Auteur: Tom Gabrièle
Date de dernière modification: 5 juin 2018

Retour sur le projet n°7 : Créez GrandPy Bot, le papy-robot

Démarche

Versionning

L'outil de versionning utilisé est git et le dépôt a été créé sur Github.

Le projet comprend 2 branches principale: **master** et **development**.

Pour chaque nouvelle fonctionnalité, une nouvelle branche a été créé à partir de la branche **development**. Une fois la fonctionnalité écrite, sa branche correspondante est fusionnée (merge) dans **development**.

La fusion dans la branche **master** n'a été faite qu'à partir de la première version fonctionnelle de l'application.

Découpage du projet

Les word files handler et data loader

La part la plus importante et la plus technique du projet repose sur l'extraction des mots ou expressions clés à partir de la question posée par l'utilisateur.

Pour cela, il faut constituer un référentiel de mots assez complet pour faciliter le travail. J'ai donc recherché un certains nombre de fichiers contenant l'ensemble des mots français, la liste des pays du monde , la liste des villes du monde…

Il est impensable de faire une recherche dans tous ces fichiers à chaque question de l'utilisateur. J'ai donc créé une base de données contenant 2 tables. La première contient les mots et leur code catégorie. La seconde liste les différentes catégories de mots.

Pour alimenter ces 2 tables, je me suis basé sur une variable de configuration qui indique pour chaque catégorie de mots, les fichiers et le handler associés. Le handler est la méthode qui permet d'extraire d'un fichier les mots à intégrer en base de données. Une instance de classe est ensuite chargée d'appeler tous les handler et de passer les requêtes SQL nécessaires.

Les parsers

Les **parsers** ont pour rôle de prendre une chaîne de caractères en entrée et d'en ressortir une liste de mots. Pour cela il faut "splitter" la chaîne transmise puis la comparer avec une liste de mots issue soit de la base de données (des villes par exemple ou des "stop words") et renvoyer uniquement les mots de la chaîne initiale qui sont présents dans les 2 listes ou l'inverse selon ce que l'on veut mettre en évidence. Par exemple:
- Si le**parser** est basé sur la liste des villes, il renvoie uniquement les noms de villes contenus dans la chaîne transmise.
- Si ce dernier est basé sur les "stop words", il renvoie tous les mots de la chaîne initiale excepté les stop words…

Chaque mot et chaque **parser** n'a pas la même importance. Pour arbitrer le choix sur les résultats les plus importants, j'utilise un **ParsingController**. Celui-ci appelle successivement tous les **parsers** et applique un

coefficient de pondération sur chaque résultat. L'ordre des mots dans la question initiale est également pris en compte.

Les API connector

Autre partie importante de l'application: l'appel aux API Google et MediaWiki. Pour chacune de ces API, j'ai écrit un connecteur qui fait un appel à l'API et renvoie une réponse structurée quel que soit le résultat de ce dernier. Les erreurs sont donc gérées.

Là encore un **ApiController** se charge de commander les connecteurs et d'agréger les résultats.

L'application web

Backend

Elle est basée sur 3 vues:
- **index**: permet d'accéder à la page unique de l'application
- **process**: permet d'appeler le **SearchConductor** qui lui même va successivement appeler le **ParsingController** puis l'**ApiController**. Le résultat structuré au format JSON est ensuite renvoyé.
- **sentences** : renvoie juste une phrase aléatoire de GrandPyBot pour introduire le lancement de la recherche.

Frontend

Le CSS se base sur bootstrap dans sa version 4 et sur un petit bout de CSS personnalisé.
Tout le code javascript est écrit avec la syntaxe ES6. Jquery n'est chargé que pour l'utilisation de bootstrap.

Test Driven Development

Les principes du TDD ont été respecté dans 90% des cas.
Ce qui veut dire que le cycle de développement de chaque fonctionnalité a été le suivant:
1. Création d'une nouvelle branche à partir de la branche **development**.
2. Écriture d'un test.
3. Lancement et échec du test.
4. Écriture du code pour faire fonctionner le test.
5. Amélioration du test.
6. Retour à l'étape 3 tant que c'est nécessaire.
7. Lancement de tous les tests du projet.
8. Correction si nécessaire.
9. Commit de la fonctionnalité.
10. Fusion avec la branche **development**

Utilisation d'un bac à sable

Comme toujours, j'ai un fichier **sandbox.py** qui me permet de construire mon "proof of concept" à chaque étape du projet, tester des librairies… Bien entendu ce fichier n'est pas versionné.

Difficultés et solutions

Blueprint puis machine arrière

Au début du projet, j'ai considéré les **parsers**, **api connectors** et **word file handler** comme des applications à part entière.

J'ai d'abord pensé à créer plusieurs applications **Flask** sous forme de micro services mais ce type d'architecture m'a finalement semblé un peu démesuré au vue de la taille du projet. Cela peut tout de même être une piste intéressante si le projet doit grandir.

J'ai ensuite découvert le concept des **Blueprints** propre à **Flask**. Il s'agit finalement de créer des sous applications dans l'application en se basant sur un système de "namespace". J'ai donc commencé à structurer mon application en utilisant les **Blueprints**.

Finalement, je me suis rendu compte que j'apportais de la complexité à là où il n'y en avait pas besoin. Les **parsers** et **api connectors** ne sont pas des applications à part entière. Ils font partie intégrante de l'application principale car celle-ci ne peut absolument pas fonctionner sans eux.

J'ai donc fait machine arrière et complètement restructuré mon application.

Optimisation de la phase de *"parsing"*

La première version du **parsing controller** remplissait son rôle mais n'était absolument pas optimisé. Plus de 10 secondes pour parser une simple phrase.

Après réflexion, j'ai trouvé le problème ! Mon code appelait plusieurs fois la même requête SQL.

Le résultat ne changeant pas d'un appel à l'autre (données relativement statiques), il m'a donc paru plus que judicieux de réduire ces nombreux appels à la base de données à un unique appel donc le résultat serait stocké dans une variable et réutilisé par les différents **parsers**.

Résultat: le temps total entre la question de l'utilisateur et la réponse n'excède pas 1s!

Une base de données trop grosse pour Heroku

Le plan gratuit pour les bases de données sur Heroku va jusqu'à 10000 lignes au total. Autant dire qu'avec mes 450233 lignes, j'étais bien au delà. J'ai donc eu une alerte lorsque j'ai déployé le projet. Certains forum indique qu'Heroku coupe l'accès au projet après quelques heures dans ce cas.
Autant dire que je n'étais pas très rassuré à quelques jours de ma soutenance.
Heureusement après des recherches plus approfondies, j'ai compris que Heroku coupe l'accès en écriture sur la base de données mais laisse l'accès en lecture. Pour ce projet, ça me suffit.
Pour info: https://devcenter.heroku.com/articles/heroku-postgres-plans#hobby-tier

Pistes d'amélioration

- Refactoring du code javascript et séparation en plusieurs fichiers.
- Utilisation de webpack pour le javascript et le css.
- Utilisation de babel pour transformer le code javascript ES6 en code ES5.
- Rendre les parsers asynchrones.
- Optimiser la taille des assets.
- Fractionner le template index en plus petites parties réutilisables.