Et votre base de données est ainsi créée.

7.3.2 Création d'un 2ème modèle : Bookmark

De la même manière, créez un modèle Bookmark :

```
package models;

import play.db.ebean.Model;
import javax.persistence.*;

@Entity
public class Bookmark extends Model {
```

```

@Id
public Long id;
public String title;
public String url;
public String details;

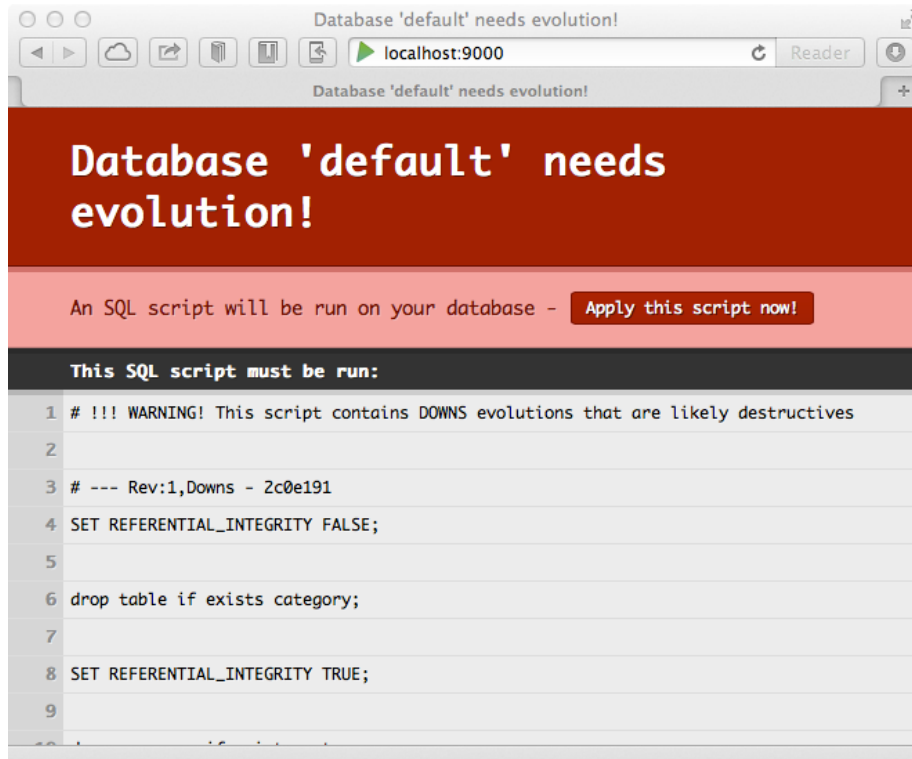
@ManyToOne
public Category category;

public static Finder<Long, Bookmark> find =
    new Finder<Long, Bookmark>(Long.class, Bookmark.class);
}

```

Remarques : Notez l'annotation `@ManyToOne` : nous expliquons à Play!> qu'un `Bookmark` peut appartenir à plusieurs `Category(ies)`.

Vous venez de créer un nouveau modèle, il est temps de “rafraîchir” à nouveau votre application. Play!> détecte la modification et vous propose encore une fois d'appliquer le script pour modifier la base de données. Acceptez :



A ce stade, nous avons nos modèles et une base de données. mettons en oeuvre la suite de notre mécanique pour pouvoir bientôt jouer justement avec ces modèles.

7.4 Les contrôleurs

Par convention, nous nommons un contrôleur d'un modèle avec le nom du modèle au pluriel, par exemple : **Bookmarks** pour le contrôleur du modèle **Bookmark** (en même temps, si vous avez envie de le nommer **BookmarksController**, rien ne vous en empêche).

7.4.1 Création du contrôleur Bookmarks

Dans `/app/controllers/` créez la classe `Bookmarks.java` :

```
package controllers;

import models.Bookmark;

import play.data.Form;
import play.mvc.Controller;
import play.mvc.Result;

public class Bookmarks extends Controller {

    public static Result add() {

        final Form<Bookmark> bookmarkForm = form(Bookmark.class).bindFromRequest();
        final Bookmark bookmark = bookmarkForm.get();

        bookmark.save();
        return redirect(routes.Application.index());

    }
}
```

Qu'avons nous fait ?

- Nous avons un contrôleur `Bookmarks` avec une méthode `add()`
- `final Form<Bookmark> bookmarkForm = form(Bookmark.class).bindFromRequest()` permet de récupérer les informations en provenance d'un POST à partir d'un formulaire html (`<form></form>`)

- `final Bookmark bookmark = bookmarkForm.get()` permet de créer une instance de `Bookmark` avec les informations en provenance du formulaire
- `bookmark.save()` permet d'ajouter le bookmark en base
- `return redirect(routes.Application.index())` fait une redirection vers `routes.Application.index()` (donc appel de la méthode `index()` du contrôleur `Application` qui se chargera d'afficher des informations dans la page, nous allons voir ça plus loin)

Modification du fichier routes

//TODO: expliquer ce que c'est qu'une route (ou pas?)

Nous allons modifier le fichier `routes` dans le répertoire `/conf` pour expliquer à Play!> que toute requête http de type POST avec une url `/bookmark/add` déclenchera la méthode `add()` du contrôleur `Bookmarks`. Donc dans le fichier `routes` ajoutez ceci :

```
# Models routes
POST /bookmark/add controllers.Bookmarks.add()
```

Votre fichier `routes` doit ressembler à ceci :

```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# Home page
GET      /                               controllers.Application.index()

# Models routes
POST /bookmark/add controllers.Bookmarks.add()

# Map static resources from the /public folder to the /assets URL path
GET      /assets/*file                  controllers.Assets.at(path="/public", file)
```

Procédons de la même manière pour créer un contrôleur `Categories`.

7.4.2 Création du contrôleur Categories

Dans `/app/controllers/` créez la classe `Categories.java` :

```

package controllers;

import models.Category;
import play.data.Form;
import play.mvc.Controller;
import play.mvc.Result;

public class Categories extends Controller {

    public static Result add() {

        final Form<Category> categoryForm = form(Category.class).bindFromRequest();
        final Category category = categoryForm.get();

        category.save();
        return redirect(routes.Application.index());

    }
}

```

Modification du fichier routes Dans le fichier routes, à la suite de notre précédente modification, ajoutons ceci :

```
POST /category/add controllers.Categories.add()
```

7.4.3 Modification du contrôleur Application

Play!> génère par défaut un contrôleur `Application` qui permet de “piloter” la page d’accueil. Si vous allez voir à nouveau le fichier `routes`, vous verrez ceci :

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

Cela signifie que dès que vous êtes à la racine de votre site (`/`, donc `http://localhost:9000`) c’est la méthode `index()` du contrôleur `Application` qui est appelée. Si vous lisez le code de `Application.java` :

```

package controllers;

import play.*;
import play.mvc.*;

```

```
import views.html.*;

public class Application extends Controller {

    public static Result index() {
        return ok(index.render("Your new application is ready.")).
    }

}
```

Vous pouvez voir que la méthode `index()` se contente de “rendre” (afficher) la vue `index` en lui passant un message (`"Your new application is ready."`). Vous trouverez la vue `index` dans le répertoire `views` sous le nom `index.scala.html`, elle contient ceci :

```
@(message: String)

@main("Welcome to Play 2.0") {

    @play20.welcome(message, style = "Java")

}
```

Petit exercice : Changez donc le message dans le contrôleur :

```
package controllers;

import play.*;
import play.mvc.*;

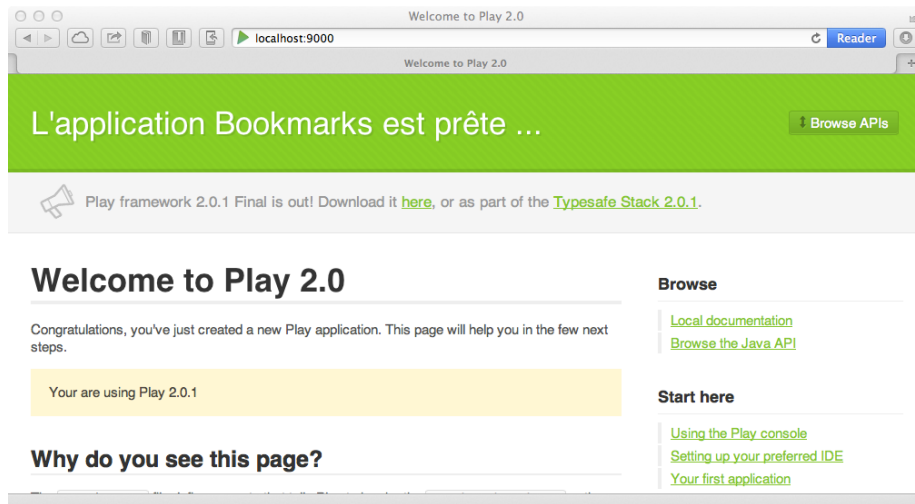
import views.html.*;

public class Application extends Controller {

    public static Result index() {
        return ok(index.render("L'application Bookmarks est prête ..."));
    }

}
```

et rafraîchissez votre page :



Maintenant, modifions vraiment Application.java Que voulons nous faire ?

En fait, je souhaite passer en paramètres de la méthode `render()` de la vue `index`, la liste des catégories et la liste des bookmarks, pour que ma vue puisse les afficher. Modifions le code du contrôleur `Application.java` de la façon suivante :

```
package controllers;

import models.Bookmark;
import models.Category;
import play.mvc.Controller;
import play.mvc.Result;
import views.html.index;

public class Application extends Controller {

    public static Result index() {

        return ok(index.render(
            "Vous pouvez commencer à saisir ...",
            Bookmark.find.fetch("category").orderBy("title").findList(),
            Category.find.orderBy("label").findList()
        ));
    }
}
```

```
//TODO : expliquer le fetch
```

Et allons tout de suite modifier notre vue, pour enfin avoir quelque chose à montrer

7.5 Les vues

7.5.1 Modification de la vue principale : index.scala.html

Notez que : il y a 2 notations possibles pour l'attribut `action` des formulaires html :

- `action="@routes.Categories.add()"`
- `action="/category/add"`

personnellement, la 2ème me semble plus appropriée (et plus élégante).

```
@(
  message: String,
  bookmarks: List[models.Bookmark],
  categories: List[models.Category]
)

@main("Gestion des bookmarks") {

  <h1>BookMarks</h1>
  <p>@message</p>
  <!-- Formulaire de saisie : Catégories -->
  <fieldset>
    <legend>Nouvelle Catégoe</legend>
    <!--<form method="post" action="@routes.Categories.add()">-->
    <form method="post" action="/category/add">
      <input name="label" placeholder="label">
      <button type="submit">Ajouter la Catégoe</button>
    </form>
  </fieldset>
  <!-- Liste des Catégories -->
  <ul>
    @for(category <- categories) {
      <li>@category.id @category.label</li>
    }
  </ul>
}
```

```

<!-- Formulaire de saisie : Bookmarks -->

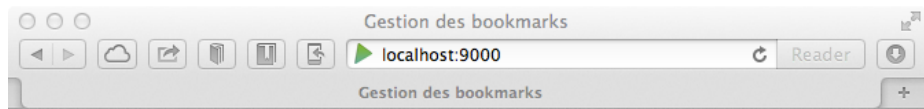
<fieldset>
  <legend>Nouveau Bookmark</legend>
  <!--<form method="post" action="@routes.Bookmarks.add()"-->
  <form method="post" action="/bookmark/add">
    <input name="title" placeholder="title">
    <input name="url" placeholder="url">
    <input name="details" placeholder="details">

    <select size="1" name="category.id">
      @for(category <- categories) {
        <option value="@category.id">@category.label</option>
      }
    </select>

    <button type="submit">Ajouter le Bookmark</button>
  </form>
</fieldset>
<!-- Liste des Bookmarks -->
<ul>
  @for(bookmark <- bookmarks) {
    <li>@bookmark.title : <a href="@bookmark.url">@bookmark.url</a> :
      @if(bookmark.category != null) {
        @bookmark.category.label
      }
    </li>
  }
</ul>
}

```

Lancez tout de suite, nous passerons aux explications plus tard, rafraîchissez donc votre page, vous devriez obtenir ceci :



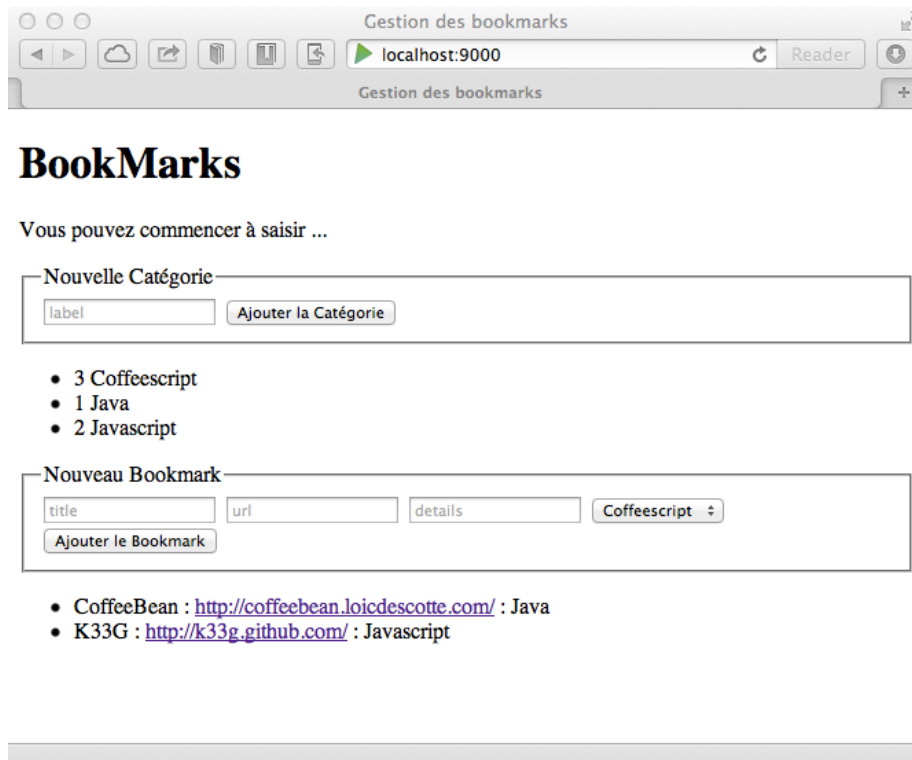
BookMarks

Vous pouvez commencer à saisir ...

Nouvelle Catégorie

Nouveau Bookmark

Et vous pouvez même commencer à saisir :



Qu'avons nous fait

- Nous avons passé en paramètres des informations à la vue :

```
@(
    message: String,
    bookmarks: List[models.Bookmark],
    categories: List[models.Category]
)
```

- Nous avons affiché le message :

```
<p>@message</p>
```

- Nous avons créé des formulaire de saisie, en leur précisant quelle méthode appeler : `@routes.Categories.add()`

```
<form method="post" action="@routes.Categories.add()">
```

```

      <input name="label" placeholder="label">
      <button type="submit">Ajouter la Catégorie</button>
    </form>

```

- Nous avons affiché des informations, comme la liste des catégories :

```

<ul>
  @for(category <- categories) {
    <li>@category.id @category.label</li>
  }
</ul>

```

Remarque : le langage utilisé pour les templates des vues est **Scala**. C’est un peu déroutant, mais vous verrez que l’on s’habitue (et que l’on peut aussi s’en passer, mais ça c’est une autre histoire).

Voilà, nous avons un embryon d’application qui fonctionne. Je vous propose maintenant d’habiller notre application pour la rendre un peu plus sexy avant de passer à des choses plus sérieuses.

8 Un peu de cosmétique

Qu’allons nous voir ?

- Une petite récréation : comment rendre un site web “beau” alors que l’on est une bille en design ?

... Ce n’est pas du Play!>, mais ça va faire joli :)

Le framework css à la mode, en ce moment c’est Twitter Bootstrap. Il permet à tout développeur web le plus nul en design de donner un aspect “pro” & “joli” à ses pages web. Alors certains me diront : “on va tous avoir des sites avec la même tête !”, et je répondrais : “certes, mais au moins, ils seront propres, sobres, ... et puis rien ne vous empêche ensuite d’aller un peu modifier les couleurs”. Toujours est-il que c’est plus agréable de travailler avec quelque chose de joli et ça me donne l’opportunité de vous expliquer où sont les ressources statiques dans une application Play!>.

8.1 Prérequis

- Allons télécharger **Bootstrap** : <http://twitter.github.com/bootstrap/assets/bootstrap.zip>
- Dé-zippez, vous obtenez un répertoire **bootstrap**
- Allez copier ce répertoire dans le répertoire **public** de notre application

8.2 Utilisation

Si vous allez dans le répertoire `app/views` de notre application, vous remarquerez le fichier `main.scala.html`. En fait on pourrait dire que le fichier (la vue) `index.scala.html` utilise `main.scala.html`. Remarquez dans `index.scala.html` la ligne :

```
@main("Gestion des bookmarks") { ... }
```

et dans `main.scala.html` :

```
@(title: String)(content: Html)
```

`index` “appelle” `main` en lui passant en paramètre le titre de la page et le contenu HTML. Et c’est dans `main.scala.html` que sont déclarées les ressources javascript et css avec la commande `@routes.Assets.at`.

8.2.1 Modifions le code de `main.scala.html`

```
@(title: String)(content: Html)

<!DOCTYPE html>

<html>
  <head>
    <title>@title</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" media="screen"
      href="@routes.Assets.at("bootstrap/css/bootstrap.css")">
    <style>
      body {
        padding-top: 60px;
        padding-bottom: 40px;
      }
    </style>
    <link rel="stylesheet" media="screen"
      href="@routes.Assets.at("bootstrap/css/bootstrap-responsive.css")">
    <link rel="shortcut icon" type="image/png"
      href="@routes.Assets.at("images/favicon.png")">
    <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")"
      type="text/javascript"></script>
  </head>
  <body>
```

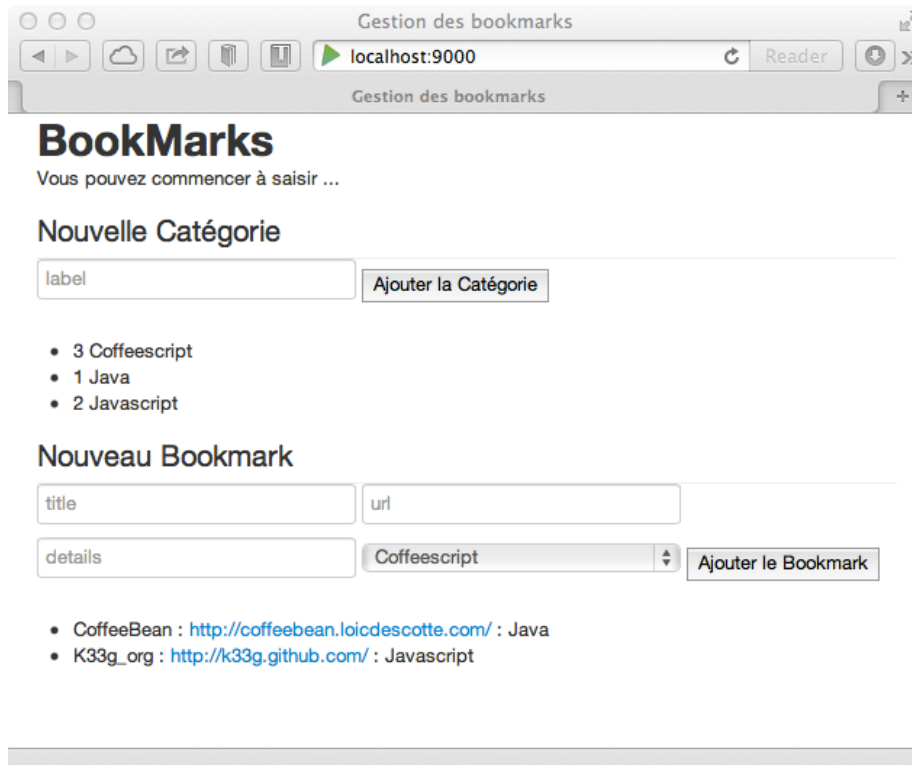
```

        @content
    </body>
</html>

```

//TODO : donner un peu d'explications

Vous pouvez rafraîchir votre page, c'est déjà beaucoup plus sympa :



Customisons légèrement `main.scala.html` Modifiez le tag `<body>` de la façon suivante :

```

<body>

    <div class="navbar navbar-fixed-top">
        <div class="navbar-inner">
            <div class="container">
                <a class="brand">@title</a>
            </div>

```

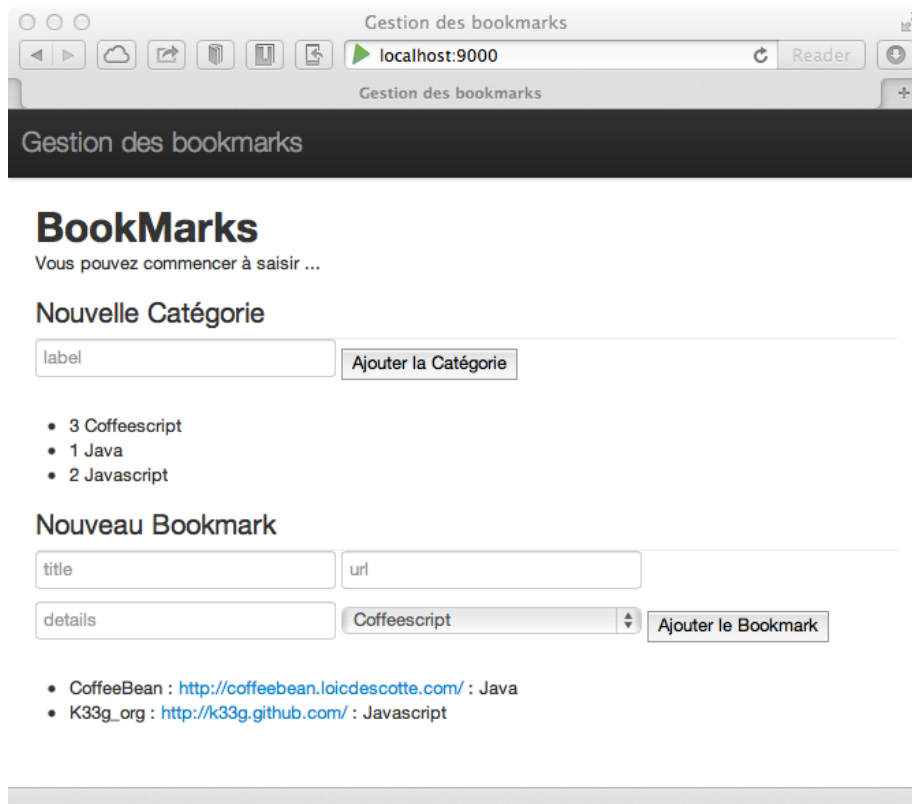
```

        </div>
    </div>
    <div class="container">
        @content
    </div>

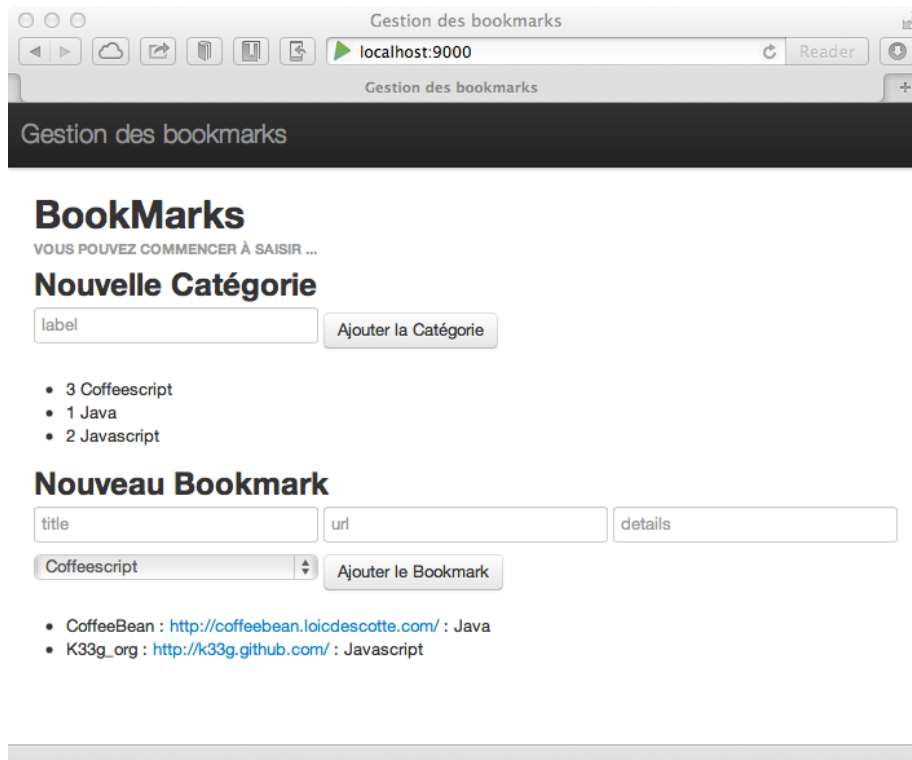
</body>

```

Vous pouvez rafraîchir votre page à nouveau, ça prend forme ... :



Allons customiser légèrement `index.scala.html`. Vous pouvez remplacer les tags `<legend>` par `<h2>`. Ajoutez la classe `btn` aux tags `<button class="btn">`, pour avoir des boutons arrondis. Remplacez le tag `<p>@message</p>` par `<h6>@message</h6>`.



Bref, amusez vous !

9 Charger des données au démarrage

Qu'allons nous voir ?

- *Comment pré-charger des données au démarrage de l'application ?*

... Très pratique à l'usage

A chaque fois que vous allez modifier vos modèles, vous allez perdre vos données. Donc nous allons voir comment charger un jeu de données au démarrage pour éviter d'avoir à tout re-saisir à chaque fois.

Dans le répertoire `/conf`, créez un fichier `initial-data.yml` avec les données suivantes :

```
# Categories
```

categories:

```
- !!models.Category
  label: Javascript

- !!models.Category
  label: Java

- !!models.Category
  label: Coffeescript
```

Puis à la racine de /app, créez une classe Global.java avec le code suivant :

```
import play.*;
import play.libs.*;

import java.util.*;

import com.avaje.ebean.*;

import models.*;

public class Global extends GlobalSettings {

    public void onStart(Application app) {
        InitialData.insert(app);
    }

    static class InitialData {

        public static void insert(Application app) {
            if(Ebean.find(Category.class).findRowCount() == 0) {

                Map<String,List<Object>> all =
                    (Map<String,List<Object>>)Yaml.load("initial-data.yml");

                // Insert categories first
                Ebean.save(all.get("categories"));

            }
        }
    }
}
```

Enfin, modifier le fichier `conf/application.conf` et décommenter la ligne suivante, ie :

```
# global=Global
```

devient

```
global=Global
```

Cette manipulation permet d’activer votre nouvelle classe au démarrage de l’application.

Vous pouvez maintenant rafraîchir votre page pour vérifier que les données sont bien chargées au démarrage de votre application.

10 Validation des données

Qu’allons nous voir ?

- *comment définir des contraintes sur un modèle*
- *comment vérifier les données d’un formulaire*
- *comment valider les données côté client en utilisant un module tiers*

10.1 Enrichissement du modèle et vérification des données

Nous voulons nous assurer que l’utilisateur entre bien un label (d’une catégorie) lors de la création d’un bookmark. Nous voulons aussi limiter la taille à 30 caractères (pourquoi pas?). Pour cela nous allons utiliser les annotations `@Required` et `@MaxLength` dans notre modèle `Category` sur le “champ” `label` (on n’oublie pas : `import play.data.validation.Constraints`):

```
package models;

import play.db.ebean.Model;
import javax.persistence.*;
import play.data.validation.Constraints;

@Entity
public class Category extends Model {
```



```

@Id
public Long id;

@Constraints.Required
@Constraints.MaxLength(30)
public String label;

// ...
}

```

Ceci nous permettra ensuite de vérifier l'intégrité des données lors de la soumission du formulaire (= lorsque l'on ajoute une catégorie). Modifions un peu le code de notre contrôleur `Categories` :

```

public class Categories extends Controller {

    public static Result add() {

        final Form<Category> categoryForm = form(Category.class).bindFromRequest();

        if (categoryForm.hasErrors()) { // <--- le code modifié, ça commence ici !

            flash("error", "Non, non, il faut saisir autre chose !");

        } else { // <--- on n'enregistre que si tout va bien
            final Category category = categoryForm.get();
            category.save();
        }

        return redirect(routes.Application.index());
    }
}

```

En cas de problème on renvoie une erreur à notre template.

Pour afficher cette erreur on peut ajouter ceci à notre fichier `index.scala.html` :

```

@if(flash.containsKey("error")) {
    <div class="alert alert-error"> <!-- ceci est un style twitter bootstrap -->
        <strong>Oups!</strong> @flash.get("error")
    </div>
}

```

```
    </div>  
}
```

Vous pouvez tout de suite essayer :

BookMarks

VOUS POUVEZ COMMENCER À SAISIR ...

Nouvelle Catégorie

Ajouter la Catégorie

- 3 Coffeescript
- 2 Java
- 1 Javascript
- 4 Ruby
- 33 Scala

BookMarks

VOUS POUVEZ COMMENCER À SAISIR ...

Nouvelle Catégorie

Ajouter la Catégorie

Oups! Non, non, il faut saisir autre chose !

- 3 Coffeescript
- 2 Java
- 1 Javascript
- 4 Ruby
- 33 Scala

Il existe d'autres annotations de validation :

- @Max et @Min pour les valeurs numériques
- @MinLength pour demander une taille longueur minimum à un champ
- @Pattern qui permet de valider des expressions régulières
- @Email pour valider le format email

On peut bien sûr écrire facilement nos propres validateurs...

10.2 Validation côté client

Avec HTML5, il est possible de valider des données d'un formulaire directement depuis le navigateur avant de les envoyer au serveur.

Il existe un module Play (développé par un de vos serveurs : @loic_d) que vous trouverez ici : <https://github.com/loicdescotte/Play2-HTML5Tags>, pour générer les bonnes balises HTML à partir des contraintes du modèle.

Avant de l'utiliser, voyons comment l'installer.

10.2.1 Installation de Play2-HTML5Tags

Pour le moment il n'existe pas de repository public pour les modules Play2!>, donc téléchargez sur le site le plugin (utilisez le bouton "zip" ou directement le lien <https://github.com/loicdescotte/Play2-HTML5Tags/zipball/master>). Une fois le module téléchargé, dézippez, allez dans le répertoire du module :

Les versions de Play, Sbt, ... ont une importance "VITALE" ;), il faut donc aller modifier quelques petits paramètres en fonction de la version de Play que vous utilisez, mais **seulement si c'est nécessaire**, ce n'est utile que dans les cas où la version du module a été développé pour une version antérieure de Play (par exemple version 2.0.1 contre version 2.0.2). Si c'est le cas vous aurez les 2 manipulations ci-dessous à effectuer :

- dans le répertoire /projet changez dans `build.properties` `sbt.version=0.11.2` par `sbt.version=0.11.3`
- dans le répertoire /projet changez dans `plugins.sbt` `addSbtPlugin("play" % "sbt-plugin" % "2.0")` par `addSbtPlugin("play" % "sbt-plugin" % "2.0.2")`

PS: bien sûr cela change en fonction des versions de Play.

Ensuite dans le répertoire du module, en mode console, tapez : `play`, cela va "mouliner" un petit moment, vous devriez ensuite obtenir un prompt avec le nom du module :

```
1. Default (sh)
k33g-orgs-MacBook-Air:Play2-HTML5Tags k33g_org$ play
[info] Loading project definition from /Users/k33g_org/Dropbox/Public/play.rules
.work/plugins/Play2-HTML5Tags/project
[info] Set current project to html5Tags (in build file:/Users/k33g_org/Dropbox/P
ublic/play.rules.work/plugins/Play2-HTML5Tags/)

  _ _ _ _ _
 | ' \ | / ' | | |
 | _/ | \_ | \ ( )
 |_ |      |_/

play! 2.0.2, http://www.playframework.org

> Type "help play" or "license" for more information.
> Type "exit" or use Ctrl+D to leave this console.

[html5Tags] $
```

Ensuite, au prompt, tapez `publish-local`, là encore cela mouline un moment, Play compile et installe le plugin pour qu'il soit utilisable par toutes vos applications Play. Vous pouvez aller vérifier dans le répertoire d'installation de Play, dans `/repository/local` vous avez maintenant un répertoire `com.loicdescotte.coffeebean` qui contient le module `html5tags_2.9.1`.

10.2.2 Déclaration de Play2-HTML5Tags

Avant de pouvoir utiliser le module, vous devez déclarer son utilisation dans votre application. Il va donc falloir modifier le fichier `/project/Build.scala` de votre application Play :

```
import sbt._
import Keys._
import PlayProject._

object ApplicationBuild extends Build {

  val appName          = "bookmarks"
```

```

val appVersion      = "1.0-SNAPSHOT"

val appDependencies = Seq(
  // Add your project dependencies here,
  "com.loicdescotte.coffeebean" % "html5tags_2.9.1" % "1.0-SNAPSHOT"
)

val main = PlayProject(appName,
  appVersion, appDependencies, mainLang = JAVA).settings(
  // Add your own project settings here
  resolvers += "Local Play Repository" at "/Users/k33g_org/play-2.0.2/repository"
)
}

```

Maintenant vous pouvez utiliser le module.

10.2.3 Utilisation de Play2-HTML5Tags

Dans le cas de notre application de gestion de bookmarks, on va pouvoir remplacer ceci :

```
<input name="label" placeholder="label">
```

Par cela :

```
@text(categoryForm("label"), 'placeholder -> "LABEL : ")
```

Et le “markup” approprié sera généré :

```
<input name="url" placeholder="url" maxlength="30" required>
```

Le navigateur vérifiera alors la présence et la longueur du champ avant d’envoyer les données au serveur, ce qui permettra à l’utilisateur d’avoir un retour d’erreur plus rapide en case de problème et d’économiser un peu de bande passante!

Mais pour que cela fonctionne, quelques manipulations sont encore nécessaires :

Créons un modèle User (il nous reservira plus tard)

```

package models;

import java.util.*;
import javax.persistence.*;

import play.db.ebean.Model;
import play.data.format.*;
import play.data.validation.*;

@Entity
public class User extends Model {

    @Id
    @Constraints.Required
    @Formats.NonEmpty
    public String email;

    @Constraints.Required
    public String name;

    @Constraints.Required
    public String password;

    public static Model.Finder<String,User> find =
        new Model.Finder(String.class, User.class);

    public static List<User> findAll() {
        return find.all();
    }

    public static User findByEmail(String email) {
        return find.where().eq("email", email).findUnique();
    }

    public String toString() {
        return "User(" + email + ")";
    }
}

```

Dans le contrôleur `Application.java`, ajoutons `form(Category.class)` en argument de la méthode `index.render()` :

```

public class Application extends Controller {

    public static Result index() {

        return ok(index.render(
            "Vous pouvez commencer à saisir ...",
            Bookmark.find.fetch("category").orderBy("title").findList(),
            Category.find.orderBy("label").findList(),
            User.find.byId(request().username()),
            form(Category.class) //<--- c'est ici
        ));
    }
}

```

Puis, dans la vue `index.scala.html`, déclarons ce nouveau paramètre :

```

@(
    message: String,
    bookmarks: List[models.Bookmark],
    categories: List[models.Category],
    user: User,
    categoryForm: Form[models.Category]
)

```

Ajoutons tout de suite après ceci (spécifique au module que nous avons installé):

```
@import html5.tags.html._
```

Maintenant nous pouvons remplacer `<input name="label" placeholder="label">` par `@text(categoryForm("label"), 'placeholder -> "saisir un label")` et vous obtiendrez ceci :

Si vous ne souhaitez pas voir apparaître les contraintes de saisie, utilisez plutôt :

```
@text(categoryForm("label"), 'placeholder -> "saisir un label", '_showConstraints -> false).
```

Remarque : Le fait de référencer un objet `categoryForm` nous permettra si on le souhaite plus tard d'éditer une catégorie existante en remplissant directement le champ du formulaire avec la valeur de notre objet. On pourrait par exemple écrire quelque chose comme ça dans le contrôleur (ceci est un exemple sans lien avec le code du tuto) :

BookMarks

VOUS POUVEZ COMMENCER À SAISIR ...

Nouvelle Catégorie

label

Maximum length: 30

Required

Ajouter la Catégorie

- 3 Coffeescript
- 2 Java
- 1 Javascript
- 4 Ruby

Figure 1:

```
public static Result edit(Long id) {  
    Form<Category> categoryForm = form(Category.class).fill(  
        Category.find.byId(id)  
    );  
    return ok(  
        edit.render(id, categoryForm)  
    );  
}
```

Le tag `text` est capable de changer le type de `<input>` si une annotation particulière est détectée.

Par exemple avec le modèle suivant :

```
@Constraints.Email  
public String contactMail;
```

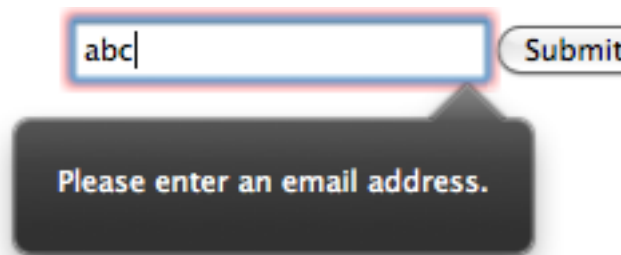
Et ce tag :

```
@text(form("contactMail"))
```

On obtiendra ceci :


```
<input type="email" id="contactMail" name="contactMail" value="">
```

Et le navigateur vérifiera le format saisi :



HTML5 reconnaît de nouveaux types de données dans les formulaires, comme les nombres, les dates, les numéros de téléphone, les URL. . . . Le module prend en charge ces types de données à travers des tags particuliers (`@number`, `@date`, `@telephone`, `@url` . . .)

Le fait de préciser le type de `<input>` permet également au navigateur, particulièrement sur mobile, d'adapter son IHM au format demandé. Par exemple pour un champ numérique :



11 Gestion de l'authentification

Qu'allons nous voir ?

- *Comment mettre en oeuvre un système simple d'authentification*

Remarque : la rédaction "complète" de ce chapitre reste encore à faire, mais les codes sont complets et vous pouvez les utiliser tels quels.

Pour que n'importe qui ne puisse pas saisir des bookmarks, nous allons mettre en place un système d'authentification.

11.1 Complétons la classe User

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.ebean.Model;
import play.data.format.*;
import play.data.validation.*;

@Entity
public class User extends Model {

    @Id
    @Constraints.Required
    @Formats.NonEmpty
    public String email;

    @Constraints.Required
    public String name;

    @Constraints.Required
    public String password;

    public static Model.Finder<String,User> find =
        new Model.Finder(String.class, User.class);

    public static List<User> findAll() {
        return find.all();
    }

    public static User findByEmail(String email) {
        return find.where().eq("email", email).findUnique();
    }

    //TODO : expliquer ce que fait le code
    public static User authenticate(String email, String password) {
        return find.where()
            .eq("email", email)
            .eq("password", password)
```

```

        .findUnique();
    }

    public String toString() {
        return "User(" + email + ")";
    }
}

```

Remarque : on a rajouté une méthode authenticate

11.2 Création d'un contrôleur Secured

```

//TODO : expliquer ce que fait le code
package controllers;

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

import models.*;

public class Secured extends Security.Authenticator {

    @Override
    public String getUsername(Context ctx) {
        return ctx.session().get("email");
    }

    @Override
    public Result onUnauthorized(Context ctx) {
        return redirect(routes.Authentication.login());
    }
}

```

11.3 Allons modifier le contrôleur Application

- Ajout de l'annotation @Security.Authenticated(Secured.class)
- Passage du user à la méthode index.render()

Code final :

```

package controllers;

import play.*;
import play.mvc.*;
import play.data.*;

import models.*;
import views.html.*;

@Security.Authenticated(Secured.class)
public class Application extends Controller {

    public static Result index() {

        return ok(index.render(
            "Vous pouvez commencer à saisir ...",
            Bookmark.find.fetch("category").orderBy("title").findList(),
            Category.find.orderBy("label").findList(),
            User.find.byId(request().username())
        ));
    }
}

```

11.4 Création d'un contrôleur Authentication

```

package controllers;

import play.*;
import play.mvc.*;
import play.data.*;

import models.*;
import views.html.*;

public class Authentication extends Controller {

    public static class AuthenticatedUser {

        public String email;
        public String password;
    }
}

```

```

        public String validate() {
            if(User.authenticate(email, password) == null) {
                return "oups! r    ! Essaye encore une fois";
            }
            return null;
        }
    }

    public static Result login() {
        return ok(
            login.render(form(AuthenticatedUser.class))
        );
    }

    //On r  cup  re les informations de login (quand le user se "signe")
    public static Result authenticate() {
        Form<AuthenticatedUser> loginForm =
            form(AuthenticatedUser.class).bindFromRequest();
        if(loginForm.hasErrors()) {
            return badRequest(login.render(loginForm));
        } else {
            session("email", loginForm.get().email);
            return redirect(
                routes.Application.index()
            );
        }
    }

    //Fermer la session
    public static Result logout() {
        session().clear();
        flash("success", "Vous   tes d  connect  (e)");
        return redirect(
            routes.Authentication.login()
        );
    }
}

```

11.5 Allons modifier les routes

# Authentication		
GET	/login	controllers.Authentication.login()
POST	/login	controllers.Authentication.authenticate()

GET /logout

controllers.Authentication.logout()

11.6 Allons modifier main.scala.html et index.scala.html

Nous avons vu que nous passions le user authentifié à la méthode `index.render()` de la vue dans le contrôleur `Application` :

```
public static Result index() {  
    return ok(index.render(  
        "Vous pouvez commencer à saisir ...",  
        Bookmark.find.fetch("category").orderBy("title").findList(),  
        Category.find.orderBy("label").findList(),  
        User.find.byId(request().username())  
    ));  
}
```

Modifions donc le code des formulaires en conséquence :

11.6.1 main.scala.html

- on ajoute `user` en paramètre : `@(title: String, user: User)(content: Html)`
- si le `user` est authentifié nous affichons ses informations et la possibilité de se “déloguer” :

```
@if(user != null) {  
    <ul class="nav">  
        <li><a>@user.name <span>(@user.email)</span></a></li>  
        <li><a href="@routes.Authentication.logout()">Logout</a></li>  
    </ul>  
}
```

Le code définitif va donner ceci :

```
@(title: String, user: User)(content: Html)  
  
<!DOCTYPE html>
```

```

<html>
  <head>
    <title>@title</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" media="screen"
      href="@routes.Assets.at("bootstrap/css/bootstrap.css")">
    <style>
      body {
        padding-top: 60px;
        padding-bottom: 40px;
      }
    </style>
    <link rel="stylesheet" media="screen"
      href="@routes.Assets.at("bootstrap/css/bootstrap-responsive.css")">
    <link rel="shortcut icon" type="image/png"
      href="@routes.Assets.at("images/favicon.png")">
    <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")"
      type="text/javascript"></script>
  </head>
  <body>

    <div class="navbar navbar-fixed-top">
      <div class="navbar-inner">
        <div class="container">
          <a class="brand">@title</a>
          @if(user != null) {
            <ul class="nav">
              <li><a>@user.name <span>(@user.email)</span></a></li>
              <li><a href="@routes.Authentication.logout()">Logout</a></li>
            </ul>
          }
        </div>
      </div>
    </div>

    <div class="container">
      @content
    </div>

  </body>
</html>

```

11.6.2 Il faut aussi modifier index.scala.html

En effet, `index` utilisant `main`, nous devons ajouter la notion de `user`, il y a juste le début à modifier :

Vous vous souvenez, dans `Application` nous avons modifié l'appel de `index.render()` :

```
@(
message: String,
bookmarks: List[models.Bookmark],
categories: List[models.Category],
user: User
)
```

et la modification précédente de `main.scala.html` implique le passage du paramètre `user` à `@main()` :

```
@main("Gestion des bookmarks", user) { ...
```

11.7 Allons créer un formulaire de login / Vue

Nous y sommes presque. Il faut créer le formulaire de login : créez dans le répertoire `views` un fichier `login.scala.html` avec le code suivant :

```
@(form: Form[Authentication.AuthenticatedUser])

@main("Authentification", null) {

  @helper.form(routes.Authentication.authenticate) {

    <h2>Qui êtes vous ?</h2>

    @if(form.hasGlobalErrors) {
      <p class="error">@form.globalError.message</p>
    }

    @if(flash.contains("success")) {
      <p class="success">@flash.get("success")</p>
    }

    <p><input type="email" name="email" placeholder="Email"
      value="@form("email").value"></p>
```



```

        <p><input type="password" name="password" placeholder="Password"></p>
        <p><button class="btn" type="submit">Login</button></p>
    }
}

```

11.8 Allons ajouter des users dans initial-data.yml

11.8.1 Ajoutons des utilisateurs :

Cela permettra de se connecter, donc dans le fichier `initial-data.yml`, ajouter ceci (ou quelque chose d'approchant) :

```

# Users

users:

  - !!models.User
    email:      ph.charriere@gmail.com
    name:       k33g
    password:   play

  - !!models.User
    email:      bob@morane.com
    name:       bob
    password:   indochine

```

Ainsi, nous aurons 2 utilisateurs au chargement de l'application. Et pour les charger, nous allons modifier le fichier `Global.java`

11.8.2 Modifions Global.java

```

import play.*;
import play.libs.*;
import java.util.*;
import com.avaje.ebean.*;
import models.*;

public class Global extends GlobalSettings {

    public void onStart(Application app) {
        InitialData.insert(app);
    }
}

```

```

static class InitialData {

    public static void insert(Application app) {

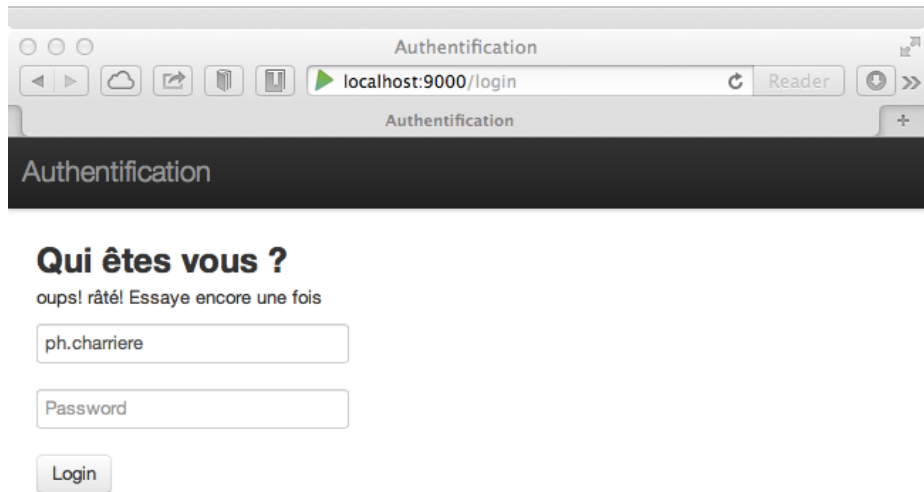
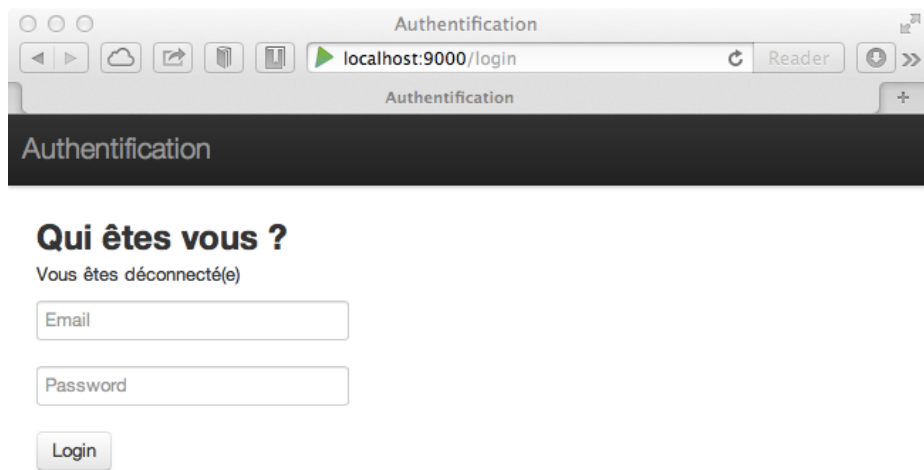
        Map<String,List<Object>> all =
            (Map<String,List<Object>>)Yaml.load("initial-data.yml");

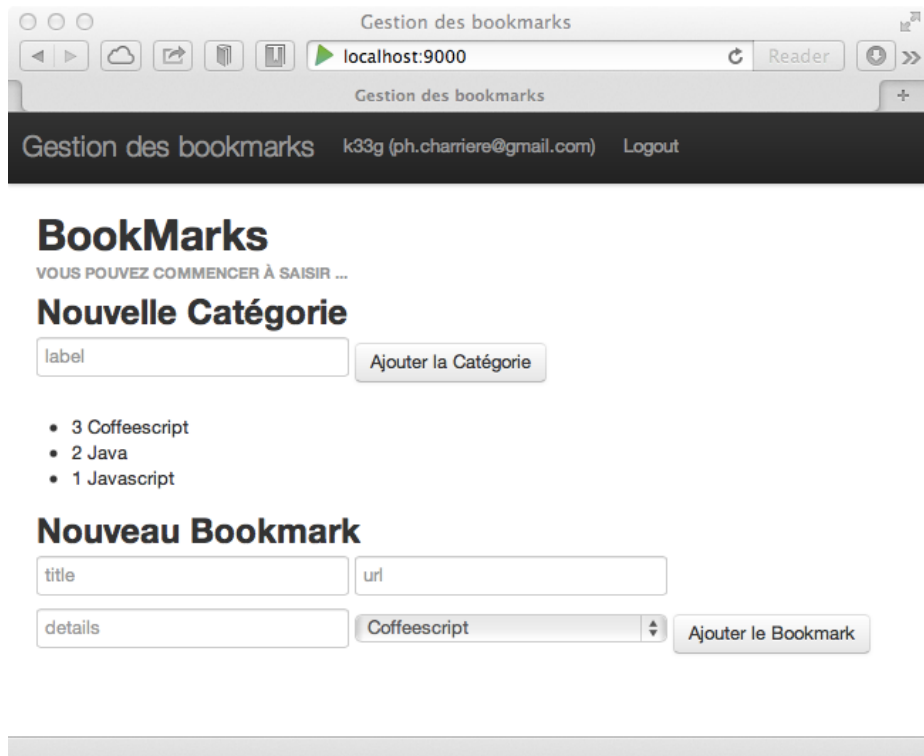
        if(Ebean.find(User.class).findRowCount() == 0) {
            Ebean.save(all.get("users"));
        }

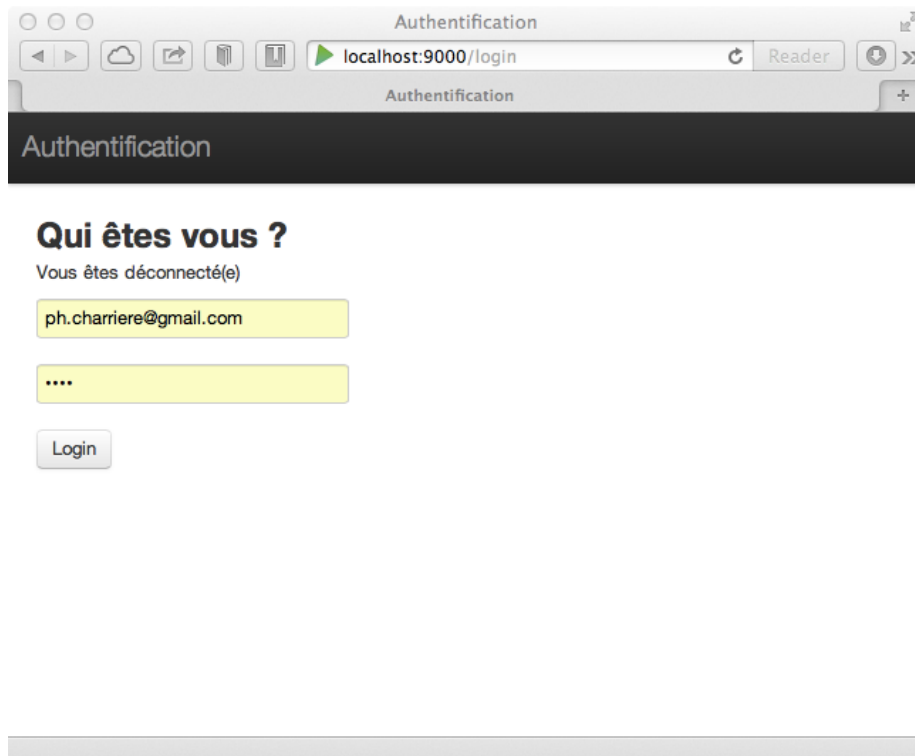
        if(Ebean.find(Category.class).findRowCount() == 0) {
            // Insert categories first
            Ebean.save(all.get("categories"));
        }
    }
}

```

C'est bon vous pouvez lancer l'application à nouveau.







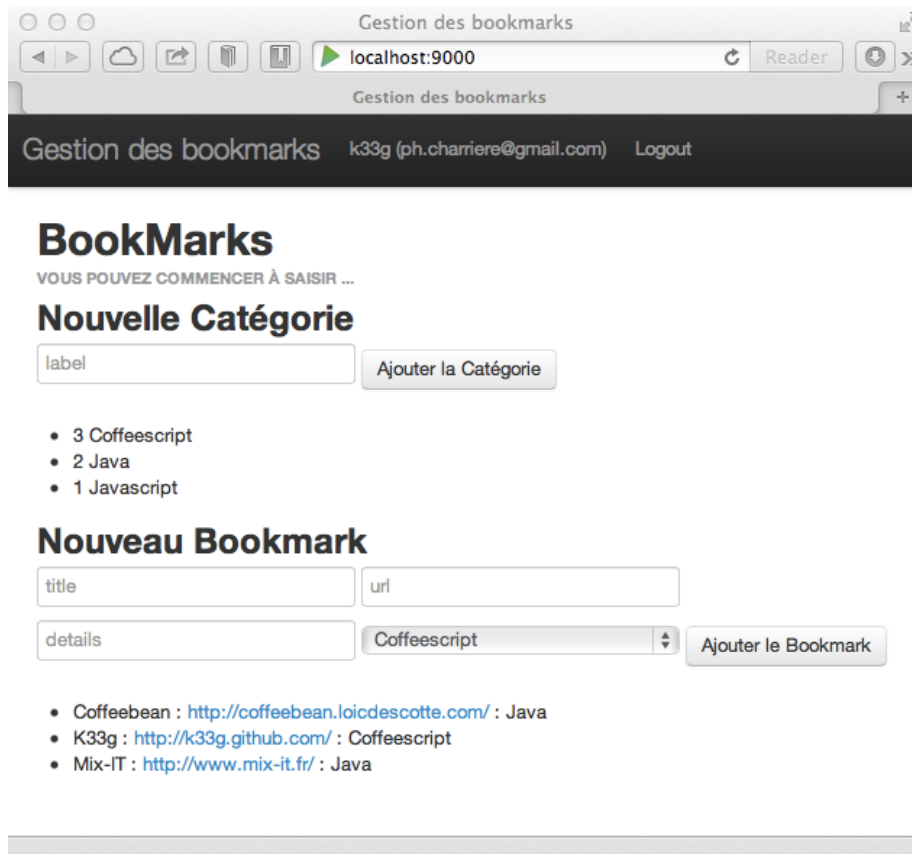
12 Services (JSON)

Qu'allons nous voir ?

- *comment faire un service json*
- *comment sécuriser ce service*
- *comment s'authentifier via ajax*
- *comment faire une "single page application"*

12.1 Primo :

Ajoutons quelques bookmarks dans notre applications :



12.2 Création de notre service JSON

Objectif : faire un service qui nous renvoie la liste des bookmarks au format JSON

12.2.1 Allons modifier le contrôleur Bookmarks

bookmarks.java dans app/controllers

import(s)

```
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;
```

```
import static play.libs.Json.toJson;
```

Ajout de la méthode `jsonList()`

```
public static Result jsonList() {
    Map<String, List<Bookmark>> data = new HashMap<String, List<Bookmark>>();
    List<Bookmark> list = Bookmark.find.orderBy("title").findList();
    data.put("bookmarks", list);
    return ok(toJson(data));
}
```

Modifions le fichier `routes` Nous ajoutons la route suivante :

```
#Services
GET /bookmarks/jsonlist controllers.Bookmarks.jsonList()
```

Donc le code final est le suivant :

```
package controllers;

import models.Bookmark;

import play.*;
import play.mvc.*;
import play.data.*;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import static play.libs.Json.toJson;

public class Bookmarks extends Controller {

    public static Result add() {

        final Form<Bookmark> bookmarkForm = form(Bookmark.class).bindFromRequest();
        final Bookmark bookmark = bookmarkForm.get();

        bookmark.save();
    }
}
```

```

        return redirect(routes.Application.index());
    }

    public static Result jsonList() {
        Map<String, List<Bookmark>> data = new HashMap<String, List<Bookmark>>>();
        List<Bookmark> list = Bookmark.find.orderBy("title").findList();
        data.put("bookmarks", list);
        return ok(toJson(data));
    }
}

```

Donc, lorsque nous appellerons l'url `localhost:9000/bookmarks/jsonlist`, nous obtiendrons la liste des bookmarks au format JSON.

12.3 Utilisation du service JSON

12.3.1 Directement avec l'url

Dans la zone de saisie de l'url de votre navigateur, saisissez donc `localhost:9000/bookmarks/jsonlist`. Et vous obtenez le flux JSON de vos bookmarks :



12.3.2 Plus utile : via une requête ajax

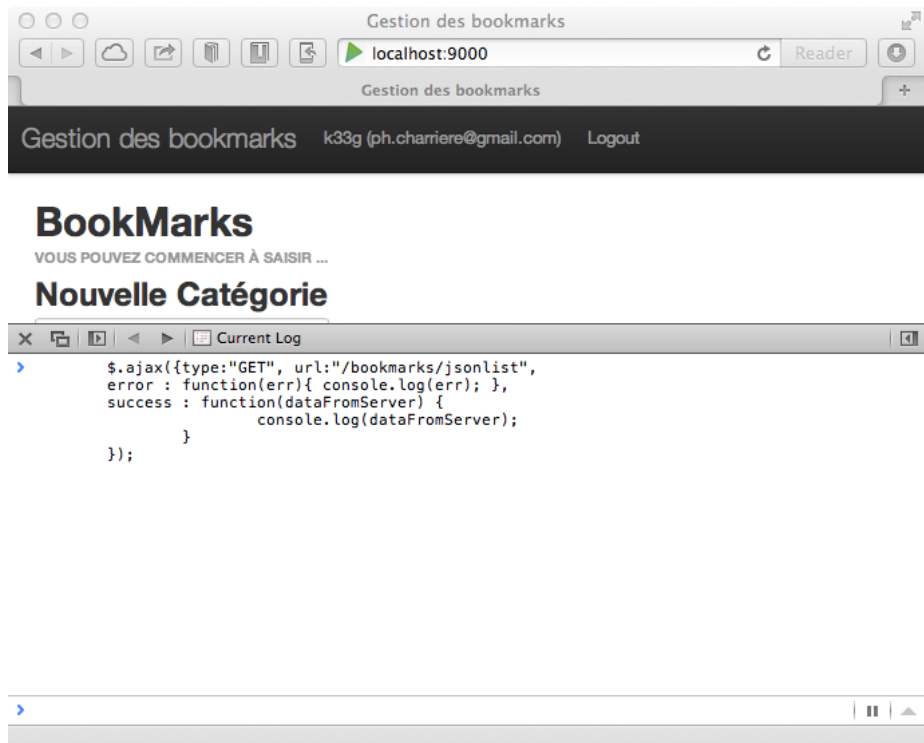
Comme vous le savez (ou pas), Play2!> est fourni avec **jQuery**, “petit” framework javascript très utile pour “jouer” avec le DOM de vos pages, mais aussi pour faire des requêtes ajax. Si vous vous souvenez, dans la vue (portion de vue) `main.scala.html` il y avait le code suivant :

```
<script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")"
  type="text/javascript"></script>
```

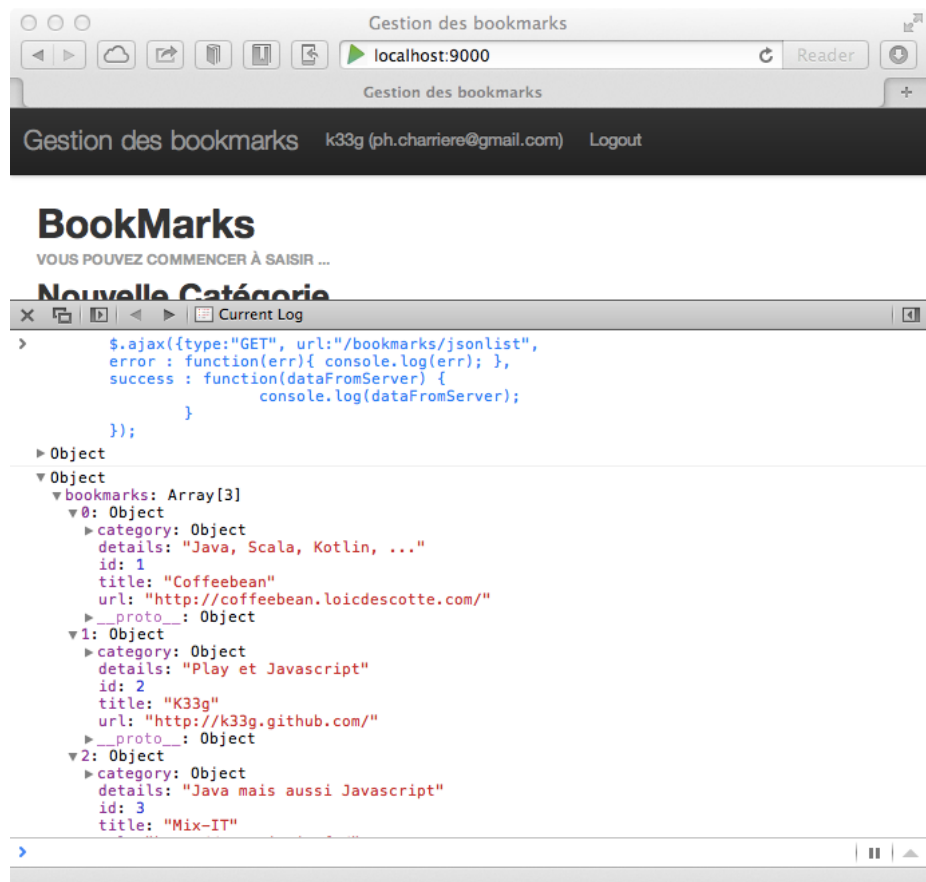
Ce qui signifie que toute vue “utilisant” `main.scala.html`, comme par exemple `index.scala.html` (vous trouverez la déclaration `@main(...)` dans le code), charge **jQuery**. Donc,

- retournez dans votre navigateur, mais cette fois-ci à la racine de votre site (probablement `http://localhost:9000`)
- ouvrez la console de votre navigateur (click droit + “Inspect Element”)
- et Tapez ceci :

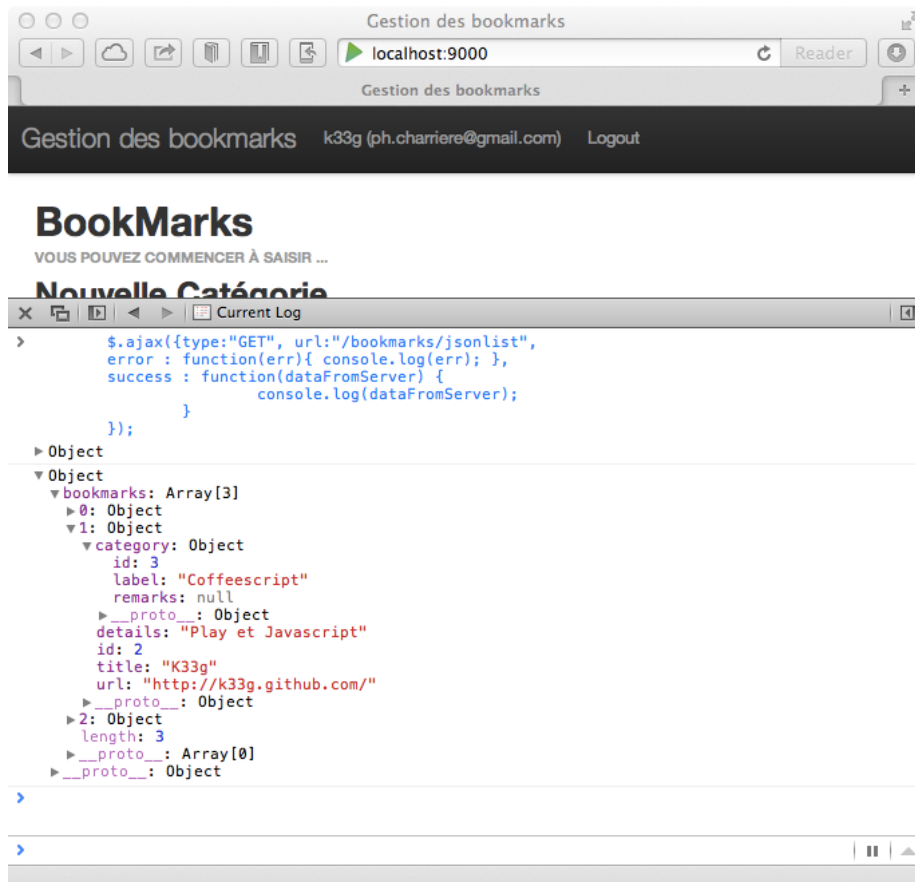
```
$.ajax({type:"GET", url:"/bookmarks/jsonlist",
  error : function(err){ console.log(err); },
  success : function(dataFromServer) {
    console.log(dataFromServer);
  }
});
```



Vous obtenez directement un objet **bookmarks** qui est un tableau d'objets avec les éléments attendus :



Et vous remarquerez ...



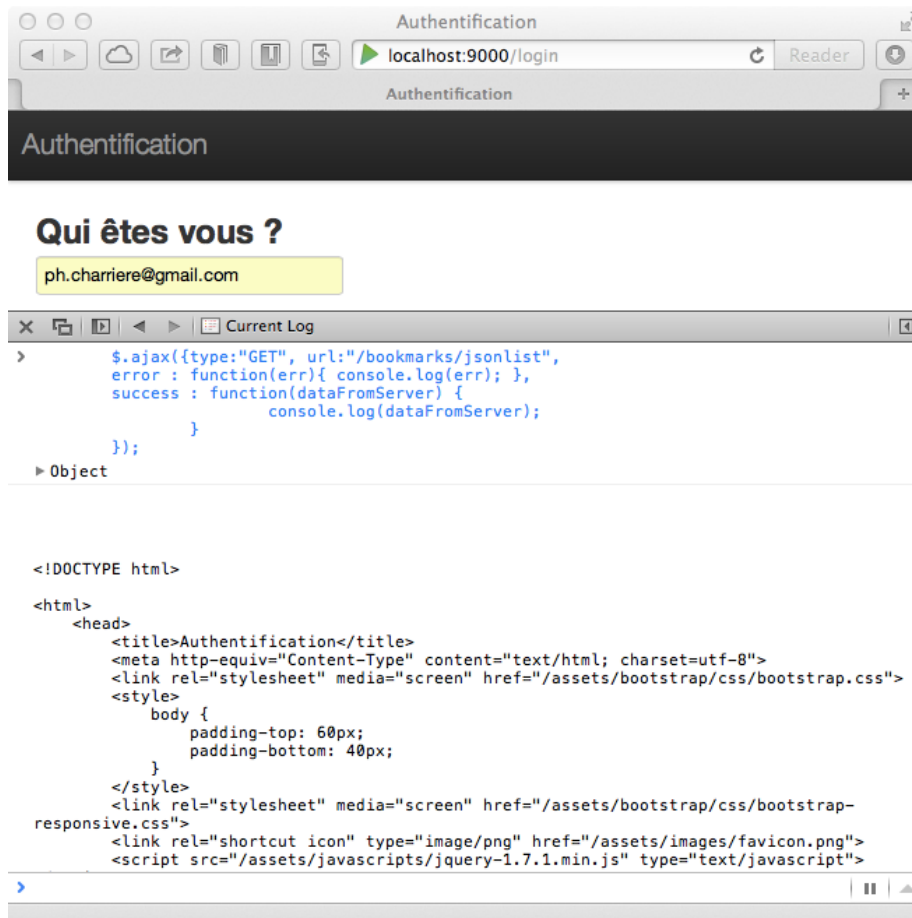
... que les objets bookmark du tableau `bookmarks` contiennent les objets “liés” `category`.

Génialement simplissime et pratique, non !?

Attention : dans mon exemple le service n’est pas sécurisé

12.4 Sécurisation

Sécurisons notre contrôleur `Bookmarks.java` en lui ajoutant l’annotation `@Security.Authenticated(Secured.class)`. Ensuite (après recompilation), retournez à la racine du site `localhost:9000` pour vous “deloguer”. Puis relancez votre requête ajax, et là vous obtenez (curieusement ?) le code HTML de la page d’authentification en retour. Ce qui est rassurant, c’est que notre service est bien sécurisé, mais le code de retour n’est pas forcément “top” à gérer.



En fait, dans notre classe `Secured.java` nous avons la méthode suivante :

```

@Override
public Result onUnauthorized(Context ctx) {
    return redirect(routes.Authentication.login());
}

```

Donc, si vous n'êtes pas authentifié, vous êtes redirigé vers la page d'authentification, d'où la récupération du code HTML dans notre requête ajax.

Nous allons donc créer une classe du même type que `Secured.java` mais dédiée aux appels JSON.

12.4.1 SecuredJson.java

```

package controllers;

```

```

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

import models.*;

import static play.libs.Json.toJson;

public class SecuredJson extends Security.Authenticator {

    @Override
    public String getUsername(Context ctx) {
        return ctx.session().get("email");
    }

    @Override
    public Result onUnauthorized(Context ctx) {
        return ok(toJson("failed"));
    }

}

```

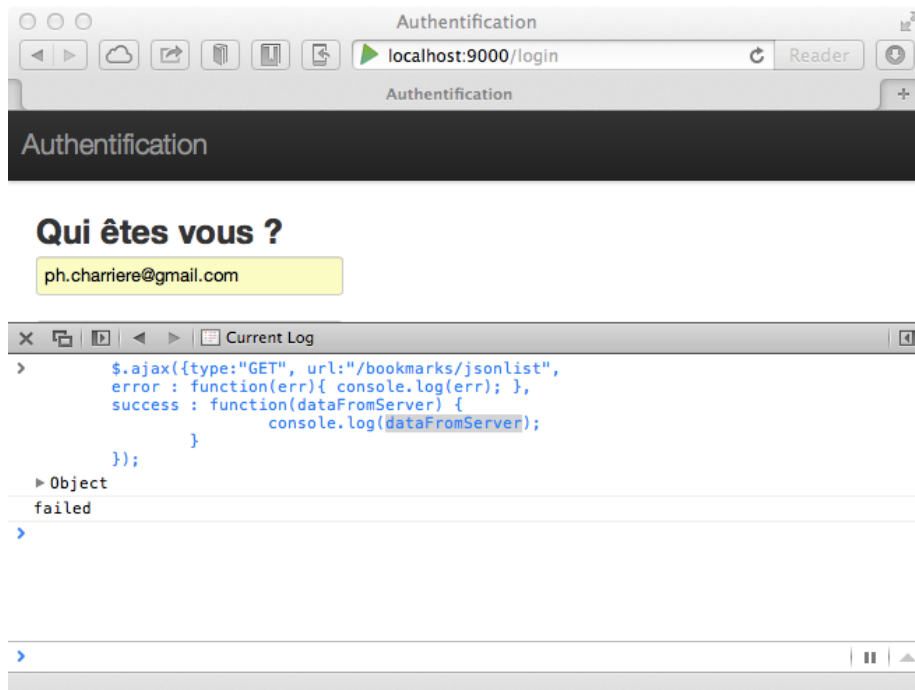
12.4.2 Modification de l'annotation dans Bookmarks.java

Remplacez `@Security.Authenticated(Secured.class)` par :

```
‘@Security.Authenticated(SecuredJson.class)’
```

12.4.3 Testons

Lancez à nouveau, dans la console du navigateur, votre requête ajax :



12.4.4 Mais comment puis m'authentifier via une requête ajax ???

Là aussi, nous allons devoir créer une classe du même type qu'`Authentication.java` mais qui ne redirige pas vers la page principale pour ne pas avoir du code HTML comme retour.

AuthenticationJson.java *Remarque : Je pourrais hériter de `Authentication.java`, mais pour le moment je vais dupliquer le code et le modifier.*

```
package controllers;

import play.*;
import play.mvc.*;
import play.data.*;

import models.*;
import views.html.*;

import static play.libs.Json.toJson;
```

```

public class AuthenticationJson extends Controller {

    public static class AuthenticatedUser {

        public String email;
        public String password;

        public String validate() {
            if(User.authenticate(email, password) == null) {
                return "oups";
            }

            return null;
        }
    }

    //On récupère les informations de login (quand le user se "signe")
    public static Result authenticate() {
        Form<AuthenticatedUser> loginForm =
            form(AuthenticatedUser.class).bindFromRequest();

        if(loginForm.hasErrors()) {
            return ok(toJson("badRequest"));
        } else {
            session("email", loginForm.get().email);
            User who = User.findByEmail(loginForm.get().email);
            return ok(toJson(who.name));
        }
    }

    //Fermer la session
    public static Result logout() {
        session().clear();
        return ok(toJson("bye"));
    }
}

```

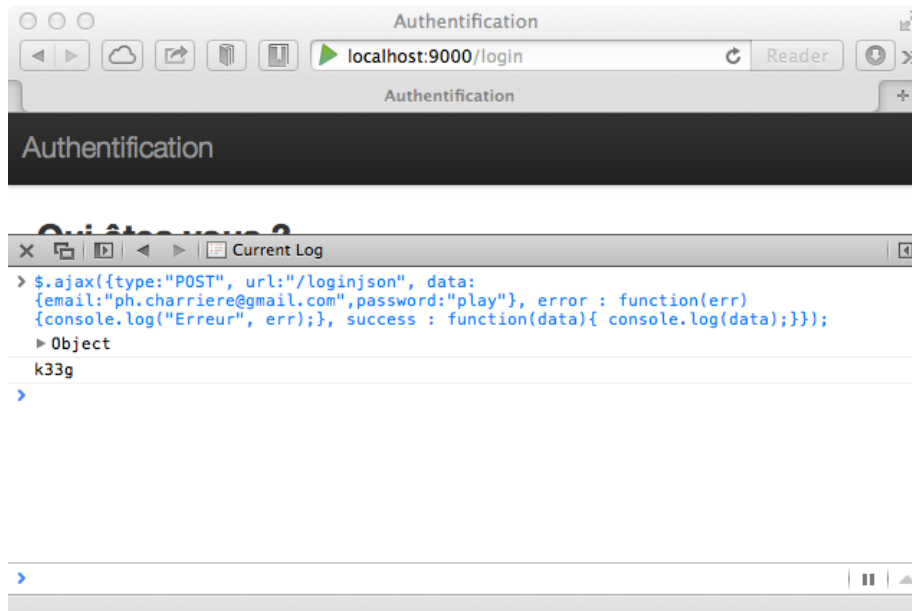
Ajoutons les routes

POST	/loginjson	controllers.AuthenticationJson.authenticate()
GET	/logoutjson	controllers.AuthenticationJson.logout()

Testons Dans la console du navigateur, essayez la commande suivante (vous ne devez pas être authentifié) :

```
$.ajax({
  type: "POST",
  url: "/loginjson", data: {email: "ph.charriere@gmail.com", password: "play"},
  error : function(err){console.log("Erreur", err);},
  success : function(data){ console.log(data);}
});
```

Vous allez obtenir ceci :

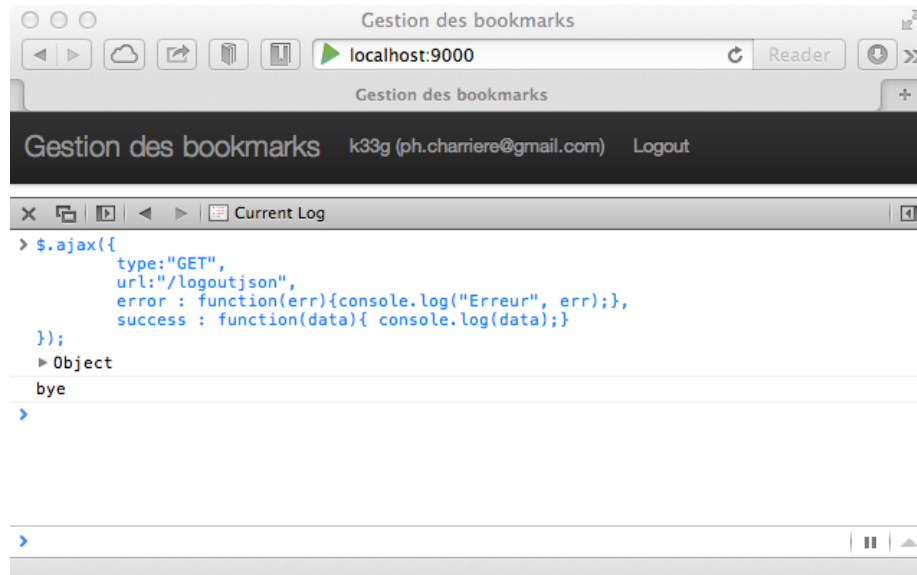


Et si vous rappelez l'url <http://localhost:9000>, vous apparaissez comme authentifié. Vous pouvez tester à nouveau votre requête pour récupérer la liste des bookmarks :

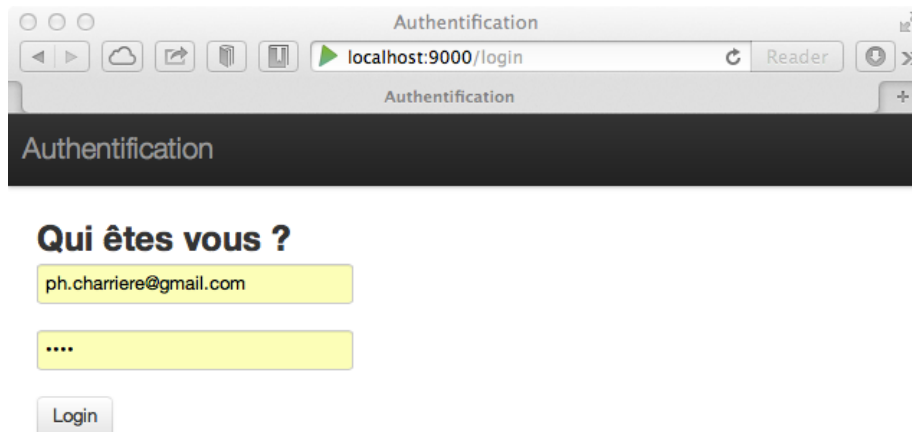
```
$.ajax({type:"GET", url:"/bookmarks/jsonlist",
  error : function(err){ console.log(err); },
  success : function(dataFromServer) {
    console.log(dataFromServer);
  }
});
```

Si vous souhaitez vous délogger (toujours via une requête Ajax) :

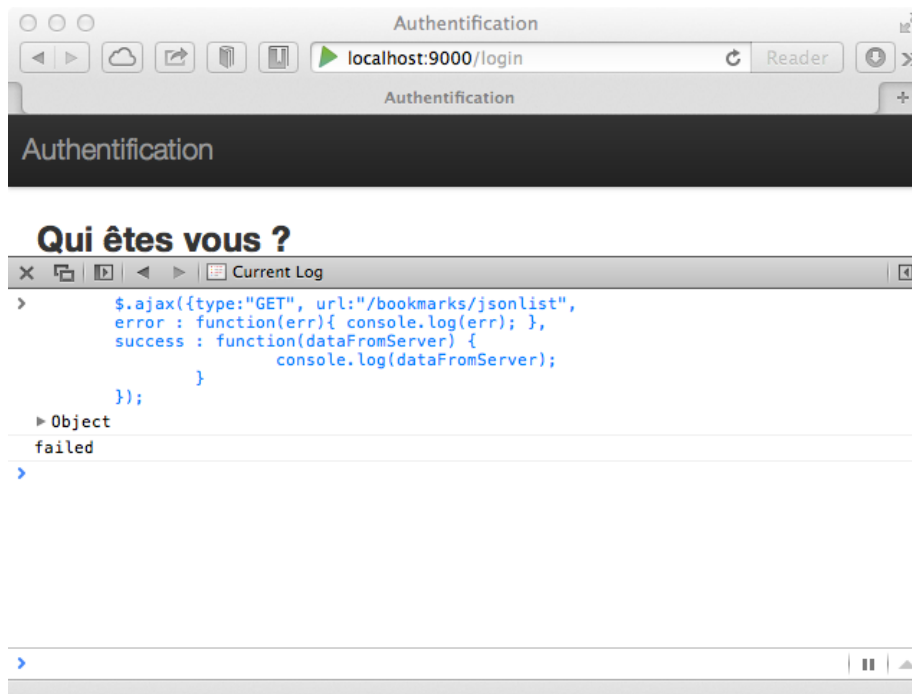
```
$.ajax({
  type: "GET",
  url: "/logoutjson",
  error : function(err){console.log("Erreur", err);},
  success : function(data){ console.log(data);}
});
```



Et si vous rappelez l'url <http://localhost:9000>, vous n'êtes plus authentifié.



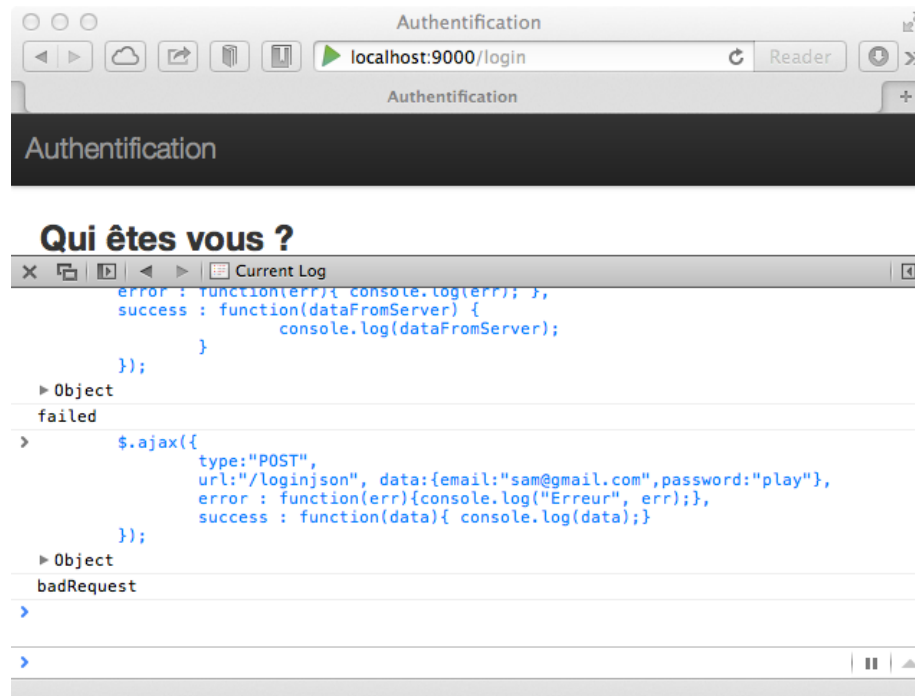
Si vous testez à nouveau votre requête pour récupérer la liste des bookmarks, vous aurez un message vous indiquant que ce n'est pas possible (**failed**) car non autorisé (cf. `SecuredJson.java`):



De la même manière, si vous tentez de vous authentifier avec un mauvais compte utilisateur :

```
$.ajax({
  type:"POST",
  url:"/loginjson", data:{email:"sam@gmail.com",password:"play"},
  error : function(err){console.log("Erreur", err);},
  success : function(data){ console.log(data);}
});
```

Vous obtiendrez un message badRequest (cf. AuthenticationJson.java) :



12.5 Utilisons tout ça ...

Mettons en applications ce que nous venons de voir pour faire quelque chose d'un peu plus "pratique" : développons une "Single page application" :

- nous allons pouvoir nous authentifier
- nous allons pouvoir charger les bookmarks
- il n'y aura aucun rechargement de page (à part la 1ère fois)

Commençons par créer un nouveau contrôleur :

12.5.1 Nouveau Contrôleur : SingleApp.java

Alors, c'est très simple, dans le répertoire `controllers`, créez un nouveau contrôleur avec le nom `SingleApp.java` :

```
package controllers;

import play.*;
import play.mvc.*;
import play.data.*;

import models.*;
import views.html.*;

public class SingleApp extends Controller {

    public static Result mainPage() {

        return ok(mainPage.render(
            "Single Page Application"
        ));
    }

}
```

12.5.2 routes

Ajoutons une route dans le fichier `routes` :

```
# Main Single Page Application
GET      /main                               controllers.SingleApp.mainPage()
```

Finalement le **gros** du travail va se faire en html et javascript. Nous disposons de jQuery (c'est fourni en standard avec Play!>), et si vous vous souvenez, nous avons installé Twitter Bootstrap. Nous allons donc voir comment faire une “single page application”, donc plus notions de templating : **je vais utiliser une vue “scala” mais qui ne contiendra que le minimum de code scala** (dans l'absolu nous pourrions utiliser une simple page html).

12.5.3 Nouvelle vue : mainPage.scala.html

Commencez par créer une nouvelle vue (dans le répertoire `views`) que vous nommerez `mainPage.scala.html`. Je suis reparti (copier/coller ... je sais) de

main.scala.html. Je ne fais que reprendre les requêtes “ajax” que nous avons utilisées précédemment pour tester nos services json.

Attention : mon code javascript n’est pas forcément compatible avec tous les navigateurs, je suis allé au plus simple et j’ai utilisé les possibilités de la dernière version de javascript (par exemple : le `forEach` sur un `array`).

```
@(title: String)
<!DOCTYPE html>

<html>
  <head>
    <title>@title</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" media="screen"
      href="@routes.Assets.at("bootstrap/css/bootstrap.css")">
    <style>
      body {
        padding-top: 60px;
        padding-bottom: 40px;
      }
    </style>
    <link rel="stylesheet" media="screen"
      href="@routes.Assets.at("bootstrap/css/bootstrap-responsive.css")">
    <link rel="shortcut icon" type="image/png"
      href="@routes.Assets.at("images/favicon.png")">
    <script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")"
      type="text/javascript"></script>
  </head>
  <body>

    <div class="navbar navbar-fixed-top">
      <div class="navbar-inner">
        <div class="container">
          <a class="brand">@title</a>

        </div>
      </div>
    </div>

    <div class="container">
      <!-- === Formulaire d'authentification === -->
      <fieldset>
        <legend>Authentification :</legend>
        <div class="well">
```

```

        <label>Email :
        <input type="email" name="email" placeholder="Email"></label>
        <label>Password :
        <input type="password" name="password"
            placeholder="Password"></label>
        <button name="login" class="btn btn-primary">Login</button>
        <button name="logout" class="btn">Logout</button>
    </div>
</fieldset>

<!-- === Les messages s'afficheront ici === -->
<div name="authentication" class="alert alert-info">
    <strong>Info : </strong> Veuillez vous authentifier s'il vous plaît
</div>

<!-- === Les bookmarks s'afficheront ici === -->
<fieldset>
    <legend>Liste des Bookmarks <button name="loadbookmarks"
        class="btn btn-inverse">Charger ...</button></legend>
    <ul name="bookmarks"></ul>
</fieldset>

</div>

</body>

<!-- === ici votre code applicatif === -->
<script>
    /*=== mon code ne s'exécute qu'une fois le DOM complètement chargé ===*/
    $(function (){

        //définition des différents éléments d'IHM
        var user = {}
        , alertAuthentication = $('div[name=authentication]')
        , loginButton = $("button[name=login]")
        , logoutButton = $("button[name=logout]")
        , loadBookmarksButton = $("button[name=loadbookmarks]")
        , email = $('input[name=email]')
        , password = $('input[name=password]')
        , labels = $('label')
        , bookmarksList = $('ul[name=bookmarks]');

        // onclick du bouton login
        loginButton.click(function(){

            user.email = email.val();

```

```

user.password = password.val();
$.ajax({
    type:"POST",
    url:"/loginjson", data:{
        email : user.email,
        password : user.password
    },
    error : function(err){console.log("Erreur", err);},
    success : function(data){
        if(data !== "badRequest") {
            alertAuthentication
                .attr('class','alert alert-success')
                .html('<strong>Bienvenue !</strong> ' + data);
            user.name = data;

            labels.hide();
            email.hide();
            password.hide();
            loginButton.hide();

        } else {
            alertAuthentication
                .attr('class','alert alert-error')
                .html('<strong>Oops !</strong> vous avez du vous tromper');
        }
    }
});

});

// onclick du bouton logout
logoutButton.click(function(){

    $.ajax({
        type:"GET",
        url:"/logoutjson",
        error : function(err){console.log("Erreur", err);},
        success : function(data){

            if(user.name) {
                alertAuthentication
                    .attr('class','alert alert-info')
                    .html('<strong>Au revoir</strong> ' + user.name);

                labels.show();
                email.show();
                password.show();
            }
        }
    });
});

```



```

        loginButton.show();

        user = {};

        bookmarksList.html('');
    }

    }

});

});

// onclick du bouton de chargement des bookmarks
loadBookmarksButton.click(function(){
    $.ajax({type:"GET", url:"/bookmarks/jsonlist",
        error : function(err){ console.log(err); },
        success : function(data) {

            if(data !== "failed") {
                bookmarksList.html('');
                data.bookmarks.forEach(function(bookmark){
                    bookmarksList.append(
                        $('<li>')
                            .append($('<b>').append(bookmark.title))
                            .append(' | ')
                            .append($('<a>').attr("href",bookmark.url)
                                .append(bookmark.url))
                            .append(' | ')
                            .append($('<i>')
                                .append(bookmark.details))
                            .append(' | (')
                                .append(bookmark.category.label).append(')')
                            );
                });
            } else {
                alertAuthentication
                    .attr('class','alert alert-error')
                    .html('<strong>Il faut être authentifié !</strong> \n
                        pour obtenir la liste des bookmarks');
            }
        }
    });
});
});

```

```

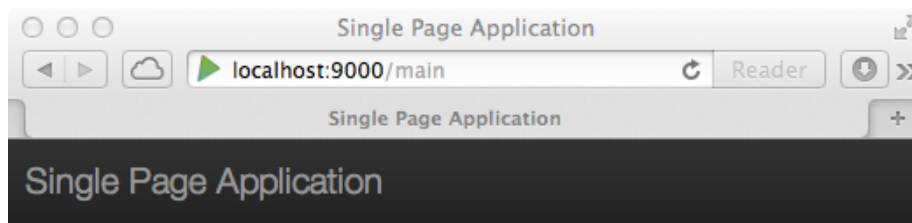
    });
</script>

</html>

```

12.6 Testons

Dans votre navigateur, appelez l'url : `http://localhost:9000/main`



Authentification :

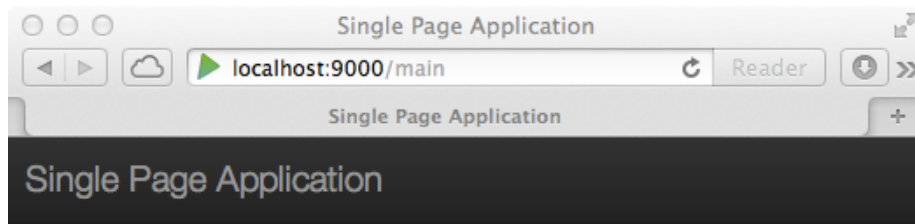
Email :

Password :

Info : Veuillez vous authentifier s'il vous plaît

Liste des Bookmarks

Si vous vous trompez en vous authentifiant :



Authentification :

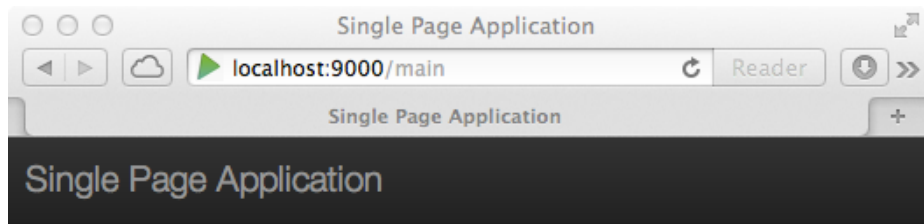
Email :

Password :

Oups ! vous avez du vous tromper

Liste des Bookmarks

Si vous essayez de charger les bookmarks alors que vous n'êtes pas authentifié :



Authentification :

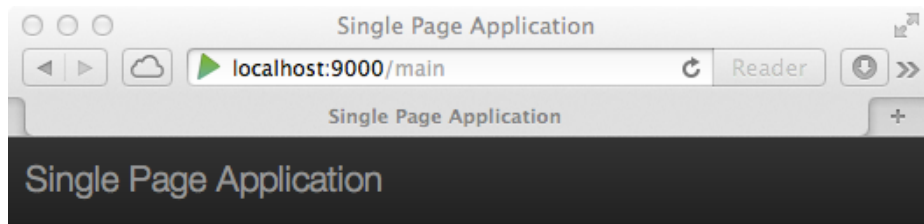
Email :

Password :

Il faut être authentifié ! pour obtenir la liste des bookmarks

Liste des Bookmarks

Si vous vous êtes correctement authentifié :



Authentification :

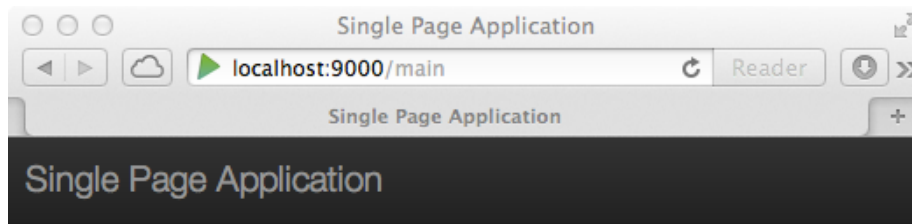
Logout

Bienvenue ! k33g

Liste des Bookmarks

Charger ...

Vous pouvez donc charger les bookmarks :



Authentification :

Logout

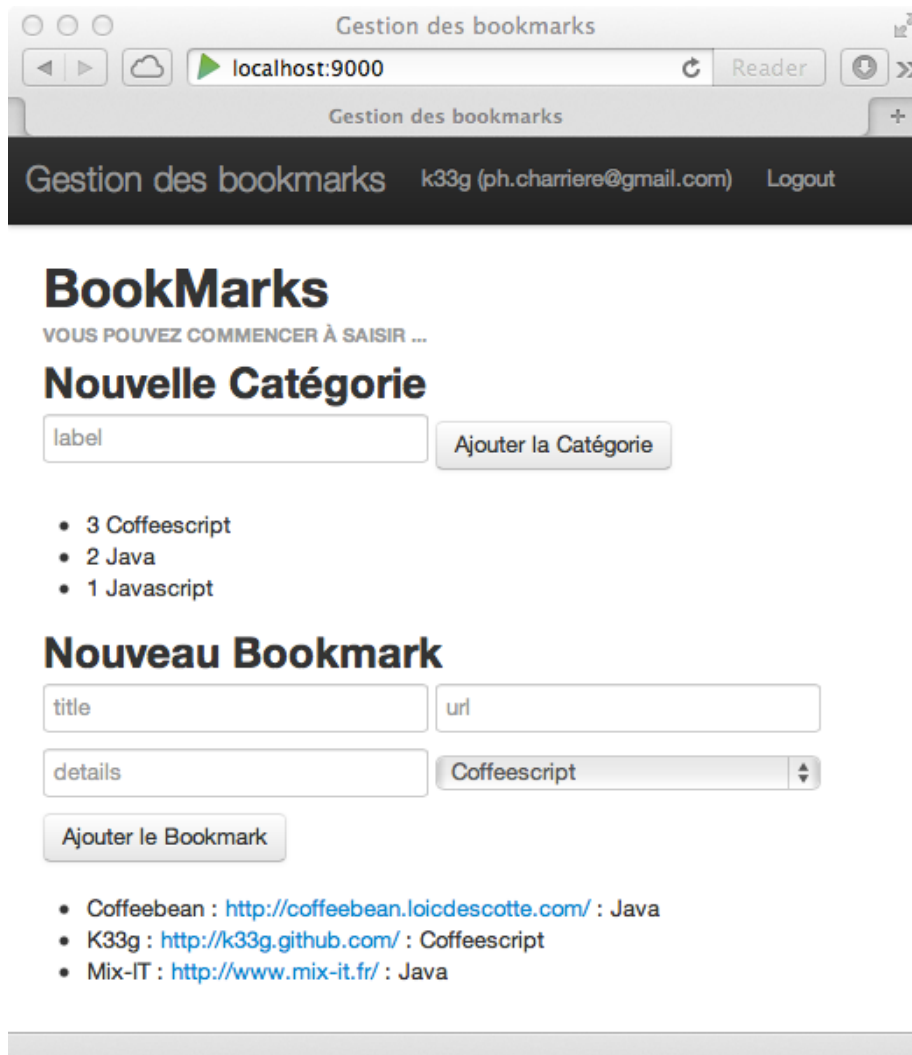
Bienvenue ! k33g

Liste des Bookmarks

Charger ...

- **Coffeebean** | <http://coffeebean.loicdescotte.com/> | *Java, Scala, Kotlin, ...* | (Java)
- **K33g** | <http://k33g.github.com/> | *Play et Javascript* | (CoffeeScript)
- **Mix-IT** | <http://www.mix-it.fr/> | *Java mais aussi Javascript* | (Java)

Si vous retournez à la racine du site : <http://localhost:9000/>, vous pouvez vérifier que vous avez effectivement été authentifié :



12.7 Conclusion

Il est donc finalement très possible de “faire” du **Play!> 2** (java) sans utiliser (ou presque) du scala dans les vues. Pensez-y lorsque vous faites du **Play!> 1** (le mécanisme décrit est tout à fait reproductible dans la version 1), cela peut faciliter vos migrations.

13 Les assets dans Play

Qu'allons nous voir ?

- *Qu'est-ce que c'est et pourquoi ?*
- *Découverte rapide de Coffeescript*
- *Utilisation de Coffeescript*
- *LESS ?*
- ...

13.1 Assets ???

Mais qu'est-ce donc ? En fait, ce sont tous les fichiers statiques (css, html, js, coffee, png, jpg ...) de votre application web Play. Vous les trouvez dans le répertoire `public` de l'application. Et pour y faire référence dans nos vues scala, nous utilisons le mot clé `@routes.Assets.at()` :

- pour les fichiers javascript : `<script src="@routes.Assets.at("javascripts/jquery-1.7.1.min.js")" type="text/javascript"></script>`
- pour les feuilles de style : `<link rel="stylesheet" media="screen" href="@routes.Assets.at("bootstrap/css/bootstrap-responsive.css")">`
- pour les images : `<link rel="shortcut icon" type="image/png" href="@routes.Assets.at("images/favicon.png")">`
- etc. ...

Mais il existe d'autres types d'assets.

13.2 Assets “compilés”

Ce sont les fichiers statiques qui subissent un retraitement avant publication, comme :

- la minification des fichiers javascript pour améliorer les temps de chargement
- la compression gzip pour les browsers qui le supportent
- la transformation de fichiers `less` en `css` (nous allons voir ce que c'est plus loin)

- la transpilation de fichiers coffeescript en javascript (ça aussi, nous allons le voir)
- ...

Remarque : Play gère la mise en cache des assets.

13.3 Mise en oeuvre

13.3.1 Préparation

Pour que Play compile les assets, il faut créer un répertoire **assets** dans le répertoire **app**, puis dans le répertoire **assets**, créez un répertoire **javascripts** et un répertoire **stylesheets**.

Nous allons commencer par des assets Coffeescript.

13.3.2 Coffeescript

Mais, tout d’abord une petite présentation de Coffeescript avant la mise en oeuvre.

Rappels Coffeescript c’est à la fois un langage et un transpiler (un run-time aussi). Vous écrivez en Coffeescript, puis vous transpilez en Javascript. Ce nouveau langage a été créé par Jeremy Ashkenas (<https://github.com/jashkenas/coffee-script> & <http://coffeescript.org/>).

Remarque : Coffeescript est “fourni” avec Play!>2.

Vous pouvez exécuter Coffeescript :

- côté client, dans le navigateur une fois transpilé en javascript ou directement en coffeescript avec un runtime javascript capable d’exécuter les script **coffee**
- côté serveur ou en mode commande avec Nodejs

Pourquoi ?

“CoffeeScript is JavaScript, the same language with a different accent”

Patrick Lee (CoffeeScript in Action)

Coffeescript simplifie le javascript, génère du javascript “propre” et apporte de nombreux “plus” pour vous faciliter la vie, simplifier votre code, en améliorer la lisibilité.

Quelques exemples avant de passer à la pratique.

Les fonctions en Coffeescript

```
addition = (a,b) ->
  a+b
```

Utilisation d'autres librairies javascript (et simplification)

```
#Ceci est une remarque :
#code js avant :
#   jQuery $(document).ready(function () {
#       some();
#       init();
#       calls();
#   });

$ -> some()
    init()
    calls()
```

Remarque : la syntaxe coffeescript facilite la création de DSL

Les “Strings” & les Interpolations Par exemple vous avez un objet :

```
bob =
  firstName : "Bob"
  lastName : "Morane"
  hello : ->
    "Hello !"
```

sont équivalent javascript serait :

```
var bob = {
  firstName : "Bob",
  lastName : "Morane",
  hello : function () {
    return "Hello !";
  }
}
```

Vous pouvez ensuite l'utiliser dans une String (et notez bien, sur plusieurs lignes) de la façon suivante :

```

console.log "
  Firstname : #{bob.firstName},
  LastName  : #{bob.lastName},
  Method (hello) : #{bob.hello()}
"

```

Les Arrays

```

buddies = [
  {name:"Bob", age:30}
  {name:"Sam", age:50}
  { name : "John", age : 20 }
]

#tous les copains de moins de 50 ans
result = (buddy for buddy in buddies when buddy.age < 50)

```

Et enfin : les CLASSES !!!

```

class Human

  #static field
  @counter : 0

  constructor : (@firstName, @lastName) ->
    #fields : @ = this
    Human.counter += 1

  #method
  hello : ->
    console.log "Hello #{@firstName} #{@lastName}"

  #static method
  @howMany : ->
    Human.counter

  Bob = new Human "Bob", "Morane"
  console.log "Human.counter #{Human.howMany()}"

```

Avec un peu d'héritage

```

class Superhero extends Human
  constructor : (@firstName, @lastName, @name) ->
  hello : ->
    super + " aka #{@name}"

```

Mise en oeuvre Nous allons re-écrire le code de notre “single page application” (cf chapitre “**Services (JSON)**”) en Coffeescript.

Préparation Tout d’abord, vous pouvez supprimer le code javascript dans la vie `mainPage.scala.html` (juste après la remarque `<!-- === ici votre code applicatif === -->`).

Ensuite, toujours dans la même vue, dans la section `<head>`, juste après la référence à jQuery, ajoutez une référence à notre futur code :

```
<script src="@routes.Assets.at("javascripts/myapp.js")" type="text/javascript"></script>
```

Nous allons ensuite créer un fichier `myapp.coffee` qui sera automatiquement compilé (transpilé) par Play en javascript.

myapp.coffee Dans le répertoire `app/assets/javascripts`, créez un fichier `myapp.coffee` avec le code suivant :

```
#=== mon code ne s'exécute qu'une fois le DOM complètement chargé ===
console.log "CoffeeScript version in progress ..."
$ ->
  #définition des différents éléments d'IHM
  console.log "dom loaded ... i hope ..."
  user = {}
  alertAuthentication = $ "div[name=authentication]"
  loginButton = $ "button[name=login]"
  logoutButton = $ "button[name=logout]"
  loadBookmarksButton = $ "button[name=loadbookmarks]"
  email = $ "input[name=email]"
  password = $ "input[name=password]"
  labels = $ "label"
  bookmarksList = $ "ul[name=bookmarks]"

  #onclick du bouton login
  console.log "OnClick login button definition ..."
  loginButton.click ->
    user.email = email.val()
    user.password = password.val()

  $.ajax
    type:"POST"
    url:"/loginjson"
    data:
      email : user.email
```

```

        password : user.password
error : (err)->
    console.log "Erreur", err
success : (data)->
    if data isnt "badRequest"
        alertAuthentication.attr("class","alert alert-success")
            .html "<strong>Bienvenue !</strong> #{data}"
        user.name = data
        labels.hide()
        email.hide()
        password.hide()
        loginButton.hide()
    else
        alertAuthentication.attr("class","alert alert-error")
            .html "<strong>Oups !</strong> vous avez du vous tromper"

#onclick du bouton logout
console.log "OnClick logout button definition ..."
logoutButton.click ->

$.ajax
    type:"GET"
    url:"/logoutjson"
    error : (err)->
        console.log "Erreur", err
    success : (data)->
        if user.name
            alertAuthentication.attr("class", "alert alert-info")
                .html "<strong>Au revoir</strong> #{user.name}"
            labels.show()
            email.show()
            password.show()
            loginButton.show()
            user = {}
            bookmarksList.html ""

#onclick du bouton de chargement des bookmarks
console.log "OnClick load bookmarks button definition ..."
loadBookmarksButton.click ->

$.ajax
    type:"GET"
    url:"/bookmarks/jsonlist"
    error : (err)->
        console.log "Erreur", err
    success : (data)->

```

```

if data isnt "failed"
  console.log data
  bookmarksList.html ""
  data.bookmarks.forEach (bookmark) ->
    bookmarksList.append $ ""
      <li><b>#{bookmark.title} |
      <a href='#{bookmark.url}'>#{bookmark.url}</a> |
      <i>#{bookmark.details}</i> |
      (#{bookmark.category.label})</li>
    ""
else
  alertAuthentication.attr("class", "alert alert-error")
  .html "
    <strong>
      Il faut être authentifié !
    </strong> pour obtenir la liste des bookmarks
  "

```

Enregistrez, relancez votre application (<http://localhost:9000/main>), et ça fonctionne comme avant. Alors, je suis d'accord, faire du Coffeescript, c'est quand même un gros changement. Mais vous n'êtes pas obligés, cependant je vous engage à donner une chance à ce langage, vous verrez, vous ne ferez plus du javascript comme avant.

Passons à une nouvelle “visions” des feuilles de styles avec **LESS**.

13.3.3 Less

Less <http://lesscss.org/> est une autre façon d'écrire vos feuilles de style. C'est un nouveau langage css, dynamique, plus pratique, avec la possibilité d'utiliser des variables, des opérations, ... Pour une présentation plus détaillée, allez faire un tour sur le blog de **Cedric Exbrayat** : <http://hypedrivendev.wordpress.com/2012/01/31/css-sucks-do-less/>.

Mise en oeuvre Alors, nous n'allons pas re-écrire Twitter Bootstrap (qui est lui aussi créé avec Less), mais ajouter des styles à notre vue principale `index.scala.html`.

Pour cela, allez d'abord dans la vue `main.scala.html` (qui est référencée dans `index.scala.html`), puis dans la section `<head>` de la vue, ajoutez une référence à notre future feuille de style :

```
<link rel="stylesheet" media="screen" href="@routes.Assets.at("stylesheets/mycss.css")">
```

myapp.coffee Dans le répertoire `app/assets/stylesheets`, créez un fichier `mycss.less` avec le code suivant :

```
@h1BckGrdColor : blue;
@h1ForeColor   : white;

@h2BckGrdColor : gray;
@h2ForeColor   : whitesmoke;

supertag {
  h1 {
    color: @h1ForeColor;
    background-color: @h1BckGrdColor;
    padding-left: 5px;
  }
  h2 {
    color : @h2ForeColor;
    background-color: @h2BckGrdColor;
    padding-left: 5px;
    margin-bottom: 5px;
  }

  input {
    background-color: yellow
  }
}
```

J'utilise donc 4 variables : `@h1BckGrdColor`, `@h1ForeColor`, `@h2BckGrdColor`, `@h2ForeColor` (cela permet de rendre le code plus paramétrable). Puis je crée un tag `supertag` et surcharge les styles `h1`, `h2` et `input`. Cela signifie que tous les tags `h1`, `h2` et `input` à l'intérieur d'un tag `supertag` prendront les styles définis dans notre fichier `less`.

index.scala.html Nous pouvons maintenant utiliser notre tag `<supertag>`. Allez modifier la vue `index.scala.html` : encadrez tous le code html par le tag `<supertag></supertag>` :

```
@(
  message: String,
  bookmarks: List[models.Bookmark],
  categories: List[models.Category],
  user: User
)
```

```

@main("Gestion des bookmarks", user) {

  <supertag>
  <h1>BookMarks</h1>
  <h6>@message</h6>
  <!-- Formulaire de saisie : Catégories -->
  <fieldset>
    <h2>Nouvelle Caté<sup>e</sup>gorie</h2>
    <form method="post" action="@routes.Categories.add()">
      <input name="label" placeholder="label">

      <button class="btn" type="submit">Ajouter la Caté<sup>e</sup>gorie</button>
    </form>
  </fieldset>

  @if(flash.containsKey("error")) {
    <div class="alert alert-error">
      <strong>Oops!</strong> @flash.get("error")
    </div>
  }

  <!-- Liste des Catégories -->
  <ul>
    @for(category <- categories) {
      <li>@category.id @category.label</li>
    }
  </ul>

  <!-- Formulaire de saisie : Bookmarks -->

  <fieldset>
    <h2>Nouveau Bookmark</h2>
    <form method="post" action="@routes.Bookmarks.add()">
      <input name="title" placeholder="title">
      <input name="url" placeholder="url">
      <input name="details" placeholder="details">

      <select size="1" name="category.id">
        @for(category <- categories) {
          <option value="@category.id">@category.label</option>
        }
      </select>

      <button class="btn" type="submit">Ajouter le Bookmark</button>
    </form>
  }
}

```



```

</fieldset>
<!-- Liste des Bookmarks -->
<ul>
  @for(bookmark <- bookmarks) {
    <li>@bookmark.title : <a href="@bookmark.url">@bookmark.url</a> :
      @if(bookmark.category != null) {
        @bookmark.category.label
      }
    </li>
  }
</ul>
</supertag>
}

```

Enregistrez. Lancez : Tadaaaaaa !

Gestion des bookmarks k33g (ph.charriere@gmail.com) Logout

BookMarks

VOUS POUVEZ COMMENCER À SAISIR ...

Nouvelle Catégorie

label

- 3 Coffeescript
- 2 Java
- 1 Javascript
- 4 Ruby

Nouveau Bookmark

- CoffeeBean : coffeebean.loicdescotte.com : Java
- K33G : www.k33g.org : Coffeescript

Oui je sais, c'est moche !

Entraînez vous maintenant, vous verrez, cela devient intéressant de faire du css ;).

14 Créer des templates Play

Qu'allons nous voir ?

- *Comment créer un template Play ré-utilisable avec giter8*

14.1 Pourquoi ?

Voilà, vous commencez à “bricoler” avec Play 2. Vous en avez fait la pub à tout vos camarades de travail, du coup, vous devez faire des démos à un peu tout le monde. Et vous vous apercevez qu'à chaque fois, vous devez à chaque fois

- copier les fichiers twitter bootstrap,
- créer un répertoire `models`,
- que finalement vous avez besoin de l'authentification, le modèle `user`, le controller associé,
- que la page de démarrage, le fichier de conf, ... ce sont toujours les mêmes
- etc. ...

Et un “ingénieur informaticien”, c'est faignant ...

La possibilité de faire des templates est prévues dans Play 2, mais n'est pas encore disponible à l'heure où j'écris (en tous les cas je ne suis pas au courant).

Cependant il y a une solution (Play 2 va probablement utiliser les mêmes composants) : **Giter8** <https://github.com/n8han/giter8>.

Giter8 est un outil en ligne de commande qui permet de générer une structure projet à partir d'un template projet hébergé sur github (donc pour le moment, pas le choix, il faut héberger ça sur github).

J'ai pu tester sous windows et OSX, je n'ai pas eu de problème. Il y a quelques pré-requis avant d'arriver à installer **Giter8**, mais vous allez voir tout est simple.

14.2 Prérequis : Installation de conscript

Conscript est un utilitaire qui va nous permettre d'installer **Giter8**.

- aller ici : <https://github.com/n8han/conscript#readme>
- télécharger “conscript runnable jar” : <https://github.com/downloads/n8han/conscript/conscript-0.4.0.jar>
- lancer **Conscript** (en général un double-click sur le jar suffit, sinon : `java -jar conscript-0.4.0.jar`)
- attendre, un message va vous avertir que **cs** a été installé
- mettre **cs** dans votre path

14.3 Installation de giter8

Tout simplement :

- en mode commande, tapez `cs n8han/giter8 ...` patientez
- c'est fini

14.4 Création/Préparation de votre template projet

Créez un répertoire (que vous penserez à pousser sous github ensuite) avec la structure suivante : j'ai appelé mon répertoire `play-java-lazy.g8` : Les templates sont sur des repositories github et ont un suffixe `.g8`.

... et il contient ceci :

```
play-java-lazy.g8
+-project
| +-plugin.sbt
|
+-src/
  +-main/
    +-g8/
      +-default.properties
      +-$application_names$/
      +-app/
      +-conf/
      +-application.conf
    +-project/
```

```
|    +-build.properties
|    +-Build.scala
|    +-plugin.sbt
|    +-public/
|    +-README.md
|
```

14.4.1 Dans /project/plugin.sbt :

```
addSbtPlugin("net.databinder" %% "giter8-plugin" % "0.4.4")
```

14.4.2 Dans /src/main/g8/default.properties :

```
description = This template generates a Java play 2.0.x project with some goodies
play_version=2.0.2
application_secret = unicornslovecats
application_name = killer_app
verbatimim = *.html *.js *.css *.coffee
```

14.4.3 Dans /src/main/g8/\$application_names\$/ :

Vous copiez les répertoires les répertoires de votre projet “template”.

14.4.4 Vérifier que /src/main/g8/\$application_names\$/project/Build.scala a bien le contenu suivant:

```
import sbt._
import Keys._
import PlayProject._

object ApplicationBuild extends Build {

    val appName          = "$application_name$"
    val appVersion       = "1.0"

    val appDependencies = Seq(
        // Add your project dependencies here,
    )

    val main = PlayProject(appName, appVersion, appDependencies, mainLang = JAVA).setting(
        // Add your own project settings here
    )
}
```

```
}
```

Ce qui est important, c'est la ligne `val appName = "$application_name$"`, cela permet de choisir le nom de votre projet à la création de celui ci (ainsi que le nom du répertoire projet).

14.4.5 Dans `/src/main/g8/$application_names$/conf/application.conf` :

Vous pouvez rendre le paramètre `application.conf` paramétrable (valeur par défaut dans `default.properties`) :

```
# If you deploy your application to several instances be sure to use the same key!
application.secret="$application_secret;format=random"$
```

14.4.6 Vérifier que dans `/src/main/g8/$application_names$/project/plugin.sbt` :

Il y a la bonne version de **Play** :

```
// Comment to get more information during initialization
logLevel := Level.Warn

// The Typesafe repository
resolvers += "Typesafe repository" at "http://repo.typesafe.com/typesafe/releases/"

// Use the Play sbt plugin for Play projects
addSbtPlugin("play" % "sbt-plugin" % "2.0.2")

sbtPlugin := true
```

14.4.7 Vérifier que dans `/src/main/g8/$application_names$/project/build.properties` :

Il y a la bonne version de **sbt** : par exemple `sbt.version=0.11.3`

Vous pouvez trouver un exemple ici : <https://github.com/k33g/play-java-lazy.g8>.
(PS: ne pas tenir compte du fichier `mycommands.scala`, c'est juste un test)

Mais pour être "plus sûr", vous pouvez aller voir l'exemple de chez **TypeSafe** :
<https://github.com/typesafehub/play-java.g8>.

14.4.8 Poussez moi tout ça sous github

- en mode commande pour les champion
- avec le fabuleux client github (pour moi)

14.5 Installation d'un template :

Maintenant que votre modèle de projet giter8 est publié sous github, vous pouvez l'installer simplement, en tapant la commande suivante dans un terminal :

```
g8 k33g/play-java-lazy.g8
```

Puis répondez aux questions, par exemple :

```
This template generates a Java play 2.0.x project with some goodies
```

```
application_secret [unicornslovecats]:  
application_name [killer_app]:  
play_version [2.0.2]: 2.0.3
```

Et voilà, votre application est créée, vous n'avez plus qu'à lancer.

15 Coder son application Play!> en Scala

15.1 Intro

Il s'agit pas de refaire un tuto entier sur Scala ou de redétailler toute l'API Play Framework en version Scala...

Par contre nous allons expliquer ce que ça apporte d'utiliser Scala à travers quelques exemples, les différences avec la version Java et surtout "pourquoi Scala colle à l'esprit de Play". Ce chapitre peut donc être vu comme un chapitre bonus, sans lien avec le tuto "Bookmarks", pour vous donner envie de découvrir Scala.

15.2 Liens entre Stateless et programmation fonctionnelle

Si vous avez lu cet ebook depuis le début, vous savez que Play!> est basé sur une architecture sans état côté serveur.

La programmation fonctionnelle permet de pousser ce concept à un niveau maximum en évitant d'utiliser des variables mutables. Les variables mutables

sont des variables dont la valeur peut changer au cours du temps. Prenez l'exemple d'un compteur que l'on incrémente.

Play!> ne conserve aucun état entre 2 requêtes HTTP. Cependant, au niveau d'une requête, une variable mutable dans une méthode constitue quelque chose qui se rapproche d'un état. Techniquement on ne parle pas du même type d'état, mais on peut rapprocher les 2 concepts... Si les accès concurrents (sur plusieurs threads) sont mal gérés, une variable mutable peut être source d'erreur. Si un client A et un client B accèdent en même temps à une telle variable et que l'un d'entre eux la modifie, que se passe-t-il? La programmation fonctionnelle permet d'éviter ce genre de soucis.

Exemple du calcul de la somme des chiffres d'une liste en programmation itérative :

```
val numbers: List(1, 2, 4)
var total = 0

for(i <- numbers){
    total += i
}

//total = 7
```

Si le total est un membre de classe susceptible d'être lu et modifié par plusieurs personnes en même temps, on va devoir jouer avec les locks pour éviter les problèmes... en plus de perdre en performance (les traitements seront bloquants) le code risque de devenir très vite compliqué pour pas grand chose.

Voici un exemple du calcul de la somme des chiffres d'une liste en programmation fonctionnelle :

```
val numbers: List(1, 2, 4)

val total = sum(0, numbers) //total = 7

def sum(total: Int, xs: List[Int]) : Int ={
    if(xs.isEmpty) total
    else sum(total+xs.head, xs.tail)
}
```

On dit d'un code basé uniquement sur des données immutables qu'il est sans effet de bords : aucun risque de fausser un résultat en affectant une valeur non désirée, ou en inversant l'ordre des affectations... Ici on calcule la somme de manière récursive (head renvoie le premier élément de la liste, tail tous les autres). Le total intermédiaire n'est jamais stocké dans une variable, il est local à chaque itération de la fonction.

Scala pousse bien sur à utiliser la deuxième solution.

Les ordinateurs possèdent un nombre toujours croissants de coeurs, les programmes optimisés utilisent donc de plus en plus d'accès concurrents. L'API Scala étant basée sur ces principes, elle permet de bénéficier directement de la force de la programmation parallèle.

Voici comment on filtre une liste en Scala :

```
val result = data.filter(line => line.contains("keyword"))
```

Pour faire la même avec un traitement parallèle il suffit d'ajouter `.par` après la référence à notre liste de données :

```
val result = data.par.filter(line => line.contains("keyword"))
```

15.3 Iteratee

Aujourd'hui on voit de plus de plus de sites web dont les données affichées se rafraîchissent en temps réel. Plutôt que de lancer des requêtes au serveur toutes les x secondes, il est possible de d'envoyer des données en "push", du serveur vers le navigateur. Pour cela il existe plusieurs solutions dont Comet, WebSocket et SSE. WebSocket et SSE (Server Sent events) font partie de la spécification HTML5 et nécessitent donc un navigateur récent. Comet (ou long polling) permet via différentes techniques de pousser des données vers le navigateur à partir d'une première connexion (une requête du navigateur) dont la durée est infinie.

Pour cela on utilise HTTP de manière asynchrone. Pour ce type de requêtes, Play!> va traiter la demande, libérer les threads de connexion, puis rappeler le client lorsque les données seront disponibles. Ceci a 2 avantages : tout est traité de manière non bloquante (le navigateur n'est pas bloqué en attente d'un résultat si on utilise les WebSockets par exemple) et on y gagne en consommation de ressources (moins de threads utilisés sur le serveur).

Play propose une API nommée Iteratee qui permet de manipuler des flux de données de manière totalement asynchrone afin de répondre à ces problématiques.

15.3.1 Un exemple avec Comet

Dans cet exemple nous allons voir commencer mixer 2 flux Twitter pour les afficher en temps réel, à l'aide de l'API Iteratee.

Vous pouvez voir la solution complète en récupérant ce [mini project](#)

15.3.2 Controleur

Définissons une méthode `comet` dans notre controleur :

```
def comet(query1: String, query2: String) = Action {  
  
  lazy val results1 = getStream(query1)  
  
  lazy val results2 = getStream(query2)  
  
  //pipe result 1 and result 2 and push to comet socket  
  Ok.stream(results1 >- results2 &> Comet(callback = "parent.messageChanged"))  
  
}
```

`query1` et `query2` sont de simples chaînes de caractères utilisées pour lancer une recherche, comme “java” ou “ruby”.

`results1` et `results2` vont contenir les résultats des recherches Twitter correspondant.

Dans la dernière ligne, `Ok.stream` va envoyer une réponse HTTP au client sous forme de chunks, c’est à dire par morceaux. Cela signifie qu’au lieu d’une réponse complète, la navigateur va recevoir des morceaux de réponse progressivement.

`results1 >- results2` va effectuer un “pipe”, pour mixer les réponses des 2 recherches. `&> Comet(callback = "parent.messageChanged")` va pousser le tout sur une socket Comet.

Voyons maintenant comment récupérer les résultats de Twitter. Pour cela nous utiliserons des `enumerators`. Les enumerators font partie de l’API Iteratee de Play!>.

Il est important de savoir que dans cette API de manipulation des données

- Iteratee représente un consommateur de données asynchrone
- Enumerator représente un fournisseur de données asynchrone

Les enumerators permettent donc de fournir des données à un iteratee qui va les consommer de manière asynchrone et non bloquante. Cela peut paraître compliqué mais ne vous inquiétez pas, le framework fera tout ce travail pour vous lorsque vous combinerez les enumeratos avec l’objet Comet.

```
private def getStream(query: String) = {  
  Enumerator.fromCallback[String]() =>  
    Promise.timeout(WS.url("http://search.twitter.com/search.json?q="+query+"&rpp=1").get)
```

```

        (response.json \\ "text").headOption.map(query + " : " + _.as[String])
      })
    )
  }

```

Ce code dit que toutes les secondes, on va demander les nouveaux tweets correspondant à nos requêtes. `Enumerator.fromCallback` attend une fonction qui retourne une “promesse” de réponse. Quand cette réponse sera prête, elle sera poussée (de manière asynchrone) à la socket comet. On combine ceci avec `Promise.timeout` pour demander les nouveaux résultats à Twitter toutes les secondes. Nous obtenons alors une promesse de réponse, et pas une réponse! C’est pour ça qu’on utilise `flatMap` pour récupérer directement la réponse contenue dans la promesse (si elle existe).

Un autre trick, `response.json \\ "text"` aide à parser la réponse JSON envoyée par Twitter à en extraire le contenu.

Adapter le contenu avec un `enumeratee` Un `enumeratee` est une sorte d’adaptateur dans l’API `Iteratee`. Nous allons utiliser ce concept pour transformer les résultats envoyés au navigateur. Voyons un exemple très simple : envoyer tous les tweets en majuscule.

```

scala val upperCase = Enumeratee.map[String] { tweet => tweet.map(_.toUpperCase)
} Note : En scala, '_' permet de représenter un élément courant. On aurait aussi
pu écrire tweet.map((tweet :String) => tweet.toUpperCase) }

```

Pour insérer cette transformation dans le pipe juste avant d’envoyer le contenu à notre socket Comet, nous devons simplement modifier notre code comme ceci :

```

Ok.stream(results1 >- results2 &> upperCase &> Comet(callback = "parent.messageChanged"))

```

Note : `&>` est juste un alias pour la méthode “through”.

Enfin on a plus qu’à utiliser la technique “iframe hack” to démarrer le streaming dans le navigateur (voir `index.scala.html` dans le projet).

Enjoy!!

Si vous voulez en savoir plus sur les `iteratees` je vous conseille [cet article](#) (en français).

15.3.3 D’autres exemples intéressants

- [Un jeu d’échecs](#) “real time” codé avec Play!>
- [Affichage d’une carte en streaming](#) - [Explicitations et code](#)
- [ReactiveMongo](#), un driver MongoDB asynchrone et non bloquant

Note : ces API [existent en Java](#) mais apportent moins de souplesse et de lisibilité que leur équivalent Scala à cause des faiblesses du langage Java. Elles sont donc plus souvent utilisées en Scala.