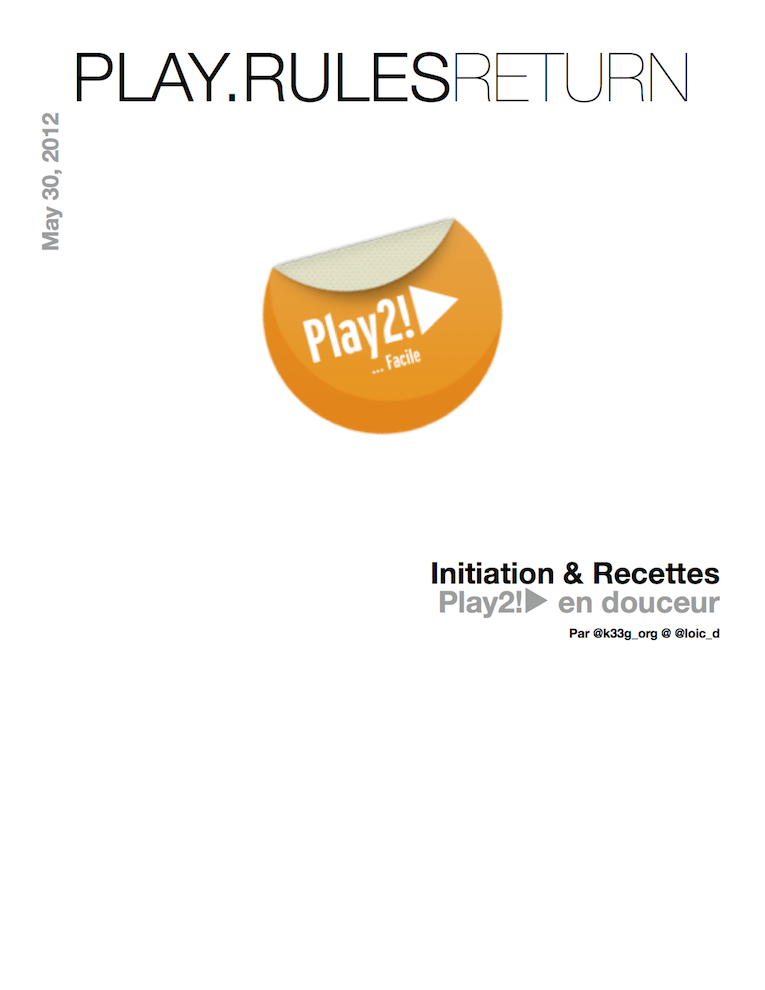
Play.Rules Return



## Remerciements

//TODO: à compléter

* @kraco\_fr : pour ses "tips Play"
* @LyonJUG : pour son équipe qui "maintient l'envie de continuer"

# Préface

Ce livre est conçu et écrit par [@k33g\_org](http://www.twitter.com/k33g_org) et [@loic\_d](http://www.twitter.com/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).

# 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

# Qui sommes-nous ?

//TODO

# Dédicaces

//TODO

# 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 regénéré 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!►.

# Remarques

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

# Introduction - Pourquoi Play!►

//TODO: ...

# Installation

**version de Play!► utilisée : 2.0.1**

/\* --- Qu'allons nous voir ? ---  
  
 - Installation de Play2!►  
 . sous OSX  
 . sous Linux  
 . sous Windows  
\*/

## 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.

## 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:/ENDROIT\_OU\_VOUS\_AVEZ\_DEZIPPE/play-2.0.1

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

où ENDROIT\_OU\_VOUS\_AVEZ\_DEZIPPE 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).

## Modification sous Linux

//TODO

## Modification sous Windows

//TODO

## 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 :



Voilà c'est prêt, nous pouvons commencer.

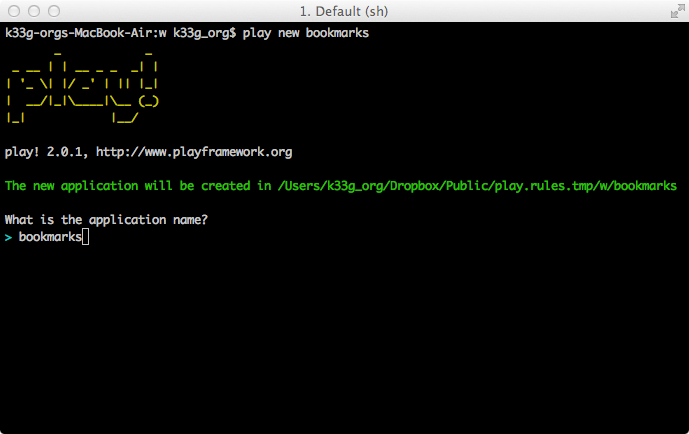
# 1er contact

/\* --- Qu'allons nous voir ? ---  
  
 - Comment générer le squelette de notre future application  
 - Comment lancer l'application  
  
 ... jusque là, ça va ;)  
\*/

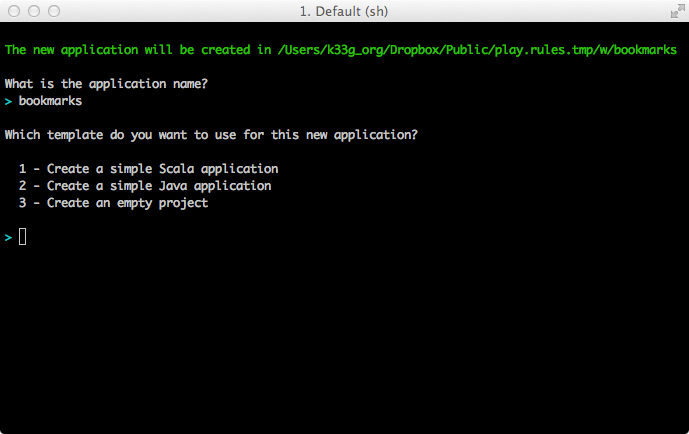
## Génération du squelette de l'application

* Installez vous dans votre répertoire de travail (cd mon\_repertoire\_de\_travail)
* Tapez la commande play new bookmarks
* validez et suivez les instructions

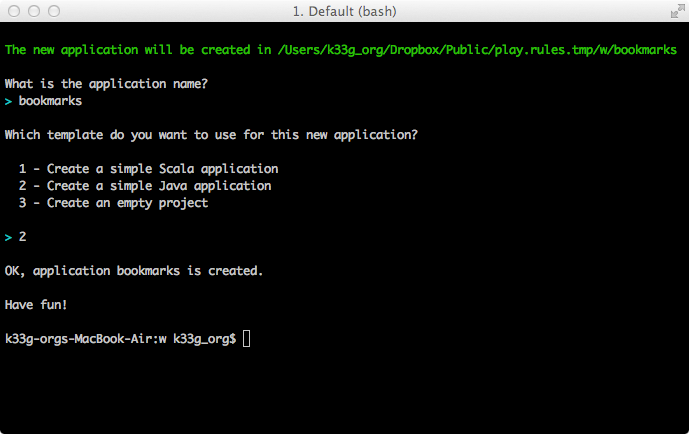
Play!► vous propose un nom par défaut pour votre application : acceptez



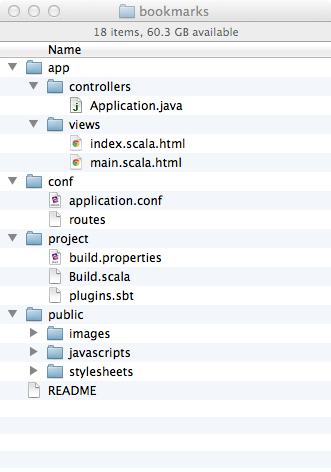
Play!► vous demande quel type de projet vous souhaitez générer, choisissez la version Java (deuxième choix donc) et validez :



C'est terminé :



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

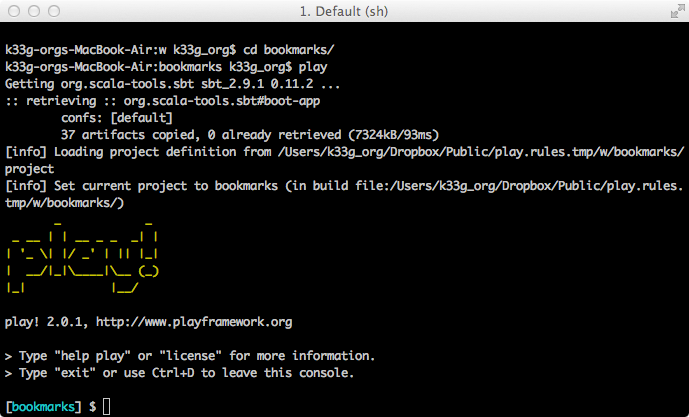


Pour plus de détail sur l'anatomie d'une application Play!►, allez faire un tour par là : <http://www.playframework.org/documentation/2.0.1/Anatomy>

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] :



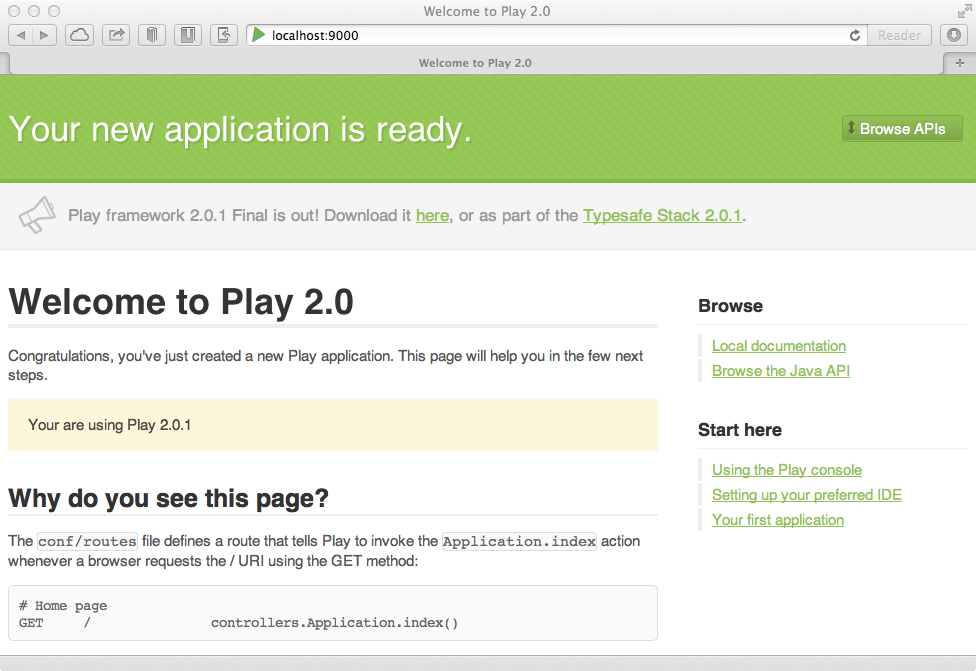
Tapez run et validez :



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 :



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

 #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 :

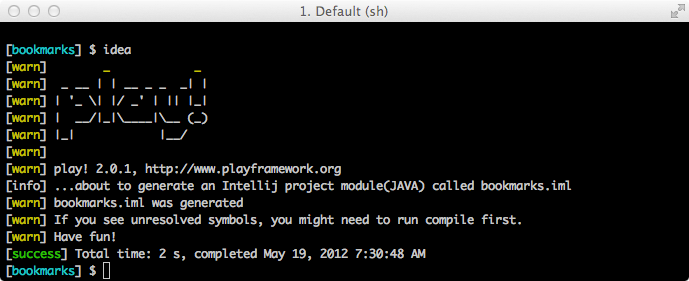
* IntelliJ
* NetBeans (il reste à ce jour encore quelques réglages/développements à réaliser)
* Eclipse

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 ...

## 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 :



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

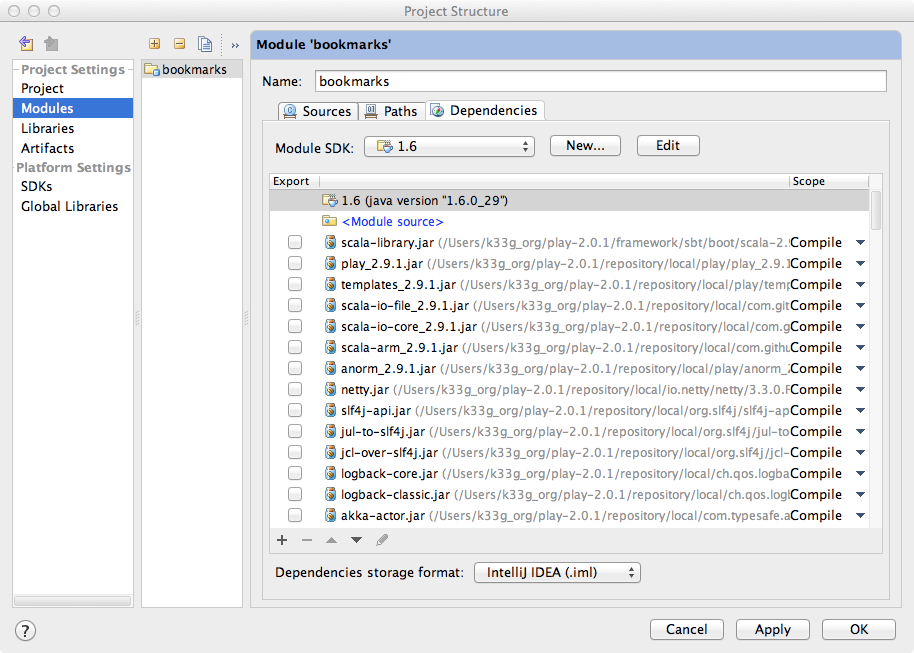
Passons au paramétrage du projet :

* 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"

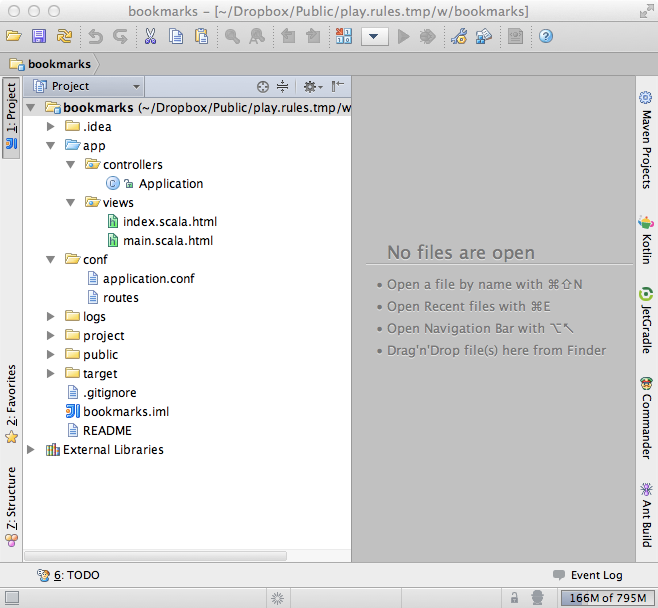
IntelliJ va vous afficher une fenêtre "Project Structure" :

* 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>

# 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.  
  
\*/

## 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.)

## Construisons les bases de notre application : paramétrages

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

### 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 souhaitions utiliser la base de données "H2 Database" en mode fichier. Comme cela toutes nos modifications seront sauvegardées.

### 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 : Dé commentez la ligne : ebean.default=models.\*

|  |
| --- |
| **Remarque :** Play!► 2 utilise **Ebean** comme framework de persistance au lieu de Hibernate+JPA (comme le faisait Play!► v°1). 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 |

## Les modèles

Nous allons étudier ça par l'exemple.

### Création d'un 1er modèle : Category

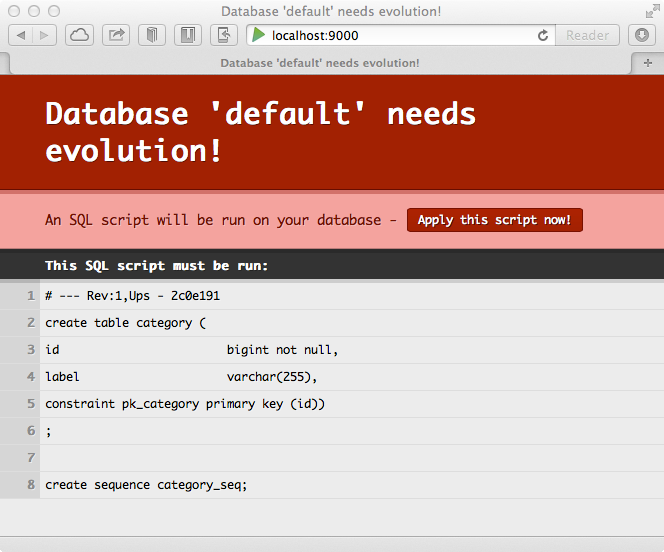
Créez une classe Category.java dans app/models :

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);  
  
 }

|  |
| --- |
| 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" per 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.

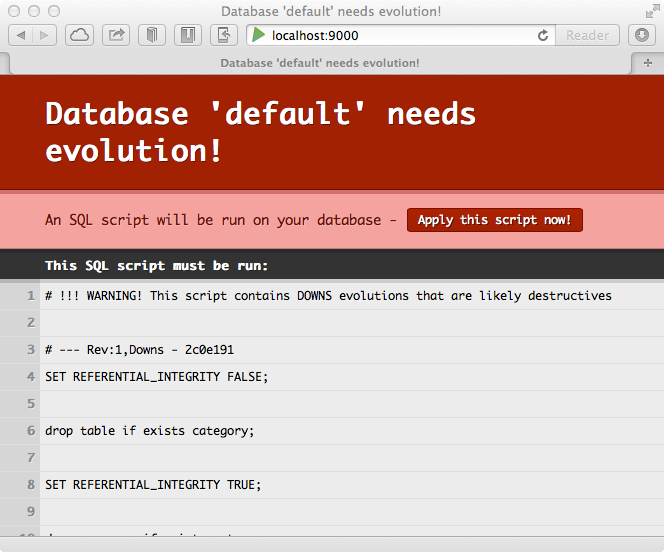
### Création d'un 2ème modèle : Bookmark

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.

## 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).

### 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 ressemblez 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.

### 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()

### 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 :

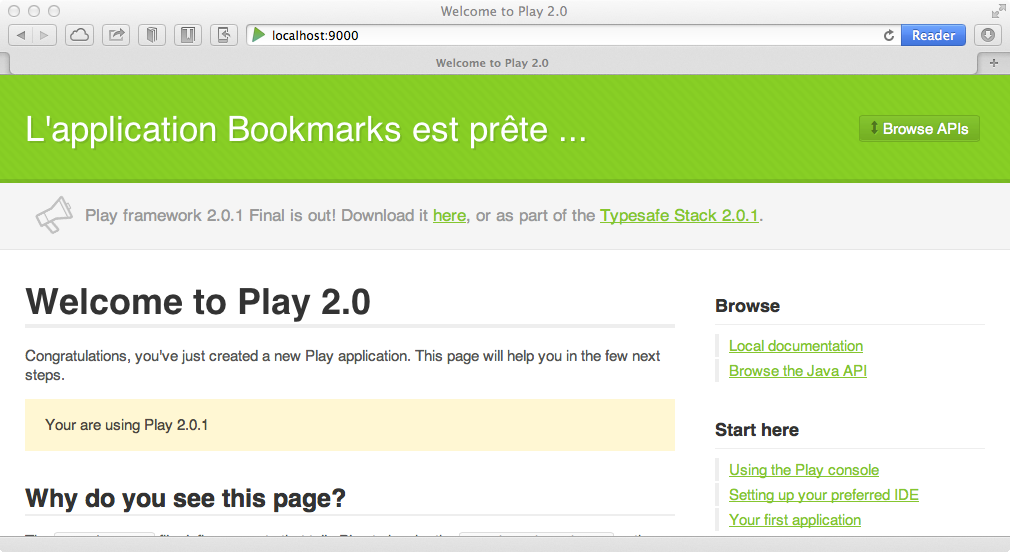
@(message: String)  
  
@main("Welcome to Play 2.0") {  
  
 @play20.welcome(message, style = "Java")  
  
}

#### 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

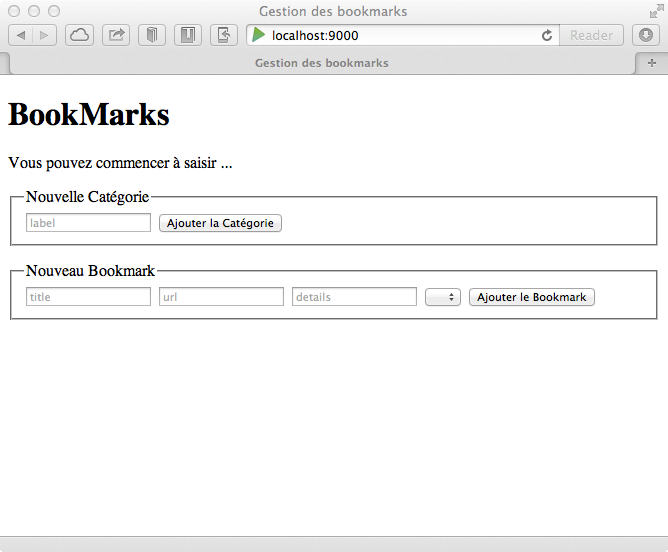
Et allons tout de suite modifier notre vue, pour enfin avoir quelque chose à montrer

## Les vues

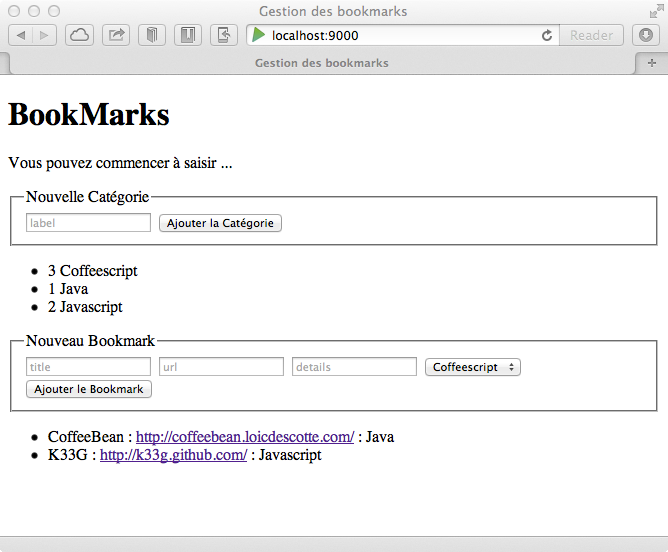
### Modification de la vue principale : index.scala.html

@(  
 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 Cat&eacute;gorie</legend>  
 <form method="post" action="@routes.Categories.add()">  
 <input name="label" placeholder="label">  
 <button type="submit">Ajouter la Cat&eacute;gorie</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()">  
 <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 :



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 Cat&eacute;gorie</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.

# 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!►.

## 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

## 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)

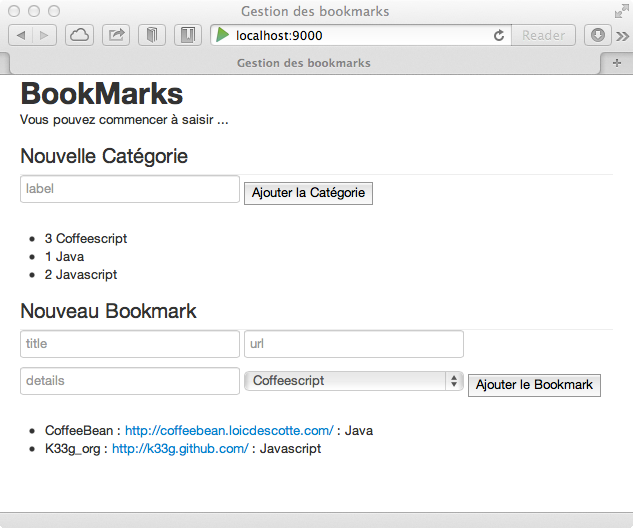
index "appelle" main en lui passant en paramètre le titre de la page et le contenu HTML. Et c'est dans main.scala.html que sont déclarées les ressources javascript et css avec la commande @routes.Assets.at.

### 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 :

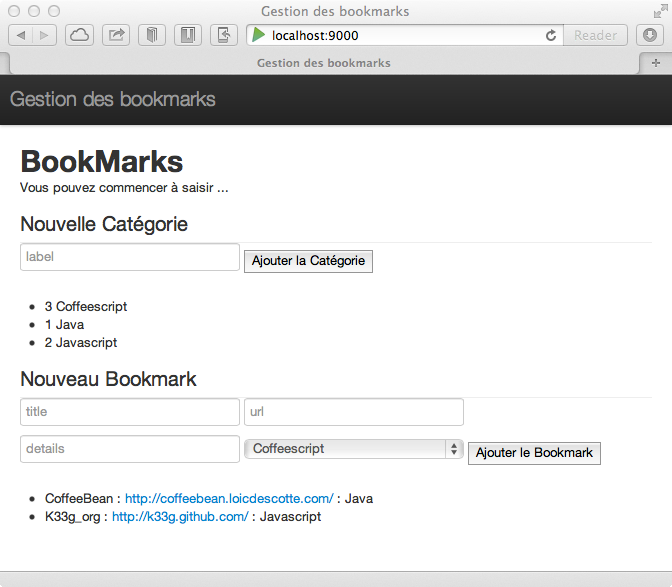


#### Customisons légèrement main.scala.html

Modifiez le tag <body> de la façon suivante :

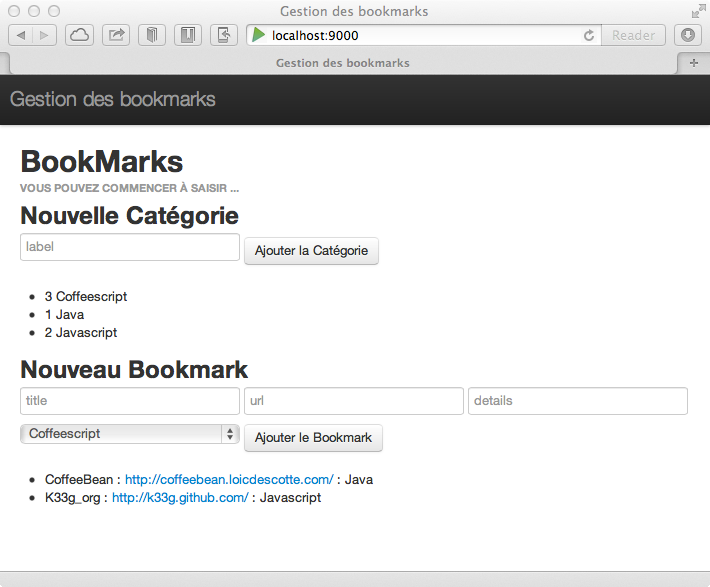
<body>  
  
 <div class="navbar navbar-fixed-top">  
 <div class="navbar-inner">  
 <div class="container">  
 <a class="brand">@title</a>  
 </div>  
 </div>  
 </div>  
 <div class="container">  
 @content  
 </div>  
  
 </body>

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 !

# 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"));  
  
 }  
 }  
  
 }  
  
 }

Vous pouvez rafraîchir votre page pour vérifier.

# Gestion de l'authentification

/\* --- Qu'allons nous voir ? ---  
  
 - Comment mettre en oeuvre un système simple d'authentification  
  
 Remarque : la rédaction 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 saisir pas saisir des bookmarks, nous allons mettre en place un système d'authentification.

## Création d'un modèle User

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

## 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());  
 }  
  
 }

## 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())  
 ));  
 }  
  
 }

## 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! râté! Essaye encore une fois";  
 }  
 return null;  
 }  
 }  
  
 public static Result login() {  
 return ok(  
 login.render(form(AuthenticatedUser.class))  
 );  
 }  
  
 //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 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 êtes déconnecté(e)");  
 return redirect(  
 routes.Authentication.login()  
 );  
 }  
  
 }

## Allons modifier les routes

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

## 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 :

### 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>

### 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) { ...

## 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>  
 }  
 }

## Allons ajouter des users dans initial-data.yml

### Ajoutons des utilisateurs :

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

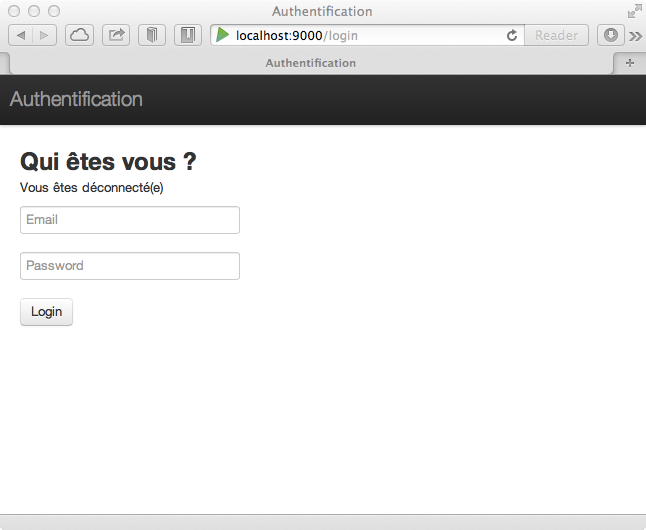
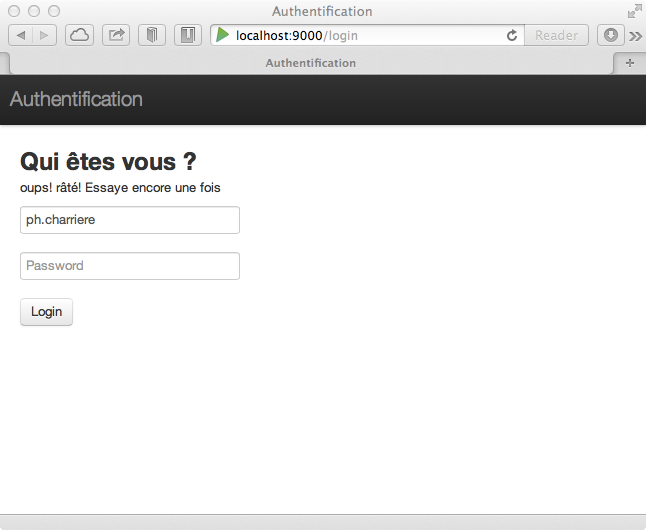
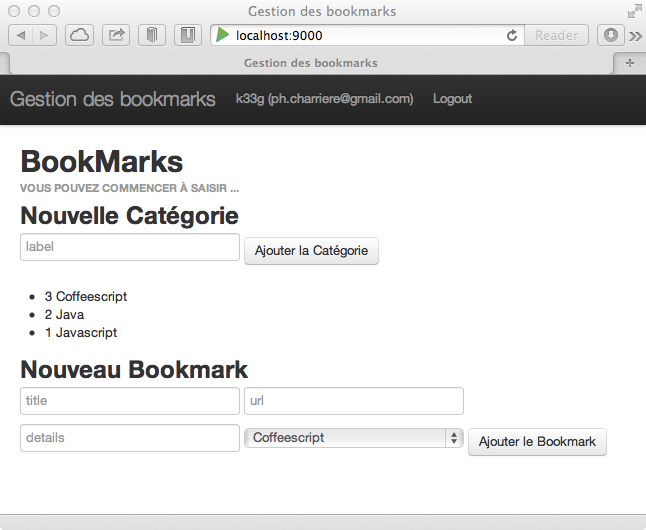
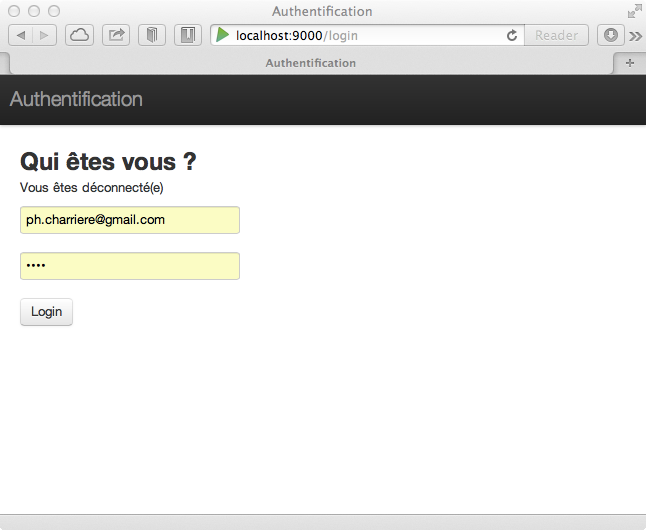
# Users  
  
users:  
  
 - !!models.User  
 email: ph.charriere@gmail.com  
 name: k33g  
 password: play  
  
 - !!models.User  
 email: bob@morane.com  
 name: bob  
 password: indochine

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

### 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.

# Services

/\* --- Qu'allons nous voir ? ---  
  
 En cours de rédaction.  
\*/