INFOM451 Conception d'applications mobiles

Projet « Gastro »
Partie 2 : futurs acteurs gustatifs exceptionnels

Gonzague Yernaux - gonzague.yernaux@unamur.be - Bureau 224

Année 2020-2021

1 Introduction

Vous avez rempli vos deux premières missions : votre compositeur de menus en Scala fait un tabac auprès de vos clients américains. Seulement, vous recevez quelques plaintes, et comme tout programmeur consciencieux, vous décidez de fournir un service clients de maintenance efficace et performant. Voici les principales plaintes reçues :

1. Tout fonctionne bien lorsque le fichier contenant les produits suit le bon format, sans erreur, que l'on a bien les droits d'écriture dessus et qu'aucun problème ne survient à l'exécution. Mais s'il y a le moindre couac, votre programme trop peu robuste finit en *crash*... et aux yeux des clients, ce genre d'incident tombe comme un cheveu dans la soupe.
2. Votre programme donne des associations de menus sympathiques, mais il y a certainement moyen d'affiner les choix d'associations de produits encore un peu... Et puis, votre programme ne donne aucune information concernant les quantités. Or vos clients aimeraient savoir exactement quelle portion ils devraient se servir de quel aliment pour éviter tout déséquilibre alimentaire.
3. Votre manager, de son côté, aimerait que votre programme puisse recevoir plusieurs demandes de menus à la fois, pour ne pas devoir attendre qu'il ait fini de composer un menu avant de lancer une seconde requête.
4. Certains clients plus paresseux que les autres aimeraient avoir trois compositions pour leur repas : entrée, plat, dessert ; chaque assiette devant être équilibrée au niveau des nutriments (mais le dessert peut bien sûr être plus sucré). Libre à vous de remplir cette mission, qui vous octroiera certainement une augmentation si votre manager en est satisfait !

Note : vous allez donc modifier directement votre code précédent !

2 Objectifs d'apprentissage

Comme pour la première partie du projet, chaque objectif d'apprentissage devra apparaître au moins une fois dans votre code et être annoncé (par un commentaire).

2.1 Gestion des erreurs

Cette seconde partie de projet vous permettra d'écrire du code Scala avec une gestion élégante des erreurs :

— comprendre l'intérêt et l'usage de Option (avec *pattern matching*)

— comprendre l'intérêt et l'usage de Try (avec *for comprehension* et *pattern matching*)

— maitriser parfaitement les *for comprehension*

Pour ce dernier point, vous traduirez l'une de vos expressions `for` en combinaison de `map`, `flatMap` et `filter`. Il faut qu'il y ait au moins un `map` et un `flatMap`.

La gestion des erreurs et exceptions peut se faire où vous le souhaitez dans le code, mais il est imposé que le type de retour de la méthode de lecture du fichier `portions.csv` soit `Try` (voir section 3.1).

2.2 Future et Promise

Vous devrez également prouver votre maîtrise des concepts de `Future` et `Promise`. Il ne suffit pas de déclarer un de ces types : il faudra surtout les utiliser à bon escient (là où ça a du sens) et correctement (bonne utilisation des méthodes, notamment *callbacks* vus au cours).

2.3 Acteurs

Le plus grand changement dans votre implémentation sera l'utilisation d'acteurs Scala pour gérer la concurrence dans le programme. Dans la section suivante, vous recevrez les instructions plus précises quant aux acteurs qui sont attendus et au nouveau fonctionnement du programme pour faire face aux plaintes des clients.

Faites attention à respecter les bonnes pratiques dans la conception de vos acteurs. Pour l'implémentation, faites usage de la librairie Akka dont la documentation est disponible à l'adresse

https ://doc.akka.io/docs/akka/current/actors.html

3 Consignes

3.1 Description du programme

Dans cette seconde partie du projet, le fonctionnement du compositeur de menus se complique. Il ne s'agit plus d'un programme mono-thread mais d'un échange permanent de messages entre acteurs. Voici les différents acteurs qui doivent se retrouver dans votre code, ainsi que leurs interactions :

1. Un acteur central, nommé le Coq, gérera différentes étapes du processus de composition de menus. Un message lui est envoyé depuis le programme principal dès qu'une requête de menu est faite par l'utilisateur. Attention, la composition d'un menu ne peut monopoliser le Coq : un bon chef doit être capable de recevoir plusieurs requêtes d'affilée du programme principal sans broncher, et répondre à chaque demande avec un bon menu sain et équilibré !
2. Le Coq fait alors appel à l'Intendant, un acteur ayant accès à tous les produits de la base de données. L'intendant, lorsqu'il est sollicité de cette façon, renvoie au Coq un produit de base autour duquel le menu sera composé. (Vous pouvez choisir si l'Intendant choisit le produit de base au hasard ou en fonction de la requête faite par le programme).
3. Le Coq fait alors appel à différents autres acteurs pour composer le menu. Chacun des acteurs en question est spécialisé dans un type d'aliment. Par exemple, l'on pourrait imaginer les acteurs suivants [1] :
   — FatDispenser : un acteur spécialisé dans les produits gras.
   — ProteinDispenser : un acteur spécialisé dans les produits protéinés.
   — SugarDispenser : un acteur spécialisé dans les produits sucrés.

1. Ce n'est qu'un exemple. Vous pouvez vous en inspirer ou non. Dans tous les cas, faites avec **au moins trois** acteurs spécialisés dans un aliment.

Ces acteurs vont typiquement chercher dans leurs produits un ou plusieurs produit(s) qui pourrai(en)t bien se marier avec le produit de base (et éventuellement avec d'autres produits déjà choisis par le Coq), et envoient le résultat au Coq. Libre à vous de les faire tous sélectionner des produits par rapport au produit de base, ou de sélectionner les produits au fur et à mesure en fonction des produits sélectionnés par les autres acteurs spécialisés déjà consultés par le Coq.

4. Le Coq évalue les produits qu'il a reçus. Peut-être y en a-t-il trop et faut-il faire le tri ? Peut-être certains produits ne se marient pas si bien, finalement ? Quoi qu'il arrive, c'est le Coq qui a le dernier mot sur quels produits seront sélectionnés pour former le menu final.
5. Le Coq envoie les références des produits sélectionnés à l'Intendant. L'Intendant répond alors avec les **quantités** des produits qu'il faut servir pour que le repas soit équilibré. Pour cela, l'Intendant doit consulter une liste où à chaque produit est associée la quantité d'une portion. Ces informations de quantités se trouvent dans le fichier `portions.csv` fourni sur WebCampus. Il vous est demandé d'implémenter la lecture de ce fichier dans l'objet GastroExtractor, sous la forme d'une méthode renvoyant un objet de type `Try[A]` ou `A` est une collection Scala adaptée.
6. Le Coq envoie le menu final ainsi que les quantités conseillées au programme principal, sous la forme d'un `Option[String]`. Le programme imprime alors un message adéquat à l'écran, en fonction du sous-type d'`Option` auquel il a affaire.

3.2 Mission transversale : les concepts Scala

Idem partie 1 du projet.

3.3 Rapport

Un rapport de maximum 4 pages A4 est attendu à l'issue de cette partie du projet. Vous y décrirez votre programme (ou en tout cas, ses spécificités par rapport aux ambiguïtés et points de choix de la section 3.1) dans un français correct (ou un anglais correct). Vous pouvez y mentionner les difficultés rencontrées, les améliorations qui pourraient être faites à votre solution, ainsi que les points forts de votre programme : en quoi les menus composés par votre programme sont-ils meilleurs que ceux d'un autre ? à quel point avez-vous amélioré le programme de la première partie du projet ? Vous pouvez en outre mentionner tout ce qui, à votre avis, devrait être communiqué à l'assistant ; quant aux détails très techniques, vous les garderez au niveau des commentaires insérés dans le code. Prêtez une attention particulière à la forme du rapport, qui est également cotée.

3.4 Délivrable attendu

Le délivrable attendu est un .zip à soumettre sur WebCampus comprenant le code (votre nouvelle version de `gastro.scala`), le rapport au format PDF et tout autre fichier nécessaire à votre implémentation. N'oubliez pas, voici une *checklist* des choses essentielles que doit vérifier votre délivrable.

— Le code compile, sans *warning* et s'exécute.

— Le code fait ce qui est spécifié dans sa documentation. Si un changement de fonctionnement devait survenir par rapport à la description de la section 3.1 (par suite d'une amélioration de l'algorithme de génération de menus par exemple), ce changement est documenté et justifié.

— Le code respecte le paradigme véhiculé par Scala.

— Le rapport est correctement écrit (soit en français, soit en anglais) et mis en forme ; il remplit le rôle décrit dans la section 3.3.

— Le code est clair, lisible, optimisé. Il s'exécute dans un temps raisonnable.