STRUCTURE MVC

Projet 9 – Le Baluchon

RAPPORT

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

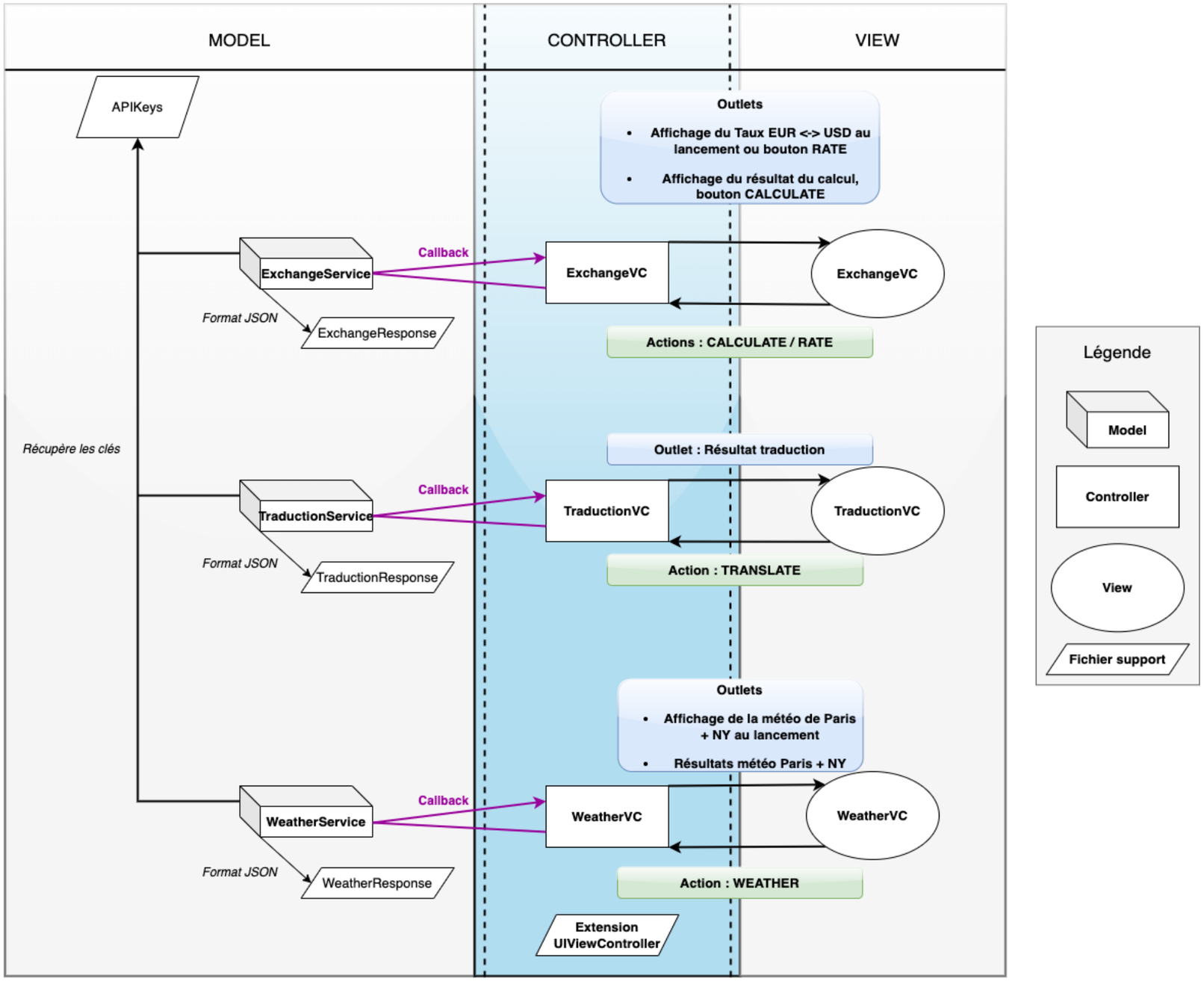
Mickaël HORN

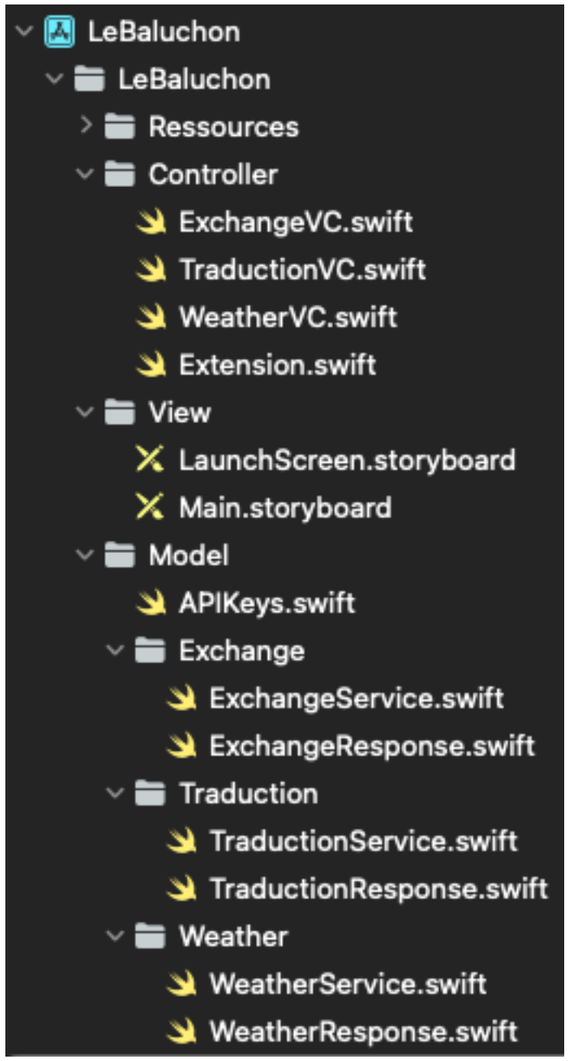
Étudiant sur le parcours « Développeur d'Applications iOS » - OpenClassrooms

Structure MVC

Comme nous pouvons le constater, il existe trois principaux MVC dans l'application LeBaluchon.
Nous allons regarder en détail chacun d'entre eux.

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 model le taux, via son viewDidLoad()).

Il peut également rafraîchir ce taux en appuyant sur le bouton RATE.
La Vue communique alors avec le Contrôleur via une Action qui correspond à l'appui de ce bouton.
Le Contrôleur (ExchangeVC) va alors demander au Model (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 Model l'envoie au Contrôleur via un callback.
Le Contrôleur va finalement, au moyen d'Outlets, afficher les nouveaux taux de change.

L'utilisateur peut bien évidemment demander, via un montant en euros, son équivalent en dollars grâce au bouton CALCULATE.

Le contrôleur va alors se servir du taux de change précédemment obtenu et afficher le résultat du calcul à l'utilisateur.

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 l'envoie au Model (TraductionService).

Comme cité dans le précédent point, la même logique est appliquée ici.
Le Model 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.

La traduction étant obtenue, le Model 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.

Le bouton WEATHER lui permet de rafraîchir les données.

Ce bouton déclenche l'Action qui demandera au Contrôleur (WeatherVC) la nouvelle météo.
Le contrôleur demandera au Model (WeatherService) qui réalisera l'appel API afin d'obtenir la météo la plus récente.

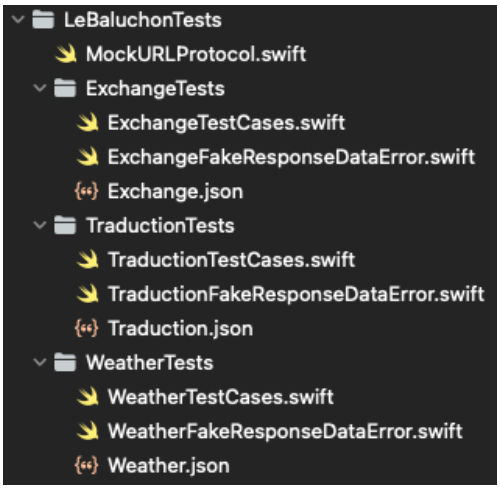
Une fois n'est pas coutume, le Model réalise l'appel API, s'appuie sur ses fichiers APIKeys et WeatherResponse pour chercher sa clé API (OpenWeatherMap) ainsi que le bon format du fichier JSON qu'il s'apprête à recevoir.

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

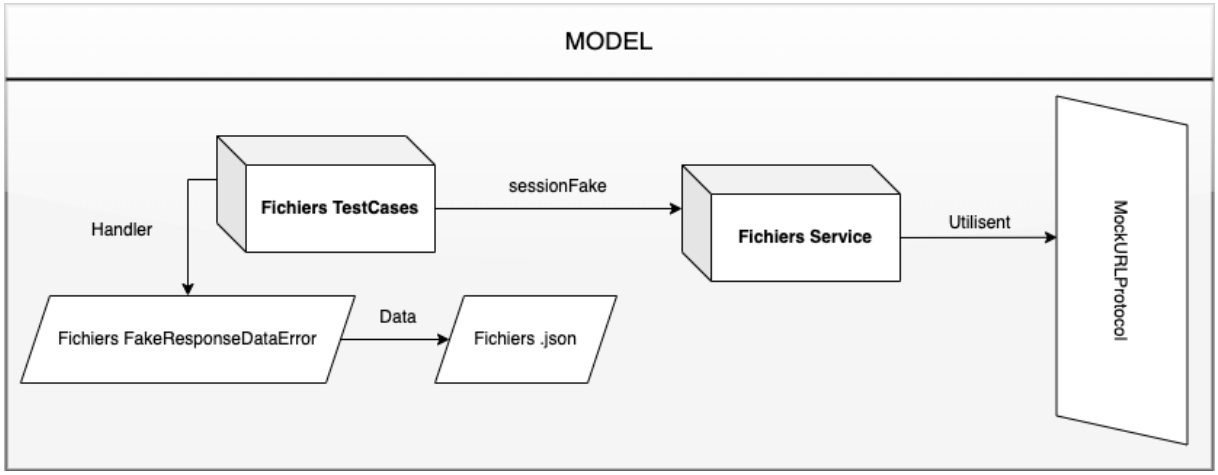
Les tests unitaires

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 Model 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 Model (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.