

*Play! Tous niveaux*



# Play!Rules

test

*Java Web trop facile !*

3Monkeys

@loic\_d, @mklabs, @k33g\_org

\*Texte temporaire\*

# Vision de [@k33g\_org](@k33g\_org)

## Comment est née l'idée de ce livre ?

De mémoire, car les choses sont allées finalement assez vite et j'ai une mémoire de poisson rouge. Cela faisait un moment que @loic\_d(1) me parlait de Play!

Framework, j'ai mis un peu de temps avant de me décider. Et là la révélation !, j'ai tellement était emballé, imaginez sortir une application web Java en 2 coups de cuillère à pot, moi le "vieux" geek de 42 balais nourri au Turbo Pascal et à VB6, issu du monde Microsoft (.Net et tout ce qui va avec), (phrase trop longue désolé) que j'ai décidé d'écrire des tutos sur mon blog au fur et à mesure de mon apprentissage.

UUn week-end @mklabs(2), qui passait par là, a relu ma prose (on fait ça de temps en temps pour voir si nos articles sont lisibles), du coup s'est mis aussi à Play!

Framework, à la fin du même week-end, il était au taquet et nous lançait un défi (à @loic\_d(3) & à moi @k33g\_org(4)). \*"Debout les gars réveillez-vous, y va falloir en mettre un coup"\* (1) \*"On écrit un livre sur Play!"\*. Et voilà ... on y est.

1: @loic\_d

2: @mklabs

3: @loic\_d

4: @k33g\_org

## Pourquoi open-source ?

Dans le désordre :

Parce que nous ne sommes pas encore trop prétentieux (même si @mklabs(1) il se la joue sur la Home de HTML5 BoilerPlate et que @loic\_d(2) c'est quand même Monsieur co-Jug-Leader de l'Alpes JUUG), je ne vais quand même pas aller sur un

projet avec des bras cassés ;)

1: @mklabs

2: @loic\_d

Parce que mon style d'écriture ne passerait probablement pas chez un éditeur

Parce que si on veut aller chez O'Reilly, il va falloir écrire en anglais, et mes blagues passent beaucoup moins bien

Parce que je suis toujours content de trouver de bons tutos gratuits par des bons p'tits gars qui on envie de partager

## Qui suis-je ?

## Dédicace

`* TODO: si j'ai envie */`

(1) C'est pas de la référence ça !? J'ai toujours aimé les citations

## Vision de [@loic\_d](@loic\_d)

## Comment est née l'idée de ce livre ?

La première réunion que nous avons organisé avec l'Alpes JUUG (Java UUser Group grenoblois) était un coding dojo sur Play avec Guillaume Bort, l'auteur du framework. Ce jour là ce fût une vrai claque et cette présentation reste pour moi parmi les plus marquantes que j'ai pu voir. Guillaume nous a présenté non pas un simple framework mais une plateforme complète pour le développement et l'exécution d'applications Web. J'ai été extrêmement enthousiasmé par sa vision des architectures Web et de la sur-complexité vendue depuis des années par les éditeurs de middleware dans le monde Java...

Depuis je n'ai pas arrêté de coder avec Play et d'écrire des article sur ce sujet pour mon blog CoffeeBean(1).

1: <http://coffeebean.loicdescotte.com>

Quand @mklabs(1) nous a soumis l'idée d'écrire cet ebook, je ne pouvais pas refuser de travailler sur un sujet aussi excitant!

1: [@mklabs](#)

## Pourquoi open-source ?

Pour coller à l'esprit de Play. Et aussi car je pense qu'avec un maximum de relecteurs et de contributeurs le résultat ne pourra être que meilleur!

## Qui suis-je ?

UUn geek, un fan de Java et depuis peu de Scala, aussi un musicien (guitare et un peu de batterie) à mes heures perdues. J'ai toujours préféré le développement backend au frontend (vive les services REST), mais je commence à prendre goût au développement JavaScript, notamment grâce aux bons conseils de @mklabs sur jQuery!.

## Dédicace

Je ne vais pas être très original sur ce coup, comme @k33g\_org(1) je dédicace ce livre aux deux autres monkeys, sans qui ce projet ne pourrait se réaliser. Grâce à @k33g\_org(2), j'ai d'excellents souvenirs des deux années que j'ai passé à Lyon : j'ai compris grâce à lui qu'on pouvait facilement mêler boulot et grosse déconnade! Quand à @mklabs(3) même si nous avons déjà échangé pas loin d'une centaine de mails, nous n'avons pas encore eu le temps de nous rencontrer physiquement au moment où j'écris ces lignes. Mais ça ne devrait pas tarder (la semaine prochaine normalement)!

1: [@k33g\\_org](#)

2: @k33g\_org

3: @mklabs

# Introduction - Pourquoi Play

Play framework est une vraie révolution dans le monde des application Web écrites en Java. Il vous fera oublier toutes les souffrances que vous avez pu vivre avec la pile Java EE classique, des frameworks comme Spring.

Architectures techniques opaques, gestion chaotique des dépendances, longues phases de compilation, redémarrage du serveur à chaque modification du code... tout ça ne sera bientôt pour vous que de mauvais souvenirs :)

## Architecture simple

Play se base sur une architecture extrêmement simple en suivant le design pattern MVC. A côté de ça il ne rajoute pas de notions de couches service, couches DAO etc.

Tout le code métier est porté par les objets du modèle, afin d'éviter le phénomène appelé Anemic Domain Model(1), qui résulte en l'écriture de classes métier contenant uniquement des champs et des accesseurs (getters et setters), donc sans traitements ni intelligence. C'est ce qui arrive dès que l'on commence à implémenter le code métier de l'application dans des couches techniques (couches service, EJB...)

1: [http://en.wikipedia.org/wiki/Anemic\\_Domain\\_Model](http://en.wikipedia.org/wiki/Anemic_Domain_Model)

Comme en Ruby On Rails, les objets du modèle sont conçus selon le pattern Active Record : ils ont la capacité de gérer eux même leur persistance dans la base de données.

On peut par exemple écrire le code suivant pour manipuler une entité "Personne" :

## Orienté REST

```
Personne p1 = Personne.findById(1);  
p1.firstName = "paul";  
p1.save();
```

Par exemple, pour afficher toutes les personnes habitant à Paris dans un annuaire,

on pourra utiliser une UURL comme

`annuaire/personnes/paris`

## Stateless et scalable

Play se veut respectueux de l'architecture du Web et donc des architectures REST. Guillaume Bort a fait le choix de ne rien stocker côté serveur.

Cela signifie qu'il n'existe pas de session utilisateur sur la partie serveur du framework.

Ceci peut sembler déstabilisant lorsque l'on a l'habitude de travailler des frameworks comme JSF ou Wicket. Mais finalement ce mode de fonctionnement simplifie vraiment les choses.

En effet on n'a pas besoin de gérer l'état du serveur, il ne fait que traiter les requêtes qui arrivent et renvoyer la réponse. Ceux qui ont déjà eu des problèmes avec Wicket et sa manie de tout garder en session, même les objets "ou of date" comprendront ce que je veux dire.

Play propose un objet "session" qui permet de stocker un identifiant de session utilisateur en écrivant dans un cookie côté client (dans le navigateur).

Pour stocker des volumes plus importants de données côté client, vous serez incité à utiliser les API de stockage de HTML 5 (web storage).

Si pour des raisons de performances vous ne voulez pas répéter trop souvent les mêmes requêtes vers la base de données, il est également possible d'utiliser un cache distribué. Play fournit une implémentation de cache par défaut.

Ceci prend tout son sens dans les environnements de type cloud ou des noeuds de serveurs peuvent être ajoutés et retirés dynamiquement selon la demande.

Autre avantage : la tolérance aux pannes. Si un serveur tombe en panne, les appels pourront passer sur un autre serveur sans que l'utilisateur s'en rende compte.

Avec des framework stateful, vous seriez obligé de dupliquer les sessions utilisateurs d'un serveur à l'autre pour que les utilisateurs ne perdent pas leur contexte de travail.

## Productif

Toute la pile est pré-configurée, de la vue à la base de données. Play suit la logique de `convention over configuration`. Ainsi, si le paramétrage par défaut vous convient, vous pourrez commencer à développer dès que vous aurez dézippé l'archive du framework! Ce principe sera également appliqué lors du développement de nos applications Play afin d'économiser des lignes de code tout au long du développement.

Play embarque son propre serveur qui est capable de compiler lui même les fichiers source et de récupérer à chaud toutes les modifications de code.

Vous n'aurez donc jamais à vous soucier des phases de compilation ou de déploiement de votre application.

Si vous ajoutez une nouvelle ligne de code, un simple "refresh" dans votre navigateur vous permettra de la voir en action. Et si jamais votre code contient une erreur, vous verrez un message clair et explicite dans votre navigateur, bien plus simple à comprendre que les traditionnelles `stack trace` que l'on rencontre habituellement lorsque l'on fait du développement JEE.

Enfin, Play propose nativement un module `CRUUD` permettant de générer les écrans, les traitements et les requêtes pour gérer les opérations basiques relatives à une entité métier (création, lecture/recherche, mise à jour, suppression).

## Modulaire et extensible

Il existe un grand nombre de modules pour ajouter des fonctionnalités au framework : déploiement sous Google APP Engine, authentification avec OAuth, validation des données côté client avec HTML5...

La communauté est très active et de nouveaux plugins arrivent régulièrement dans le dépôt officiel.



De plus le framework, bien que `full stack`, n'est pas monolithique, il est possible de n'utiliser que les parties de Play qui nous intéressent et de l'utiliser conjointement à d'autres technologies. On pourrait par exemple imaginer n'utiliser que la partie contrôleur de Play pour exposer des services REST à un front end écrit en HTML/JavaScript et s'appuyer sur des services Spring pour la partie métier.

## Pur Java

Play est écrit en Java et il est compatible avec toutes vos bibliothèques Java préférées.

De plus Play facilite l'utilisation de Java grâce à un certain nombre d'astuces. Il génère par exemple automatiquement les accesseurs (getters et setters) dans les classes Java dans le but d'améliorer la lisibilité du code.

## 5 trucs cool que l'on peut faire avec Play

Les exemples suivants sont tirés du site officiel de Play<sup>(1)</sup> et montrent en quelques lignes l'esprit et la simplicité du framework.

1: <http://www.playframework.org/documentation/1.0/5things>

### 1. Mapper des paramètres HTTP et une méthode Java

L'URL suivante

```
articles/archive?date=08/01/08&page=2
public static void archive(Date date, Integer page) {
    List<Article> articles = Article.fromArchive(date, page);
    render(articles);
}
public class Person {
    public String name;
    public Integer age;
}
public static void add(Person p) {
    p.save();
}
```

### 2. Redirection vers une action, en appelant simplement une méthode Java

```

<form action="/Directory/add" method="POST">
  Name: <input type="text" name="p.name" />
  Age: <input type="text" name="p.age" />
</form>
public static void show(Long id) {
  Article article = Article.findById(id);
  render(article);
}
public static void edit(Long id, String title) {
  Article article = Article.findById(id);
  article.title = title;
  article.save();
  show(id);
}

```

Dans les templates, on peut utiliser une syntaxe équivalente pour générer un lien :

### 3. Ne vous répétez pas en passant des paramètres aux templates

```

<a href="@{Article.show(article.id)}">${article.title}</a>
That will generate the following HTML:
<a href="/articles/15">My new article</a>
Article article = Article.findById(id);
User user = User.getConnectioned();
Map<String, Object> model = new HashMap<String, Object>();
model.put("article", article);
model.put("user", user);
render(model);
Article article = Article.findById(id);
User user = User.getConnectioned();
render(article, user);

```

### 4. JPA sous steroids

Il est vraiment facile d'utiliser l'API de mapping objet/relationnel JPA avec Play. Rien à configurer, Play synchronisera la base (également configurée et démarrée automatiquement en mode développement) avec vos objets.

En plus, si vous utilisez la classe Model de Play, le code sera encore simplifié :

### 5. UUploaderez facilement des fichiers

```

public void messages(int page) {
  User connectedUser = User.find("byEmail", connected());
  List<Message> messages = Message.find(
    "user = ? and read = false order by date desc",
    connectedUser
  ).from(page * 10).fetch(10);
  render(connectedUser, messages);
}
Le formulaire HTML :
<form action="@{Article.uploadPhoto()}" method="POST" enctype="multipart/

```

```
form-data">
    <input type="text" name="title" />
    <input type="file" id="photo" name="photo" />
</form>
Et le code Java :
public static void uploadPhoto(String title, File photo) {
}
```

# Annexes

## L'internationalisation

Dans le cas d'une application ou d'un site multilingue, on doit être capable de traduire facilement le contenu de nos pages.

Pour le message d'accueil de notre application, on peut par exemple écrire :

```
<h1>&{welcome}</h1>
```

Les paramètres entourés de `&{}` seront traduits à partir des clés définies dans les fichiers de configuration de Play.

Les clés pour la langue par défaut se trouvent dans le fichier `/conf/messages` :

```
welcome=Welcome on Vote4Music!
```

On peut ensuite définir un fichier par langue supplémentaire, par exemple `messages\_fr` pour le français.

Ce mécanisme peut être utilisé pour traduire toutes sorte de clés. On peut par exemple afficher les valeurs de l'énum Genre dans notre application en modifiant la casse :

```
<h1>Top albums in &{genre} for ${year}</h1>
```

On renseigne ces clés dans le fichier `messages` :

```
ROCK=Rock
```

```
METAL=Metal
```

HIP\_HOP=Hip Hop

WORLD=World

POP=pop

JAZZ=Jazz

BLUUES=Blues

OTHER=Other

TODO more tips to come

# On se met en jambes

Tout ce qui va se dire ici doit être oublié une fois que vous passerez au chapitre 1. Le chapitre "zéro" n'est là que pour désacraliser la bête. Vous ne vous sentez pas à l'aise avec le développement web en java, le pattern MVC vous donne des boutons (faut reconnaître qu'une indigestion de Struts c'est moyen), écrire des requêtes SQL vous angoisse, vous êtes débutant etc. ... Mais vous voulez vous y coller tout de suite (l'est-y pas tout plein de motivation !). Et bien, ce chapitre est pour vous (vous allez très vite comprendre pourquoi)

## Play! c'est quoi ?

On l'a déjà dit dans l'intro ! Mais je vous donne ma définition : *"c'est le moyen de se la jouer rapidement en java-web, alors que l'on n'est pas forcément le meilleur de l'équipe"*

Lorsque vous aurez lu les chapitres du niveau supérieur et passé tous les boss de fin de niveau, vous verrez aussi que c'est un excellent framework qui vous permet de pondre des applications web de qualité, robustes, maintenables, "scalables" ... (Oh p... j'ai l'impression de répondre à un appel d'offre).

Mais ça c'est pour mes petits camarades @loic\_d(1) et @mklabs(2). Du coup vous aurez compris que c'est moi qui ne suis pas le meilleur de l'équipe ;)

1: [@loic\\_d](#)

2: [@mklabs](#)

## Bon on s'y colle ?

Nous allons voir comment :

installer Play!

définir le cahier des charges de notre 1ère application

générer le squelette de l'application

paramétrer votre IDE préféré (en fait je vais vous dire quel IDE utiliser)

## Pré-requis

Vous devez avoir installé Java sur votre joujou préféré.

\*Si vous ne savez pas faire, créez une "issue" sur le repository git du bouquin : [https://github.com/3monkeys/play.rules/issues\(1\)](https://github.com/3monkeys/play.rules/issues(1)), ça veut dire que c'est utile à rajouter dans le bouquin.\*

1: <https://github.com/3monkeys/play.rules/issues>

## Installation

Télécharger Play Framework : [http://www.playframework.org/download\(1\)](http://www.playframework.org/download(1))

1: <http://www.playframework.org/download>

dézipper quelque part

ajouter à votre path : c'est mieux

> sous windows ça devrait donner ceci : (dans les variables utilisateur) si vous avez dézippé dans C:\play

```
crØer PLAY_HOME = C:\play
ajouter %PLAY_HOME% au path : PATH = C:\bla bla bla;%PLAY_HOME%
sudo pico ~/.bash_profile
PLAY_HOME=/Users/ton_user_name/play; export PLAY_HOME
export PATH=$PATH:$PLAY_HOME
```

\* TODO \*/

en mode commande : tapez `play` pour voir. Si tout va bien, vous aurez ceci :

```
~
î K'_ \KK'_ ' KKK_K
```

```
^ K __/K_K____K__ (_)
```

```
~ |_| |__/  
^ play! 1.2.1, http://www.playframework.org
```

```
^
```

```
^ UUsage: play cmd [app_path] [--options]
```

```
^
```

```
~ with, new      Create a new application  
~      run      Run the application in the current shell  
~      help      Show play help
```

\*\*\*Remarque :\*\*\* \*En ce qui me concerne, je fais toutes les manipulations sous OSB, mais globalement la logique est la même sous Linux ou Windows. Si vous avez un soucis, créez une "issue" sur le repository git du bouquin : [https://github.com/3monkeys/play.rules/issues\(1\)](https://github.com/3monkeys/play.rules/issues(1)).\*

1: <https://github.com/3monkeys/play.rules/issues>

## L'Appel d'Offres

\*Ou comment je vais être hors sujet pendant quelques minutes ... Imaginons ...\*

Vous êtes Philou, chef de projet technique dans une ch'tite SSII parisienne, plutôt orienté (vous) technologies .Net. Pour des raisons personnelles (Dulcinée, marre de Panam, du RER, ...) vous décidez de retourner à la campagne (euh en province pardon) et intégrez une agence régionale d'une grande SSII nationale.

### 1er jour : confrontation avec votre chef d'agence

\*\*Lui : \*\* \*Tu sais, Philou, chez nous on est tous ingés, car on doit savoir tout faire, et on est tour à tour Chef de projet, architecte, développeur, ...\*

\*\*Vous : \*\* \*Ah c'est pour ça que sur mon contrat de travail il y a marqué ingénieur



d'études ?\*

\*\*Lui : \*\* \*Exactement ! Je suis content que tu adhères à notre façon de penser, c'est très corporate, notre collaboration va être particulièrement enrichissante.\*

\*\*Vous : \*\* \*...\*

\*\*Lui : \*\* \*Par contre en ce moment nous n'avons pas de missions .Net, tu sais ce qui est porteur chez nos clients, c'est Java. Et à ce titre, je souhaiterais que tu sois formé à Java. Tu n'y vois pas d'inconvénient ?\*

\*\*Vous : \*\* \*Bien au contraire !\* `[Motivation mal dissimulée]` \*Et chez vous, les formations sont faites en interne ou par le biais d'un organisme ?\*

\*\*Lui : \*\* \*Philou, par expérience, la formation la plus efficace, c'est l'autoformation, c'est de cette manière que tu retiendras le mieux les choses.\*

\*\*Vous : \*\* \*...\* `[je suis un lapin de 6 semaines et j'aime ça]`

\*\*Lui : \*\* \*Allez pas de chichi entre nous, je vais être sympa, tu peux faire ton intercontrat chez toi, tu seras plus tranquille, tu as un PC bien sûr ? Au fait tu pourrais poser des jours de congés par anticipation ? Genre je te permet de faire 5 jours en interco à la maison et toi tu fais le 2ème pas, tu poses 5 jours supplémentaires ?\*

\*\*Vous : \*\* \*Non. Je vous demande même pas si vous avez des bouquins sur Java ?\*

\*\*Lui : \*\* \*Non, ne demande pas. Allez, je t'appelle dès qu'une mission en adéquation avec ton profil se présente.\*

Et les semaines, mois, ... passent à apprendre Java à la terrasse du café en face de chez vous, le tout entrecoupé de quelques missions à forte valeur ajoutée autour de technologies d'avenir telles MS Access, Delphi, ...

\*Toute ressemblance avec des personnages existants serait totalement fortuite.\*

Pétage de plombs

Bon, apprendre Java tout seul, bof ... C'est le printemps, vous êtes dans une région magnifique où le réseau halieutique (les rivières, les étangs, ...) est très riche. Grande décision : vous avez décidé de vous mettre à la pêche "sportive" et fait \$ Java ! C'est parti pour les grandes balades au bord de l'eau.

Après quelques jours sur le même spot, vous faites la connaissance de Julo qui fait partie de l'amicale des pêcheurs de l'Azergues. Quelques canettes de bières plus tard, vous tentez d'expliquer à Julo ce qu'est un ingénieur informaticien. C'est à ce moment là qu'il sort "La Poire Maison" de sa besace car ça y'est vous êtes "potes de pêche".

1 heure après vous êtes LA PERSONNE qui va faire le site web de gestion des prises de pêches des concours de l'amicale des pêcheurs de l'Azergues et \*en PLA6 ! Framework\* bien sûr.

Rendez-vous, donc, dimanche au local de l'amicale pour établir les spécifications, capturer les besoins, définir les exigences utilisateurs (eh oui, moi la poire ça me rend loquace)

## Le Cahier des Charges

Plusieurs fois par an, l'amicale des pêcheurs de l'Azergues organise un concours de pêche. L'amicale souhaiterait pouvoir gérer les inscrits et leurs prises lors de chacun des concours.

C'est court (eh oui le client n'est pas mature dans l'expression de ses besoins), mais ça sera suffisant pour la création de notre application.

Maintenant, j'arrête de délirer et nous repassons à la technique.

## Créer le squelette de l'application

Nous avons donc installé Play, nous allons commencer à bosser :

Créer un workspace (le répertoire qui hébergera tes applications), et "aller dedans"

> sous OSB :

```
mkdir play_projects
cd play_projects
md c:\\play_projects
cd c:\\play_projects
```

> \* En mode commande, taper `play new azerguespeche` où "azerguespeche" est le nom de notre application

si tout va bien, ceci devrait s'afficher :

```
~
â K'_\KK'_ KKK_K
```

```
â K __/K_K____K__ (_)
```

```
~ |_| |__/
â play! 1.2.1, http://www.playframework.org
```

```
â
```

```
â The new application will be created in /UUsers/k33g_org/Dropbox/play_projects/
azerguespeche
```

```
â What is the application name? [azerguespeche]
```

> \* Valider. Si tout va bien, ceci devrait s'afficher :

```
â
```

```
â OK, the application is created.
```

```
â Start it with : play run azerguespeche
```

```
â Have fun!
```

```
â
```

Lancer l'application : `play run azerguespeche`, vous allez obtenir dans la console :

~  
^ K'\_ \KK' \_' KKK\_K

^ K \_\_/K\_K\_\_\_\_K\_\_ (\_)

~ | \_ | | \_ /  
^ play! 1.2.1, <http://www.playframework.org>

^

^ Ctrl+C to stop

^

Listening for transport dt\_socket at address: ( 000

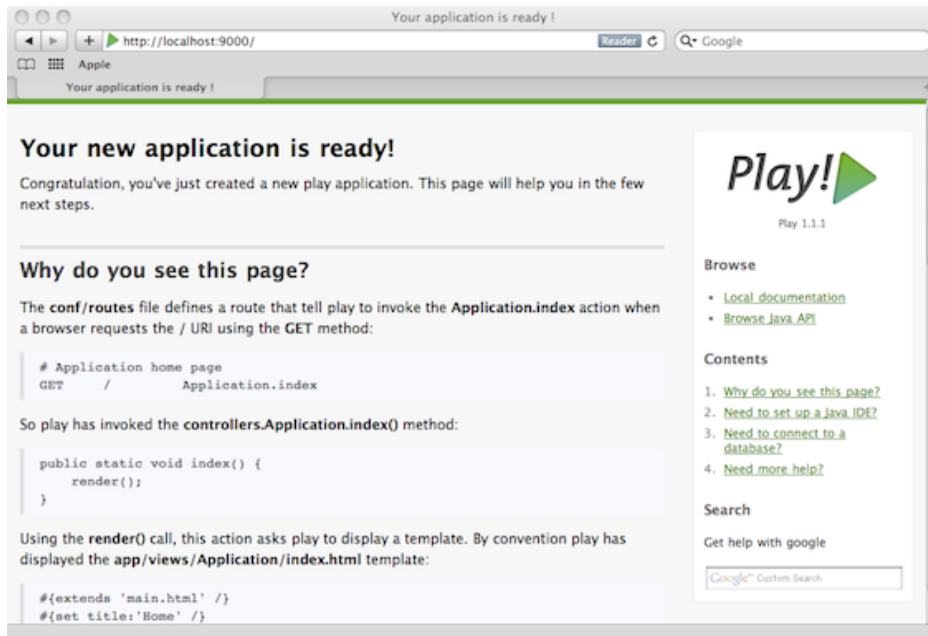
05:44:3( ,610 INFO ^ Starting /UUsers/k33g\_org/Dropbox/play\_projects/  
azerguespeche

05:44:40,21à WARN ^ 6 ou're running Play! in DEV mode

05:44:40,35â INFO ^ Listening for HTTP on port à000 (Waiting a first request to  
start) ...

lancer le navigateur : [http://localhost:à000/\(1\)](http://localhost:à000/(1))

1: <http://localhost:à000/>



\*\*Wouaaoo ! Vous êtes trop forts, on va bientôt pouvoir commencer.\*\*

## Paramétrage de l'IDE

En ce qui concerne l'IDE, vous pouvez très bien utiliser un simple éditeur de code avec colorisation syntaxique (Vous avez par exemple KomodoEdit qui fonctionne sur toutes les plateformes qui est assez sympa ... et open-source), mais c'est vraiment si vous voulez vous la jouer en démo ou que vous connaissez Java par coeur ou que vous codez sur un eeeepc â01.

Play! propose les commandes nécessaires pour transformer votre projet en projet Eclipse, NetBeans ou IntelliJ. Je suis particulièrement accro à NetBeans, mais pour avoir utilisé la version Community d'IntelliJ (Win, Tux, OSB), je vous conseille fortement de choisir cet IDE, étant donné que la version open source suffit largement pour faire du Play!, pourquoi se priver ?

Voyons donc comment faire pour transformer notre squelette d'application en projet IntelliJ (bien sûr vous avez téléchargé IntelliJ : [http://www.jetbrains.com/idea/\(1\)](http://www.jetbrains.com/idea/(1))) :

1: <http://www.jetbrains.com/idea/>

Tout d'abord, nous devons arrêter notre application : faire `Control+c` dans la console pour quitter

En mode commande : `play idealize azerguespeche` (pour NetBeans ça serait `play netbeansify links`)

~  
^ K' \_ \KK \_ ' KKK\_K \_

^ K \_ /K\_K \_ K \_ ( \_)

~ | \_ | | \_ /  
^ play! 1.2.1, <http://www.playframework.org>

^

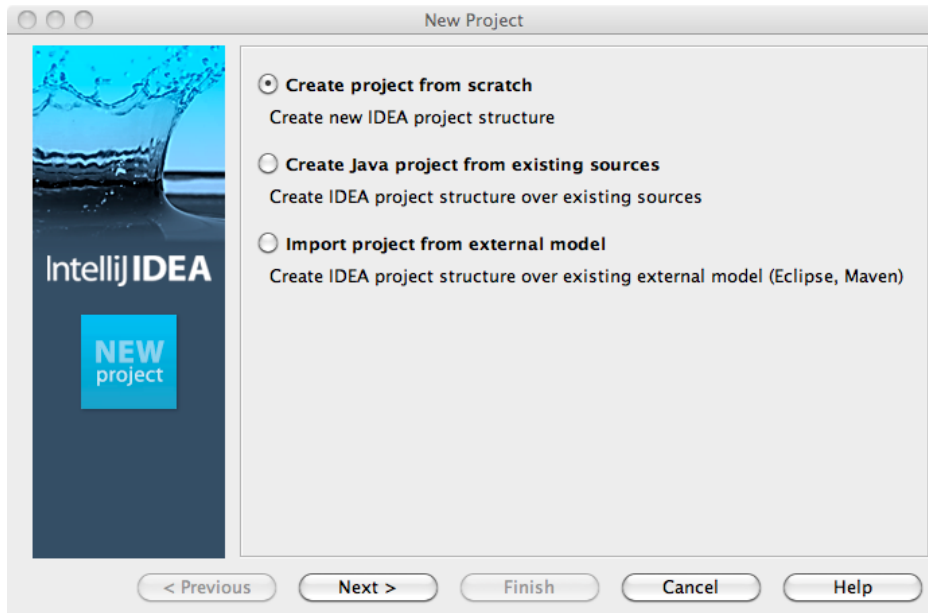
^ OK, the application is ready for IntelliJ Idea

^ UUse File/New Module/Import Existing module

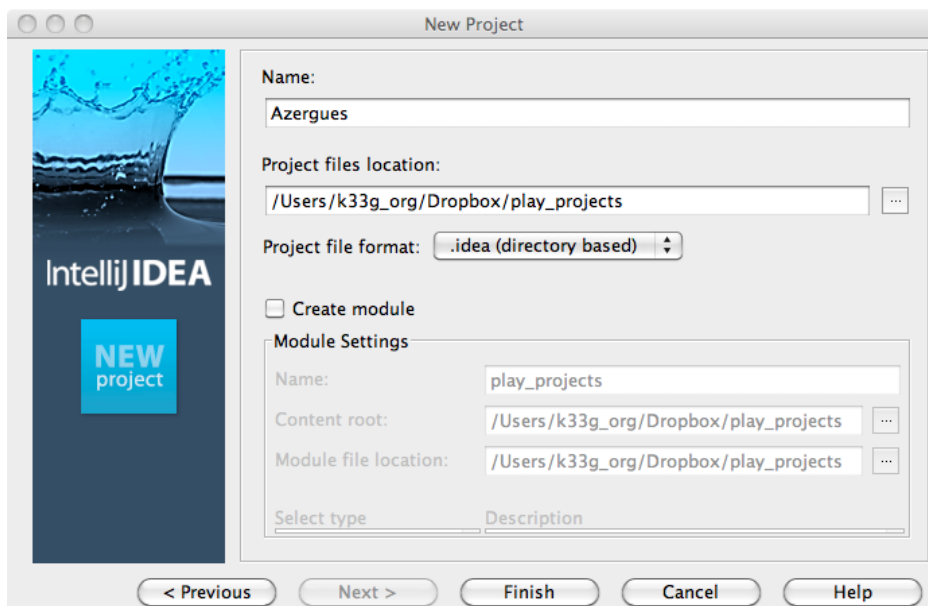
^

**\*\*Puis on lance IntelliJ et on fait les manipulations suivantes : \*\***

Créer un nouveau projet :



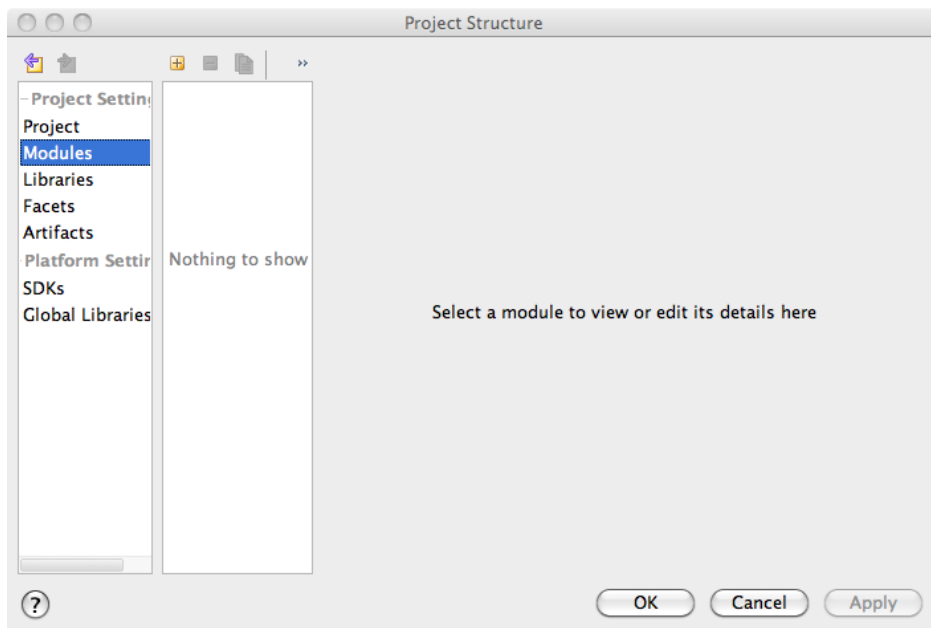
Saisir le nom du projet `Azergues` et l'endroit où vous souhaitez le sauvegarder, puis cliquez sur `Finish` :



Ensuite une fenêtre s'affiche, vous permettant d'ajouter un module

Sélectionnez `Modules`

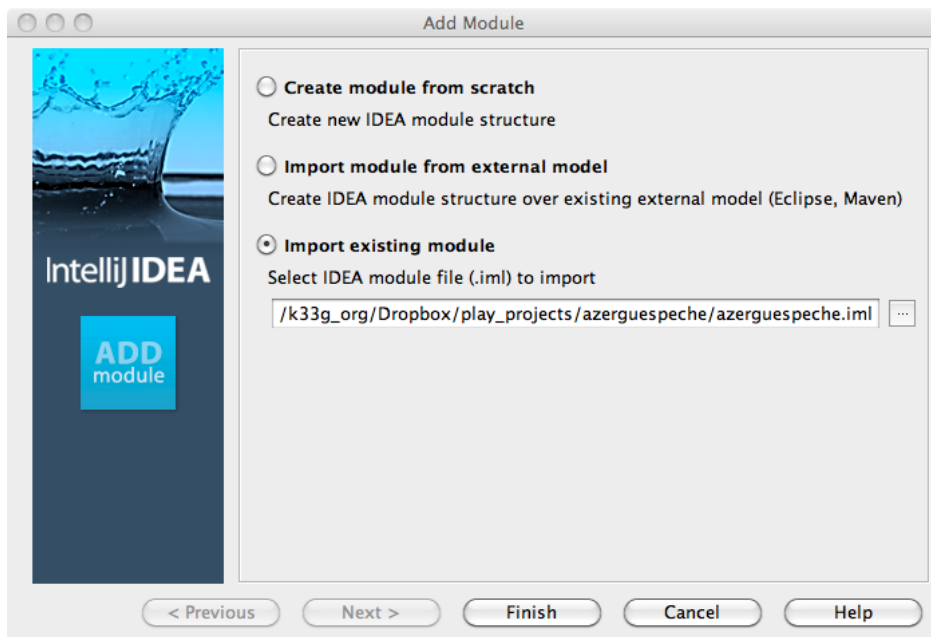
Cliquez sur le `a` en haut (un peu) à gauche



Sélectionnez le choix `Import existing module`

Précisez le chemin du fichier module `azerguespeche.iml` (qui a été généré par Play! lors de la commande `play idealize azerguespeche`)

Puis cliquez sur `Finish`

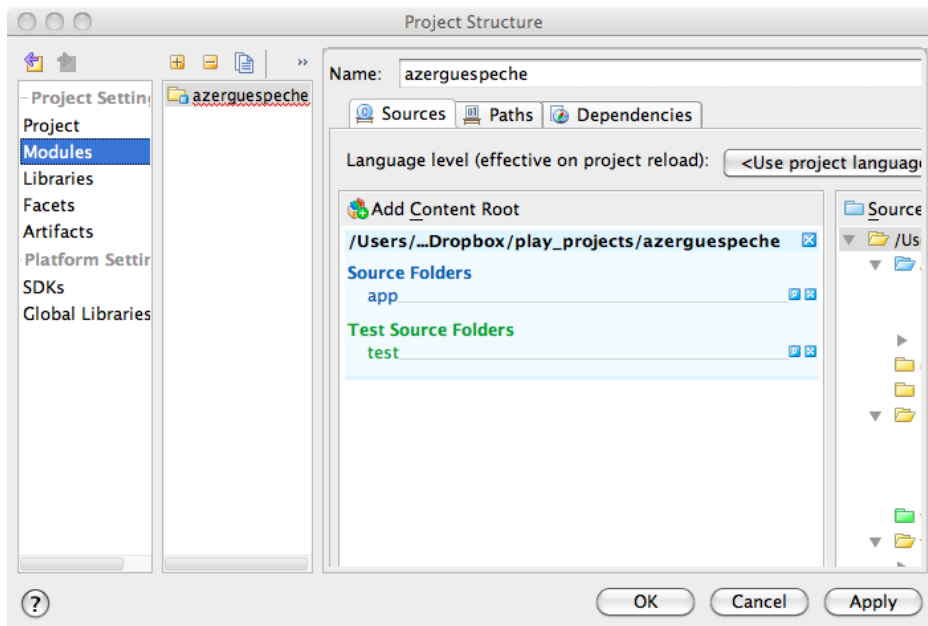


La fenêtre se réactualise avec un onglet `Sources`

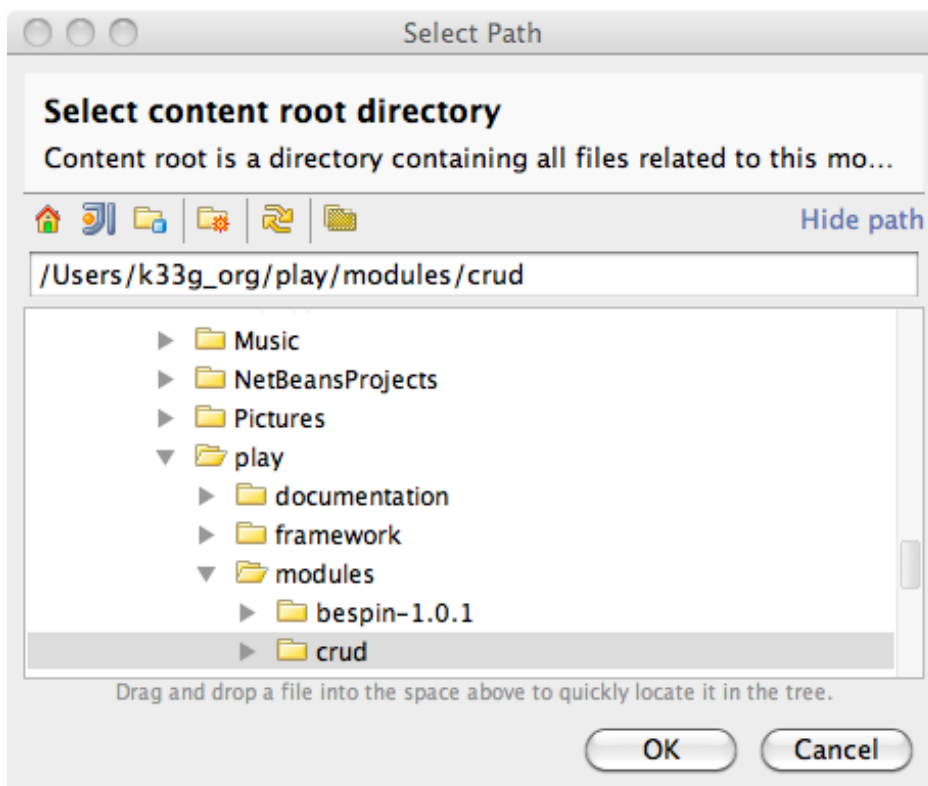
Dans l'onglet, cliquez sur `Add Content Root`. Nous allons ajouter les dépendances



à certains modules embarqués dans Play!.

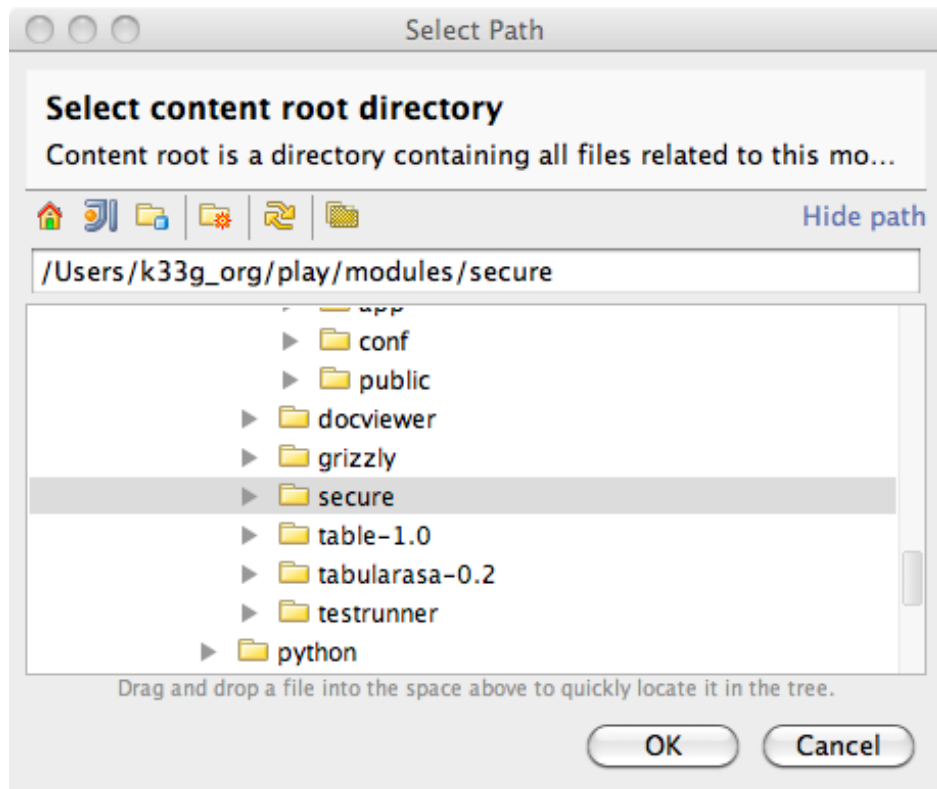


Précisez le répertoire du module `CRUUD` (présent dans le répertoire d'installation de Play!). Ce module nous permettra de générer automatiquement des écrans de saisie et de visualisation.



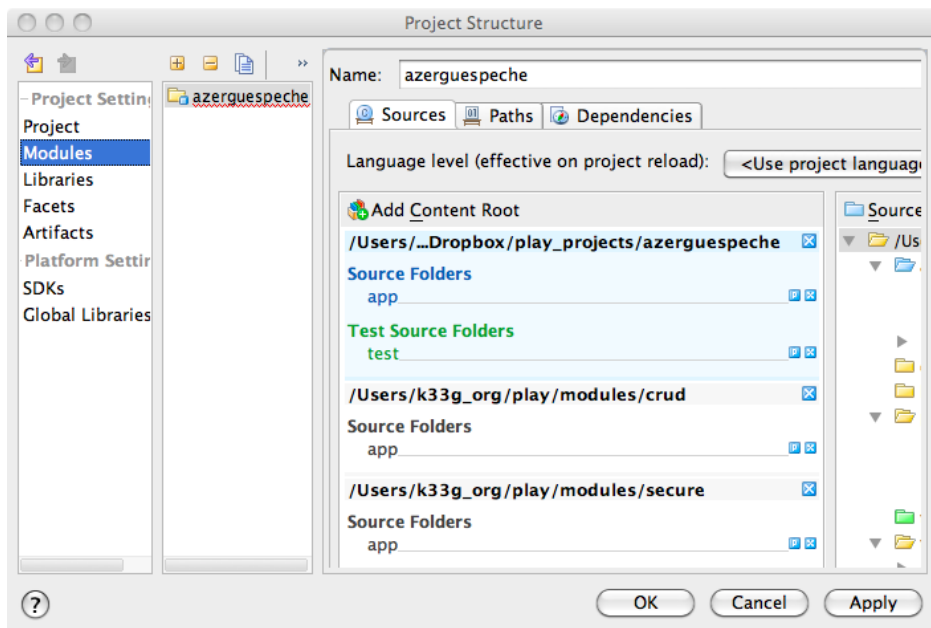
Précisez le répertoire du module `Secure` (présent dans le répertoire d'installation de

Play!). Ce module nous permettra de gérer facilement l'authentification.

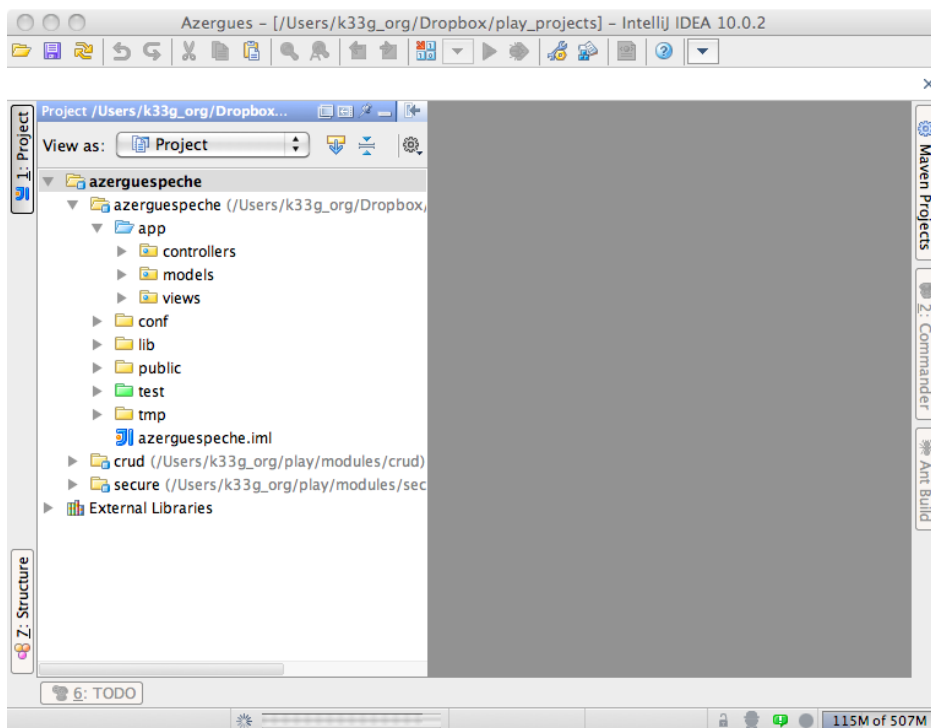


Votre projet est paramétré

Cliquez sur OK (ça va mouliner quelques secondes)



Et hop! Vous arrivez dans l'arborescence de votre projet :



\*\*Voilà, nous sommes prêts à démarrer, nous pouvons passer à l'étape suivante.\*\*

\*\*Rendez-vous donc au chapitre suivant : ch02-Premiere-application(1).

1: [ch02-Premiere-application.md](#)



❌ Faut bosser maintenant !!!

=

Dans ce chapitre nous allons :

créer notre 1ère page selon les préceptes MVC (kézaco ?)

construire les bases de notre application

ces bases seront suffisantes pour avoir quelque chose qui tourne à montrer à nos copains les pêcheurs.

## Introduction

\*Remarque : si ça vous ennuie, il est tout à fait possible de passer au § suivant\*

Alors, avant de remettre les mains dans le code, nous allons parler théorie. Et plus particulièrement "design pattern" (\*definition ici : [http://fr.wikipedia.org/wiki/Design\\_pattern](http://fr.wikipedia.org/wiki/Design_pattern)(1) (1)\*) et encore plus particulièrement : \*\*pattern MVC\*\*, sûrement le pattern que j'ai le plus détesté, que j'ai trouvé être le pattern le plus crétin et le plus improductif de l'histoire des patterns.

1: [http://fr.wikipedia.org/wiki/Design\\_pattern](http://fr.wikipedia.org/wiki/Design_pattern)

Je m'explique, j'étais architecte .Net (v1 1.1), fan invétéré du modèle évènementiel d'ASP.Net (2) et en changeant de boîte (celle dont je parlais dans le chapitre précédent), j'ai hérité du pilotage d'une TMA Java avec des applications Web. Mon nouveau boss m'a dit : \*"Tu n'es pas là pour coder, tu es là pour faire du management et de la relation client, donc pas besoin de connaître Java"\*, \*\*Moi (dans ma tête) :\*\* \*"wahouuu, trop sympa, ... mais en même temps, l'équipe va me prendre pour un c..."\*.

Ca n'a pas loupé, dès le 1er jour :

à ma demande \*"Combien de temps pour modifier la mire de login là, pour changer la police et donner le focus sur la zone de saisie du user ?"\*,

**\*\*mon ingé préféré :** **\*\*** "tu sais java, c'est une techno très aboutie, mais qui n'est pas simple, mais si il n'y a pas de problème, d'ici la fin de la semaine cela devrait être bon ... tu sais STRUUTS c'est pas pour les gamins" **\*** `(ndla : on est lundi)`

**\*\*Moi (dans ma tête) :** **\*\*** "comme un doute ... C'est sympa de passer pour c... auprès du client dès le premier jour ..." **\***

Je vous épargne la suite. Les anciens (la meilleure équipe que j'ai pu avoir (3)) de ce projet reconnaîtront sûrement la personne à laquelle je fais allusion.

bon j'en étais où ?

Ah oui, c'est alors que je décide de me mettre sérieusement à Java, et surtout de prendre les choses à l'envers et de brillamment m'auto-former à **\*\*STRUUTS\*\*** (achat de bouquins, etc. ...) d'urgence, car 5 jours pour une f@&t ing login box, ça me fait mal.

## Petit rappel : MVC

**\*\*STRUUTS\*\*** c'est (c'était?) le framework web Java par excellence qui met en oeuvre le pattern **\*\*MVC\*\*** : **\*\*Modèle Vue Contrôleur\*\*** (Model View Controller). Il permet de séparer les responsabilités en 3 couches (m... j'en avais 5 moi ...) :

**\*la vue\*** : c'est l'IHM, elle va recevoir des infos du **\*\*contrôleur\*\*** (**\***"tiens affiche moi ça"**\***), elle va envoyer des infos au contrôleur (**\***"au fait, on m'a cliquée dessus, le gars il voudrait la liste des clients"**\***)

**\*le contrôleur\*** : c'est lui donc qui reçoit des infos de la **\*\*vue\*\***, qui va aller récupérer des données métiers chez le **\*\*modèle\*\*** (**\***"j'ai besoin pour la vue de la liste des clients"**\***), et va les renvoyer à la vue et éventuellement appliquer des traitements à ces données avant de les renvoyer.

**\*le modèle\*** : c'est vos objets clients, fournisseurs, users, ... avec toute la mécanique qui sert à les sauvegarder, retrouver, modifier, supprimer ...

## Suite de l'introduction (où j'ai donc décidé d'apprendre

## STRUUTS)

Et là c'est le drame ! Après avoir passé le cap de l'installation de mon serveur d'application (ça a bien changé maintenant, mais quand en plus on est habitué à IIS ...), après avoir installé et paramétré tous les bons plugins Eclipse, ce fut le 1er contact avec **STRUUTS**.

Alors je ne suis pas complètement idiot, j'ai codé mes 1ers écrans et ma mire de login (en passant il ne faut pas 5 jours), mais alors qu'est ce que j'ai trouvé ça fastidieux et inutile ! Je ne sais combien de fichier pour un seul écran! **STRUUTS** pas pour moi! (et là je suis tombé amoureux des **JSF**, si !).

Et bien vous allez voir (pour ceux qui ne connaissent pas) que :

## Avec Play!, MVC c'est limpide !

Dans le chapitre précédent, nous avons initialisé notre application **AzerguesPêche**, il est temps maintenant de coder notre première page en mode MVC, comme ça après on n'en parle plus de MVC. Ami lecteur-codeur-newbee, prépare toi à entrer dans le monde de **Java a WEB** sans mal ni effort avec Play! Framework (si c'est pas un slogan de bouquin ça!).

Tout d'abord relancez votre application si elle ne tourne plus (play run azerguespeche`)

Ouvrez votre projet dans votre ide préféré (cf. chapitre précédent : 01-Preparation.fr.md(1))

[1: 01-Preparation.fr.md](#)

Observons la structure de notre future "killer-app"

## Structure d'azerguespeche et 1ère page

Dans IntelliJ vous pouvez voir dans l'arborescence de votre projet que vous avez plusieurs répertoires :

azerguespeche

app

controllers

models

views

## 1er modèle

```
- Application  
{{{ java
```

```
package models;
```

```
public class Version {
```

```
    public String reference;  
    public String name;  
    public Version() {  
        this.reference="v° zero";  
        this.name="proto pour les copains";  
    }  
{{{
```

**\*\*Remarque :\*\*** vous pouvez voir qu'il n'y a pas de getter ni de setter, juste des "champs" publics. Dans la majeure partie des cas vous n'en n'avez pas besoin, alors à quoi bon ? Sachez cependant que Play! va les générer lui même à la compilation (vous ne verrez rien dans le code). Au besoin vous pouvez les écrire vous même si nécessaire.

## Et le contrôleur ?

Cette fois-ci nous allons modifier le contrôleur `Application` dans le répertoire `controllers` :

**\*Si vous n'avez touché à rien vous devez avoir le code suivant :\***



^^^ java

```
package controllers;
```

```
import play.*;
```

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

```
import java.util.*;
```

```
import models.*;
```

```
public class Application extends Controller {
```

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

^^^

\*Modifions la méthode `index()` :\*

^^^ java

```
public class Application extends Controller {
```

```
    public static void index() {  
        Version version = new Version();  
        render(version);  
    }
```

^^^

## Allons modifier la vue Application

Ouvrez la page `index.html` du répertoire `views/Application/`

Vous devez avoir le code suivant :

```
e {extends 'main.html' /}
```

```
e {set title:'Home' /}
```

```
e {welcome /}
```

Que vous allez remplacer par :

```
e {extends 'main.html' /}
```

```
e {set title:'Azergues Pêche' /}
```

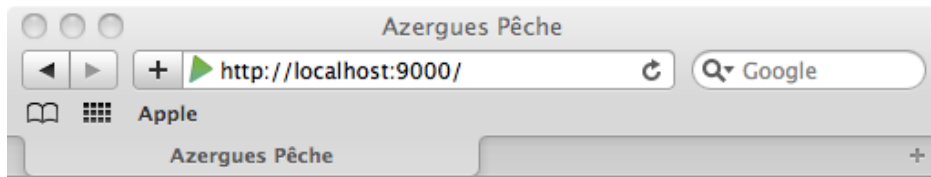
```
<B>Version : ${version.reference} ${version.name}</B>
```

**Remarque :** `e {extends 'main.html' /}` signifie que `index.html` hérite de `main.html` (si vous allez regarder dans `main.html`, vous verrez que c'est là que sont déclarées les ressources css, js, etc. ...)

**Remarque bis :** \*Que vient-on de faire ?\*, eh bien, lorsque nous allons nous connecter, le contrôleur `Application` va instancier le modèle `Version` qui sera affiché dans la vue `Application` (`index.html`) grâce à la méthode `render(version)` utilisée dans le contrôleur.

Si c'est vrai ! Vous n'avez qu'à appeler l'url `http://localhost:8000/(1)` dans votre navigateur :

1: <http://localhost:8000/>



**Version : v° zero proto pour les copains**

\*Et là je sens que une pointe de fierté et de satisfaction monter en vous ... non ???\*

Donc, comme vous venez de le voir, faire discuter un contrôleur avec un modèle et une vue, ça n'a rien de bien compliqué, surtout avec Play!. Là sans vous en apercevoir, vous avez compris MVC.

**\*\*Question :\***

\*Euh ... et Play!, comment il sait qu'il faut ouvrir la page index.html ?\*

\*c'est une excellente question !\*

Allez ouvrir le fichier `routes` dans répertoire `conf` de votre projet, il contient le code suivant :

e Routes

e This file defines all application routes (Higher priority routes first)

e î î î î

e Home page

GET / Application.index

e Map static resources from the /app/public folder to the /public path

```
GET /public/      staticDir:public
*                /{controller}/{action}      {controller}.{action}
```

e Home page

GET / Application.index

explique que quand on appelle la racine du site dans l'url (le "/" tout seul), alors on ouvre la page index.html du répertoire Application (c'est une **convention**)

## **Vous êtes prêts ! ... pour aller plus loin : des objets, des écrans ...**

Où nous allons créer les bases d'**Azergues Pêche** grâce au module CRUUD de Play

## **Préparation**

c'est dans le chapitre suivant : ch03-Premiere-application-CRUUD(1).

1: [ch03-Premiere-application-CRUUD.md](#)

(1) : oui, je sais, ce n'est pas la meilleure source d'information qui soit, ni la plus fiable, mais bon ... n'hésitez pas à poster vos définitions dans les "issues" du projet : [https://github.com/3monkeys/play.rules/issues\(1\)](https://github.com/3monkeys/play.rules/issues(1)).

1: <https://github.com/3monkeys/play.rules/issues>

(2) : je le suis encore un peu, je dois l'avouer, mais on ne se refait pas.

(3) : sans aucune ironie

Le SOCLE de notre application grâce au module CRUUD

=

Dans ce chapitre nous allons :

paramétrer notre projet pour utiliser la base de données embarquée : HSQLDB

déclarer le module CRUUD

créer des classes "Models"

utiliser le module CRUUD pour générer notre IHM de saisie

apporter quelques modifications pour améliorer l'affichage et valider les saisies

Et tout ceci facilement et rapidement

## Préparation de notre environnement

### Nous allons avoir besoin d'une base de données

\* chercher dans `/conf/application.conf` ceci :

```
# Database configuration
# ~~~~~
# Enable a database engine if needed.
#
# To quickly set up a development database, use either:
#   - mem : for a transient in memory database (HSQL in memory)
#   - fs  : for a simple file written database (HSQL file stored)
# db=mem
```

### Nous allons activer le module CRUUD

\*C'est quoi CRUUD ?\*, alors déjà, ça veut dire **C**reate **R**ead **UU**pdate **D**elete, le module CRUUD de Play! va permettre de générer automatiquement pour vous les écrans de saisie des données à partir de votre modèle objet avec toute la mécanique qui va bien pour sauvegarder vos modifications. Je n'en dis pas plus, les exemples qui vont suivre parleront d'eux-même.

Donc pour activer le module CRUUD :

\* aller dans `/azerguespeche/conf/application.conf`

\* ajouter ceci :

```
# Import CRUD module
module.crud=${play.path}/modules/crud
```

\* ajouter ceci :

```
# Import CRUD routes
* /admin module:crud
GET /admin module:crud`
```

\*\*Avant de continuer, arrêtez puis relancez votre application :\*\*

donc avec la commande : `play run azerguespeche` en mode console.

Vous devriez voir apparaître un message du type : `0â:4( :26,643 INFO Module crud is available (/UUsers/k33g\_org/play/modules/crud)`

## Il est temps de créer notre modèle objet (model)

Pour se mettre en jambes nous allons créer des pêcheurs et des poissons. les sources des classes vont ici `/azerguespeche/app/models/`, (donc dans IntelliJ, vous faites un click-droit sur le répertoire models a new a Java Class)

### La classe Pecheur

aura le code suivant :

```
		/// java
```

```
package models;
```

```
import javax.persistence.*;
```

```
import play.db.jpa.*;
```

@Entity

```
public class Pecheur extends Model{
```

```
    public String identifiant;  
    public String nom;  
    public String prenom;  
    public Pecheur(){  
    }  
}
```

\*ce constructeur n'est pas obligatoire, c'est pour plus tard\*/

```
    public Pecheur(String identifiant, String nom, String prenom) {  
        this.identifiant = identifiant;  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
}
```

î î î

\*Note : tous les modèles sont précédés par l'annotation `@Entity`, cela permet à Play! de savoir que c'est un modèle qui sera "persistable" en base de données.\*

\*Note bis : tous les modèles héritent de la classe `Model`, ce qui leur affecte différents "comportements", comme la méthode `save()`.\*

\*Note (encore) : vous pouvez remarquer que l'on n'utilise pas de getter ni de setter, mais directement des champs publics, Play! s'occupera de les générer à la compilation.\*

## La classe Poisson

î î î java

```
package models;
```

```
import javax.persistence.*;
```

```
import play.db.jpa.*;
```

@Entity



```
public class Poisson extends Model{
```

```
    public String identifiant;  
    public String nom;  
    public Poisson() {  
    }  
}
```

\*ce constructeur n'est pas obligatoire, c'est pour plus tard\*/

```
    public Poisson(String identifiant, String nom) {  
        this.identifiant = identifiant;  
        this.nom = nom;  
    }  
}
```

^^^

ça c'est fait.

## Maintenant, au tour des contrôleurs

dans `/azerguespeche/app/controllers/` créer les 2 classes "controllers"  
correspondant à nos 2 classes "models" précédentes

> pour info le contrôleur de Pecheur.java s'appellera Pecheur\*\*s\*\*.java (c'est la  
norme, on parle aussi de \*\*conventions\*\*)

Vous allez voir, c'est tout simple avec le mode CRUUD :

### La classe Pecheur\*\*s\*\*

^^^ java

```
package controllers;
```

```
import play.*;
```

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

```
public class Pecheurs extends CRUUD {
```

```
}
```

^^^

## La classe Poisson\*\*s\*\*

^^^ java

```
package controllers;
```

```
import play.*;
```

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

```
public class Poissons extends CRUD {
```

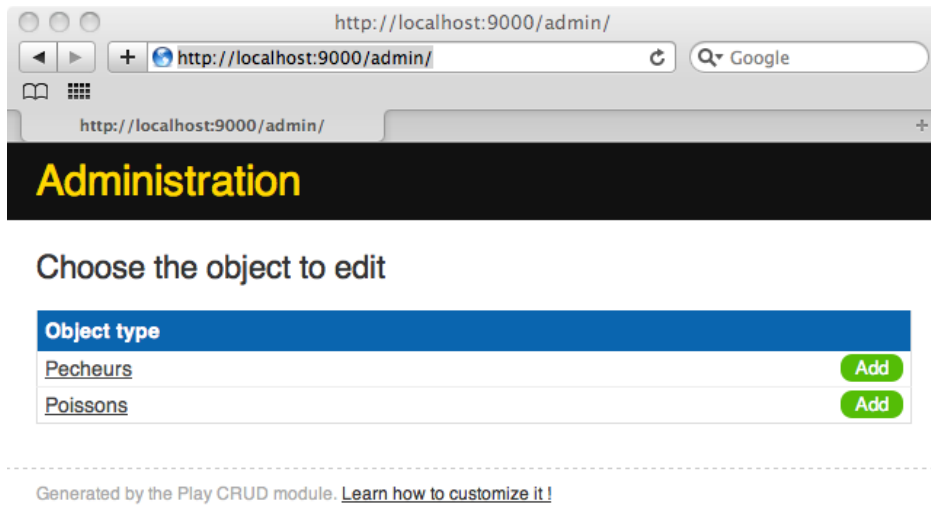
```
}
```

^^^

**\*\*C'est tout ?! ?\*\*** ... Ben voui ! C'est pas la classe ça ?

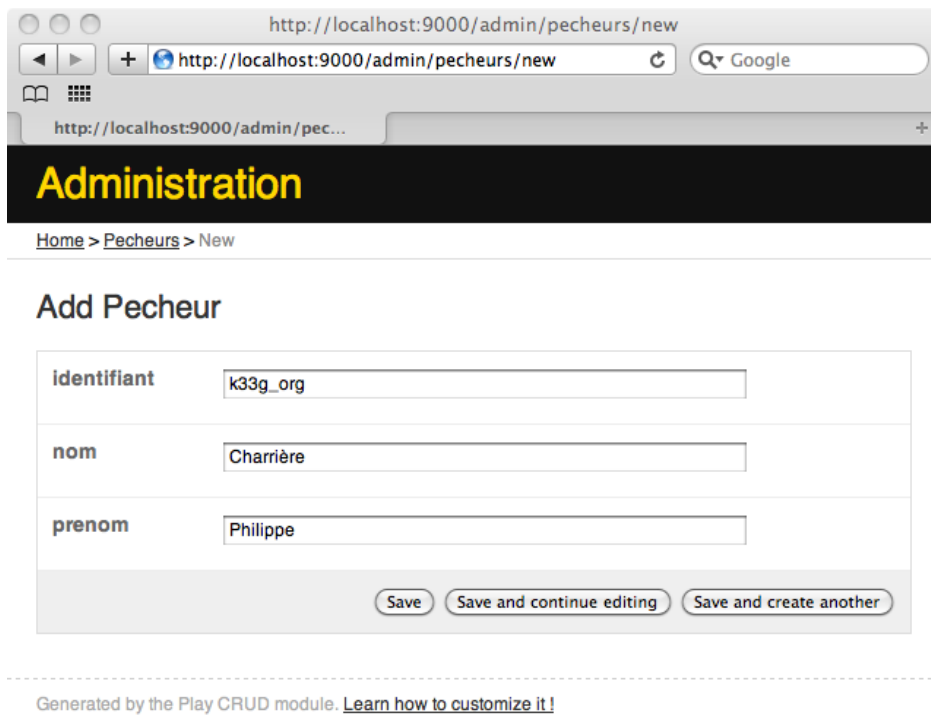
Vérifions quand même : [http://localhost:8000/admin/\(1\)](http://localhost:8000/admin/(1)), et là vous devriez avoir l'écran suivant :

1: <http://localhost:8000/admin/>



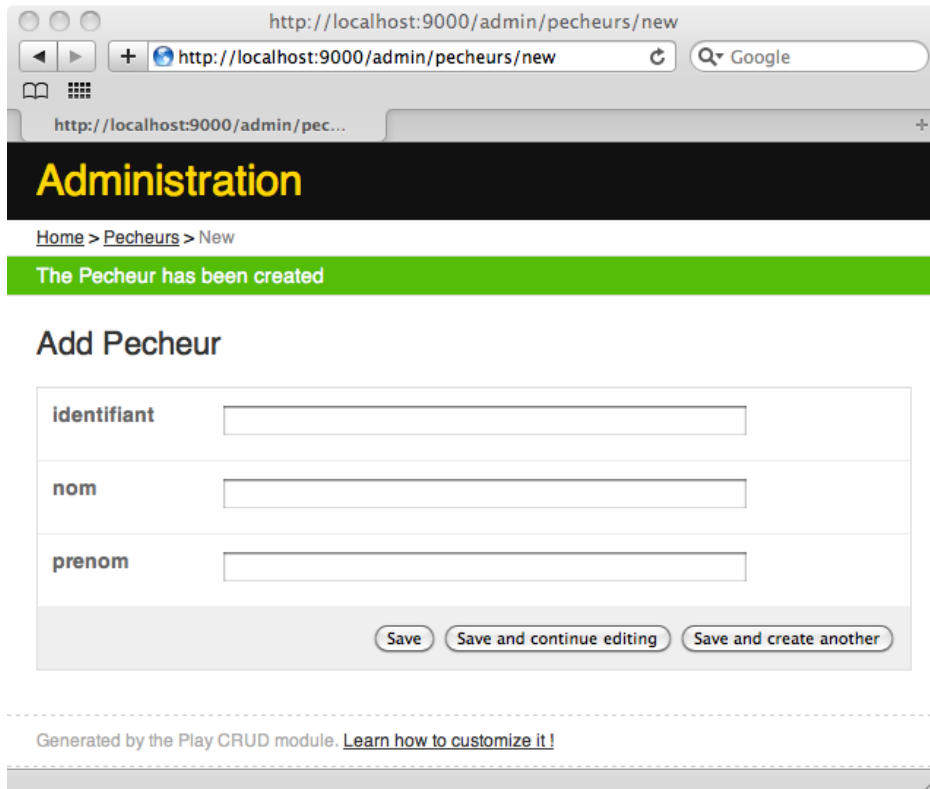
Pour être plus sûr : cliquez sur "add" à droite, sur la ligne "Pêcheur", nous allons ajouter quelques "amis pêcheurs" :

Vous pouvez voir que Play! s'est débrouillé comme un chef à partir de la structure de votre classe `Pêcheur`



Lorsque vous sauvegardez ("Save and create another"), Play! vous affiche un joli

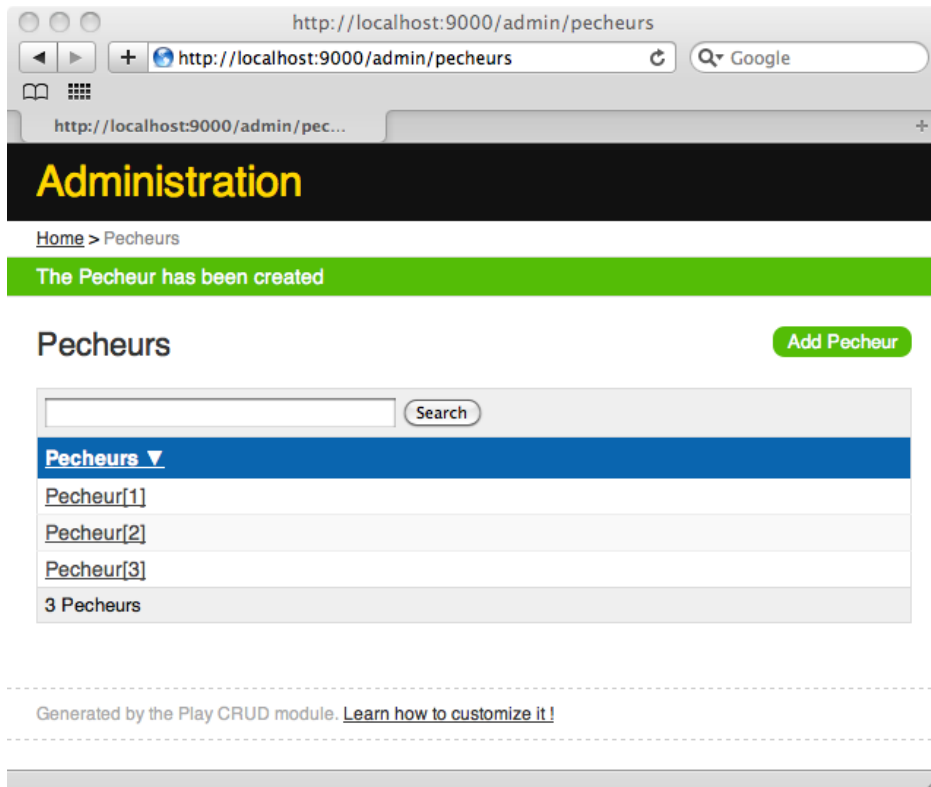
message d'encouragement (en vert)



The screenshot shows a web browser window with the address bar at `http://localhost:9000/admin/pecheurs/new`. The page has a black header with the word "Administration" in yellow. Below the header is a breadcrumb trail: `Home > Pecheurs > New`. A green message box states "The Pecheur has been created". Below this is a section titled "Add Pecheur" containing a form with three input fields: "identifiant", "nom", and "prenom". At the bottom of the form are three buttons: "Save", "Save and continue editing", and "Save and create another". At the very bottom of the page, a footer line reads: "Generated by the Play CRUD module. [Learn how to customize it !](#)".

Créons encore 2 pêcheurs ... : `loic\_d` et `mklabs`

Si vous revenez à la liste des pêcheurs (cliquez sur pecheurs en haut : `Home > Pecheurs`) vous obtenez ceci et c'est laid ! :



Retournons vite modifier le code de nos modèles :

Nous allons nous contenter de rajouter (surcharger en fait) la méthode `toString()` de nos 2 modèles

## Dans Classe Pecheur

```java

@Override

```
public String toString() {
    return identifiant+" : "+nom+" "+prenom;
}
```

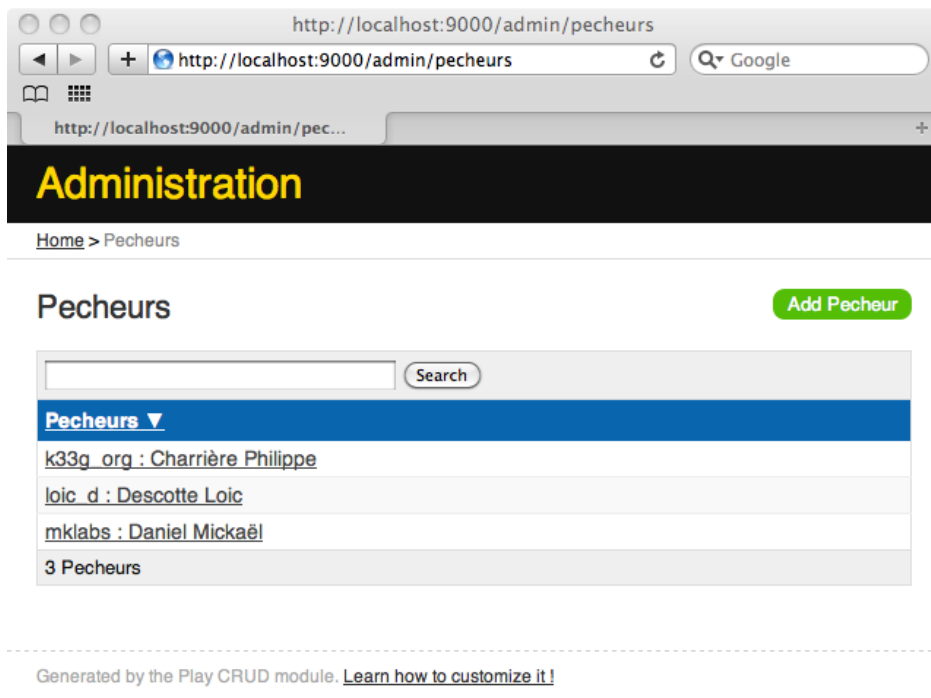
## Dans Classe Poisson

```java

@Override

```
public String toString() {
    return identifiant+" : "+nom;
}
```

Raffraichissez la page :



C'est mieux, non ?

Nous allons créer quelques poissons, mais avant de continuer ...

## Comment rendre la saisie obligatoire ?

Tout simplement en utilisant l'annotation `@Required` au dessus des champs obligatoires (dans notre classe `Poisson`) et en ajoutant la référence suivante dans le code de la classe : `import play.data.validation.Required;`, nous aurons donc le code suivant :

```
@@@ java
```

```
package models;
```

```
import javax.persistence.*;
```

```
import play.data.validation.Required;
```

```
import play.db.jpa.*;
```

@Entity

```
public class Poisson extends Model{
```

```
    @Required
    public String identifiant;
    @Required
    public String nom;
    public Poisson() {
    }
    public Poisson(String identifiant, String nom) {
        this.identifiant = identifiant;
        this.nom = nom;
    }
    @Override
    public String toString() {
        return identifiant+" : "+nom;
    }
}
```

î î î

\*Note : faite donc la même chose pour la classe `Pecheur`, ça sera fait\*

Vérifions si cela fonctionne en allant créer quelques poissons :

http://localhost:9000/admin/poissons/new

http://localhost:9000/admin/poissons/new

Google

http://localhost:9000/admin/poi...

## Administration

Home > Poissons > New

### Add Poisson

|             |                      |
|-------------|----------------------|
| identifiant | <input type="text"/> |
|             | Required.            |
| nom         | <input type="text"/> |
|             | Required.            |

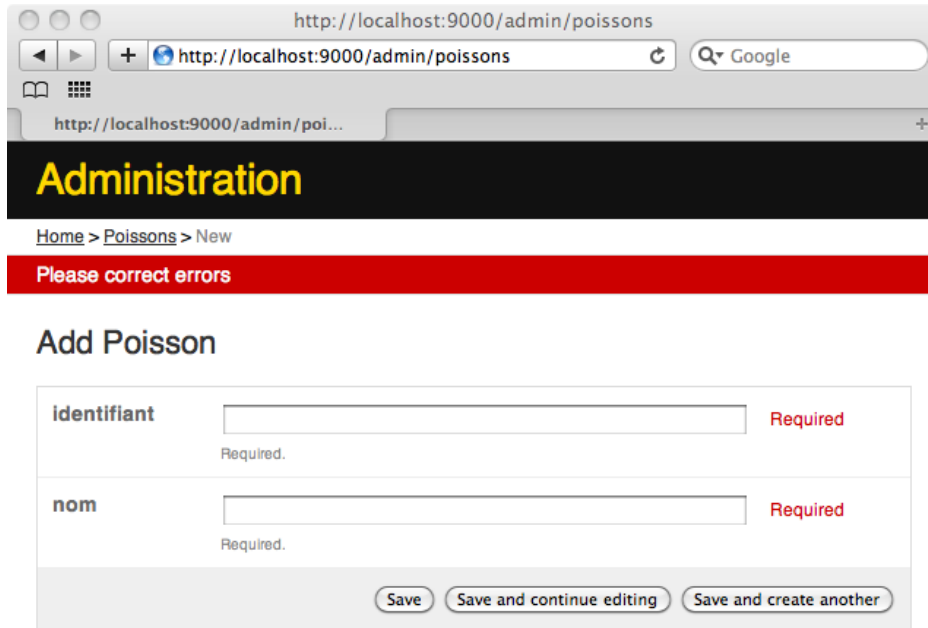
Save Save and continue editing Save and create another

Generated by the Play CRUD module. [Learn how to customize it!](#)

Vous pouvez déjà remarquer que l'annotation "Required" apparaît en dessous des

champs de saisie.

Essayez de sauvegarder sans avoir rien saisi :



The screenshot shows a web browser window with the URL `http://localhost:9000/admin/poissons`. The page title is "Administration". Below the title, there is a breadcrumb trail: `Home > Poissons > New`. A red banner at the top of the form area displays the message "Please correct errors". The form is titled "Add Poisson" and contains two input fields: "identifiant" and "nom". Both fields are marked as "Required" in red text. Below the "nom" field, there is a "Required." label. At the bottom of the form, there are three buttons: "Save", "Save and continue editing", and "Save and create another".

Generated by the Play CRUD module. [Learn how to customize it !](#)

Play! vous affiche un message d'erreur, et où sont les champs obligatoires, tout ça juste avec une simple annotation !

Bon, maintenant, il faut les saisir ces poissons ...

## Comment "exiger" un format de saisie

Vous pouvez aussi déclarer qu'un type de format est nécessaire comme l'e-mail, un chiffre, etc. ...

Pour cela allons modifier la classe `Pecheur`` en ajoutant un champ email et un champ département.

pour l'email, nous utiliserons l'annotation `@Email`` (la référence à déclarer dans le code sera : ``import play.data.validation.Email;``)

pour le département, c'est encore plus facile, le simple fait de le typer en ``Integer``



suffit

Le code de notre classe `Pecheur` va ressembler à ceci :

î î î java

```
package models;
```

```
import javax.persistence.*;
```

```
import play.data.validation.Email;
```

```
import play.data.validation.Required;
```

```
import play.db.jpa.*;
```

```
@Entity
```

```
public class Pecheur extends Model{
```

```
    @Required
    public String identifiant;
    @Required
    public String nom;
    @Required
    public String prenom;
    @Email
    public String email;
    public Integer departement;
    public Pecheur(){
    }
    public Pecheur(String identifiant, String nom, String prenom) {
        this.identifiant = identifiant;
        this.nom = nom;
        this.prenom = prenom;
    }
    @Override
    public String toString() {
        return identifiant+" : "+nom+" "+prenom;
    }
}
```

î î î

retournez dans votre appli web [http://localhost:8000/admin/pecheurs\(1\)](http://localhost:8000/admin/pecheurs(1)) :

1: <http://localhost:9000/admin/pecheurs>

sélectionnez un pêcheur à modifier :

http://localhost:9000/admin/pecheurs/1

Administration

Home > Pecheurs > k33g\_org : Charrière Philippe

Edit Pecheur [Delete Pecheur](#)

|             |  |
|-------------|--|
| identifiant | <input type="text" value="k33g_org"/>  |
|             | Required.                              |
| nom         | <input type="text" value="Charrière"/> |
|             | Required.                              |
| prenom      | <input type="text" value="Philippe"/>  |
|             | Required.                              |
| email       | <input type="text"/>                   |
|             | Valid email address.                   |
| departement | <input type="text"/>                   |
|             | Numeric.                               |

[Save](#) [Save and continue editing](#)

Vous pouvez remarquer que le formulaire précise qu'il souhaite une adresse mail valide pour l'email et une valeur de type numérique pour le département.

Soyons donc débile :

pour le mail, saisir : ph.charriere\_gmail.com

pour le département, saisir : Rhône

cliquer sur "Save"

The screenshot shows a web browser window with the URL `http://localhost:9000/admin/pecheurs/1`. The page title is "Administration". The breadcrumb trail is "Home > Pecheurs > k33g\_org : Charrière Philippe". A red banner at the top says "Please correct errors". The form is titled "Edit Pecheur" and has a "Delete Pecheur" button. The form fields are:

| Field       | Value                  | Validation                |
|-------------|------------------------|---------------------------|
| identifiant | k33g_org               | Required.                 |
| nom         | Charrière              | Required.                 |
| prenom      | Philippe               | Required.                 |
| email       | ph.charriere_gmail.com | Invalid email address.    |
| departement | Rhône                  | Incorrect value. Numeric. |

At the bottom of the form are two buttons: "Save" and "Save and continue editing".

Eh oui, Play! propose bien un module "anti-débile" et vous explique ce qu'il faut saisir avant d'enregistrer.

## Quelques remarques

### Recherches

Je ne sais pas si vous avez vu, mais dans les pages du module CRUD il y a un "petit" module de recherche, on l'essaie ? En images :

http://localhost:9000/admin/poissons

http://localhost:9000/admin/poissons

# Administration

[Home](#) > Poissons

The Poisson has been created

## Poissons

Add Poisson

| Poissons ▼                               |
|--|
| <a href="#">PRC : Perche commune</a>     |
| <a href="#">BRO : Brochet</a>            |
| <a href="#">SAN : Sandre</a>             |
| <a href="#">TRF : Truite Fario</a>       |
| <a href="#">TRA : Truite Arc-en-ciel</a> |
| <a href="#">BBS : Black-Bass</a>         |
| <a href="#">SIL : Silure</a>             |
| 7 Poissons                               |

Generated by the Play CRUD module. [Learn how to customize it !](#)

http://localhost:9000/admin/poissons?search=Truit

http://localhost:9000/admin/poissons?search=Truit

# Administration

[Home](#) > Poissons

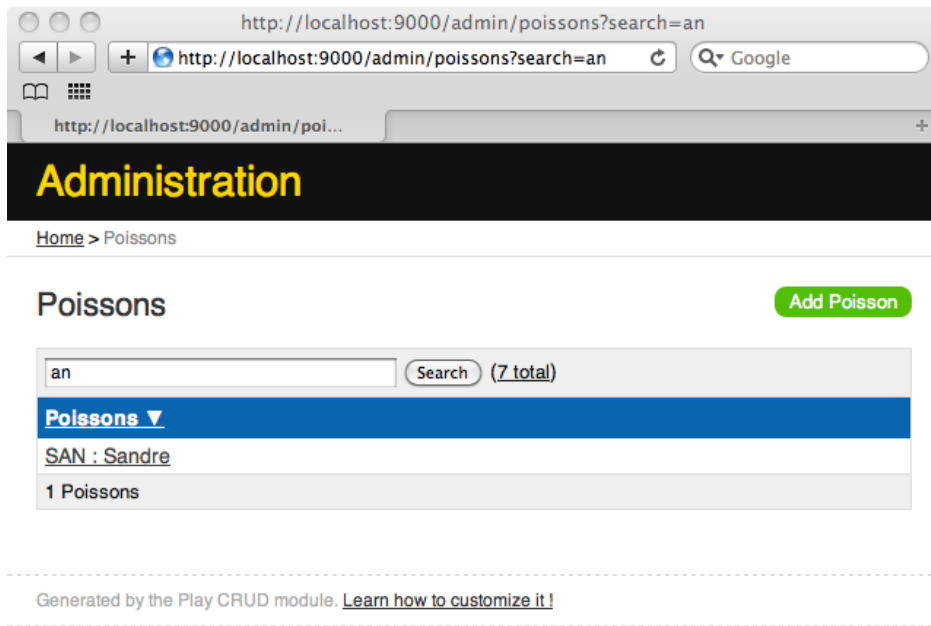
## Poissons

Add Poisson

(7 total)

| Poissons ▼                               |
|--|
| <a href="#">TRF : Truite Fario</a>       |
| <a href="#">TRA : Truite Arc-en-ciel</a> |
| 2 Poissons                               |

Generated by the Play CRUD module. [Learn how to customize it !](#)



Plutôt sympa, vous n'avez rien eu à coder pour ça :)

## Modèles

Vous avez aussi noté (j'espère) que toutes les modifications apportées aux classes modèles étaient répercutées à la fois sur la partie base de données et la partie affichage, et tout cela sans gros effort.

## La suite (c'est fini pour cette partie)

Nous verrons comment :

créer de nouveaux modèles et des relations entre eux (nous aborderons donc quelques notions de JPA)

aller plus loin dans la personnalisation de nos écrans CRUUD

c'est dans le chapitre suivant : ch04-Premiere-application-CRUUD-part2(1).

1: [ch04-Premiere-application-CRUUD-part2.md](#)

Vous n'avez pas encore le statut "demi-dieu" de la programmation Web en java \*(1)\*, mais je sens déjà quelque chose frétiller en vous :) Non ? Vous avouerez que pour le moment c'est assez facile et sans effort et pourtant ça a déjà de la gueule. J'espère que cela vous donne envie de continuer.

(1) : ne vous inquiétez pas, je n'ai pas cette prétention non plus ;), mais nous allons progresser ensemble

Note pour le chargement du module CRUUD :

Je propose d'utiliser la nouvelle méthode de chargement des modules avec le fichier dependencies.yml.

En effet, le chargement du module CRUUD par le fichier application.conf semble déprécié depuis la version 1.2.1 de play et provoque l'affichage du message suivant:

```
`1à:40:30,â26 WARN â Declaring modules in application.conf is deprecated. UUse dependencies.yml instead (module.crud)`
```

Cela donnerai donc pour activer le module CRUUD :

\* aller dans `/azerguespeche/conf/dependencies.conf`

\* ajouter ceci sur une nouvelle ligne à la suite de `- play` :

```
play -> crud  
# Import CRUD routes  
* /admin module:crud
```

\*\*Avant de continuer, arrêtez votre application :\*\*

En ligne de commande, taper : `play dependencies`

Cette commande permet d'installer les modules et/ou bibliothèques externes déclarés dans le fichier dependencies.yml.

Donc si tout se passe bien vous devriez voir un message indiquant que le module CRUUD a été installé.

Notre application CRUUD ... la suite

=

## Introduction

Dans l'épisode précédant nous avons initié notre application grâce au module CRUUD. Cette fois ci nous allons aller un peu plus loin. Nous verrons comment "franciser" notre site, comment lier les modèles et enfin modifier nos formulaires (enfin un formulaire).

## Tout d'abord quelques réglages

Dans application.conf cherchez le texte suivant :

```
e il( n
```

```
e î î î î î
```

```
e Define locales used by your application.
```

```
e 6 ou can then place localized messages in conf/messages.{locale} files
```

```
e application.langs=fr,en,ja
```

Ajoutez à la suite la ligne suivante :

```
application.langs=fr
```

Créer un fichier `messages.fr` dans le répertoire `conf` de l'arborescence du projet

Puis, cherchez le texte suivant : (en général c'est juste après)

```
e Date format
```

```
e î î î î î
```



```
date.format=yyyy-MM-dd
```

```
e date.format.fr=dd/MM/yyyy
```

Transformez moi ça en :

```
e Date format
```

```
e {{{{{
```

```
date.format=yyyy-MM-dd
```

```
date.format.fr=dd/MM/yyyy
```

Nous venons d'expliquer à Play que notre locale est française, et nous avons précisé le format de date, cela aura donc une incidence sur les écrans de saisie.

## Evolutions de notre modèle :

Nous allons créer des compétitions, qui serviront à enregistrer les prises faites par chacun des pêcheurs. Je ne vais pas gérer les inscriptions, on dit que l'on saisit uniquement les pêcheurs et leurs prises lors des compétitions.

Nous aurons donc : une compétition est composée de prises, une prise c'est un poisson pris par un pêcheur.

Créer les 2 Classes `Competition` et `Prise` dans le répertoire `models` :

## La Compétition

```
{{{{ java
```

```
package models;
```

```
import javax.persistence.*;
```

```
import play.data.validation.Required;
```

```

import play.db.jpa.*;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

@Entity

public class Competition extends Model {

    @Required
    public String nom;
    @Required
    public Date date;
    /*Il y a plusieurs prises dans une compétition
    on fait le lien avec la propriété 'competition' de la classe Prise
    */
    @OneToMany(mappedBy="competition", cascade=CascadeType.ALL)
    public List<Prise> prises = new ArrayList();
    public Competition() {
    }
    public Competition(String nom, Date date) {
        this.nom = nom;
        this.date = date;
    }
    @Override
    public String toString() {
        return nom + " : " + date;
    }
}

```

## La Prise

java

```

package models;

import javax.persistence.*;

import play.data.validation.Required;

```

```
import play.db.jpa.*;
```

```
@Entity
```

```
public class Prise extends Model {
```

```
    /* une prise est faite par un seul p cheur */
    @OneToOne
    @Required
    public Pecheur pecheur;
    /* on ne prend qu'un seul poisson   la fois */
    @OneToOne
    @Required
    public Poisson poisson;
    /* plusieurs prises dans une comp tition
       cette prise appartient   1 seule comp tition
    */
    @ManyToOne
    @Required
    public Competition competition;
    public Prise() {
    }
    public Prise(Pecheur parQui, Poisson poissonPris, Competition
pendantCompetition) {
        this.pecheur = parQui;
        this.poisson = poissonPris;
        this.competition = pendantCompetition;
    }
    @Override
    public String toString() {
        return "Prise{" +
            "par : " + pecheur.nom + " " + pecheur.prenom +
            ", poisson : " + poisson.nom +
            ", pendant : " + competition.toString() +
            '}' ;
    }
}
   
```

**\*\*Remarque :** nous venons d'utiliser des annotations JPA pour "lier" nos mod les : @OneToOne, @ManyToOne, @OneToMany ...

Vous pouvez trouver une description de ces annotation ici [http://www.oracle.com/technetwork/middleware/ias/toplink-jpa-annotations-0 6251.html\(1\)](http://www.oracle.com/technetwork/middleware/ias/toplink-jpa-annotations-0 6251.html(1))

1: <http://www.oracle.com/technetwork/middleware/ias/toplink-jpa-annotations-0à6251.html>

## N'oublions pas les contrôleurs

Il n'y a pas grand chose à écrire, mais sans ça cela ne fonctionnera pas.

Donc tout bêtement, créer les 2 Classes `Competitions` et `Prises` dans le répertoire `controllers` :

### Contrôleur Competitions

```
ïïï java
```

```
package controllers;
```

```
import play.*;
```

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

```
public class Competitions extends CRUD {  
  
}
```

```
ïïï
```

### Contrôleur Prise

```
ïïï java
```

```
package controllers;
```

```
import play.*;
```

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

```
public class Prises extends CRUUD {  
  
}
```

^^^

Vous pouvez lancer l'application et vérifier le bon fonctionnement de nos modifications.

saisissez quelques compétitions

saisissez quelques prises

Nous en aurons besoin pour la suite (pour le moment, je vous passe les "screenshots", nous n'avons rien fait de très compliqué).

## On passe une vitesse : Modifions les formulaires CRUUD !

Sérieux, on va avoir la prétention de modifier un "truc" qui fonctionne bien (très bien même), qui a été codé par des "pros" ?

Ben voui, on va se le permettre (et en plus Play est fait pour ça).

### Modifications simples

Comme vous avez pu vous en apercevoir, dans les formulaire CRUUD les libellés des champs de saisie, correspondent au nom des propriétés des classes modèles, donc pas d'accent, pas d'espace, que du brut de décoffrage, donc bof. Il y a un moyen simple pour changer ceci.

Dans le répertoire `conf` de l'arborescence, il y a le fichier `messages.fr`, dans lequel vous pouvez saisir des libellés "plus parlants" qui seront utilisés (entre autre) par les formulaires CRUUD.

**\*\*Remarque :\*** il est aussi possible de saisir dans le fichier `messages` (sans extension) mais cela sera valable pour toutes les langues

Saisissons dans ce fichier, ceci :

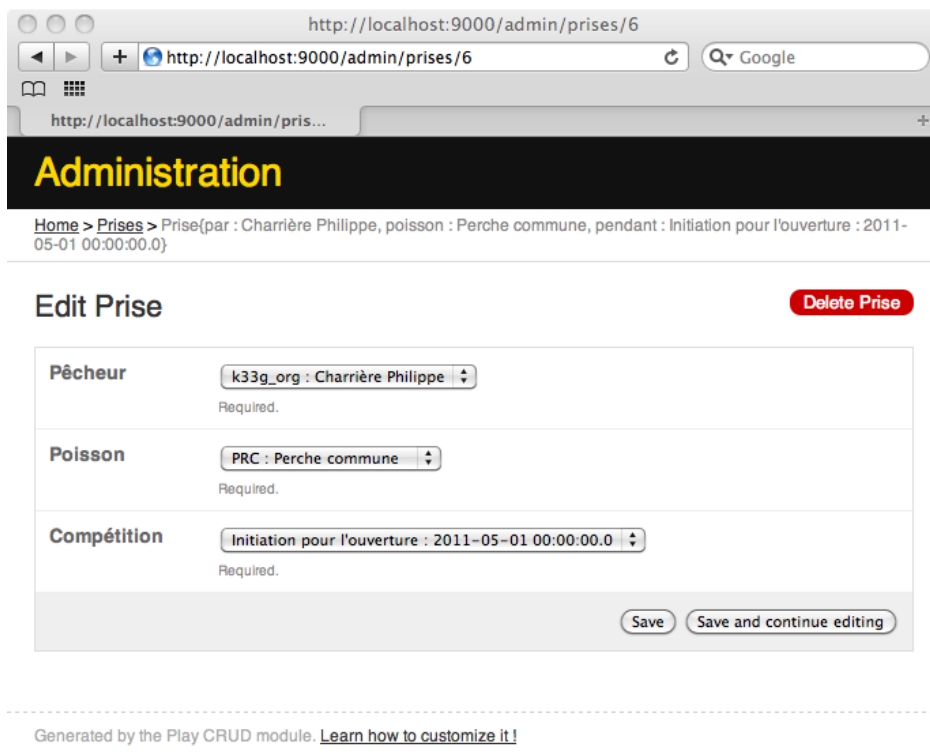
pecheur = Pêcheur

prise = Prise

poisson = Poisson

competition = Compétition

Allez faire un tour dans l'écran des prises :



The screenshot shows a web browser window with the URL `http://localhost:9000/admin/prises/6`. The page has a black header with the word "Administration" in yellow. Below the header, there is a breadcrumb trail: `Home > Prises > Prise(par : Charrière Philippe, poisson : Perche commune, pendant : Initiation pour l'ouverture : 2011-05-01 00:00:00.0)`. The main content area is titled "Edit Prise" and features a red "Delete Prise" button. The form contains three required fields: "Pêcheur" with a dropdown menu showing "k33g\_org : Charrière Philippe", "Poisson" with a dropdown menu showing "PRC : Perche commune", and "Compétition" with a dropdown menu showing "Initiation pour l'ouverture : 2011-05-01 00:00:00.0". At the bottom of the form are two buttons: "Save" and "Save and continue editing". Below the form, there is a footer that reads: "Generated by the Play CRUD module. [Learn how to customize it!](#)".

C'est plus pro, ça coûte pas cher, et c'est facile, voire trop facile! (la magie de Play!)

**\*\*Remarque :** il est possible d'avoir autant de fichiers `messages.xxx` que de langues.

## Modifications plus "profondes"

J'aimerais (nous aimerions) bien que l'écran d'une compétition affiche la liste des prises de la compétition (ce qui n'est pas le cas actuellement si vous avez bien suivi).

Le module CRUUD permet de générer automatiquement le code d'une vue (liées à son controller), et donc de se passer de la version "générée à la volée"

arrêtez l'application

allez dans le répertoire de l'application : `cd azerguespeche/`

tapez la commande `play crud:ov --template Competitions/show`

Cela vient de créer dans le répertoire `views` un répertoire `Competitions` avec un template `show.html` avec le code suivant :

```
!!! html
```

```
e {extends 'CRUUD/layout.html' /}
```

```
<div id="crudShow" class="{type.name}">
```

```
<h2 id="crudShowTitle">{{crud.show.title, type.modelName}}</h2>
```

```
<div class="objectForm">
```

```
e {form action:@save(object._key()), enctype:'multipart/form-data'}
```

```
e {crud.form /}
```

```
<p class="crudButtons">
```

```
    <input type="submit" name="_save" value="{ 'crud.save', type.modelName}" />
    <input type="submit" name="_saveAndContinue" value="{ 'crud.
saveAndContinue', type.modelName}" />
</form>
```

```
</div>
```

```
e {form @delete(object._key())}
```

```
<p class="crudDelete">
```

```
    <input type="submit" value="{ 'crud.delete', type.modelName}" />
</form>
```

```
</div>
```

```
î î î
```

vous pouvez remonter d'un cran : ``cd ..``

relancer votre application : ``play run azerguespeche`` (ça sera fait)

**\*\*Donc\*\***, dorénavant, lorsque vous utiliserez le formulaire d'édition des compétitions, ce sera ce template qui sera utilisé.

**\*\*Remarque : \*\*** pour modifier le formulaire de liste (toujours du module CRUUD), la commande serait la suivante : ``play crud:ov --template Competitions/list``, pour modifier le template CRUUD general (layout.html) dont héritent toutes les vues CRUUD, la commande serait la suivante : ``play crud:ov --layout`` ... Amusez vous (sauvegardez avant).



Nous allons créer un "tag" qui va permettre d'afficher la liste des prises d'une compétition

Dans un 1er temps, remplacer dans `show.html`, `{crud.form /}` par `{crud.form}` `{/crud.form}`,

Puis au sein de la nouvelle balise, saisissez le code suivant :

```
!!! html
```

```
{crud.custom 'prises'}
```

```
<div>
  <ul>
    #{list items:object.prises, as:'prise'}
      <li>#{prise.toString()}</li>
    #{/list}
  </ul>
</div>
#{/crud.custom}
```

**Remarque :** `object` représente l'objet (l'instance de classe) lié au formulaire.  
Dans le cas qui nous intéresse, c'est une compétition. Or, il se trouve que la classe `Competition` a une propriété `prises` qui contient la liste des prises d'une compétition.

Donc pour avoir la liste, il suffit d'appeler `object.prises` et de le parcourir avec `{list}{/list}`.

Au final, votre template devrait ressembler à ceci :

```
!!! html
```

```
{extends 'CRUUD/layout.html' /}
```

```
<div id="crudShow" class="{type.name}">
```

```
<h2 id="crudShowTitle">&{'crud.show.title', type.modelName}</h2>
```

```
<div class="objectForm">
```

```
e {form action:@save(object._key()), enctype:'multipart/form-data'}
```

```
    #{crud.form}
    #{crud.custom 'prises'}
    <div>
      <ul>
        #{list items:object.prises, as:'prise'}
        <li>${prise.toString()}</li>
      #{/list}
    </ul>
  </div>
  #{/crud.custom}
#{/crud.form}
  <input type="submit" name="_save" value="&{'crud.save', type.modelName}" /
>
  <input type="submit" name="_saveAndContinue" value="&{'crud.
saveAndContinue', type.modelName}" />
e {/form}
```

```
</div>
```

```
e {form @delete(object._key())}
```

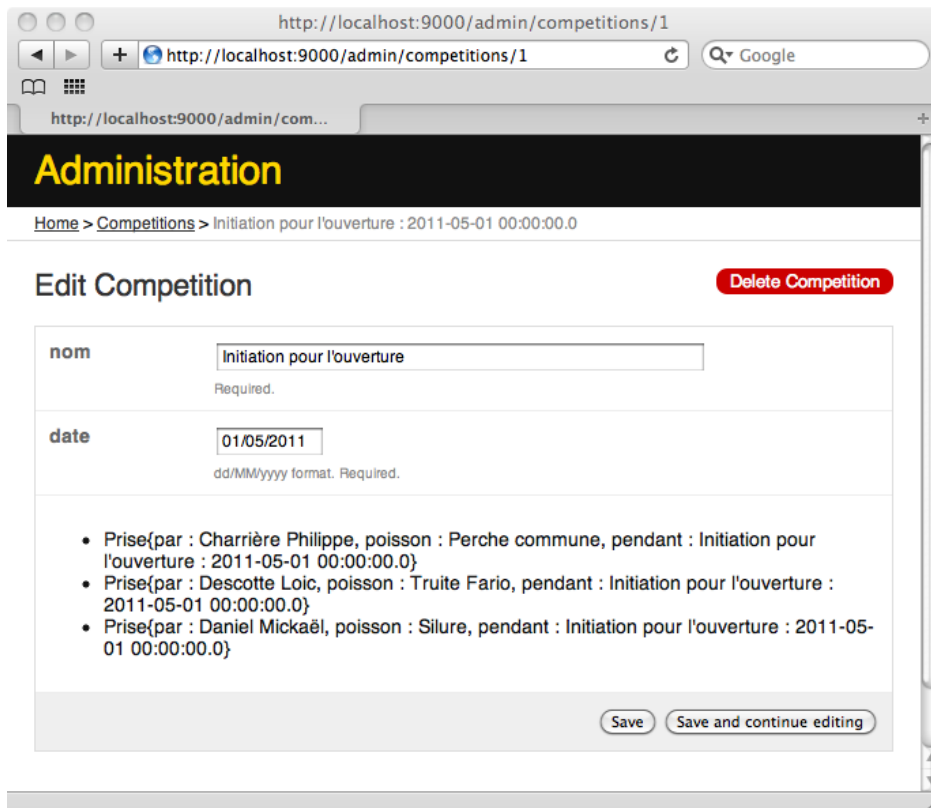
```
<p class="crudDelete">
```

```
  <input type="submit" value="&{'crud.delete', type.modelName}" />
e {/form}
```

```
</div>
```

```
î î î
```

Et si tout va bien, vous devriez obtenir cela :



Ok, ça casse pas 3 pattes à un canard esthétiquement parlant (encore que j'ai vu beaucoup plus moche sur des projets facturés ;) ), mais avouez que l'effort est bien faible au regard du résultat. Est-ce que vous commencez à vous sentir plus "à l'aise" avec Play! ?

Encore quelques étapes, et vous en saurez assez pour commencer à "bidouiller" des choses un peu plus "trapues".

## La suite (c'est fini pour cette partie)

Nous verrons comment :

Créer des services

Créer "from scratch" des pages qui utilisent ces services

**\*\*Remarque : N'hésitez pas\*\*** à aller ici : [https://github.com/3monkeys/play.rules/issues\(1\)](https://github.com/3monkeys/play.rules/issues(1)) pour poser vos questions ou faire vos remarques, je ne suis pas forcément

toujours assez clair, et surtout je peux faire des erreurs.

1: <https://github.com/3monkeys/play.rules/issues>

# Mise en place de l'application Vote4Music

Dans la partie précédente nous avons vu comment générer une partie des traitements de notre application grâce au module CRUUD.

Nous allons maintenant apprendre à développer une petite application entièrement "à la main".

Le but de cette webapp est d'offrir la possibilité de parcourir une CDthèque, d'en ajouter de nouveaux et de voter pour vos albums préférés. Elle notre fil conducteur tout au long de cette partie.

Le code complet de l'application est disponible ici(1)

1: <https://github.com/loicdescotte/vote4music>

## Le modèle de données

### La classe Album

La classe Album contient les informations suivante :

Nom de l'album (obligatoire)

Référence à l'artiste (obligatoire)

Année de sortie (obligatoire)

Le genre (ou style de musique)

Le nombre de votes que cet album a reçu

Voici le code de cette classe :

```
ïïï java
```

```
@Entity
```

```
public class Album extends Model {
```

```
@Required
```

```
public String name;
```

```
@Required
```

```
@ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
```

```
public Artist artist;
```

```
@Required
```

```
public Date releaseDate;
```

```
@Enumerated(EnumType.STRING)
```

```
public Genre genre;
```

```
}
```

```
ïïï
```

Nous verrons le code métier de cette classe dans la suite du chapitre.

## La classe Artist

La classe Artist est définie comme ceci :

```
ïïï java
```

```
public class Artist extends Model{
```

```
    @Required
    @Column(unique = true)
    public String name;
```

```
}}
```

## L'enum Genre

Le genre est une simple Enum, définie comme cela :

```
}}} java
```

```
public enum Genre {
```

```
    ROCK, METAL, JAZZ, BLUES, POP, WORLD, HIP_HOP, OTHER
```

```
}}
```

Vous pouvez bien sur ajouter autant que genres que vous voulez.

## Définition des routes

Les routes que nous allons définir permettront :

De consulter les albums

De rechercher des albums

De consulter le top 10 par genre

D'ajouter de nouvelles entrées dans la bibliothèque d'albums

Le fichier `routes` de notre application se présente ainsi :

e UUser pages

GET	/	Application.
index		
POST	/album	Application.

```

save
  GET      /albums                                Application.
list
  GET      /search                                Application.
search
  GET      /topalbums                             Application.
listByGenreAndYear
  GET      /album/new                             Application.
form
  GET      /album/{id}                            Admin.form
  POST     /api/album                             Application.
saveAlbumJson
eVote

  POST     /vote                                  Application.vote
  GET      /public/                               staticDir:
public
  *        /{controller}/{action}                {controller}.
{action}

```

## La page d'accueil

La page d'accueil permet d'accéder aux principales fonctionnalités de l'application :

Le formulaire de création d'albums

La liste des albums

Le top 10 par genre

Les fonctions d'administration (que nous verrons au chapitre 3)

Pour le top 10, vous pouvez choisir un style de musique. Pour cela, le template Play utilise l'enum Genre :

```

{{{ html

```

```

<label for="genre">Genre:</label>

```

```

<select id ="genre" name="genre">

```

```

    #{list models.Genre.values(), as:'genre'}
    <option value="{{$genre}}">{{$genre.toString().toLowerCase()}}</option>
    #{/list}
{{{

```



N.B. : Le langage d'expression utilisé dans les templates est Groovy(1). C'est un langage à typage dynamique très proche de Java, qui nous permet de manipuler facilement les objets renvoyés par le contrôleur.

1: <http://groovy.codehaus.org/>

## Le formulaire d'ajout

Dans le contrôleur, on crée une méthode pour obtenir le formulaire :

```
/// java
```

```
public static void form() {
```

```
    render();
```

```
///
```

En se référant aux routes, on voit que cette méthode est invoquée lorsque l'on utilise le verbe HTTP GET. C'est la méthode utilisée par le navigateur lorsque l'on tape une URL ou lorsque l'on clique sur un lien.

On utilise ensuite POST pour envoyer les données au contrôleur (voir le fichier de routes plus haut) Voici le code du formulaire :

```
/// html
```

```
{extends 'main.html' /}
```

```
{set title:'Album form' /}
```

```
<h1>Please write information about your favorite album</h1>
```

```
{form @Application.save(), id:'form',
method:'POST', enctype:'multipart/
form-data'}
```

```
<input type="hidden" name="album.id" value="{album?.id}"/>
```

```
<p class="field">
```

```
    <label for="name">Album Name:</label>
    <input type="text" name="album.name" id="name" value="{album?.name}"/>
    <span class="error">${errors.forKey('album.name')}</span>
```

```
<p class="field">
```

```
    <label for="artist">Artist:</label>
    <input type="text" name="artist.name" id="artist" value="{album?.artist?.
name}"/>
    <span class="error">${errors.forKey('artist.name')}</span>
```

```
<p class="field">
```

```
    <label for="genre">Genre:</label>
    <select id="genre" name="album.genre">
      #{list models.Genres.values(), as:'genre'}
      #{if album?.genre == genre}
        <option value="{genre}" selected="selected">${genre.toString().
toLowerCase()}</option>
      #{/if}
      #{else}
        <option value="{genre}">${genre.toString().toLowerCase()}</option>
      #{/else}
    </select>
```

```
<p class="field">
```

```
    <label for="release-date">Release date</label>
    <input type="text" name="album.releaseDate" id="release-date"
value="{album?.releaseDate?.format('yyyy-MM-dd')}" />
    <span class="error">${errors.forKey('album.releaseDate')}</span>
```

```
<p class="buttons">
```

```
    <a href="/albums" class="button">Cancel</a>
    <span>or</span>
    <input type="submit" class="button" value="Save this album"
```

```
id="saveAlbum"/>
{/form}
```

î î î

Ce formulaire nous permettra aussi bien de créer des utilisateurs que de les mettre à jour. C'est pour cette raison que nous utilisons une syntaxe comme `album?.name` pour la valeur des champs : si l'album existe déjà on affiche son nom. Sinon, on n'affiche rien. On retrouve également la sélection des genres à partir de l'Enum, comme sur la page d'accueil.

Pour permettre à l'utilisateur de sélectionner une date à l'aide d'un widget, on ajoute ce code JavaScript à notre template :

î î î html

```
{set 'moreScripts'}
```

```
<script src="@{'public/javascripts/jquery.validate.js'}"></script>
```

```
<script>
```

```
$(document).ready(function() {
    $("#form").validate();
});
$(function() {
    $("#release-date").datepicker({dateFormat: 'yy-mm-dd', showAnim:
'fadeIn'});
});
```

```
{/set}
```

î î î

Ce script utilise jQuery, comme tous les exemples de code JavaScript que nous

verrons dans ce chapitre.

Enfin, définissons la méthode du contrôleur qui va nous permettre d'enregistrer un album dans la base :

^^^ java

```
public static void save(@Valid Album album, @Valid Artist artist, File cover) {
```

```
    if (Validation.hasErrors()) {  
        render("@form", album);  
    }  
    album.artist = artist;  
    album.replaceDuplicateArtist();  
    album.save();  
    list();  
}
```

^^^

La première ligne de cette méthode vérifie que les valeurs envoyées au contrôleur sont conformes au modèle défini dans les classes Album et Artist (par exemple le nom obligatoire pour l'album).

Dans le cas contraire, on retourne au formulaire, qui affichera les erreurs grâce aux balises d'erreur que l'on écrit, comme

```
<span class="error">${errors.forKey('album.name')}</span>
```

La méthode replaceDuplicateArtist de la classe Album permet d'éviter les doublons de nom d'artistes dans la base de données :

^^^ java

```
public void replaceDuplicateArtist() {
```

```
    Artist existingArtist = Artist.find("byName", name).first();  
    if (existingArtist != null) {  
        artist = existingArtist;  
    }  
}
```

^^^

On accède à la base de données en utilisant les méthodes statiques fournies par la classe `Model`. La méthode `find` permet de passer des requêtes pour obtenir des entités enregistrées précédemment.

A la fin de l'action `save`, on retourne à la liste d'albums pour voir apparaître le nouvel élément enregistré.

Vous vous demandez peut être comment les transactions en base de données sont gérées dans cet exemple. La méthode `save` est bien transactionnelle. En fait dès qu'il a besoin d'accéder à la base de données, Play ouvre une transaction en début de requête HTTP, qui sera terminée en fin de requête. Si quelque chose se passe mal durant cet intervalle de temps, un rollback sera effectué.

Autre point important, on a utilisé la syntaxe `byName` pour écrire notre requête. Cette syntaxe supporte également des cas plus avancés.

On peut utiliser les mots clés suivants pour générer des requêtes :

`LessThan` (inférieur)

`LessThanEquals` (inférieur ou égal)

`GreaterThan` (supérieur)

`GreaterThanEquals` (supérieur ou égal)

`Like` (\*)

`NotEqual` (différent)

`Between` (compris entre 2 valeurs)

`IsNotNull` (non null)

`IsNull` (null)

Les mots clés peuvent être liés avec des `"And"`. On peut par exemple écrire

```
`find("byNameAndGenre", name, genre)` ou  
`find("byNameLikeAndGenreIsNotNull", name, genre)`.
```

La méthode `find` prend un nombre indéfini de paramètres (grâce à la syntaxe `...`) :

î î î java

```
JPAQuery find(String query, Object... params);
```

î î î

\*Il existe différents types de 'like' selon la sensibilité qu'on veut donner à la casse. Le mot clé `Like` va chercher des mots clés en minuscule dans la base, `Ilike` est complètement insensible à la casse, alors que `Elike` et équivalent au `like` SQL n'effectue aucune conversion.

## Lister et rechercher des albums

On utilise jQuery et le plugin datatables pour améliorer le rendu du tableau des résultats. Ce plugin permet d'afficher des liens pour trier le tableau, et ajoute la pagination des données.

Ce plugin est très simple à utiliser, il suffit d'écrire ces quelques lignes pour l'activer :

î î î js

```
$(document).ready(function(){  
  
    $('#albumList').dataTable();  
});
```

î î î

Ceci suffit à ajouter des fonctions de pagination et de tri à un simple tableau HTML. Notre tableau est défini comme ceci :

î î î html

```
<table id="albumList">
```

```

<thead>
  <tr>
    <th>Artist</th>
    <th>Album</th>
    <th>Release date</th>
    <th>Genre</th>
    <th>Number of votes</th>
  </tr>
</thead>
#{list _albums, as:'album'}
<tr id="album-#{album.id}">
  <td>#{album.artist.name}</td>
  <td>#{album.name}</td>
  <td>#{album.releaseDate.format('yyyy-MM-dd')}</td>
  <td>#{album.genre.toString()}</td>
  <td>
    <span id="nbVotes#{album.id}">#{album.nbVotes}</span>
    <a id="#{album.id}-clickVote" class="voteLink" href="#">Vote for it!
</a>
  </td>
</tr>
#{/list}

```

î î î

Nous plaçons ce code dans un fichier nommé `albumtable.tag`, séparé du reste de notre page, afin de pouvoir de réutiliser dans d'autres contextes :

Pour intégrer ce tag Play à notre page, on écrit la directive suivante :

```
e {albumtable albums:albums/}
```

Par défaut, on affiche les 100 derniers résultats trouvés dans la base de données :

î î î java

```

public static void list() {

    List<Album> albums = Album.all().fetch(100);
    render(albums);
}

```

î î î

Au dessus de notre tableau, nous définissons un champ de recherche qui permettra d'envoyer des filtres au serveur :

î î î html

```
{form @search() }
```

```
<input type="text" id="filter" name="filter"/>
```

```
<input type="submit" value="Filter" class="button" id="submitFilter">
```

```
{/form }
```

î î î

La variable `filter` est récupérée dans le contrôleur. Elle permet de trouver des noms d'albums ou d'artistes correspondant à la saisie de l'utilisateur.

Comme dans le cas précédent, on ne ramène que 100 résultats à la fois côté client. Si l'utilisateur a besoin de parcourir plus de résultats pour trouver ce qu'il cherche, on l'incite à utiliser le formulaire de recherche pour affiner les résultats.

Cette solution est plus simple pour nous du point de vue du code, par rapport à l'option qui consisterait à rappeler le serveur lors des clics sur les liens de pagination pour aller au delà de 100 résultats.

Le contrôleur intercepte l'appel de cette manière:

î î î java

```
public static void search(String filter) {
```



```
List<Album> albums = Album.findAll(filter);  
render(albums);  
{{{
```

La méthode `findAll` est définie comme ceci :

```
{{{ java
```

```
<T> List<T> findAll();
```

```
}}}
```

Le mécanisme d'inférence de type nous permet de récupérer une liste correctement typée (ici, `List<Album>`).

La classe Album définit la méthode de recherche avec un filtre sur le nom :

```
{{{ java
```

```
public static List<Album> findAll(String filter) {
```

```
    String likeFilter = "%" + filter + "%";  
    List<Album> albums = find("byNameLike", likeFilter).fetch(100);  
    return albums;  
}
```

```
}}}
```

Selon nos besoins, on peut bien sûr enrichir les filtres et les requêtes pour obtenir des résultats plus précis.

## Le top 10

Cette fonction de l'application permet d'afficher les 10 albums ayant reçu le plus de votes, pour une année et un genre donnés.

Sur la page d'accueil, on ajoute la possibilité de choisir le genre et l'année durant laquelle sont sortis les albums :

```
{{{ html
```

## {form @listByGenreAnd6 ear() }

```
<label for="year">Release 6 ear</label>
```

```
<select id="year" name="year">
```

```
    #{list controllers.Application.getYearsToDisplay(), as:'year'}
    <option value="${year}">${year}</option>
    #{/list}
<br/>
```

```
<label for="genre">Genre:</label>
```

```
<select id="genre" name="genre">
```

```
    #{list models.Genre.values(), as:'genre'}
    <option value="${genre}">${genre.toString().toLowerCase()}</option>
    #{/list}
<input type="submit" class="button" value="View"/>
```

## {/form}

^^^

On rend cette fonctionnalité accessible depuis le contrôleur :

^^^ java

```
public static void listByGenreAnd6 ear(String genre, String year) {
```

```
    notFoundIfNull(genre);
    notFoundIfNull(year);
    List<Album> albums = Album.findByGenreAndYear(genre, year);
    render(genre, year, albums);
    ^^^
```

Les paramètres `genre` et `year` sont obligatoires. Cela veut dire que si on appelle ce

contrôleur dans ces paramètres, il renverra une erreur 404 (not found).

La classe Album définit les méthodes nécessaires à cette recherche :

^^^ java

```
public static List<Album> findByGenreAndYear(String genre, String year) {  
  
    List<Album> albums;  
    Genre genreEnum = Genre.valueOf(genre.toString().toUpperCase());  
    albums = find("genre = ? order by nbVotes desc", genreEnum).fetch(100);  
    albums = filterByYear(albums, year);  
    return albums;  
}
```

La syntaxe de notre requête est un peu différente de celle que l'on a utilisé dans les exemples précédents. La méthode `find` est capable de traiter différents types de syntaxe. Ici on utilise la syntaxe standard JPQL (JPA Query Language), plus adaptée pour faire des requêtes avancées.

Pour proposer les dates disponibles depuis le contrôleur, on calcule un intervalle de dates allant de l'album le plus récent à l'album le plus ancien.

Si la base est vide on donne des valeurs par défaut :

^^^ java

```
public static List<String> getYearsToDisplay() {  
  
    List<String> years = new ArrayList<String>();  
    for (int i = Album.getFirstAlbumYear(); i <= Album.getLastAlbumYear();  
i++) {  
        years.add(String.valueOf(i));  
    }  
    Collections.reverse(years);  
    return years;  
}
```

La classe Album implémente les méthodes `getFirstAlbumYear` et `getLastAlbumYear`, qui récupèrent ces valeurs dans la base de données :

^^^ java

```
private static SimpleDateFormat formatYear = new SimpleDateFormat("yyyy");
```

```
public static int getFirstAlbumYear() {
```

```
    Date result = (Date) em().createQuery("select min(a.releaseDate) from  
Album a").getSingleResult();  
    if (result != null)  
        return Integer.parseInt(formatYear.format(result));  
    return 1990;
```

```
public static int getLastAlbumYear() {
```

```
    Date result = (Date) em().createQuery("select max(a.releaseDate) from  
Album a").getSingleResult();  
    if (result != null)  
        return Integer.parseInt(formatYear.format(result));  
    return Integer.parseInt(formatYear.format(new Date()));
```

^^^

La librairie `lambdaj` nous aide à filtrer l'ensemble des albums récupérés pour une année donnée. Grâce à elle, nous pouvons écrire nos filtres comme dans un langage fonctionnel, en évitant de créer des boucles pour parcourir la collection d'albums dans le but de la trier. Dans cet exemple, on utilise la fonction ``select`` :

^^^ java

```
public static List<Album> filterByYear(List<Album> albums, String year) {
```

```
    return select(albums, having(on(Album.class).getReleaseYear(),  
equalTo(year)));
```

^^^

N.B. : On aurait pu se passer de cette librairie, appliquer les filtres à l'aide d'une requête en base de données. Mais cet exemple nous permet de voir comment intégrer d'autres librairies à notre application Play, tout en obtenant un code intéressant du point de vue de la syntaxe.

Pour que Play puisse bénéficier de lambdaj, on ajoute cette ligne à la section `require` du fichier dependencies.yml :

com.googlecode.lambdaj -> lambdaj 2.2

## La fonction de vote

Voyons maintenant une fonctionnalité clé de cette application, le vote!

Cette méthode du contrôleur permet d'enregistrer un vote pour un album :

``` java

```
public static void vote(String id) {  
  
    Album album = findById(Long.parseLong(id));  
    album.vote();  
    renderText(album.nbVotes);  
}
```

```

Si vous avez une bonne mémoire, vous vous souvenez qu'on avait ajouté une route "catch all" à notre fichier de configuration `routes` :

e Catch all

```
*           /{controller}/{action}           {controller}.  
{action}
```

Dans le cas présent, la méthode `vote` sera accessible depuis l'URL `/application/vote`.

La classe Album définit cette méthode pour mettre à jour le compteur des votes d'une instance d'album:

``` java

```
public void vote() {
```

```
    nbVotes++;  
    save();
```

```
}}
```

Les entités du modèle pouvant auto-gérer leur état dans la base de données, on peut directement appeler la méthode `save` pour sauvegarder ce nouvel état.

La méthode du contrôleur renvoie directement le nouveau score de l'album au format texte. On récupérera cette réponse dans notre client HTML pour mettre à jour les informations affichées à l'écran.

Le bouton de vote est accessible dans la liste des albums :

```
html
```

```
<td>
```

```
    <span id="nbVotes${album.id}">${album.nbVotes}</span>  
    <a id="${album.id}-clickVote" class="voteLink" href="#">Vote for it!</a>
```

```
</td>
```

On crée aussi une `div` pour afficher un message en cas de succès :

```
html
```

```
<div id="voteInfo" class="info">One vote added!</div>
```

Cette section sera masquée par défaut, à l'aide de CSS :

```
css
```

```
info {
```

```
    display: none;
```

```
}
```

Ce code JavaScript permet d'intercepter les clicks et de rafraîchir l'écran :

î î î js

On récupère les span dont l'id commence par "nbVotes" pour trouver la zone à mettre à jour

```
var nbvotes = $('span[idç="nbVotes"]');
```

```
clickVote = function() {
```

```
    var id = t.attr('id').split('-')[0],
    voteTarget = nbvotes.filter("[id$=" + id + "]");
    $(this).hide();
    $.ajax({
        url: '/application/vote',
        type: "POST",
        data: {id: id},
        complete: function(req) {
            var newTotal = req.responseText;
            if (req.status === 200) {
                voteTarget.text(newTotal);
                voteInfo.slideDown("slow").delay(3000).slideUp("slow");
            }
        }
    });
});
```

```
$('#a.voteLink').click(clickVote);
```

î î î

## Gestion des pochettes d'albums

On veut maintenant ajouter la possibilité d'attacher l'image d'une pochette aux albums.

On enrichit la classe Album d'un nouveau champ :

ïïï java

```
public boolean hasCover = false;
```

ïïï

Ce booléen nous permettra de savoir si l'album possède une pochette ou non.

On ajoute une colonne à la liste des albums. Lors de l'affichage, on effectue le test suivant :

ïïï html

```
<td>
```

```
    #{if album?.hasCover}
    <span class="cover"><a href="#">Show cover</a></span>
    #{/if}
```

ïïï

Lors du survol de ce lien, on affiche une miniature de la pochette avec un peu de JavaScript :

ïïï js

```
$('.cover').each(function(i, val) {
```

```
    var t = $(this);
    var album = t.closest('tr').attr("id");
    var id = album.match(/album-(\\d)/)[1];
    displayCover(id, t);
```

Affichage de l'image

```
displayCover = function(id, albumMarkup){
```

```
    var root = '/public/shared/covers';
    var markup = '<img src="' + root + '/' + id + '" width="200"
```



```

height="200">';
    albumMarkup.bt(markup, {
        width: 200,
        fill: 'white',
        cornerRadius: 20,
        padding: 20,
        strokeWidth: 1,
        trigger: ['mouseover', 'click']
    });
}

```

Ce code récupère une image dans un répertoire du serveur et effectue son rendu à l'aide du plugin jQuery bt (BeautyTips).

Voyons maintenant comment enregistrer l'image dans ce répertoire lors de la création d'un album.

## UUpload et sauvegarde d'une image

On ajoute un champ dans le formulaire de création (et d'édition) de l'album :

```

<div class="form">
    <p class="field">
        <label for="cover">Cover</label>
        <input type="file" id="cover" name="cover" accept="image/*"/>
        <#if album?.hasCover>
            <br/>
            
        </if>
    </p>
</div>

```

Ce champ permet d'uploader une image. En mode édition, si une image est enregistrée elle sera affichée.

On modifié également la méthode `_save_` du contrôleur pour traiter cet upload :

ïïï java

```
public static void save(@Valid Album album, @Valid Artist artist, File cover) {
```

```
    if (Validation.hasErrors()) {
        render("@form", album);
    }
    album.artist = artist;
    album.replaceDuplicateArtist();
    album.save();
    if (cover != null) {
        String path = "/public/shared/covers/" + album.id;
        album.hasCover = true;
        File newFile = Play.getFile(path);
        if (newFile.exists())
            newFile.delete();
        cover.renameTo(newFile);
        album.save();
    }
    list();
}
```

ïïï

Comme vous pouvez le voir il suffit d'ajouter un paramètre de type `File` à la méthode `save` puis de le traiter avec les méthodes `Play.getFile` (pour déterminer le chemin de destination du fichier) et `renameTo`.

# Services Web

## REST et RESTful, qu'est ce que c'est?

On appelle service RESTful un service web respectant le style d'architecture REST.

REST (Representational State Transfer) est un modèle d'architecture orienté ressources. Ceci signifie qu'au lieu d'exposer des méthodes comme lorsque l'on utilise le protocole SOAP, on va exposer des ressources. Chaque ressource possède une URL qui l'identifie.

Contrairement à SOAP, REST s'appuie uniquement sur le protocole HTTP et ne propose aucune couche au dessus de ce protocole. Tout est faisable à partir des opérations fournies par de base par HTTP : GET, PUT, POST, DELETE, etc.

Pour récupérer une collection, on effectue un GET sur l'URL appropriée.

La réponse contiendra un ensemble d'éléments, décrits par exemple en BML ou en JSON. Pour chaque élément, une URL est définie. Il sera donc possible d'effectuer un appel GET sur un élément en particulier pour ne récupérer que celui ci. Une opération de type PUT sur le même élément permettra de mettre à jour ses données. De la même façon, une opération DELETE supprimera l'élément.

REST est en fait le modèle sur lequel le web lui même est construit : les sites et les pages web étant des ressources accessibles via des URL, depuis un navigateur grâce à des opérations HTTP.

Pour la sécurité il est possible de s'appuyer sur l'authentification HTTP, ou encore sur le SSL avec HTTPS. Comme vous pouvez le voir, tout est fait pour utiliser au maximum ce que le web nous fournit depuis toujours, sans sur-couche supplémentaire.

## Play et les services REST

Les URL de Play étant RESTful par essence, il devient très facile de créer une petite API REST/BML conjointement à l'interface Web d'une application Play!.

Voyons comment procéder.

## Exposer des données avec un service REST

Gardons l'exemple de notre bibliothèque musicale. Notre modèle comporte des albums, des artistes et des genres.

Pour rappel, la classe Album se présente comme ceci :

```
ïïï java
```

```
@Entity
```

```
public class Album extends Model {
```

```
public String name;
```

```
@ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
```

```
public Artist artist;
```

```
public Date releaseDate;
```

```
@Enumerated(EnumType.STRING)
```

```
public Genre genre;
```

```
ïïï
```

Nous voulons définir une URL qui renvoie lors d'un GET la liste des albums au format BML pour un genre donné.

Pour cela nous devons modifier le fichier routes :

```
GET /albums/{genre} Application.list
```

La première ligne correspond à la page HTML(non présentée dans cet article) affichant la liste des albums disponibles : le format n'étant pas spécifié, le rendu se

fera avec une page HTML.

Ici c'est la deuxième ligne qui nous intéresse. Le paramètre `(format:'xml')` indique que la méthode `render` du contrôleur devra chercher un fichier nommé listBml.xml.

Le paramètre {genre} sera récupéré dans l'URL et passé au contrôleur.

NB :

Il est possible d'utiliser une seule méthode dans le contrôleur si les paramètres requis et les traitements sont identiques pour les 2 types de rendus.

Dans notre cas il se peut qu'on ajoute des paramètres à la version HTML ultérieurement, sans vouloir impacter le rendu BML, par exemple :

GET /albums/{genre}/{first}/{count} Application.list

J'ai donc opté pour une séparation du rendu dans deux méthodes distinctes.

Le code de la méthode Application.listBml est le suivant :

``` java

public static void listBml(String genre) {

```
    Genre genreEnum = Genre.valueOf(genre.toString().toUpperCase());  
    List<Album> albums= Album.find("byGenre",genreEnum).fetch();  
    render(albums);
```

```

Je recherche simplement les albums correspondant au genre passé en paramètre, et je demande le rendu de la liste. Au passage on voit la simplicité d'utilisation de JPA avec Play! Le rendu sera fait dans le fichier portant le nom de la méthode et l'extension xml : listBml.xml.

Ce template, placé dans le repertoire app/views, est défini comme ceci :

î î î xml

```
<albums>
```

```
{list albums, as:'album'}
```

```
{/list}
```

```
    <album>
      <artist>${album.artist.name}</artist>
      <name>${album.name}</name>
      <release-date>${album.releaseDate.format('yyyy')}</release-date>
      <genre>${album.genre.toString()}</genre>
    </album>
î î î
```

Voilà, cela suffit pour exposer nos albums en BML. En respectant le pattern d'URL défini dans le fichier routes, par exemple en appelant `http://localhost:à000/albums/rock`, on obtient le résultat suivant :

î î î xml

```
<albums>
```

```
<album>
```

```
    <artist>Nirvana</artist>
    <name>Nevermind</name>
    <release-date>1991</release-date>
    <genre>ROCK</genre>
  </album>
  <artist>Muse</artist>
  <name>Origin of Symmetry</name>
  <release-date>2001</release-date>
  <genre>ROCK</genre>
</album>
  <artist>Muse</artist>
  <name>Black Holes and Revelations</name>
  <release-date>2006</release-date>
  <genre>ROCK</genre>
</albums>
```

î î î

## Envoi de données à travers un service REST

Dans la première partie nous avons vu comment exposer des données au format BML avec Play.

Maintenant nous allons effectuer l'opération inverse, l'envoi d'un contenu BML au contrôleur Play, à travers une UURL RESTful.

On veut par exemple envoyer le contenu suivant en POST avec un content type application/xml :

î î î xml

<album>

```
<artist>Metallica</artist>
<name>Death Magnetic</name>
<release-date>2008</release-date>
<genre>METAL</genre>
```

î î î

Pour cela on ajoute la ligne suivante au fichier routes pour autoriser l'opération POST sur l'url `/album` :

POST /api/album Application.saveBml

La méthode `saveBml` récupère le contenu de la requête dans la variable `request.body`.

Elle parse ensuite le contenu pour créer un album et l'enregistrer dans la base. La classe play.libs.BPath facilite le parcours de documents BML :

î î î java

```
public static void saveBML(){
```

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
Document document = null;
try{
    DocumentBuilder builder = factory.newDocumentBuilder();
    document = builder.parse(request.body);
}
catch(Exception e){
}
Element albumNode = document.getDocumentElement();
Node artistNode = BPath.selectNode("artist", albumNode);
```

```
String artistName = BPath.selectText("name",artistNode);
```

```
Artist artist = new Artist(artistName);
```

get the name

```
String albumName = BPath.selectText("name", albumNode);
```

```
Album album = new Album(albumName);
```

get the date

```
String date = BPath.selectText("release-date",albumNode);
```

```
DateFormat dateFormat = new SimpleDateFormat("yyyy");
```

```
try {
```

```
    album.releaseDate = dateFormat.parse(date);
```

```
} catch (ParseException e) {
```

```
    Logger.error(e.getMessage());
```

```
}
```

genre



```
String genre = BPath.selectText("genre", albumNode);

Genre genreEnum = Genre.valueOf(genre.toString().toUUpperCase());

album.genre = genreEnum;

sauvegarde

album.artist = artist;

album.save();

}
```

î î î

NB: il est bien sûr possible d'obtenir un code moins verbeux en dé-sérialisant l'objet à l'aide d'un outil comme JABB ou BStream, mais ce n'est pas l'objet de ce chapitre.

Lorsqu'on écrit le code `album.artist=artist`, la méthode `setArtist` est appelée automatiquement par Play (le code est modifié au runtime). On peut ainsi vérifier le fait que l'artiste existe ou non dans la base, pour savoir si on doit créer une nouvelle instance d'artiste ou récupérer l'artiste existant.

La méthode `save()` de la classe `Album` s'occupe alors d'enregistrer l'album en base, ainsi que l'artiste si il est inconnu dans la bibliothèque(à l'aide d'un cascade JPA).

î î î java

```
public void setArtist(Artist artist){

    List<Artist> existingArtists = Artist.find("byName", artist.name).fetch();
    if(existingArtists.size()>0){
        this.artist=existingArtists.get(0);
    }
    else{
        this.artist=artist;
    }
}
```

î î î

Notre API REST/BML nous permet donc maintenant de lire la liste des albums de notre bibliothèque musicale et d'ajouter des albums.

Vous pouvez tester l'envoi de contenu BML avec l'application rest-client(1) ou avec le plugin Poster de Firefox(2):

1: <http://code.google.com/p/rest-client/>

2: <https://addons.mozilla.org/de/firefox/addon/poster/>

## Services REST/JSON

Dans la première partie de ce chapitre, nous avons vu comment créer des services REST envoyant et consommant des messages au format BML.

Voyons maintenant comment faire la même chose avec JSON.

## Le format JSON

Définition de wikipedia :

`JSON (JavaScript Object Notation) est un format de données textuel, générique, dérivé de la notation des objets du langage ECMAScript. Il permet de représenter de l'information structurée.`

L'avantage de JSON par rapport à BML est d'être un peu moins verbeux et directement interprétable dans un navigateur à l'aide de JavaScript.

Si on écrit cette ligne dans le fichier routes :

```
GET /api/albums.json      Application.listAlbumsInJson
  @@ java
```

```
public static void listAlbumsInJson(){
```

```
List<Album> albums = Album.findAll();
renderJSON(albums);
}
}
```

L'appel de l'URL `http://monappli/albums.json` renverra directement notre liste d'objets albums au format JSON. Difficile de faire plus simple!

Autre astuce (que j'ai découvert grâce site [zengularity.com](http://zengularity.com)) : pour déterminer directement le format de données à partir de l'URL, il est possible d'utiliser cette syntaxe dans le fichier routes :

```
GET /api/albums.{<json&xml>format} Application.listAlbums
```

En appelant `/albums.xml`, Play appellera la méthode `listAlbums` avec le paramètre 'format' initialisé à 'xml', et en appelant `/albums.json` ce même paramètre aura la valeur 'json'.

On peut ensuite s'en servir dans le contrôleur :

```
}
}
```

```
public static void listAlbums() {
```

```
List<Album> albums = Album.all().fetch();
if(request.format.equals("json"))
    renderJSON(albums);
    render(albums);
}
}
```

Si vous tapez l'URL `/albums.xml`, Play cherchera un fichier de template BML nommé `listAlbums.xml` (une autre extension fonctionnerait aussi) pour effectuer le rendu.

## Recevoir un message JSON

Maintenant que nous savons exposer des données au format JSON à travers un

service REST, voyons comment envoyer des données au serveur en utilisant le même format.

Cette méthode du contrôleur permet de résoudre cette problématique :

```
ïïï java
```

```
public static void saveAlbumJson() {  
  
    Gson gson = new Gson();  
    Album album = gson.fromJson(new InputStreamReader(request.body), Album.  
class);  
    album.replaceDuplicateArtist();  
    album.save();  
ïïï
```

En récupérant l'objet `request.body`, on peut analyser le flux entrant et enregistrer un album dans la base de données.

Attention, pour que cette méthode fonctionne, il faudra respecter la structure de la classe Album lors de l'envoi des données en JSON.

Si on veut fournir un point d'entrée unique pour enregistrer de nouveaux albums en BML ou JSON, on peut écrire une méthode qui se chargera de rediriger vers le bon traitement selon le `_content type_` demandé dans la requête HTTP.

A la place de la ligne `POST /api/album Application.saveBml` dans le fichier routes, on écrit :

```
POST /api/album Application.saveByApi
```

On ajoute ensuite cette méthode dans le contrôleur :

```
ïïï java
```

```
public static void saveAlbumByApi() {  
  
    if (request.contentType.equalsIgnoreCase("application/xml"))
```

```
saveAlbumBml();
```

```
else if (request.contentType.equalsIgnoreCase("application/json"))
```

```
saveAlbumJson();
```

```
}
```

## Appeler un service externe avec Play.libs.WS

Play inclue également une librairie pour écrire des clients de services REST. Pour appeler un service externe, on peut écrire :

```
ïïï java
```

```
import play.libs.WS;
```

```
HttpResponse response = WS.url("http://api.twitter.com/1/statuses/user_timeline/  
loic_d.json").get();
```

```
ïïï
```

On peut ensuite récupérer le résultat comme ceci :

```
ïïï java
```

```
JsonElement json = response.toJson();
```

```
ïïï
```

Cette librairie est aussi compatible avec les services renvoyant du BML ou du texte brut :

```
ïïï java
```

```
String content = response.getString();
```

```
Document xml= response.getBml();
```

```
^^^
```

# Authentification et sécurité

Maintenant que nous savons comment développer une application Web avec Play, voyons comment gérer la sécurité et l'authentification à l'aide du module Secure.

Nous allons étudier le cas suivant : notre application est publique, on peut y naviguer sans être authentifié. Mais elle possède également des fonctions d'administrations, affichées lorsque l'on s'identifie comme admin. Pour accéder à ces fonctions, il existe une URL qui permet d'accéder à un formulaire d'authentification.

Play permet d'écrire les informations de session utilisateur dans un cookie. Ce cookie est signé, il n'est donc pas modifiable côté client, par contre il n'est pas crypté, il ne faut donc pas écrire d'informations sensible à l'intérieur (pas de mot de passe par exemple). Dans notre exemple, on souhaite utiliser le cookie de session pour stocker le fait que l'utilisateur soit identifié comme un administrateur ou non.

Une des choses que l'on souhaite ajouter à l'application web si l'utilisateur est admin est un lien "Supprimer" dans le tableau html qui liste nos entités métiers (on liste des albums de musique pour reprendre les exemples précédents). On peut donc utiliser le code suivant:

```
# {if session.get("username").equals("admin")}  
e {/if}
```

Mais on se retrouve vite confronté à un problème, un clic sur ce lien mène à une URL comme celle ci :

admin/delete?id=11

Même si le lien est masqué, n'importe qui peut entrer cette URL dans son browser pour supprimer l'entité de son choix. Nous devons donc aussi protéger la méthode delete côté serveur.

Le module Secure de Play va nous permettre de faire ça de manière élégante. Il propose également un formulaire de login prêt à l'emploi qui permet de mettre les informations dont on a besoin dans le cookie de session.

## Mise en oeuvre du module Secure

Pour activer le module secure, on commence par modifier le fichier dependencies.yml pour y ajouter la ligne suivante dans la section `require` :

```
play -> secure
e Import Secure routes
```

```
* / module:secure
```

Toujours dans ce fichier, on ajout les identifiants d'admin :

```
e Admin tokens
```

```
application.admin=admin
```

```
application.adminpwd=admin
```

On déclare ensuite un contrôleur d'administration pour toutes les actions que l'on veut restreindre. On ajoute l'annotation `@With` à ce contrôleur pour lui dire qu'il doit s'appuyer sur le contrôleur du module Secure :

```
@@@ java
```

```
@With(Secure.class)
```

```
public class Admin extends Controller {
```

```
}
```

```
@@@
```

On ajoute ensuite un contrôle sur l'action delete en utilisant l'annotation `@Check` :

```
@@@ java
```



```
Check("admin")
```

```
public static void delete(Long id) {
```

```
}
```

```
}}}
```

On redefinie également la méthode check en créant une nouvelle classe dans le package contrôler, héritant de la classe Secure.Security :

```
}}}
```

```
static boolean check(String profile) {
```

```
    if(profile.equals("admin"))  
        return session.get("username").equals("admin");  
    return false;
```

```
}}}
```

Ce code permet de demander au module Secure de vérifier que l'utilisateur en session est bien éadminè lorsque l'annotation @check(éadminè) est trouvée.

Dans la même classe, on redéfinie la méthode authentifier. C'est sur cette méthode que le formulaire d'authentification du module Secure s'appuie pour laisser passer ou non l'utilisateur :

```
}}}
```

```
static boolean authentifier(String username, String password) {
```

```
    return Play.configuration.getProperty("application.admin").  
equals(username)&& Play.configuration.getProperty("application.adminpwd").  
equals(password);
```

```
}}}
```

Avec cette configuration, si on essaie d'entrer l'UURL /admin/delete?id=11, on arrivera directement sur le formulaire d'authentification pour prouver que l'on est bien administrateur.

Et bien sur si le mot de passe et l'utilisateur entrés ne sont pas les bons, on ne passe pas.

On aimerait maintenant pouvoir aller directement sur ce formulaire pour mettre en session utilisateur les informations concernant notre identité.

Il suffit d'ajouter le code suivant dans le contrôleur Admin pour exposer le formulaire de login à l'UURL /admin/login :

```
ïïï java
```

```
public static void login() {
```

```
Application.list();
```

```
}
```

```
ïïï
```

Toutes les méthodes que l'on définit dans ce contrôleur étant soumises à un contrôle de l'utilisateur en session, vous vous retrouverez directement sur le formulaire d'authentification.

L'utilisateur sera ensuite redirigé vers l'écran principal de l'application (la liste des albums dans cet exemple).

Pour terminer, on souhaite permettre à un utilisateur identifié en tant qu'admin de se déconnecter.

Pour cela rien de plus simple, il suffit d'ajouter un lien au template main.html, dont toutes les pages héritent.

On ajoute le code suivant :

î î î html

<body>

```
# { if session.get( "username" ).equals( "admin" ) }  
  <div align="right">  
    <a href="@{Secure.logout()}">Logout</a>  
  </div>  
# { /if }  
</body>
```

î î î

Et voilà, vous savez maintenant comment ajouter des fonctions d'administration et de la sécurité à un site public avec Play!

## L'internationalisation

Dans le cas d'une application ou d'un site multilingue, on doit être capable de traduire facilement le contenu de nos pages.

Pour le message d'accueil de notre application, on peut par exemple écrire :

```
<h1>#{welcome}</h1>
```

Les paramètres entourés de `#{}` seront traduits à partir des clés définies dans les fichiers de configuration de Play.

Les clés pour la langue par défaut se trouvent dans le fichier `/conf/messages` :

```
welcome=Welcome on Vote4Music!
```

On peut ensuite définir un fichier par langue supplémentaire, par exemple `messages\_fr` pour le français.

Ce mécanisme peut être utilisé pour traduire toutes sorte de clés. On peut par

exemple afficher les valeurs de l'énum Genre dans notre application en modifiant la casse :

```
<h1>Top albums in &{genre} for ${year}</h1>
```

On renseigne ces clés dans le fichier `messages` :

ROCK=Rock

METAL=Metal

HIP\_HOP=Hip Hop

WORLD=World

POP=pop

JAZZ=Jazz

BLUUES=Blues

OTHER=Other

# Tester notre application

Play intègre un framework de tests permettant de lancer différents types de tests.

## Tests unitaires

Les tests unitaires testent une partie précise du code de notre application.

Voici un exemple de test unitaire :

``` java

```
public class CoreTest extends UUnitTest {

    @Test
    public void filterByYearTest() {
        List<Album> albums = new ArrayList<Album>();
        Album album1 = new Album("album1");
        Calendar c1 = Calendar.getInstance();
        c1.set(2010, 1, 1);
        album1.releaseDate= c1.getTime();
        albums.add(album1);
        Album album2 = new Album("album1");
        Calendar c2 = Calendar.getInstance();
        c2.set(2009, 1, 1);
        album2.releaseDate= c2.getTime();
        albums.add(album2);
        albums = Album.filterByYear(albums, "2010");
        assertTrue(albums.size()==1);
    }
}
```
```

Cette classe hérite de la classe UUnitTest fournie par Play. La méthode `filterByYearTest` permet de tester la méthode `filterByYear` de la classe Album.

## Tests fonctionnels

Les tests fonctionnels permettent de tester l'application à partir de son contrôleur en se basant sur le fichier `routes` pour résoudre les URL d'appel.

Ce test permet par exemple d'utiliser un service REST et valider la réponse obtenue :

```java

```
public class ApplicationTest extends FunctionalTest {

    @Before
    public void setUp() {
        Fixtures.deleteAll();
    }
    @Test
    public void testYML() {
        Response response = GET("/api/albums.xml");
        assertNotNull(response);
        String xmlTree = response.out.toString();
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        Document document = null;
        try {
            DocumentBuilder builder = factory.newDocumentBuilder();
            document = builder.parse(new ByteArrayInputStream(xmlTree.getBytes()));
        } catch (Exception e) {
            Logger.error(e.getMessage());
        }
        Element rootNode = document.getDocumentElement();
        assertTrue(rootNode.getElementsByTagName("album").getLength() == 2);
    }
}
```

```

La méthode setUp permet de réinitialiser les données avant chaque méthode de test.

Avec les méthodes GET et POST, on peut facilement tester le comportement de nos pages web.

On peut également vérifier le bon fonctionnement de nos services REST :

```java

@Test

```
public void testJsonApi() {
    assertFalse(artists.out.toString().contains("john"));
    assertFalse(albums.out.toString().contains("album1"));
    String album1 = "{ \"name\": \"album1\", \"artist\": { \"name\": \"john\" },
```

```
\\"releaseDate\\":\\"12 sept. 2010 00:00:00\\",\\"genre\\":\\"ROCK\\"}";
```

```
    POST("/api/album", "application/json", album1);
    artists = GET("/api/artists.json");
    assertTrue(artists.out.toString().contains("john"));
    assertTrue(albums.out.toString().contains("album1"));
}
public void testXmlApi() {
    Response artists = GET("/api/artists.xml");
    assertFalse(artists.out.toString().contains("john"));
    assertFalse(albums.out.toString().contains("album1"));
String album1 = "<album><artist><name>john</name></artist><name>album1</name><release-date>2010</release-date><genre>ROCK</genre><nvVotes>0</nvVotes></album>";
```

```
    POST("/api/album", "application/xml", album1);
    artists = GET("/api/artists.xml");

    assertTrue(artists.out.toString().contains("john"));
    assertTrue(albums.out.toString().contains("album1"));
}
```

## Tests Selenium

Ces tests permettent de simuler des clicks dans l'application à l'aide de l'outil Selenium.

Ce code permet de déclencher la création d'un album, puis de vérifier sa présence dans la liste des albums :

```
e {fixture delete:'all', load:'data.yml' /}
```

```
e {selenium}
```

Ouverture de la page d'accueil

```
open('/')
```

```
waitForPageToLoad(3000)
```

```
assertNotTitle('Application error')
```

Ouverture de la liste des albums

```
open('/albums')
```

```
waitForPageToLoad(3000)
```

```
assertTextPresent('coolAlbum')
```

Création d'un album

```
click('link=New album')
```

```
waitForPageToLoad('3000')
```

```
type('name', 'black album')
```

```
type('artist', 'metallica')
```

```
click('release-date')
```

```
type('release-date', '1àà0-01-01')
```

```
click('saveAlbum')
```

```
waitForPageToLoad('3000')
```

```
assertTextPresent('metallica')
```

```
e {/selenium}
```



La directive ``fixture delete:'all', load:'data.yml'`` vide la base de données puis charge le fichier ``data.yml``. Ce fichier se présente comme ceci :

Artist(joe) :

name: joe

Album(coolAlbum) :

name: coolAlbum

artist: joe

releaseDate: 2010-11-12 00:00:00

genre: ROCK

## Lancer les tests

Pour lancer les tests, entrez la commande suivante dans votre terminal : ``play test``

Puis tapez l'URL ``http://localhost/@tests`` dans votre navigateur.

Vous verrez apparaître cette page :

A partir de cet écran, vous pouvez lancer les tests et obtenir un rapport d'erreur (si il y en a)!

Plutôt pratique non?

N.B. : Si vous désirez connaître la couverture de tests de votre application, il existe un module Play pour ça!

Le module Cobertura(1) est capable de générer un rapport de couverture en analysant votre code. Quand le module est actif, il génère automatiquement ce rapport dans le répertoire ``test-result`` de l'application.

1: <http://www.playframework.org/modules/cobertura>

Pour installer ce module, ajoutez cette ligne au fichier ``dependencies.yml`` :

# Jobs et traitements asynchrones

## Les jobs

```
require:
  play -> cobertura 2.1
```

## Au démarrage de l'application

Le code suivant permet de charger un jeu de données au démarrage de l'application :

```
@@@ java
```

```
@OnApplicationStart
```

```
public class PopulateOnStart extends Job {

    public void doJob() {
        if(Album.count() == 0) {
            Fixtures.load("init-data.yml");
        }
    }
}
@@@
```

Pour que ça fonctionne il suffit de déposer le fichier init-data.yml dans le répertoire conf/

Voici un exemple de fichier yaml :

Artist(joe) :

```
name: joe
name: coolAlbum
artist: joe
releaseDate: 2010-11-12 00:00:00
genre: ROCK
```

## Effectuer des traitements périodiques

```
name: superAlbum
artist: joe
releaseDate: 2011-10-09 00:00:00
genre: ROCK
@@@ java
```

```
@Every("1h")
```

```
public class ReloadData extends Job {
```

```
    public void doJob() {  
        Fixtures.deleteAll();  
        Fixtures.load("data.yml");  
    }  
}
```

On peut imaginer beaucoup d'applications possibles pour ce genre de traitements périodiques. On pourrait par exemple envoyer un résumé d'activité par mail tous les lundi à l'ensemble des utilisateurs.

Pour définir finement la périodicité on peut utiliser la syntaxe CRON avec l'annotation `@On``. Par exemple, `@On("0 0 ( * * ?)")` déclenchera le traitement tous les jours à ( h.

## Traitements asynchrones : WebSockets

Il arrive que l'on doive effectuer des traitements longs dans une application web : génération d'un gros rapport PDF, contact d'un serveur distant pour obtenir des informations...

Lors d'un traitement long, on ne veut pas que le navigateur reste en suspension en attendant la réponse HTTP après qu'on ait lancé la requête. Il pourrait déclencher une erreur de timeout si le temps d'attente était trop long.

Pour résoudre cette problématique, on peut utiliser des traitements asynchrones. Le plus simple pour implémenter ce genre de fonctionnement avec Play est d'utiliser les WebSocket HTML5.

Ce procédé crée un mode de communication bidirectionnel entre le navigateur et le serveur. Dès que le serveur aura fini son action, il notifiera le navigateur sans que celui-ci soit obligé de garder une connexion HTTP ouverte pendant toute la durée

du traitement. Le serveur est alors capable de pousser une information au client dès qu'il en a besoin ("push" de données).

Un autre cas d'usage habituel pour les communications client-serveur bilatérales et asynchrones est l'implémentation d'applications de 'chat' : lorsqu'on envoie un message à un correspondant, on ne sait pas à quel moment on va recevoir une réponse. Il faut donc que le serveur soit capable de pousser un nouveau message vers notre client à n'importe quel moment.

Voyons comment implémenter un exemple très simple de communication asynchrone avec cette API :

## Déclaration de la WebSocket dans le navigateur

On crée une nouvelle page dans le dossier app/views. On l'appelle par exemple firstWebSocket.html.

On ajoute cette méthode au contrôleur principal de notre application :

```java

```
public static void firstWebSocket() {
```

```
    render();  
}
```

Ce code javascript utilise l'API WebSocket HTML5 pour ouvrir la communication entre le navigateur et le serveur :

```javascript

```
var socket = new WebSocket('@@{AsyncController.asyncMessage()}');
```

```
socket.onmessage = function(event) {  
    display(event.data);  
}  
var display = function(event) {  
}
```

Voici le code complet de notre page :

î î î html

```
{extends 'main.html' /}
```

```
<h1>Test</h1>
```

```
<div id="message"></div>
```

```
{set 'moreScripts'}
```

```
<script>
```

```
    var socket = new WebSocket('@@{AsyncController.asyncMessage()}');
    socket.onmessage = function(event) {
        display(event.data);
    }
    $('#message').append('' + event + '');
}
```

```
{/set}
```

î î î

Lorsqu'un message est reçu, il est affiché dans la div `e message`

## Implémentation côté serveur

On crée l'objet liveStream qui sera une sorte de file d'attente de messages. Dès que la file reçoit un message, l'objet outboud (hérité de la classe `WebSocketController`) est invoqué pour envoyer ce message au client :

î î î java

```
public class AsyncController extends WebSocketController {
```

```
    public static EventStream<String> liveStream = new EventStream<String>();
    public static void asyncMessage() {
        while (inbound.isOpen()) {
            String message = await(liveStream.nextEvent());
            if (message != null) {
                outbound.send(message);
            }
        }
    }
}
```

On met à jour les routes pour spécifier l'utilisation du protocole `WS` (WebSocket) au lieu de `HTTP` :

```
WS          /asyncTest
AsyncController.asyncMessage
```

## Push de données depuis le serveur

Voici le code de notre traitement long (on simule une longue durée avec un sleep):

ïïï java

```
public static void publishEvent(String message) throws IOException {
```

```
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        Logger.error(e.getMessage());
    }
    AsyncController.liveStream.publish(message);
}
```

On ajoute cette méthode au contrôleur principal de notre application.

Une fois le traitement terminé, on publie un message dans notre file d'attente.

## Mise en action

Pour tester le fonctionnement de notre WebSocket, on ouvre la page `http://localhost:à000/Application/firstWebSocket`

Pour l'instant la zone de messages est vide.

Dans une console, on peut envoyer un message à l'aide du client HTTP cURL :

```
curl -d message=coucou http://localhost:à000/Application/publishEvent
```

Ceci va déclencher l'action de "push" que l'on a écrit. Si on revient 5 secondes après sur notre page Web, le message "coucou" est apparu!

On voit qu'il est tout à fait possible de pousser des données vers une page, même si

celle ci n'a effectué aucune requête. Pas mal non?

Ce mode de fonctionnement peut être utile par exemple dans le cas où on lance des traitements programmés à l'avance à l'aide de jobs (voir le chapitre 3 de cette partie).

Imaginons un job qui mette à jour les données de l'application une fois par jour à heure fixe. A la fin de ce traitement, on pourrait notifier le client que les données ont été modifiées.

Mais il est également possible de lancer l'action depuis la page courante. On ajoute ce lien :

{{{ html

```
<a id="longTask" href="#">Long task</a>
```

}}

Un clic sur ce lien lance la tâche asynchrone :

{{{ javascript

```
$(document).ready(function() {  
  
    $('#longTask').click(  
        function() {  
            $.post('@{Application.publishEvent()}', { message: 'Ok it  
works!!! ' } );  
        }  
    );  
});
```



# Découverte des modules

Il existe un grand nombre de modules complémentaires pour ajouter des fonctionnalités à Play. Nous allons en voir quelques exemples.

## Validation côté client avec HTML5

La spécification HTML 5 prévoit la possibilité de valider les données d'un formulaire HTML côté client, directement dans le navigateur avant d'envoyer les données vers un serveur.

Il existe un module pour Play qui permet de faire un mapping entre les annotations de validation du modèle (qui servent normalement à valider les données côté serveur) et le rendu HTML, pour intégrer cette fonctionnalité.

Pour activer ce module, après l'avoir téléchargé il suffit d'ajouter cette ligne dans le fichier `dependencies.yml`, dans la section `require``:

```
play -> html5validation 1.2
{{{ java
```

```
@Entity
```

```
public class Album extends Model {
```

```
    @Required
    public String name;
{{{
```

Dans le formulaire HTML, on peut utiliser un nouveau tag, `e {input}` :

```
{{{ html
```

```
e {input for:'album.name', type:'text', id:'name' /}
```

```
{{{
```

Ce tag sera traduit en une balise input classique, avec un attribut indiquant que le champ est obligatoire :

!!! html

```
<input type="text" name="album.name" value="album?.name" id="name"
required="required"/>
```

Le rendu est le suivant si on valide le formulaire sans remplir le champ obligatoire :



Album Name:

Artist:

Veuillez compléter ce champ.

Le tag input supporte un grand nombre d'options et plusieurs types d'annotations de validation, comme @Match pour valider une expression régulière ou @Email.

Toutes ces options sont décrites dans cette documentation(1).

1: <http://www.playframework.org/modules/html5validation-1.0/home>

Si votre navigateur ne supporte pas la validation HTML5, aucun soucis car la validation côté serveur sera exécutée dans tous les cas. J'ai testé avec Chrome 10 et Firefox 4 beta 12 et cela fonctionne parfaitement sur ces navigateurs.

Home page du module HTML5 Validation(1)

1: <http://www.playframework.org/modules/html5validation>

## Elastic Search

Elastic Search est un framework construit au dessus de Lucene(1).

1: <http://lucene.apache.org/java/docs/index.html>

Il offre la possibilité d'effectuer des recherches "à la google" sur nos entités métier. Pour que cela fonctionne il suffit de les annoter avec `@ElasticSearchable`.

Le moteur Lucene permet par exemple de faire des recherches :

tolérantes aux fautes de frappes ou d'orthographe

basées sur un dictionnaire de synonymes (on peut taper indifféremment 'rue' ou 'avenue' pour rechercher une adresse)

basées sur la prononciation des mots (recherche phonétique)

Elastic Search est basé sur une architecture REST et est capable d'indexer du contenu sous plusieurs formes, notamment à partir de flux JSON. Il offre une grande souplesse d'utilisation car il ne demande de respecter un schéma pour les données, contrairement à une base de données relationnelle. En mode production il est capable de fonctionner en multi-instances. L'index est réparti sur plusieurs noeuds, qui peuvent être répliqués pour résister aux pannes. Ce genre d'architecture est particulièrement adapté aux environnements cloud et permet de répondre à de fortes charges et de grosses volumétries sans sacrifier les performances.

Avec l'API Java fournie par Elastic Search, on peut écrire ce genre de requêtes :

```java

```
QueryBuilder qb = filteredQuery(  
    termQuery("name", name),  
    rangeFilter("nbVotes")  
        .from(100)  
        .to(90)  
);
```

Mais il n'est pas nécessaire de maîtriser l'API Elastic Search pour profiter de ce module : celui ci propose également un mode inspiré du module CRUUD. En héritant de la classe `ElasticSearchController` et en utilisant l'annotation du même nom pour indiquer le type d'entité à rechercher, on peut générer tous le code et les

écrans nécessaires pour la création et la recherche de nos entités :

î î î java

```
@ElasticSearchController.For(Album.class)
```

```
public class AlbumSearch extends ElasticSearchController {  
  
}
```

î î î

Si vous souhaitez conserver le comportement par défaut du module, rien à ajouter dans cette classe! Mais comme pour le module CRUUD vous pouvez surcharger ce comportement si vous le désirez.

On peut également surcharger les vues et créer de nouveaux templates en créant un répertoire ELASTIC\_SEARCH sous `app/views` dans l'arborescence de notre application.

Vous pouvez voir ce module en action dans cette vidéo(1)

1: [http://www.youtube.com/watch?v=pHpvNKO1mTE&feature=player\\_detailpage](http://www.youtube.com/watch?v=pHpvNKO1mTE&feature=player_detailpage)

Home page du module Elastic Search(1)

1: <http://www.playframework.org/modules/elasticsearch-0.0.3/home>

## Et plein d'autres modules!

Il existe un tas d'autres modules et de nouveaux arrivent fréquemment grâce à la communauté des développeurs Play.

On trouve par exemple :

GAE(1), pour déployer son application sur Google APP Engine, le cloud de Google

1: <http://www.playframework.org/modules/gae>

Morphia(1), pour utiliser très simplement la base de données NoSQL MongoDB (base orientée documents) avec Play

1: <http://www.playframework.org/modules/morphia>

Tabula Rasa(1), pour intégrer le plugin DataTables de JQuery au CRUUD de Play

1: <http://www.playframework.org/modules/tabularasa-0.2/home>

OAuth(1), pour authentifier des utilisateurs auprès de fournisseurs d'identité OAuth, comme Twitter ou Google

1: <http://www.playframework.org/modules/oauth>

Pdf(1), pour générer des documents... PDF!

1: <http://www.playframework.org/modules/pdf>

Akka(1), pour faciliter la programmation concurrente, en utilisant le modèle des acteurs

1: <http://www.playframework.org/modules/akka>

Et beaucoup d'autres...

La liste complète des modules disponibles est consultable ici(1).

1: <http://www.playframework.org/modules>

# Play et Scala

Play propose d'utiliser au choix les langages Scala ou Java pour développer des applications.

Play-Scala est plus qu'un simple module, c'est une version complète du framework dédiée à ce langage. Cette version possède un site dédié qui présente ses avantages et ses fonctionnalités : [scala.playframework.org](http://scala.playframework.org/)(1)

1: <http://scala.playframework.org/>

Le code de l'application vote4music est disponible dans sa version Scala ici(1)

1: <https://github.com/loicdescotte/vote4music-scala>

## Le langage Scala

Scala est un langage pour la machine virtuelle Java (JVM) qui marie les caractéristiques des langages orientés objet et des langage fonctionnels.

C'est un langage très différent de Java, il demande donc un temps d'adaptation pour les développeurs Java. Cependant, nous allons voir à travers quelques exemples que c'est un langage très intéressant qui peut nous aider à améliorer sensiblement la lisibilité et l'expressivité du code, grâce à l'approche fonctionnelle.

## Quelques exemples de code

On peut facilement filtrer une collection avec la fonction filter :

```
/// java
```

```
var numbers = Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Récupération des nombres pairs

```
var evenNumbers = numbers.filter(x => x % 2 == 0)
```

^^^

On peut même simplifier l'écriture de cette fonction avec le caractère joker '\_' :

^^^ java

```
var evenNumbers = numbers.filter(_ % 2 == 0)
```

^^^

Cette fonction nous sera utile dans l'application vote4music, notamment pour trier les albums par année :

^^^ java

```
albums = albums.filter(x => formatYear.format(x.releaseDate).equals(year))
```

^^^

Pour tirer des albums en fonction du nombre de votes, dans le sens décroissant, on peut écrire :

^^^ java

```
albums.sortBy(_._nbVotes).reverse
```

^^^

Pour trouver l'intervalle compris entre 2 entiers, ici le plus ancien album et le plus récent, il suffit d'écrire :

^^^ java

```
val first = Albums.firstAlbumYear
```

```
val last = Albums.lastAlbumYear
```

Utilisation de la fonction to

```
years = first.to(last).toList
```

^^^

En Java, ces exemples auraient nécessité de passer par d'horribles boucles for (ou par l'utilisation de bibliothèques comme Guava ou lambdaJ).

Voyons maintenant quelles sont les spécificités de Play-Scala et comment porter entièrement vote4music avec ce nouveau langage.

## Accès à la base de données avec l'API Anorm

Play-Scala intègre une API qui permet d'effectuer très facilement des requêtes SQL et de mapper les résultats dans des objets Scala.

### Définition d'une entité

Une entité est une classe du modèle :

^^^ java

```
case class Artist(var id:Pk[Long], @Required var name: String){  
  
}
```

^^^

Le mot clé `case` en Scala ajoute des facilités pour faire du pattern matching sur les classes et leurs attributs. Ces facilités seront exploitées par Anorm lors de l'exécution des requêtes.

N.B : Comme vous le voyez nous pouvons continuer à utiliser les annotations de



validation du modèle, comme `@Required`.

## La classe Magic

En créant un objet Scala qui hérite de la classe `Magic`, on obtient des méthodes pour manipuler les objets dans la base.

Si on déclare :

```
``` java
```

```
object Album extends Magic[Album]
```

```
```
```

On peut par exemple écrire :

```
``` java
```

récupération du premier élément

```
Album.find().first()
```

recherche par genre

```
Album.find("genre = {g}").on("g" -> "ROCK").list()
```

insertion

```
Album.create()
```

mise à jour

```
Album.update()
```

```
```
```

## Requêtes plus complexes

On a souvent besoin de ramener plus d'un type d'objet à la fois. Cette méthode permet de récupérer tous les albums et les artistes dans la base de données :

```java

```
def findAll:List[(Album,Artist)] =
```

```
    SQL(
        """
        select * from Album al
        join Artist ar on al.artist_id = ar.id
        order by al.nbVotes desc
        limit 100;
        """
    ).as( Album ~< Artist ^^ flatten * )
```

La dernière ligne utilise des 'matchers' définis par le framework pour mapper les résultats vers nos objets du modèle et les ranger dans une liste de paires d'éléments (un album et son artiste).

Pour effectuer une recherche, on écrit la requête suivante :

```java

```
def search(filter: String):List[(Album,Artist)] = {
```

```
    val likeFilter = "%".concat(filter).concat("%")
    SQL(
        """
        select * from Album al
        join Artist ar on al.artist_id = ar.id
        where al.name like {n}
        or ar.name like {n}
        order by al.nbVotes desc
        limit 100;
        """
    ).on("n"->likeFilter)
    as( Album ~< Artist ^^ flatten * )
```

```

La suppression des albums se déroule comme ceci :

```
def delete(id: Option[Long]) = {
```

```
id.map(id => Album.delete("id={c}").onParams(id).executeUpdate())  
Action(Application.list)
```

Option est un type Scala qui permet d'éviter les erreurs liées aux pointeurs nuls (NullPointerException). Quand on récupère un objet de type Option, il peut avoir la valeur `Some` ou `None`. La méthode `map` appliquée à cette option nous permet de traiter les résultats différents de `None`.

La méthode `Action` permet d'effectuer une redirection vers une autre action du contrôleur.

## Le moteur de template

Play-Scala propose un moteur de template full Scala. Ce moteur permet d'écrire du code Scala autour de notre code HTML pour définir l'affichage des pages :

```
@@@ html
```

```
@(messages:List[String])
```

```
<ul>
```

```
@messages.map{ message =>
```

```
<li>@message</li>
```

```
}
```

```
</ul>
```

```
@@@
```

Ce template prend en paramètre une liste de messages.

La méthode map permet ici de parcourir la liste des éléments.

Pour effectuer le rendu d'un template, on appelle une méthode qui porte le même nom que le fichier HTML contenant ce template :

@@@ java

```
def index = html.index(messages)
```

@@@

Ces méthodes sont générées automatiquement par le compilateur.

## Tags et layouts

Comme dans tout moteur de template qui se respecte, il est possible de créer des tags réutilisables pour factoriser du code.

Par exemple, pour définir un tag qui affiche les informations concernant une personne, on crée un fichier `persons.scala.html` :

@@@ html

```
@(person:Person)
```

```
<span>Name: </span> <span>@person.name</span>
```

```
<span>Address: </span> <span>@person.address</span>
```

```
<span>Phone Number: </span> <span>@person.phoneNumber</span>
```

@@@

On peut ensuite appeler ce tag depuis une autre page :

@@@ html

```
@persons(person=p)
```

î î î

On utilise également ce procédé pour créer des layouts.

Pour cela, on crée un fichier `layout.scala.html` :

î î î html

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

```
<h1>@title</h1>
```

```
<div id="menu">
```

```
</div>
```

```
<div id="content">
```

```
  @content
```

```
<div id="footer"> Mon super site</div>
```

î î î

Pour créer une nouvelle page qui utilise ce layout, on procède comme ceci :

î î î html

```
@main(title = "maPage") {
```

^^^

Le titre sera remplacé par "ma page" et le contenu prendra place dans la balise `content`.

## Un exemple concret

Pour mettre à jour un album, on récupère l'album et l'artiste depuis la base de données puis on les transmet au template approprié :

^^^ java

```
def form(id: Option[Long]) = {  
  
    val album = id.flatMap( id => Album.find("id={id}").onParams(id).first())  
    val artist = album.flatMap( album => Artist.find("id={id}").  
onParams(album.artist_id).first())  
    html.edit(album, artist)  
}
```

^^^

On utilise ici flatMap pour récupérer un album ou un artiste à partir de son id. Si on avait utilisé map, on aurait récupéré une option contenant le résultat de la fonction passée en paramètre. flatMap permet de récupérer directement la valeur retournée (dans les cas différents de None) au lieu d'une option.

Pour parcourir une liste de résultats, par exemple un objet de type List[Album, Artist], on procède comme ceci :

^^^ java

```
def list() = {  
  
    html.list(Album.findAll)  
}
```

^^^

^^^ html

```
@(entries:List[(models.Album,models.Artist)])
```

```
@import controllers._
```

```
<table id="albumList">
```

```
  <thead>
    <tr>
      <th>Album</th>
      <th>Artist</th>
      <th>Cover</th>
      <th>Release date</th>
      <th>Genre</th>
      <th>Number of votes</th>
    </tr>
  </thead>
  @entries.map { entry =>
    <tr id="album-@entry._1.id">
      <td>@entry._1.name</td>
      <td>@entry._2.name</td>
      <td>
        @if(entry._1.hasCover){
          <span class="cover"><a href="#">Show cover</a></span>
        }
      </td>
      <td>@Option(entry._1).map(_.releaseDate.format("yyyy-MM-dd"))</td>
      <td>@entry._1.genre</td>
      <td>
        <span id="nbVotes@entry._1.id">@entry._1.nbVotes</span>
      </td>
    </tr>
  }
  {{{
```

Le template définit un paramètre `entries` qui correspond à la liste des tuples d'albums et d'artistes renvoyée par `Album.findAll`.

A l'intérieur d'un tuple, on accède à un album avec l'expression `entry.\_1` et à un artiste avec `entry.\_2`.

## Authentification à l'aide des traits

Les traits sont une variante plus puissante des interfaces et classes abstraites de Java. Ils permettent aussi d'implémenter facilement le principe du design pattern "décorateur" dont le but est d'ajouter dynamiquement des comportements à des

objets.

Nous allons voir comment les utiliser pour écrire un mécanisme d'authentification simple.

Ce sera l'équivalent du module secure que nous avons utilisé dans la version Java. Ce mécanisme permettra d'accéder aux fonctions d'administration de l'application.

On définit le trait Secure, qui hérite de `Controller`. Ce trait définit le comportement suivant : avant chaque action (grâce à l'annotation `@Before`), on vérifie qu'il existe un utilisateur dans la session (le cookie stocké dans le navigateur).

Si un nom d'utilisateur est disponible on le stocke dans la variable `connectedUser` :

``` java

```
trait Secure extends Controller {
```

```
@Before def check = {
```

```
    session("username") match {  
        case Some(username) => Continue  
        case None => Action(Authentication.login)  
    }
```

```
def connectedUser = session.get("username").asInstanceOf[String]
```

```
}
```

```

On définit ensuite un deuxième trait qui hérite du comportement de `Secure` et qui vérifie que l'utilisateur qui a ouvert une session est bien l'admin :

``` java

```
trait AdminOnly extends Secure {
```

```
@Before def checkAdmin = {
```



```

    if (!connectedUser.equals("admin")) Forbidden else Continue
}

```

î î î

On peut maintenant écrire le contrôleur qui nous permettra d'identifier l'administrateur et de mettre son login dans la session si ses identifiants sont corrects :

î î î java

```

object Authentication extends Controller {

```

```

import views.Authentication._

```

```

def login = {

```

```

    html.login()

```

```

def logout() = {

```

```

    session.clear()

```

```

    Action(Admin.login)

```

```

def authenticate() = {

```

```

    val username = params.get("username")

```

```

    val password = params.get("password")

```

```

    Play.configuration.getProperty("application.admin").equals(username) &&

```

```

    Play.configuration.getProperty("application.adminpwd").equals(password)

```

```

match {

```

```

    case true => session.put("username", username)

```

```

                Action(Application.index)

```

```

    case false => flash.error(Messages.get("error.login"))

```

```

                html.login()

```

```

    }

```

î î î

Le template `login.scala.html` est un formulaire très simple qui transmet le login et le mot de passe saisis à l'action `authenticate` :

î î î html

```

@form(controllers.Authentication.authenticate()){

    @if(flash.get("error")){
        <p class="error">
            @flash.get("error")
        </p>
    }
    <p id="username-field">
        <label for="username">User name</label>
        <input type="text" name="username" id="username"/>
    </p>
    <p id="password-field">
        <label for="password">Password</label>
        <input type="password" name="password" id="password" value="" />
    </p>
    <p id="signin-field">
        <input type="submit" id="signin" value="Sign in" />
    </p>
}
}
}

```

N.B : le scope `flash` est utilisé lorsque l'on veut récupérer une information dans la requête suivant le traitement. On s'en sert ici pour afficher le message d'erreur si les identifiants saisis sont invalides.

Enfin, on peut écrire le contrôleur `Admin` qui contiendra les méthodes pour lesquelles le rôle d'administrateur est nécessaire :

```

}
}
}

```

```

object Admin extends Controller with AdminOnly {

```

```

import views.Application._

```

```

def delete(id: Option[Long]) = {

```

```

    id.map(id => Album.delete("id={c}").onParams(id).executeUpdate())
    Action(Application.list)

```

```

}

```

```

}
}
}

```

Grâce au mot clé `with` ce contrôleur hérite des comportements définis dans nos traits.

## Les tests

Le framework de test de Play-Scala est un bon d'exemple des avantages de ce langage. La syntaxe offerte par Scala donne des tests vraiment expressifs et simples à lire :

```java

```
test("collections") {  
  
    var albums=Albums.findAll()  
    (albums.size) should be (2)  
    var artists=Artists.findAll()  
    (artists.size) should be (1)  
    Artist artist = artists(1)  
    artist.name should include ("Joe")  
}
```

```

Il est également possible d'écrire des tests dans le style BDD (Behavior Driven Developpment), en combinant le code des tests avec du texte représentant les comportements attendus :

```java

```
val name = "Play.Rules"
```

```
"Play.Rules" should "not contain the B letter" in {
```

```
    name should not include ("X")  
    it should "have 10 chars" in {
```

```
        name should have length (10)  
    }
```

```

Si on applique ça à notre application, on peut par exemple écrire ce test :

^^^ java

it should "return right albums with years" in {

```
val artist = new Artist("joe")
val c1 = Calendar.getInstance()
c1.set(2010,1,1)
val album = new Album("album1", c1.getTime, "ROCK", false)
val c2 = Calendar.getInstance()
c2.set(2009,1,1)
val album2 = new Album("album2", c2.getTime, "ROCK", false)
val albums = List(new Tuple2(album,artist),new Tuple2(album2,artist))
val filteredAlbums = Album.filterByYear(albums, "2010")
filteredAlbums.size should be (1)
}
```

NB : un Tuple2 est une paire d'éléments en Scala

## Installer le module Scala

Pour installer ce module, il suffit d'ajouter cette ligne dans le fichier `dependencies.yml`, dans la partie `require` : `play -> scala 0.à.1`.

Ca y'est vous êtes armés pour développer en Scala!

## Apprendre Scala

Si vous désirez apprendre ce langage, il existe un e-book gratuit(1) (en anglais)

1: <http://programming-scala.labs.oreilly.com/index.html>

Home page du module Scala(1)

1: <http://scala.playframework.org/>

# Créer son propre module

Maintenant que nous avons vu un certain nombre de modules existants, voyons comment créer un module personnel pour répondre à nos propres besoins!

## Exemple 1 : Créer un tag personnalisé

Dans le chapitre précédent, je vous ai présenté le module HTML5Validation, qui ajoute la validation des données côté navigateur à l'aide d'un tag, en se basant sur les annotations du modèle.

Nous allons maintenant étudier le code de ce module pour comprendre comment réaliser ce genre de tags.

### Structure du module

La structure d'un module ressemble à celle d'une application Play. Dans ce module nous trouvons les répertoires suivants :

- \* app

- \* app/tags

- \* documentation

Si le module avait défini des vues et des contrôleurs on les aurait retrouvés dans le répertoire app comme dans une application Play classique. Dans le cas de ce module on ne trouve qu'un tag, que l'on va étudier.

La classe `HTMLValidationTags` étend `FastTags`. Cette dernière permet de créer rapidement un tag en décrivant simplement son comportement en Java.

Le tag comporte une seule méthode publique, qui sera appelée pour effectuer le rendu :

```
@@@ java
```

```
public static void _input(final Map<?, ?> args, final Closure body, final PrintWriter
out,
```

```
        final ExecutableTemplate template, final int fromLine) {
}
```

î î î

Les paramètres `args` et `body` correspondent au contenu du tag dans la vue HTML. Dans notre cas, le tag est toujours fermé à la fin de sa déclaration, il n'y a donc pas de body.

Exemple : `e {input for:'user.name', id:'6 ourID', class:'class1 class2' /}`

Le paramètre `out` permet d'écrire dans la sortie HTML. `FromLine` sert à spécifier la section du template dans laquelle on exécute le code. On s'en sert par exemple pour préciser la ligne d'erreur lors de la levée d'une exception.

Le nom du tag est défini par le nom de la méthode, sans le underscore.

Le corps de la méthode écrit dans le flux sortie HTML en fonction des paramètres d'entrée :

î î î java

```
out.print("<input");
```

Print standard attributes

```
printStandardAttributes(args, out);
```

Print validation attributes

```
printValidationAttributes(args, out);
```

Close input tag

```
out.println(">");
```

^^^

Si on a un modèle comme ceci :

^^^ java

```
@Required
```

```
@Match("[a-z]*")
```

```
public String name;
```

^^^

Alors le template `{input for:'user.name', id:'6 ourID', class:'class1 class2' /}` rendra le code HTML suivant : `<input name="user.name" value="{user?.name}" id="6 ourID" class="class1 class2" required pattern="[a-z]*">`

Ce code permet de déterminer le champ que l'on est entrain de manipuler :

^^^ java

```
final String fieldname = args.get("for").toString();
```

```
final String[] components = fieldname.split("\\\\.");
Class<?> clazz = null;
for (final Class<?> current : Play.classloader.getAllClasses()) {
    if (current.getSimpleName().equalsIgnoreCase(components[0])) {
        clazz = current;
    }
}
final Field field = clazz.getField(components[1]);
```

Et voici le code utilisé pour détecter et traiter l'annotation `@Required` :

^^^ java

```
if (field.isAnnotationPresent(Required.class)) {
```

```
        printAttribute("required", "required", out);  
    }  
}
```

Pour `@Match` :

``` java

```
if (field.isAnnotationPresent(Match.class)) {  
  
    final Match match = field.getAnnotation(Match.class);  
    printAttribute("pattern", match.value(), out);  
}
```

N.B. : Vous pouvez voir le détail des méthodes `printAttribute`,  
`printStandardAttributes` et `printValidationAttributes` dans le code du module(1).

1: <https://github.com/oasits/play-html5-validation>

## Exemple 2 : Amélioration du module CRUUD avec JQuery UI

Il y a quelques temps, la team jQuery UI a annoncé(1) qu'ils avaient commencé à sérieusement et officiellement travailler sur le widget Grid de la librairie. Le développement prend place au sein de la branche grid du repository github, le répertoire nous intéressant le plus étant grid-datamodel(2).

1: <http://blog.jqueryui.com/2011/02/unleash-the-grid/>

2: <https://github.com/jquery/jquery-ui/tree/grid/grid-datamodel>

Dans cette partie, nous nous intéresserons à l'implémentation grid-datamodel et son intégration dans le module CRUUD de Play.

Nous nous concentrerons à configurer le widget UI Grid et adapter légèrement le



CRUUD généré par Play pour permettre l'utilisation du widget en mode BHR (Ajax).

## Le plan

Le module CRUUD de Play comporte un ensemble de fichier de templates (views/tags) permettant de gérer et afficher les données du modèle. Dans l'exemple que nous nous apprêtons à mettre en place ici, cela signifie générer une table à partir des données de notre modèle. Le module CRUUD par défaut utilise des paramètres pour permettre pagination, recherche/filtre et tri. Play génère alors la table correspondante pour une "page" unique. Ainsi, même si votre modèle comporte des milliers d'objets, Play générera la table correspondante avec seulement une vingtaine de ligne (configurable).

Dans le cadre de ce tutoriel, les étapes que l'on devra mettre en place se résumeront à:

1. Configuration d'une application exemple, le but étant de définir des données avec lesquelles travailler.
2. Création de nos contrôleurs et modification des templates utilisé par le module CRUUD. Cette étape nous permettra de fournir un service dont le retour est une réponse JSON représentant les données de notre modèle.
3. La création d'un module Play très simple dont le seul but est de contenir les assets (fichiers statiques) nécessaire au widget grid et de les rendre disponible au reste de l'application via l'utilisation d'une route particulière ``/grid/``
4. Configuration du widget grid et du datasource pour utiliser le service fourni par le module Play.

Le code de cet exemple est disponible ici :

```
git clone git://github.com/mklabs/play-ui-grid.git
```

## Mise en place de l'application exemple

Nous utiliserons un exemple de gestion des fuseaux horaires (timezone locale) pour jouer avec un widget grid. Cela nous permettra d'avoir facilement plusieurs milliers

d'enregistrement avec lesquels travailler.

Nous partons d'une application Play vierge pour laquelle nous activerons le module CRUUD (voir partie 0 du livre).

Nous allons désormais nous occuper de la création de notre modèle. Dans notre exemple, il s'agit de timezones, le modèle est simple: timeZoneId, name, language et un offset.

^^^ java

```
@Entity
public class LocalisedTimeZone extends Model {
    public String timeZoneId;
    public String name;
    public String language;
    public int offset;
    public LocalisedTimeZone(TimeZone zone, Locale locale) {
        this.timeZoneId = zone.getID();
        this.name = zone.getDisplayName(locale);
        this.language = locale.getDisplayLanguage();
        this.offset = zone.getRawOffset() / 3600000;
    }
}
```

Vient ensuite la dernière partie de la mise en place de notre exemple d'application avec la définition d'un Job pour charger les données au démarrage :

```
~~~ java
public class Bootstrap extends Job {
    @Override
    public void doJob() {
        if (LocalisedTimeZone.count() == 0) {
            for (String id : TimeZone.getAvailableIDs()) {
                final TimeZone zone = TimeZone.getTimeZone(id);
                new LocalisedTimeZone(zone, Locale.ENGLISH).save();
                new LocalisedTimeZone(zone, Locale.FRENCH).save();
                new LocalisedTimeZone(zone, new Locale("nl")).save();
            }
        }
    }
}
```

Okay, notre modèle est prêt à être utilisé. Ils nous manque encore le contrôleur CRUUD pour afficher le tout.

## Contrôleurs, et modifications des vues du module CRUUD

## Contrôleurs

Penchons nous désormais sur le code du contrôleur, la partie de l'application qui permet d'offrir à nos vue les données du modèle sous format JSON (nous voulons faire de UUI Grid un consommateur de ce "service").

^^^ java

```
public class CrudJson extends CRUUD {

    public static void listJson(int page, String search, String
searchFields, String orderBy, String order) {
        ObjectType type = ObjectType.get(getControllerClass());
        notFoundIfNull(type);
        if (page < 1) {
            page = 1;
        }
        final List<Model> objects = type.findPage(
            page,
            search,
            searchFields,
            orderBy,
            order,
            (String) request.args.get("where")
        );
        renderJSON(objects);
    }
}
^^^
```

UUI Grid attend simplement un tableau d'objets. Parfait, c'est exactement le format JSON renvoyé par `renderJSON`.

Cette classe `CrudJson` est conçue pour être étendue par les véritables contrôleurs de notre application. Dans cette exemple, il s'agira de `LocalisedTimeZone`.

^^^ java

```
public class LocalisedTimeZones extends CrudJson {}
```

^^^

UUne ligne... Je sais pas vous, mais il me plaît beaucoup ce contrôleur!

Il nous reste une étape à ne pas oublier avec la configuration des routes des contrôleurs. Ici, nous ne disposons que d'un seul contrôleur. Aussi, nous pourrions nous contenter de:

```
GET /localeOrWhateverName.json LocalisedTimeZone.listJson
```

Dans la plupart des cas, cela suffirait à répondre à nos besoin. Cela dit, dans la pratique, les applications ne disposant que d'un modèle/contrôleur ne sont pas légion, et pour chaque objet de notre modèle, une route serait nécessaire. Ceci étant, il existe également la méthode suivante, se reposant sur la convention de nommage de vos contrôleurs et permettant une approche un peu plus DR6:

```
e {crud.types}
```

```
GET /${type.controllerName}.json ${type.controllerClass.name.substring(12).  
replace('$','')}.listJson
```

```
e {/crud.types}
```

Chacun des contrôleurs créés se verra alors attribuer une route automatiquement de la forme `/controllername.json` pointant sur l'action listJson (celle de CrudJson).

Ici, un rapide test vers `localhost:à000/localisedtimezones.json(1)` devrait nous donner quelque chose comme:

1: <http://localhost:à000/localisedtimezones.json>

```
GET      /admin/  
LocalisedTimeZones.index  
GET      /admin/localisedtimezones  
LocalisedTimeZones.list  
GET      /admin/localisedtimezones/new  
LocalisedTimeZones.blank  
GET      /admin/localisedtimezones/{id}  
LocalisedTimeZones.show  
GET      /admin/localisedtimezones/{id}/{field}  
LocalisedTimeZones.attachment  
GET      /admin/localisedtimezones/{id}/edit  
LocalisedTimeZones.edit  
POST     /admin/localisedtimezones
```

```

LocalisedTimeZones.create
  POST      /admin/localisedtimezones/{id}
LocalisedTimeZones.save
  DELETE    /admin/localisedtimezones/{id}
LocalisedTimeZones.delete
  GET       /localisedtimezones.json
LocalisedTimeZones.listJson
UUn tour à l'adresse localhost:à000/admin/(1) devrait vous donner:

```

1: <http://localhost:à000/admin/>



Page à partir de laquelle nous pouvons accéder à la liste des Timezone. Par défaut, la vue list du module crud n'affiche qu'une colonne contenant le résultat de la méthode `toString()` de l'objet.

## Vues

Maintenant, jetons un œil à notre vue custom list.html. Il s'agit de la vue responsable de la génération de notre table HTML (`app/views/CRUUD/list.html`). Le module CRUUD offre un moyen simple et efficace de surcharger des composants du module comme les views ou tags avec la commande `play crud:ov --template CRUUD/list`. Cela indiquera à Play de vous fournir une copie conforme de ce template dans votre propre répertoire, que l'on peut ensuite modifier à souhait. Aucune configuration supplémentaire n'est à apporter, le système de module implique que Play cherche d'abord toute ressource au sein du répertoire de votre appli, puis ensuite au sein des modules configurés. Pratique, puissant, flexible, élégant, le système de modules de Play est une petite merveille... mais je m'égare, continuons :)

Pour modifier la vue list.html, Play propose la commande `play crud:ov` :

```
> play crud:ov
```

```

~
î K' _ \KK' _' KKKK_K

î K __/K_K____K__ (_)
```

~ | \_ | | \_ /  
^ play! 1.1.1, <http://www.playframework.org>

^

^ Specify the template to override, ex : -t UUsers/list

^

^ UUse --css to override the CRUUD css

^ UUse --layout to override the CRUUD layout

^

Ainsi la commande:

```
play crud:ov --t CRUUD/list
```

demandera à Play de copier le template CRUUD par défaut list.html dans le répertoire  
`app/views/CRUUD/list.html` de notre application.

```
#{extends 'CRUD/layout.html' /}  
<div id="crudList" class="${type.name}">  
  <div id="crudListSearch">  
    #{crud.search /}  
  </div>  
  <table class="crud-grid">  
    <caption>UI Grid integration with Play! Crud module</caption>  
    <thead>  
      <tr>  
        <th data-field="timeZoneId">yayTimezone</th>  
        <th data-field="name">Name</th>  
        <th data-field="language">language</th>  
        <th data-field="offset">offset</th>  
      </tr>  
    </thead>  
    <tbody>  
    </tbody>  
  </table>  
  <div id="crudListPagination">  
    #{crud.pagination /}  
  </div>  
</div>
```

Ensuite, nous aurons à configurer les assets (ressources statiques CSS/JS)

nécessaires à UI Grid (dans `app/views/CRUUD/list.html`):

```
#{set 'css'}
  <link rel="stylesheet" type="text/css" media="screen" href="/grid/css/
themes/base/jquery.ui.all.css">
  <link rel="stylesheet" type="text/css" media="screen" href="/grid/js/
grid-datamodel/grid.css">
#{/set}
#{set 'js'}
  <script src="/grid/js/ui/jquery.ui.core.js"></script>
  <script src="/grid/js/ui/jquery.ui.widget.js"></script>
  <script src="/grid/js/jquery.tmpl.js"></script>
  <script src="/grid/js/grid-datamodel/dataitem.js"></script>
  <script src="/grid/js/grid-datamodel/datasource.js"></script>
  <script src="/grid/js/grid-datamodel/datastore.js"></script>
  <script src="/grid/js/grid-datamodel/grid.js"></script>
#{/set}
```

## Création du module crud-grid

Ce module prendra soin de l'import des ressources statiques et de la configuration des routes correspondante pour permettre à notre application d'y avoir accès.

Cette étape n'est absolument pas nécessaire, et vous pourriez vous contenter de stocker ces fichiers au sein de votre répertoire `/public/`.

Toutes les ressources sont récupérées à partir du chemin `/grid/` qui est une route définie par le module crud-grid. Il prendra soin de faire correspondre toute ressources statiques du répertoire `/app/public/` au chemin `/grid/`. Il aura aussi pour rôle de contenir tout code relatif à l'intégration du widget grid (comme la définition du contrôleur spécial CrudJson qui nous permet de renvoyer une représentation JSON et grid-compliant de notre modèle).

```
play new-module crud-grid
```

Pour faciliter sa réutilisation, on déplace la classe CrudJson de l'application vers le répertoire app/controllers du module que l'on vient de créer.

Il nous faut maintenant importer les routes du module au sein de l'application, ceci est fait dans le fichier `conf/application.conf` :

```
module.crud-grid=./crud-grid
```

Le but ici, est de fournir un module contenant tous les fichiers nécessaires au fonctionnement de UI Grid. Ce module crud-grid contiendra alors les fichiers de la

branche de jQuery UI relative au développement du widget grid (css et js) tout en fournissant une route particulière, ce qui permet à notre application de charger ces fichiers à partir de celle-ci.

Ceci est fait avec la modification du fichier route du module `crud-grid/conf/routes` avec quelque chose comme:

```
# Map the static resources from the /app/public folder to the /grid path
GET      /grid/          staticDir:app/public
# Grid
*        /              module:crud-grid
#{set 'js'}
  <script src="/grid/js/ui/jquery.ui.core.js"></script>
  <script src="/grid/js/ui/jquery.ui.widget.js"></script>
  <script src="/grid/js/jquery.tmpl.js"></script>
  <script src="/grid/js/grid-datamodel/dataitem.js"></script>
  <script src="/grid/js/grid-datamodel/datasource.js"></script>
  <script src="/grid/js/grid-datamodel/datastore.js"></script>
  <script src="/grid/js/grid-datamodel/grid.js"></script>
#{/set}
```

N.B : la plupart des modules que l'on rencontrera seront des modules que l'on peut appeler "techniques", cad permettant ou facilitant l'intégration de couche "techniques" que ne propose pas Play par défaut (gae, sienna, pdf, etc.). Mais on peut imaginer que, dans le cadre d'une application assez large pour s'y prêter, l'utilisation de module pour compartimenter "fonctionnellement" l'application est possible (un module admin, un module gestion, un module facturation, etc.).

## UI GRID!

Nous avons créé notre application exemple avec de nombreuses données à afficher, nous nous sommes occupé du contrôleur permettant de renvoyer une représentation du modèle sous format JSON et préparer nos vues pour être utilisé avec UI Grid, nous avons enfin créé un module custom permettant de contenir toutes les ressources nécessaires à UI Grid pour fonctionner. Il nous reste alors à configurer ce fameux widget.

Voici le script responsable de la configuration et de l'initialisation du widget et du datasource associé:

```
<script>
$(document).ready(function() {
  var search = $('#crudListSearch'),
  table = $('.grid-table'),
  pagination = $('#crudListPagination'),
  datasource = $.ui.datasource({
    type: 'timezones',
    data: {search: 'français'},
    source: '/localisedtimezones.json'
  })
})
```



\* les données sont récupérées en passant par des appels BHR (pas de rechargement de page)

\* la pagination est gérée en interceptant les clicks des liens de pagination (les liens générés par le CRUUD de play) pour effectuer une nouvelle requête au datasource en passant les paramètres correspondant: ``datasource.get({page: page});``

\* la fonctionnalité de filtre est supportée. Tout comme les liens, l'event submit du formulaire est intercepté pour demander au datasource de faire une nouvelle requête en passant une fois encore les paramètres correspondants et attendues par le module Play: ``datasource.get({search: $.trim($(this).find('input[name="search"]').val()));``

Un tour à l'adresse [http://localhost:8000/admin/localisedtimezones\(1\)](http://localhost:8000/admin/localisedtimezones(1)) devrait vous donner:

1: <http://localhost:8000/admin/localisedtimezones>

Administration  
Playing with jQuery UI Grid and Ajax datasource

Theme: Smoothness

Home » LocalisedTimeZones

Search

UI Grid integration with Play! Crud module

| YayTimezone       | Name                               | language | offset |
|-------------------|------------------------------------|----------|--------|
| Etc/GMT+12        | GMT-12:00                          | français | -12    |
| Etc/GMT+11        | GMT-11:00                          | français | -11    |
| MIT               | Heure des Samoa occidentales       | français | -11    |
| Pacific/Apia      | Heure des Samoa occidentales       | français | -11    |
| Pacific/Midway    | Heure standard de Samoa            | français | -11    |
| Pacific/Niue      | Heure de Niue                      | français | -11    |
| Pacific/Pago_Pago | Heure standard de Samoa            | français | -11    |
| Pacific/Samoa     | Heure standard de Samoa            | français | -11    |
| US/Samoa          | Heure standard de Samoa            | français | -11    |
| America/Adak      | Heure normale d'Hawaï-Aléoutiennes | français | -10    |
| America/Atka      | Heure normale d'Hawaï-Aléoutiennes | français | -10    |
| Etc/GMT+10        | GMT-10:00                          | français | -10    |
| HST               | Heure normale d'Hawaï              | français | -10    |
| Pacific/Fakaofo   | Heure de Tokelau                   | français | -10    |
| Pacific/Honolulu  | Heure normale d'Hawaï              | français | -10    |
| Pacific/Johnston  | Heure normale d'Hawaï              | français | -10    |
| Pacific/Rarotonga | Heure des Îles Cook                | français | -10    |
| Pacific/Tahiti    | Heure de Tahiti                    | français | -10    |
| SystemV/HST10     | Heure normale d'Hawaï              | français | -10    |
| US/Aleutian       | Heure normale d'Hawaï-Aléoutiennes | français | -10    |
| US/Hawaii         | Heure normale d'Hawaï              | français | -10    |
| Pacific/Marquesas | Heure des Marquises                | français | -9     |
| AST               | Heure normale d'Alaska             | français | -9     |
| America/Anchorage | Heure normale d'Alaska             | français | -9     |
| America/Juneau    | Heure normale d'Alaska             | français | -9     |
| America/Nome      | Heure normale d'Alaska             | français | -9     |
| America/Yakutat   | Heure normale d'Alaska             | français | -9     |
| Etc/GMT+9         | GMT-09:00                          | français | -9     |
| Pacific/Gambier   | Heure de Gambi                     | français | -9     |
| SystemV/YST9      | Heure normale d'Alaska             | français | -9     |

1800 LocalisedTimeZones

12345678910Last<>Next>

Generated by the Play CRUUD module. [Learn how to customize it](#)

Améliorations possibles

"Et le tri?" me direz-vous. Et bien, ce sera une bonne occasion de continuer l'investigation! Pour le moment, nous nous arrêtons à ce niveau là de l'expérimentation. Ce module est donc encore perfectible, mais il nous a permis de mettre en évidence un bon nombre de principes qui devraient vous donner les billes pour créer de nouveaux modules. Il ne reste plus qu'à laisser parler votre imagination!

## Références

Cette expérience est grandement inspirée par ces deux excellents articles de Lunatech Research(1) parlant de l'intégration du plugin datatable avec Play:

1: [//www.lunatech-research.com/editorials/tags/play](http://www.lunatech-research.com/editorials/tags/play)

\* Integrating Play framework with jQuery DataTables(1)

1: [//www.lunatech-research.com/archives/2011/01/20/playframework-jquery-datatables](http://www.lunatech-research.com/archives/2011/01/20/playframework-jquery-datatables)

\* Ajax DataTables with the Play framework(1)

1: [//www.lunatech-research.com/archives/2011/02/02/ajax-datatables-playframework](http://www.lunatech-research.com/archives/2011/02/02/ajax-datatables-playframework)

## Aller plus loin

Pour aller plus loin dans la création de modules, vous pouvez jeter un oeil à [ce tutoriel]() qui explique comment modifier le rendu des pages en utilisant des annotations sur le modèle.