STRUCTURE MVC

Projet 9 – Le Baluchon

RAPPORT

Présentation de la structure MVC et comment communiquent-ils entre eux.

Mickaël HORN

Étudiant sur le parcours « Développeur d'Applications iOS » - OpenClassrooms

Table des matières

Introduction

Il existe plusieurs MVC dans l'application LeBaluchon.

Pour un souci de clarté, nous allons les diviser en deux parties :

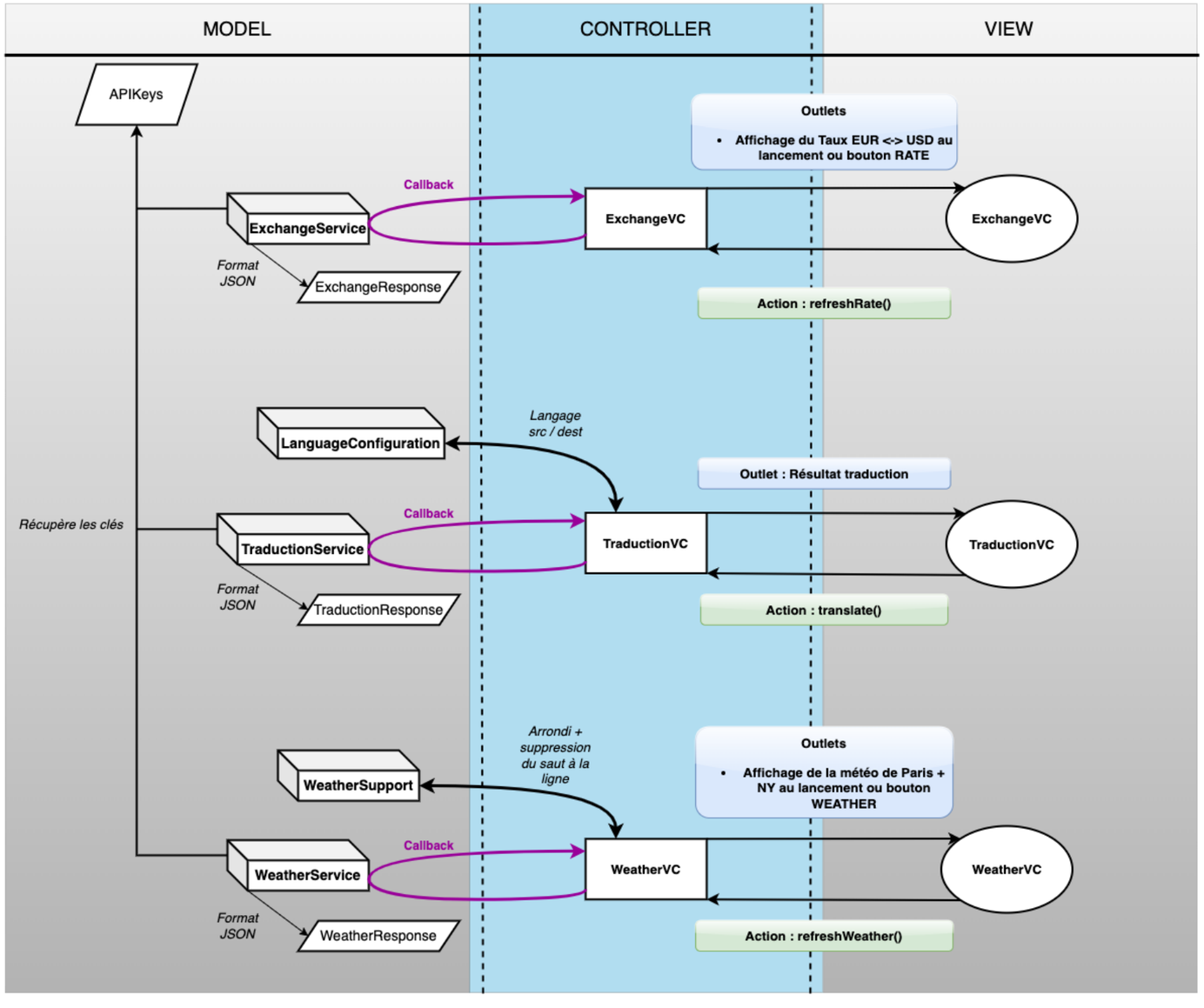
1. MVC principaux (concernant les appels API)
2. MVC secondaires

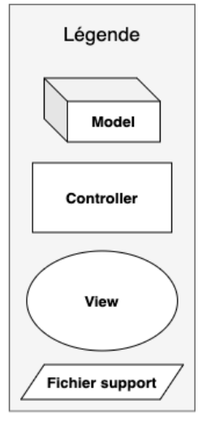
Pour chacune des parties, un schéma sera présenté ainsi que des explications les concernant.

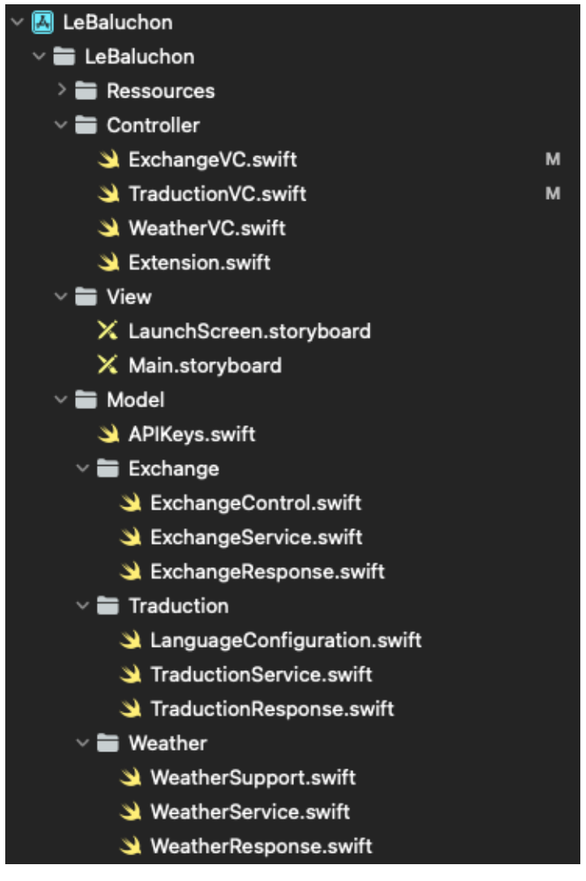
MVC Principaux

Nous sommes dans le cœur du projet, ce sont les MVC liés aux appels API.
Nous allons les décortiquer par fonctionnalités, c'est-à-dire une partie taux de change, traduction et météo.

Schéma et Arborescence







MVC Exchange

Tout d'abord, le MVC concernant la fonctionnalité du taux de change.

Quand l'utilisateur arrive sur l'application, c'est le premier écran auquel il a à faire.
Dès son arrivée, il se voit afficher le taux de change EUR vers USD, et vice-versa (le Contrôleur demande au Modèle le taux, via son viewDidLoad()).
Il peut également rafraîchir ce taux en appuyant sur le bouton RATE.

Dans les deux cas, il s'agit de la même opération.

La Vue communique alors avec le Contrôleur via une Action qui correspond à l'appui de ce bouton (refreshRate()).
Le Contrôleur (ExchangeVC) va alors demander au Modèle (ExchangeService) le nouveau taux de change.

Comme pour les autres MVC, ExchangeService va alors lancer l'appel API grâce à ses fichiers de supports tels que APIKeys, où il va aller chercher sa clé API (fixer.io) et ExchangeResponse, qui contient le format JSON du résultat que l'on attend, nécessaire pour la réception de celui-ci.

Une fois le résultat obtenu, le Modèle l'envoie au Contrôleur via un callback.
Le Contrôleur va finalement, au moyen d'Outlets, afficher les nouveaux taux de change.

MVC Traduction

Cette fois, l'utilisateur rentre le texte de son choix en français dans une textView et appuie sur le bouton TRANSLATE pour obtenir la traduction en anglais (l'inverse est également possible en cliquant sur le bouton avec les flèches afin d'échanger les langues).

La Vue communique avec le Contrôleur (TraductionVC) via une Action lancée par le bouton CALCULATE pour demander la traduction, en transmettant le contenu de la textView.

Le Contrôleur réceptionne l'Action et demande au Modèle (TraductionService) la traduction demandée.

Comme cité dans le précédent point, la même logique est appliquée ici.
Le Modèle va réaliser l'appel API ou il disposera du fichier APIKeys pour sa clé API (Google Traduction) et du fichier TraductionResponse afin d'assurer la bonne réception du JSON.

Il y a également une communication s'effectuant entre le Contrôleur et un autre Modèle (LanguageConfiguration).
En effet, nous avons besoin de connaître le langage source et destination et c'est ici que l'on va obtenir l'information.

La traduction étant obtenue, le Modèle l'envoie au Contrôleur au moyen d'un callback, qui va afficher grâce à un Outlet, la traduction que l'utilisateur a initialement demandée.

MVC Météo

À son arrivée sur l'onglet Météo, l'utilisateur se voit normalement afficher la météo de Paris ainsi que de New York.

Il peut manuellement faire la même chose en appuyant sur le bouton WEATHER, qui déclenchera l'Action chargée de rafraîchir les données météos.

Le contrôleur (WeatherVC) demande au Modèle (WeatherService) de lui fournir la météo des deux villes.

Une fois n'est pas coutume, le Modèle réalise l'appel API, s'appuie sur ses fichiers APIKeys et WeatherResponse pour chercher sa clé API (OpenWeatherMap) ainsi que le bon format du fichier JSON qu'il s'apprête à recevoir.

Une autre communication a lieu, celle du Contrôleur et d'un autre Modèle (WeatherSupport).
Cela va nous permettre d'arrondir les températures à l'entier le plus proche, ainsi que de supprimer le dernier saut à la ligne.

Le dernier point peut paraître un peu flou mais est en réalité simple.
En effet, lorsque j'affiche les descriptions météos (pluvieux, ensoleillé, etc.), je les concatène dans la propriété « text » d'un Outlet, en prenant soins de rajouter un saut de ligne pour espacer chacune des descriptions.

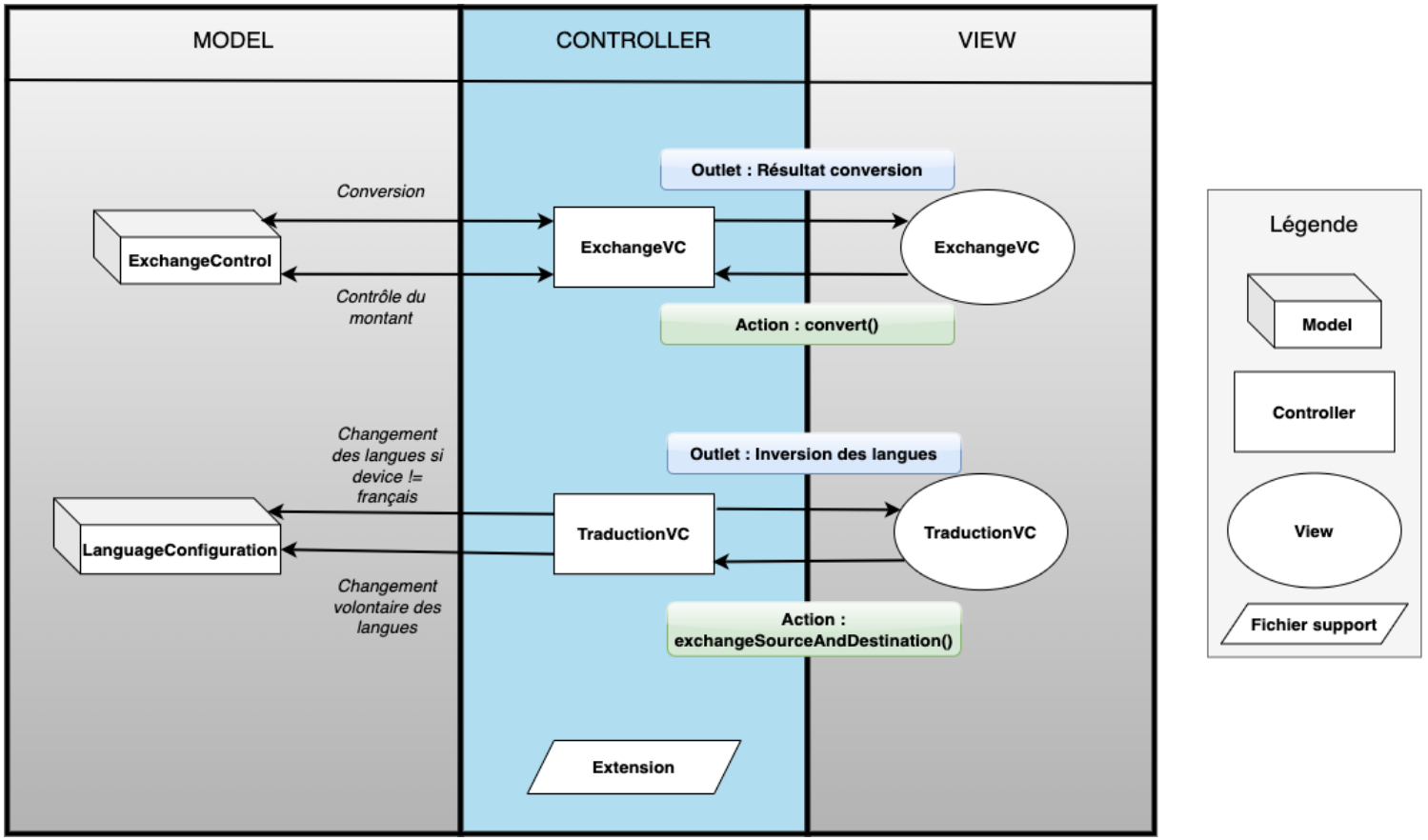
Arrivé à la dernière, je me dois d'enlever ce saut de ligne inutile !

Une fois les données reçues, le Modèle, grâce au callback, envoie la réponse au Contrôleur qui modifie la Vue grâce aux Outlets qui afficheront la météo à l'utilisateur.

MVC Secondaires

Cette partie entre en détail dans les MVC dits « secondaires » présentant le reste des fonctionnalités.
Nous y retrouverons certains Contrôleurs et Vues abordées au point précédent.

Schéma



MVC Exchange

Quand l'utilisateur aura obtenu les taux de changes EUR <-> USD, fonctionnalité vue au chapitre précédent, il peut maintenant demander la conversion d'un montant de son choix.

Il rentre alors son montant en euros dans une zone de texte, et clique sur le bouton CONVERT.

L'action du bouton CONVERT va alors demander au Contrôleur (ExchangeVC) le résultat en dollars.

Plusieurs choses se passent ici.

La première est que le Contrôleur va demander au Modèle (ExchangeControl) de vérifier si le montant saisi par l'utilisateur est correct.
C'est-à-dire qu'il y en a bien un dans un premier temps, puis qu'il soit correct dans sa forme.

La deuxième est que si le contrôle du montant a été un succès, le Contrôleur peut alors demander au Modèle de convertir le montant initialement en euros, en dollars en prenant bien entendu dans le calcul, le taux de change.

Le Contrôleur affiche alors le résultat à l'utilisateur sur la Vue.

MVC Traduction

Pour la fonctionnalité de traduction, l'utilisateur peut inverser les langues, et ainsi demander une traduction inversée (Bonus choisis, voir le document
« Horn_Mickael_2_bonus_102022 »).

En cliquant sur le bouton en forme de flèches, le Contrôleur appelle le Modèle (LanguageConfiguration) pour échanger les langues sources et destinations.

La Vue se met alors à jour avec la bonne configuration.

Pour terminer, en fonction de la langue du téléphone de l'utilisateur, l'interface s'adapte automatiquement.

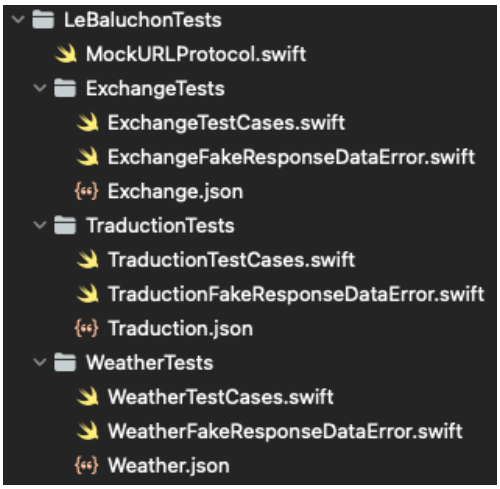
Si la langue est le français, alors la configuration par défaut ne bouge pas (source = français, destination = anglais).

En revanche, si le device est en anglais, on inverse automatiquement les langues pour que l'anglais devienne la langue source, et le français celle de destination.

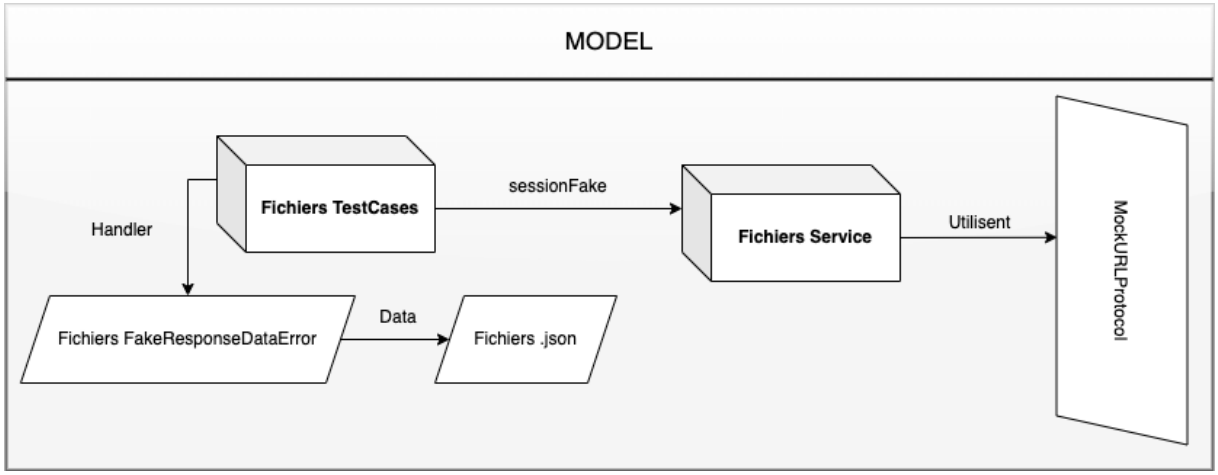
Les tests unitaires

Ce ne sont pas à proprement parler de MVC complets, ils ne disposent ni de Contrôleurs, ni de Vues.
Cependant, ils restent assimilables à des Modèles et il est nécessaire que l'on s'y intéresse.

Arborescence



Schéma



Détails

Chaque fonctionnalité a son dossier associé, composé des fichiers TestCases (les tests), FakeResponseDataError (que l'on va utiliser pour nos tests) et les fichiers .json (où se trouve le résultat d'un appel que FakeResponseDataError récupèrera pour alimenter le paramètre Data).

Lorsqu'on réalise des tests unitaires dans une application iOS, nous ne pouvons pas nous permettre de dépendre de la qualité du réseau et ainsi, d'attendre que les réponses arrivent.

Pour cela, il nous faut lancer les appels API mais sans vraiment les lancer.
C'est-à-dire qu'on va intercepter le lancement qui normalement est assuré par la méthode dataTask d'URLSession.

MockURLProtocol est là pour ça, c'est lui qui va se charger de l'appel et faire en sorte qu'il ne se lance jamais.
Pour cela, nous allons lui fournir les paramètres Data, Response et Error, qui normalement sont réceptionnés à la suite d'un appel.

Déroulement des tests unitaires

Tout d'abord, on rédige notre test en prenant soin de laisser la main à notre MockURLProtocol, chargé d'intercepter l'appel.

Ensuite, on fournit les réponses souhaitées à notre MockURLProtocol par le moyen du Handler (qui contient Response, Data et Error).

Donc si par exemple on veut tester le cas où un appel se déroule sans encombre, on indiquera à notre Handler une réponse valide, des bonnes données et aucune erreur.
Ces éléments sont contenus dans les fichiers FakeResponseDataError et pour le cas des data, ils iront les chercher dans les fichiers .json.

Via notre SessionFake, on lance l'appel API des Modèles (les fichiers Service).
Grâce à MockURLProtocol, l'appel n'a pas lieu et à la place, il va transmettre le Handler à la fonction qui gère l'appel, et celle-ci va s'exécuter normalement.