HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Professeur: Nastaran Fatemi
Assistant: Miguel Santamaria

SCALA: Play Framework

Contents

1 Play Framework **2**
1.1 Création d'un projet Play 2
1.2 Exécution du serveur Play 3
1.3 Quelques brèves explications avant d'aller plus loin... 4
1.4 Ajouter de nouvelles dépendances 7
1.5 Exemple d'application Play Framework 8

2 Références **9**

1 Play Framework

Play Framework est un framework web open source écrit en Scala (depuis la version 2.0) et qui permet d'écrire des applications web non seulement en Scala, mais aussi en Java. Play cherche à optimiser la productivité en priorisant les conventions plutôt que la configuration manuelle. A l'aide de cette technologie de qualité basée sur le patron d'architecture MVC (Modèles-Vues-Contrôleurs), il est possible de réaliser non-seulement des REST API, mais aussi des applications web fullstack, voire même les deux en même temps !

Il s'agit d'un framework qui utilise la JVM mais qui se différencie des autres frameworks du même type de par son choix volontaire de ne pas utiliser le moteur Java de Servlet. Ceci dit, si vous faites partie des chanceux ayant suivi le cours d'AMT du semestre passé, vous trouverez tout de même des similitudes avec le fonctionnement de JavaEE et de Spring-Boot.

Depuis la version 2.0 du framework, les processus de production et de déploiement sont gérés par SBT, ce qui simplifie amplement les tâches.

Play Framework prend en charge la librairie Scala **Slick**, qui permet de questionner une base de données. Celle-ci est présentée plus en détails dans le tutoriel réservé à cet effet, car nous allons tout d'abord créer notre tout premier projet Play Framework !

1.1 Création d'un projet Play

La création d'un nouveau projet Scala Play se déroule de la même manière que pour n'importe quel autre projet dans IntelliJ (**File | New | Project... | Scala | Play 2.x**).

A noter que la création d'un premier projet Play prendra un temps relativement long pour s'exécuter, car SBT va télécharger un grand nombre de ressources. Lorsque le processus est terminé, vous devriez obtenir l'arborescence de fichiers suivante :

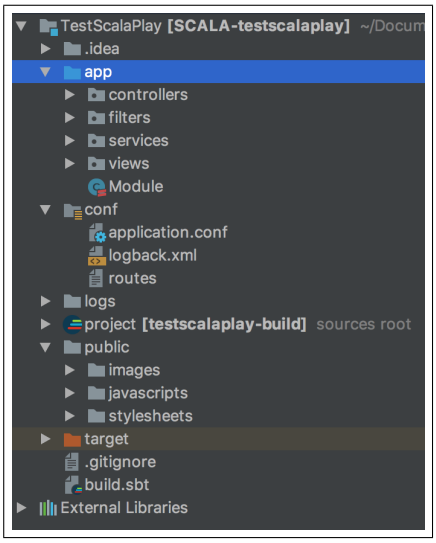

Figure 1: arborescence de fichiers d'un projet Play Framework vierge

Parmi les dossiers et fichiers importants qui nous intéressent, vous trouverez les suivants :

- **app** : ce dossier contient le code back-end du serveur et possède déjà une arborescence permettant de gérer des contrôleurs, des filtres, des services et des vues Scala qui seront compilées en HTML lors du déploiement.
- **conf** : contient notamment le fichier de configuration de l'application, et le fichier de configuration des routes.
- **public** : contient toutes les ressources (« assets ») nécessaires au front-end de l'application (JavaScript, CSS, images, polices d'écriture, etc.).
- **target** : contient le code qui sera exécuté par le serveur et qui est produit par le build process SBT.
- **build.sbt** : ce fichier contient toute la configuration et les dépendances nécessaires à SBT pour le build process (en comparaison à Maven, il s'agit de la version « pom.xml » de SBT).

1.2 Exécution du serveur Play

Une configuration d'exécution *Play2Run* doit normalement elle-aussi avoir été automatiquement générée par SBT lors de la création du projet. Celle-ci permet de lancer ou de débugger le serveur Play.

Figure 2: exécution *Play2Run*

En exécutant la procédure (bouton « play »), le serveur doit se lancer, et vous devriez pouvoir accéder à l'application via l'URL http://localhost:9000/.

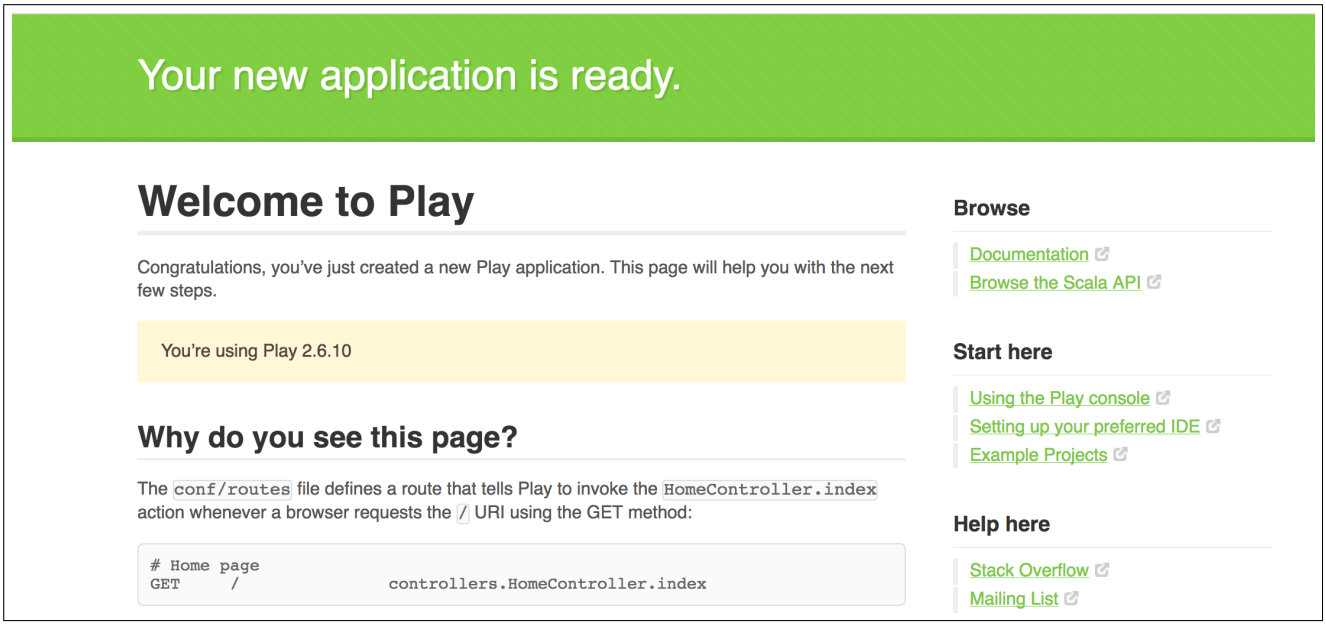

Figure 3: contenu initial par défaut d'un nouveau projet Play Framework

Vous pouvez désormais parcourir les explications présentées sur la page, et jouer avec le code de l'application depuis IntelliJ. A noter qu'une compilation continue est mise en place par défaut, vous évitant de devoir à chaque fois redémarrer le serveur pour appliquer vos changements.

Prêtez une attention particulière à la gestion asynchrone de requêtes présentée dans le contrôler **Async**. Vous pourrez y voir qu'un système de promesses est disponible.

Le contrôleur **Count** présente un exemple d'injection de dépendance.

A noter que l'application générée possède donc plusieurs composants qui peuvent s'avérer inutiles, et qui sont présents ici à titre d'exemples. Si vous souhaitez partir sur une application vierge, vous pouvez cloner le contenu de la branche *fb-empty-play-app* disponible sur ce repo GitHub.

1.3 Quelques brèves explications avant d'aller plus loin...

Comme vu auparavant, Play Framework est basé sur le patron d'architecture MVC, ce qui signifie pour rappel que les données transitent de la vue au modèle via le contrôleur, et inversément.

Dans le jargon du framework, une application typique contient une liste de **contrôleurs** (une classe par contrôleur dans le dossier *app/controllers*), qui contiennent eux-mêmes une liste d'**actions** (les méthodes de ces classes, en quelque sorte). Chacune de ces actions représente un endpoint de l'application (que ce soit un endpoint REST, ou alors un endpoint permettant d'accéder à une vue HTML), accessible via une URI grâce au **routeur** du framework (*conf/routes*). Ces actions envoient une réponse HTTP pouvant contenir ou non les données d'une **vue** (qui est un mélange de Scala et d'HTML, à la manière des fichiers JSP de JavaEE) pour afficher une page ; si tel est le cas, la vue sera compilée par le serveur afin d'obtenir de l'HTML pur qui sera lu par le client (le navigateur, en général).

Ainsi, pour créer une nouvelle page dans notre site, il faudra ajouter une action dans un contrôleur, créer une vue associée, et configurer le routeur pour pouvoir mapper une URI avec notre nouvelle action.
Tout cela peut vous paraître un peu confus au premier abord, c'est pourquoi nous allons ensemble créer une nouvelle page « test » sur notre site généré ! Vous trouverez le code complet de cet exemple sur ce repo GitHub, dans la branche *fb-example-new-page* :

1. Nous allons tout d'abord commencer par **créer une nouvelle vue** :
   (a) Dans *app/views*, créer un nouveau fichier *test.scala.html* ; ce fichier représente donc notre vue, et possède une extension de type *scala.html*, indiquant ainsi au serveur que le fichier devra être compilé. Cela permettra notamment d'utiliser le caractère `@` qui permet au serveur d'interpréter du code Scala, ce qui va s'avérer très pratique ! Par la suite, une vue pourra être référencée à l'aide de son nom, qui est généré selon la chaîne de caractère située avant le « scala.html » dans le nom du fichier (ici, *test*).

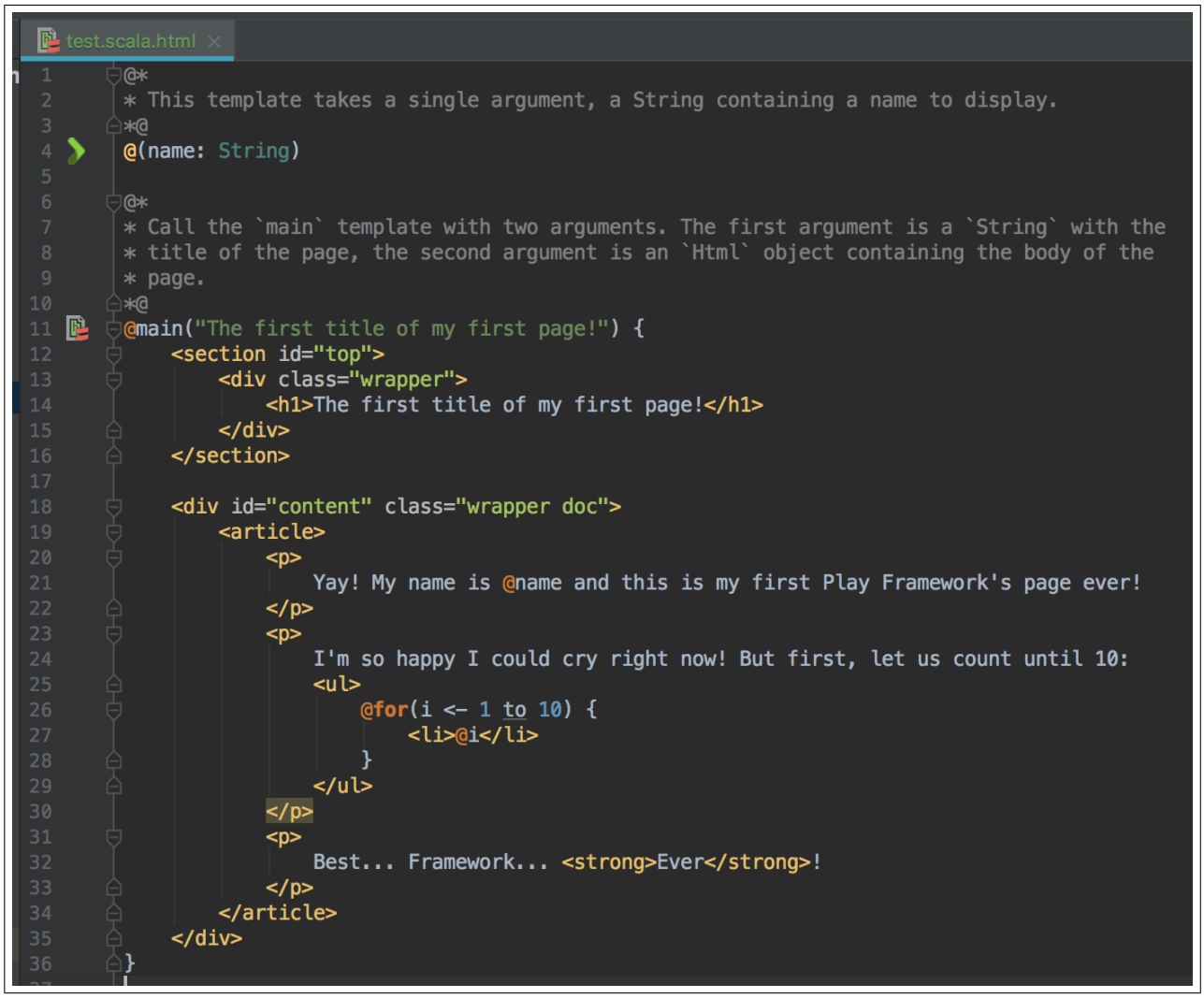

Figure 4: vue *test.scala.html*

(b) Dans cet exemple, nous allons passer un paramètre `name` de type `String` à notre vue depuis notre contrôleur ; il faut donc le déclarer à l'aide de la première ligne de code : `@(name: String)`. Ainsi, nous pourrons par la suite afficher le contenu de ce paramètre en entrant `@name`.

(c) Nous appelons ensuite le template *main* (qui correspond donc à la vue *main.scala.html* et qui contient le code de base d'une page) dans lequel nous injectons du code HTML. Ce template prend deux paramètres : le titre de la page dans le navigateur, ainsi que le fameux code à injecter.

2. Pour **créer la nouvelle action** qui appellera notre vue, nous allons ici réutiliser le contrôleur qui existe déjà (**HomeController**) :

(a) Dans *app/controllers*, ouvrir le fichier *HomeController*, et y ajouter une nouvelle action en-dessous de l'action *index*.

```
/**
  * Create an Action that renders an HTML page based on the "test" view.
  * The configuration in the `routes` file means that this method
  * will be called when the application receives a `GET` request with
  * a path of `/test`.
  */
def test = Action {
  Ok(views.html.test("Mordecai"))
}
```

Figure 5: action *test* du contrôleur **HomeController**

(b) Cette action renvoie une requête HTTP contenant un statut « 200 OK » et du code HTML basé sur la vue *test*, en lui passant le paramètre « Mordecai » de type `String`. Vous pouvez trouver les différents codes HTTP renvoyables nativement depuis une vue ici.

(c) Dans le cas où l'IDE ne reconnaîtrait pas la vue, il suffit de (re)lancer le serveur pour initialiser la phase de compilation et l'écriture des résultats dans le dossier *target*.

3. Il ne nous reste plus qu'à **créer une route** pour lier l'action à l'URI */test* :

(a) Dans *conf*, ouvrir le fichier *routes*, et y ajouter la nouvelle route en-dessous de la route */*.

```
# An example controller showing a sample home page
GET     /                           controllers.HomeController.index
# An action that shows the new "test" page
GET     /test                       controllers.HomeController.test
```

Figure 6: route */test*

(b) Cette route permet de rediriger une requête HTTP de type GET sur l'action *test* du contrôleur **HomeController**.

4. Vous pouvez finalement accéder à votre nouvelle page via l'URL http://localhost:9000/test.

Figure 7: Nouvelle page *test*

5. Si vous souhaitez générer un lien sur cette page dans une balise `<a>` d'une autre page par exemple, vous pouvez le faire en utilisant le routeur, à l'aide du code `@routes.HomeController.test()`. Dans cet exemple, nous allons générer un lien dans la vue *welcome.scala.html*, accessible depuis la racine du site.

```
<article>
  <h1>Welcome to Play</h1>
  Enter content here!
  <br/>
  <a href="@routes.HomeController.test()">Go to my test page!</a>
</article>
```

Figure 8: Génération d'un lien vers la nouvelle page *test*

1.4 Ajouter de nouvelles dépendances

Pour ajouter de nouvelles dépendances au projet, il suffit de les ajouter dans le fichier "build.sbt" (par exemple : `libraryDependencies += "mysql" % "mysql-connector-java" % "5.1.24"`), puis de rafraîchir le projet en appuyant sur le bouton "Refresh all sbt projects" de la fenêtre SBT dans IntelliJ (si celle-ci n'est pas affichée, cliquer sur **View | Tool Windows | sbt**).

Il est possible d'activer la compilation SBT automatique pour un projet, en se rendant dans **File | Settings | Languages & Frameworks | Play2** ou dans **IntelliJ IDEA | Preferences... | Languages & Frameworks | Play2** selon votre OS, et en cochant la case **Use Play 2 compiler for this project**.

1.5 Exemple d'application Play Framework

Vous trouverez un exemple d'application Play Framework qui gère une liste d'étudiants ainsi qu'une liste de cours (Ultimate HEIG-VD Manager 2018 ©) sans base de données sur ce repo GitHub, dans la branche *fb-basic-play-app*.

L'application est composée d'une REST API (dont les endpoints sont accessibles via les actions situées dans les contrôleurs **StudentsController** et **CoursesController**) et de deux pages HTML (situées dans le contrôleur **HomeController**) :

- **index**, la page principale, depuis laquelle l'utilisateur peut consulter les données, les modifier en cliquant dans les cellules d'un tableau, et les supprimer.
- **about**, une page secondaire qui sert surtout à illustrer la génération de liens hypertexte vers une action du Framework à l'aide du routeur.

Les classes de gestion des données sont situées dans les deux services *app/services/Student* et *app/services/Course*.

Afin de pouvoir utiliser du JSON pour communiquer avec la REST API, nous avons besoin de mécanismes de sérialisation et de désérialisation. Ceux-ci sont implémentés dans les contrôleurs **StudentsController** et **CoursesController** à l'aide de trois méthodes à chaque fois. En regardant dans le **StudentsController**, nous pouvons voir :

1. Une fonction `studentToJson` qui va permettre de sérialiser un objet de type `Student` en JSON.
2. Une fonction `jsonToStudent` qui va permettre de désérialiser du JSON en un objet de type `Student`.
3. Une méthode `validateJson` qui va tenter de désérialiser du JSON en un objet de type donné, en utilisant l'implémentation du `Reads` de ce dernier (notre fonction `jsonToStudent`), à condition qu'elle existe. Cette méthode est passée en paramètre des actions qui requiert une désérialisation de données JSON (comme par exemple `createStudent`).

Finalement, afin de pouvoir utiliser le mécanisme de génération d'URI du routeur dans le code JavaScript (très apprécié de tous les étudiants, bien-entendu) du client, il faut générer une ressource *router* en créant une action qui invoque une version JavaScript du générateur du routeur.
Dans l'exemple, il s'agit de l'action *javascriptRoutes* du contrôleur *HomeController*, accessible via l'URI */javascriptRoutes*, qui est chargée sous forme de script dans les en-têtes des pages HTML à l'aide de la balise `<script type="text/javascript" src="@routes.HomeController.javascriptRoutes"> </script>` située dans le template *main.scala.html*.
Grâce à ces étapes, il est désormais possible d'invoquer le générateur du routeur en appelant `jsRoutes.controllers.NomDuController.nomDeLAction()`. Vous trouverez des exemples dans le fichier *public/javascripts/actions.js*.

2 Références

- https://fr.wikipedia.org/wiki/Play_framework
- https://playframework.com/
- https://www.playframework.com/documentation/2.6.x/ScalaJavascriptRouting
- https://www.playframework.com/documentation/2.6.x/ScalaJsonHttp