Flux de données et programmation

Acteurs et services en Akka

1 Consignes

Le travail demandé est d'une part de produire un code approfondissant quelques unes des notions abordées pendant ce cours et d'autre part d'expliquer ce que fait votre code en cinq minutes, au vidéo projecteur.

Vous travaillerez seuls mais vous pouvez vous entraider.

Le code que vous rendrez n'aura pas besoin d'être une application complète, ni de correspondre à un projet particulier. Votre code doit :

1. utiliser Akka (utiliser la documentation et celle de l'API) ;
2. si possible communiquer ses affichages en HTTP avec Spray ou Akka HTTP (c'est la même chose) ;
3. être éventuellement complété par une interface utilisateur très sommaire dans le navigateur, utilisant Riotjs et jQuery ;
4. compiler et fonctionner, même s'il ne fait rien d'intéressant.
5. être rendu dans un dépôt git, lisible sur votre compte à l'université ou sur un github ou ailleurs.
6. Vous aurez à d'expliquer votre code. Ne copiez-collez pas du code sans le comprendre.

Votre code doit donc faire quelque chose, mais cela peut être quelque chose de bizarre, inutile et incomplet.

Vous pouvez créer un seul acteur dont vous montrerez le fonctionnement, en imaginant qu'il sera intégré plus tard à un système plus vaste.

Par exemple, votre code pourrait :

— donner la liste des anagrammes d'un mot fourni par l'utilisateur ;
— compléter cela par la liste des émissions de radio qui mentionnent ce mot ou l'un de ses anagrammes dans leur titre ou leur résumé ;
— Lorsque le mot où l'un de ses anagrammes est le nom d'une station de métro, donner les prochains départs depuis cette station (`https://github.com/pgrimaud/horaires-ratp-api`) ;
— etc.

Si cela est possible nous intégrerons vos différents acteurs dans un système d'acteurs plus vaste mais sans avoir une application précise en tête.

2 Mise en marche

Vous utiliserez un template `sbt` pour `spray` (`https://github.com/spray/spray-template`). Au besoin un projet `sbt` peut être utilisé directement dans IntelliJ IDEA.

N'hésitez pas à faire des branches ou si vous préférez à multiplier les copies de ce template pour faire vos essais. Ne commitez que des fichiers sources, pas des fichiers générés.

Question A. Lister le contenu. Votre premiere tâche sera d'identifier le rôle des différents fichiers dans ce template. Vous devriez avoir sept fichiers importants (outre la doc).

Question B. Revolver. Vous pouvez normalement lancer le code en tapant `run` dans `sbt`. Ce projet utilisant `revolver`, vous pouvez lancer le programme avec `re-start` et le stopper avec `re-stop`. vous pouvez encore le lancer avec `~re-start` de façon à ce que le projet recompile et relance l'application à chaque modification de fichier source. Faites un essai (ouvrez votre navigateur à l'url indiquée).

Vous trouverez une présentation sur Akka et Spray ici : `http://spray.io/wjax/`. En particulier la page 45 vous donne un exemple de routage des requêtes HTTP entrantes en utilisant l'opérateur `~` entre clauses `path`. Il y a également du jsonp, que nous verrons plus loin.

Question C. Ping. Faites en sorte de répondre avec un contenu html de votre choix à une requête sur l'url `ping`.

Question D. Test. Quel code source est exécuté lorsque vous lancez la commande `test` dans `sbt` ? Ajouter un nouveau `test` qui vérifie que le service fonctionne également pour l'url `/ping`.

Question E. Choisir les dernières versions. Commencez par changer les numéros de version de façon à utiliser les dernières versions stables de tous les composants et dépendances du projet. Dans quels fichiers devez vous intervenir (trois fichiers) ? Vérifiez que cela compile et s'exécute correctement. Si ça ne fonctionne pas vous pouvez ramener un composant à son numéro de version d'origine (en général, un changement de majeur dans la version d'un composant nécessite une réécriture du code qui l'utilise).

Pour répondre un contenu json à une requête HTTP, il faut avant tout construire un type pour ce contenu. Cela se fait avec une *case class* (`Reponse` dans le code ci-dessous).

Il faut également définir une façon de sérialiser les données de ce type sous forme de chaîne (opération appelée marshalling). Cela se fait en construisant une valeur implicite et en l'important dans le contexte de la réponse (c'est une tournure assez habituelle en Scala mais elle n'est pas simple à comprendre).

L'objectif global de cette façon de procéder est de laisser l'api http en bordure de l'application de façon à ne pas interférer avec la logique de l'application.

Commençons par ajouter la bibliothèque `spray-json` au projet en ajoutant dans `build.sbt` :

```
"io.spray" %% "spray-json" % sprayV,
```

On modifie `MyService.scala` pour ajouter un service de réponse en json et jsonp à des requêtes GET sur l'url `/api`.

Premièrement, il faut ajouter les imports suivants :

```
import spray.json._
import spray.json.DefaultJsonProtocol._
import spray.httpx.SprayJsonSupport
```

Deuxièmement, il faut définir le type de données que nous utiliserons pour les réponses (`Reponse`) et sa sérialisation implicite.

```
case class Reponse(entiers: List[Int], graine: Int, mot: String)

object JsonReponse extends DefaultJsonProtocol with SprayJsonSupport {
  implicit val json = jsonFormat3(Reponse)
}

import JsonReponse._
```

Troisièmement il faut donner les instructions de routage pour l'url `api` dans le trait `MyService` (on a adapté ici le code de la présentation citée plus haut).

```
pathPrefix("api") {
    jsonpWithParameter("callback") {
      path("numbers") {
        post {
          parameter("seed".as[Int]) { seed =>
            validate(seed >= 0, "query parameter 'seed' must be >= 0") {
              complete {
                Reponse((1 to seed).toList, seed, "hello")
              }}}}}}
```

Question F. Essayer. Tester la requête sur l'api en fournissant le nom de la callback et aussi sans le fournir.

Question G. Passer la demande à un autre acteur. Pour la clarté du code, nous voulons séparer la logique des routes de celle de la construction des réponses, en particulier si l'api s'enrichit. Il est possible de le faire en faisant appel à une méthode de `MyService` mais il est plus intéressant de savoir comment le faire en demandant à un autre acteur de formuler la réponse. Cela se fait en utilisant le *pattern ask* (page 95 de la doc) qui produit une promesse de résultat (un future).

Il y a quelques subtilités à connaître pour utiliser ce pattern. Supposons que vous ayiez déclaré un type de message `Seed` et un type d'acteurs `SeedHandler` répondant à ces messages par une `Reponse`.

```
case class Seed(n: Int)

class SeedHandler extends Actor {
  def receive = {
    case Seed(n) => sender ! Reponse((1 to n).toList, n, "hello")
  }
}
```

Dans `myRoute` vous pouvez alors passer la question à un acteur de type `SeedHandler` et utiliser sa réponse au lieu de la formuler directement.

```
complete {
  (seedHandler ? Seed(seed)).mapTo[Reponse]
}
```

Vous avez besoin pour cela de faire trois choses dans `MyService` :

1. Créer l'acteur `seedHandler`
2. définir un temps au delà duquel on n'attendra plus la réponse de `seedHandler` (on utilise ici une valeur implicite)
3. Définir un *dispatcher* dans le contexte implicite (il est nécessaire pour gérer la réalisation de la promesse de réponse).

```
implicit val timeout = Timeout(5 seconds)
implicit def ec = actorRefFactory.dispatcher

lazy val seedHandler = actorRefFactory.actorOf(Props[SeedHandler])
```

Bon travail !