Professeur : Nastaran FATEMI
Assistant : Christopher MEIER

Labo 4
Bot-Tender : Future

Objectifs

Dans les itérations précèdentes de Bot-tender, on supposait une commande prête immédiatement. L'objectif de ce labo est d'intégrer une gestion de commandes asynchrones en utilisant la classe `Future` de l'API Scala.

Ce laboratoire intègre le travail implémenté lors du Labo 1, 2 et 3.

Indications

Le code source du labo vous est fourni avec un commit sur le repo git. Celui-ci contient la pré-implémentation du laboratoire 3.

Pour récupérer ces nouveautés il vous suffit de faire `> git pull upstream` puis de régler les éventuels conflits de *merge*.

- Ce laboratoire est à effectuer par groupe de 2, en gardant les mêmes formations que pour le labo 3. Tout plagiat sera sanctionné par la note de 1.
- Votre implémentation du labo 3 doit être intégrée à ce laboratoire. Si vous le souhaitez vous pouvez utiliser l'implémentation d'un de vos collègues (n'oubliez pas d'indiquer la source en commentaire).
- En plus du code source, il faut rendre un fichier README contenant une explication de vos choix architecturaux et d'implémentation (pas plus que 2 pages).
- Faites en sorte d'éviter la duplication de code.
- Préférez un style de programmation fonctionnel.
- N'utiliser pas `Future.onComplete` ou `Await`.
- On vout fournit la classe `MainFuture` comme point d'entrée pour ce labo.

Description

1. Une commande peut prendre un certain temps avant d'être prête.
2. Directement après la commande, le bot indique qu'elle est en cours.
3. Chaque type de produit à un temps de préparation qui lui est propre.
   — Par exemple : une bière Farmer peut prendre plus de temps à préparer qu'une bière Boxer.
   — On vous fournis la fonction `Utils.FutureOps.randomScheduled` qui simule un temps d'attente (éventuellement avec échec).
4. On peut préparer plusieurs types de produits en même temps, par contre les produits d'un même type sont préparés les uns après les autres
   — Par exemple : Si on commande 2 croissants maison et 1 croissant caillier, la préparation du croissant caillier et du premier croissant maison commence en même temps alors que la préparation du deuxième croissant maison commence quand le premier croissant maison est prêt.
5. La préparation d'un produit peut échouer, dans ce cas on sert une commande partielle avec les produits qui ont pu être préparés.
6. Dès que la commande est prête (complète ou partielle), le bot envoi un message qui mentionne l'utilisateur qui a effectué la commande ainsi que le statut de la commande. La gestion des race conditions lors de l'ajout des messages est implémenté pour vous dans la classe `Data.MessageConcurrentImpl`.
7. Le prix des produits servis est prélevé lorsque la commande est prête. Il faut donc modifier `AccountService` pour supporter un accès concurrent.
   — Par exemple en utilisant `scala.collection.concurrent`

Voir les figures 1 à 4 pour un exemple d'exécution.

Concurrence

L'API Scala fournit une collection qui permet l'accès et la modification concurrente à ces éléments.

La méthode `updateWith` de `TrieMap` est particulièrement intéressante car elle permet la mise à jour atomique et conditionnelle d'une valeur de la collection.

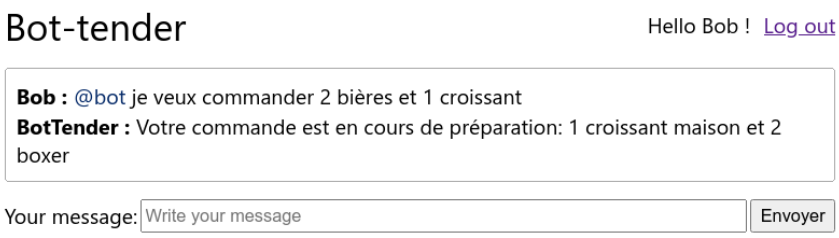

FIGURE 1 – Une capture d'une commande en cours de préparation

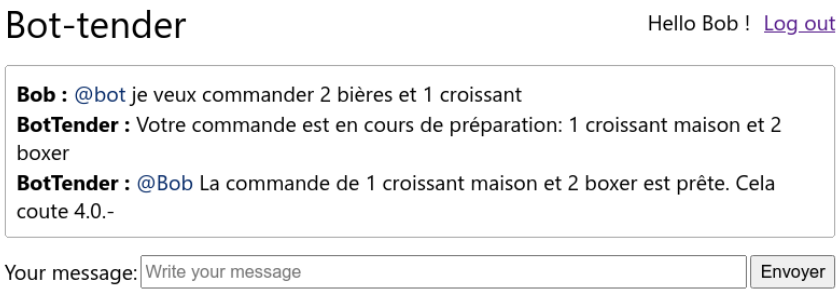

FIGURE 2 – Une capture d'une commande servie

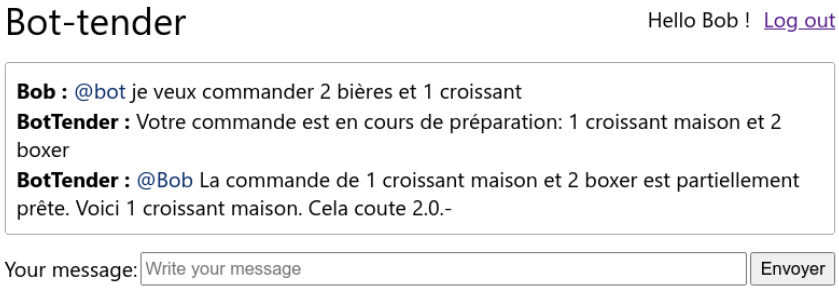

FIGURE 3 – Une capture d'une commande partiellement servie

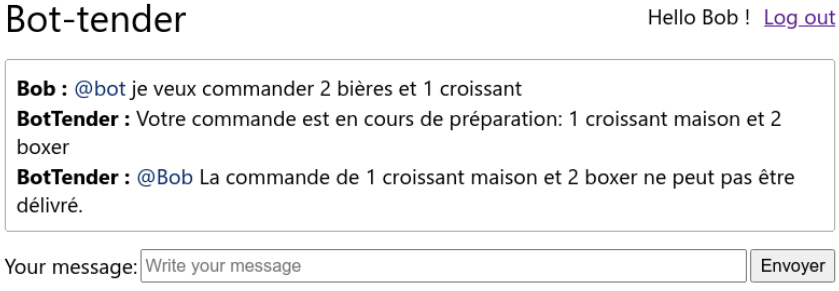

FIGURE 4 – Une capture d'une commande qui a échouée