Séparation des Layers.

Pour commencer, il faut comprendre le découpage qu'on effectue entre la récupération des datas (**DataAccess** et le traitement de la data (**Business**).

Si on prends comme exemple le cas ou on veut utiliser l'API de Netflix pour afficher la liste des différents médias.

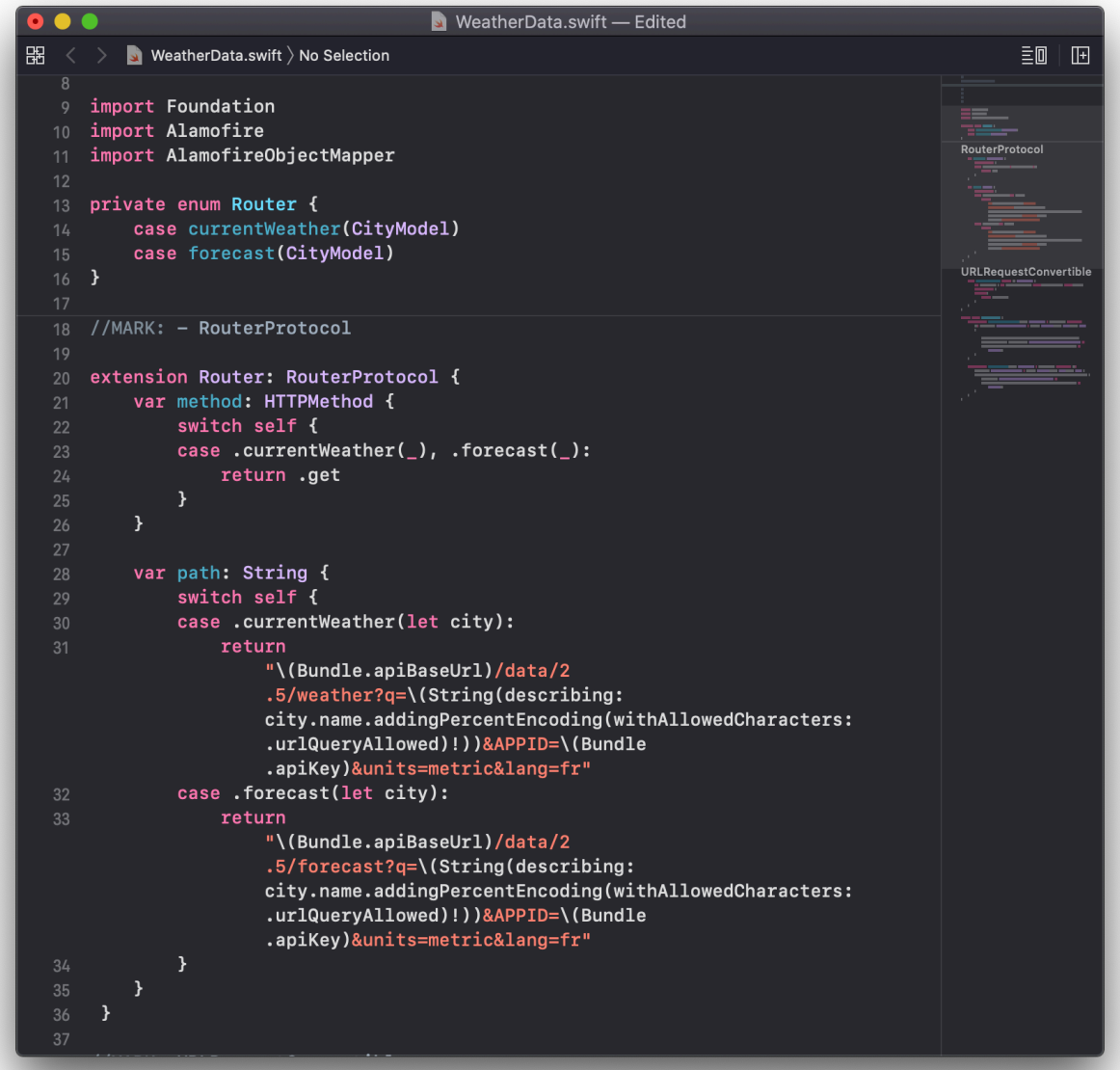
Admettons que l'api nous retourne uniquement un gros fichier JSON contenant toutes les datas, si on se base sur le *MVC (Model, View, Controller)* traditionnel d'Apple, si on veut créer plusieurs fonctions qui permettent de faire un tri par genre, par type de média (film, série, etc…) ou par durée, le traitement pourra seulement être effectué dans le **Controller** sauf que celui-ci se retrouvera vite surchargé ce qui sera lourd et peu lisible par toi et aussi par les autres développeurs.

DataAccess

Cette classe nous permet de récupérer les datas via le Framework *Alamofire.*

Prenons l'example du SwiftStarter:

Router




```
import Foundation
import Alamofire
import AlamofireObjectMapper

private enum Router {
    case currentWeather(CityModel)
    case forecast(CityModel)
}

//MARK: - RouterProtocol

extension Router: RouterProtocol {
    var method: HTTPMethod {
        switch self {
        case .currentWeather(_), .forecast(_):
            return .get
        }
    }

    var path: String {
        switch self {
        case .currentWeather(let city):
            return
                "\(Bundle.apiBaseUrl)/data/2
                .5/weather?q=\(String(describing:
                city.name.addingPercentEncoding(withAllowedCharacters:
                .urlQueryAllowed)!))&APPID=\(Bundle
                .apiKey)&units=metric&lang=fr"
        case .forecast(let city):
            return
                "\(Bundle.apiBaseUrl)/data/2
                .5/forecast?q=\(String(describing:
                city.name.addingPercentEncoding(withAllowedCharacters:
                .urlQueryAllowed)!))&APPID=\(Bundle
                .apiKey)&units=metric&lang=fr"
        }
    }
}
```

Le Router (cf: https://github.com/Alamofire/Alamofire/blob/master/Documentation/AdvancedUsage.md#routing-requests) nous permet de définir les routes que nous allons utiliser lors des appels à une API.

Dans cet exemple, nous allons faire 2 Appels de type *GET* qui sont la récupération de la météo actuelle (currentWeather) et la prévision météo (forecast).

Nous définissons donc un enum contentant les *cases currentWeather* et *forecast.*

Ensuite, afin de définir quel méthode de requête (method) nous allons utiliser ainsi que l'url de notre requête (path), on crée une extension de notre *Router* qui se conformera au protocole *RouterProtocol.*

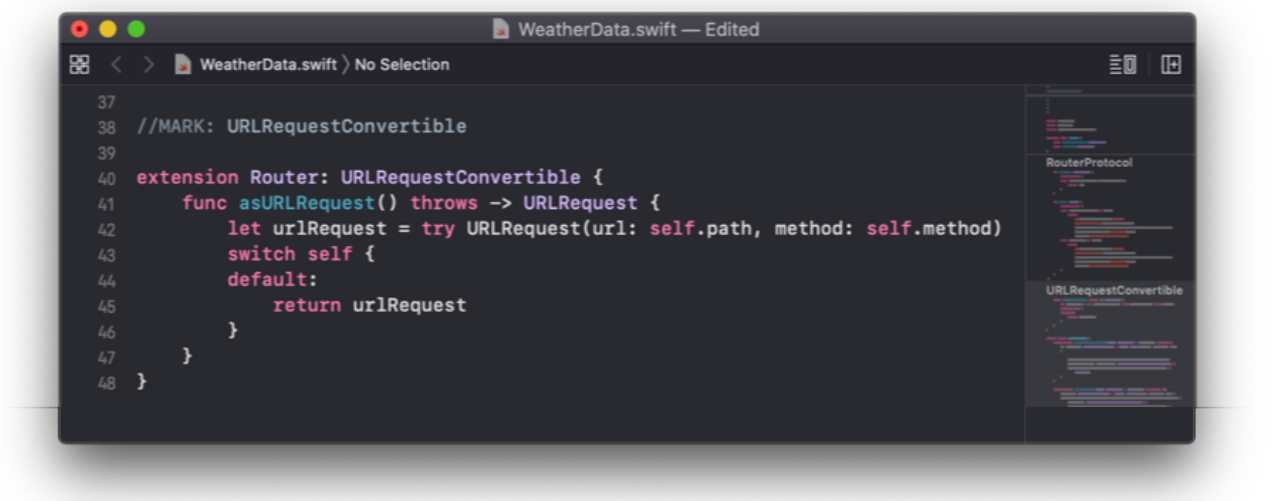
Method

- On retourne la méthode à utiliser en fonction de la requête voulue (case).

Path

- On retourne l'url à utiliser en fonction de la requête voulue (case) en passant si il le faut une variable qui sera utilisée dans l'url.

Enfin, on crée une second extension de notre *Router* qui se conformera au protocole *URLRequestConvertible.*

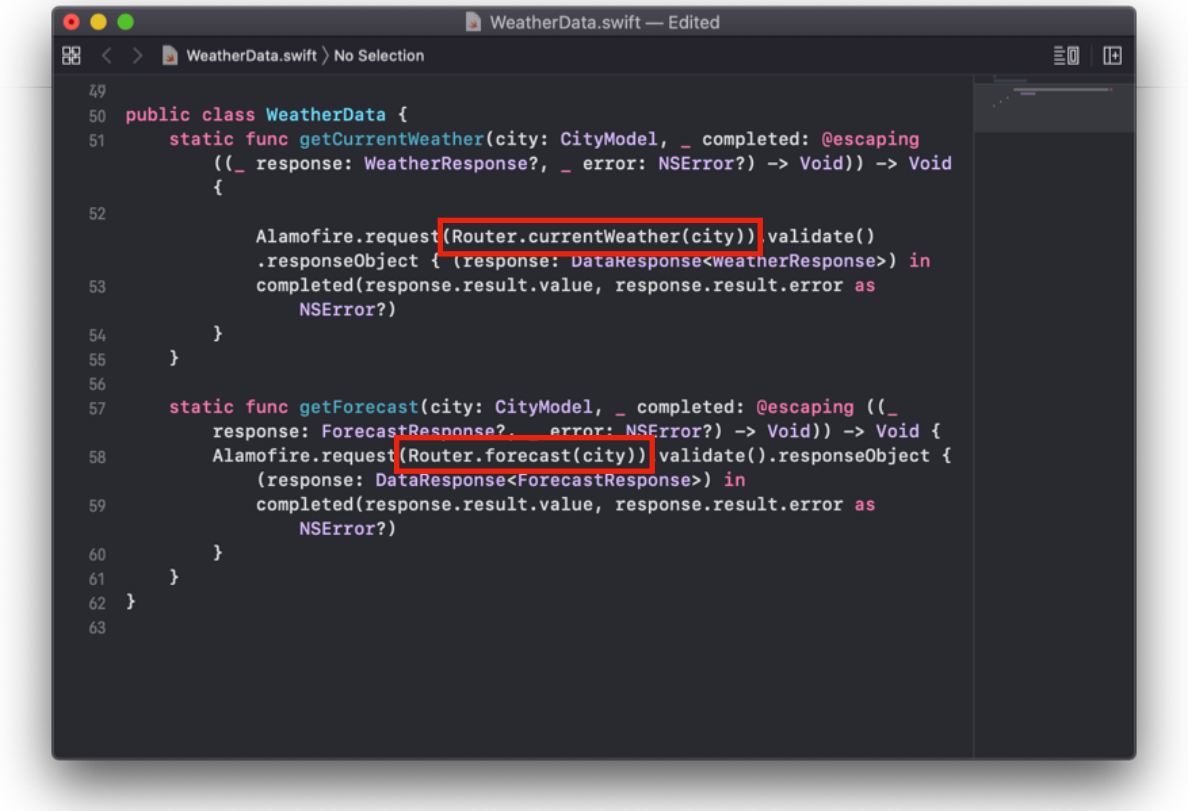


Cette extension nous permet de définir une fonction *asURLRequest* qui permet de créer notre URLRequest.

Notre *URLRequest* utilise le *path* et la *method* que nous avons défini à l'extension du *RouterProtocol.*

La classe Data

C'est dans cette classe que nous allons définir les différentes méthodes qui seront appelées par notre Business.



Les méthodes que nous définissons ici peuvent prendre des variables en paramètre et elles doivent obligatoirement prendre en paramètre un *completionHandler.*

Ce *completionHandler* prendra en paramètre optionnel une réponse et une erreur.

Pour faire un appel à notre Api, nous avons juste à appeler la méthode request d'Alamofire avec comme paramètre un des cases que nous avons défini dans notre Router.

La méthode *request* d'Alamofire nous retournera une DataResponse qui sera mappé à notre Model via AlamofireObjectMapper.

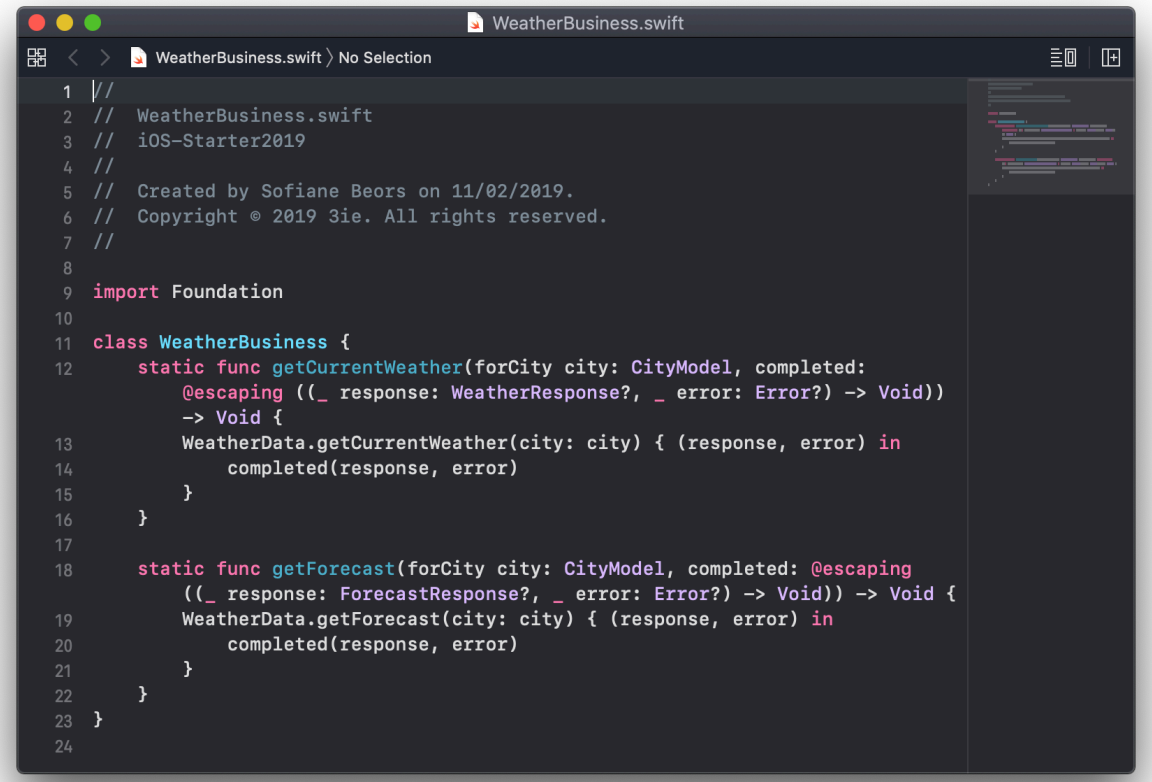
Nous avons donc juste à passer à notre *completionHandler* la response mappé que nous avons reçu et l'erreur le cas échéant.

Business

Cette classe nous permet de faire du traitement sur nos datas, on l'appelle donc la couche Logique.

C'est dans cette classe que nous allons définir les différentes méthodes qui seront appelées par notre ViewController.

Les méthodes que nous définissons ici appellerons les méthodes correspondantes définis dans notre DataAccess et prendront elles exactement les mêmes paramètres.



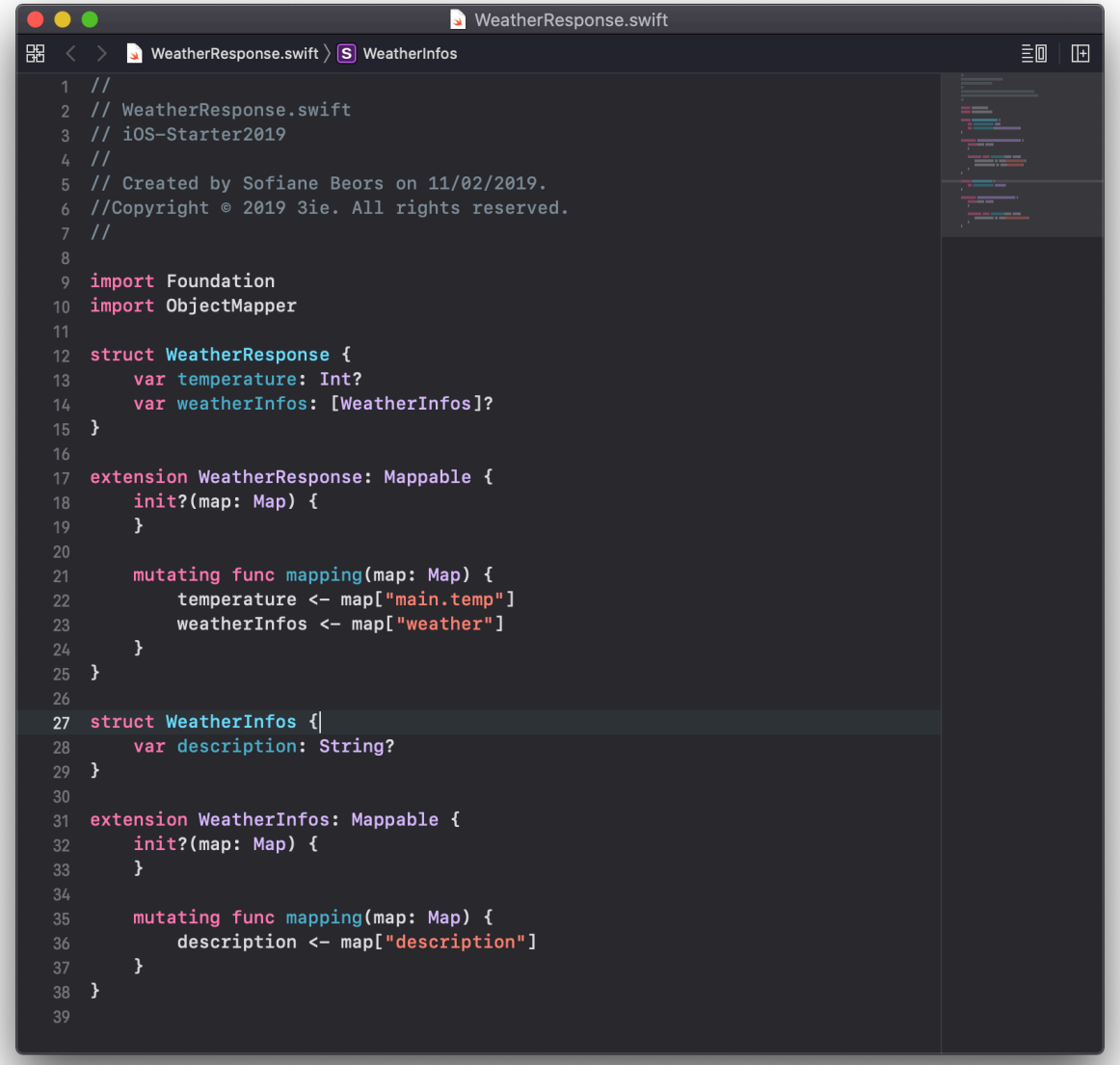

```swift
//
//  WeatherBusiness.swift
//  iOS-Starter2019
//
//  Created by Sofiane Beors on 11/02/2019.
//  Copyright © 2019 3ie. All rights reserved.
//

import Foundation

class WeatherBusiness {
    static func getCurrentWeather(forCity city: CityModel, completed:
        @escaping ((_ response: WeatherResponse?, _ error: Error?) -> Void))
        -> Void {
        WeatherData.getCurrentWeather(city: city) { (response, error) in
            completed(response, error)
        }
    }

    static func getForecast(forCity city: CityModel, completed: @escaping
        ((_ response: ForecastResponse?, _ error: Error?) -> Void)) -> Void {
        WeatherData.getForecast(city: city) { (response, error) in
            completed(response, error)
        }
    }
}
```

Nous appelons donc la méthode correspondante de notre Data Access dans chaque méthode et nous passons dans notre *completionHandler* la réponse que nous avons reçu et l'erreur que nous avons reçu.

C'est ici que nous allons par exemple créer des méthodes pour filtrer notre contenu et toute la partie algorithmique liée au datas se feront dans cette **classe.**

Model

Cette structure nous permet de convertir les propriétés contenues dans le JSON de notre réponse en une classe bien structurée grâce à l'aide du Framework ObjectMapper.



Dans cet exemple, nous souhaitons obtenir de notre JSON la température ainsi qu'un tableau contenant toutes les informations relatives à la météo.

Pour ce faire, nous avons juste à définir une structure contenant comme variable le nom des propriétés que nous souhaitons associé au type de la valeur.

Mappable

Afin que le framework se charge de parser correctement notre JSON et d'y associer les propriétés aux bonnes variables, nous devons nous conformer au protocole *Mappable.*

```
17  extension WeatherResponse: Mappable {
18      init?(map: Map) {
19      }
20
21      mutating func mapping(map: Map) {
22          temperature <- map["main.temp"]
23          weatherInfos <- map["weather"]
24      }
25  }
```

Cette extension nous donne droit à 2 méthodes.

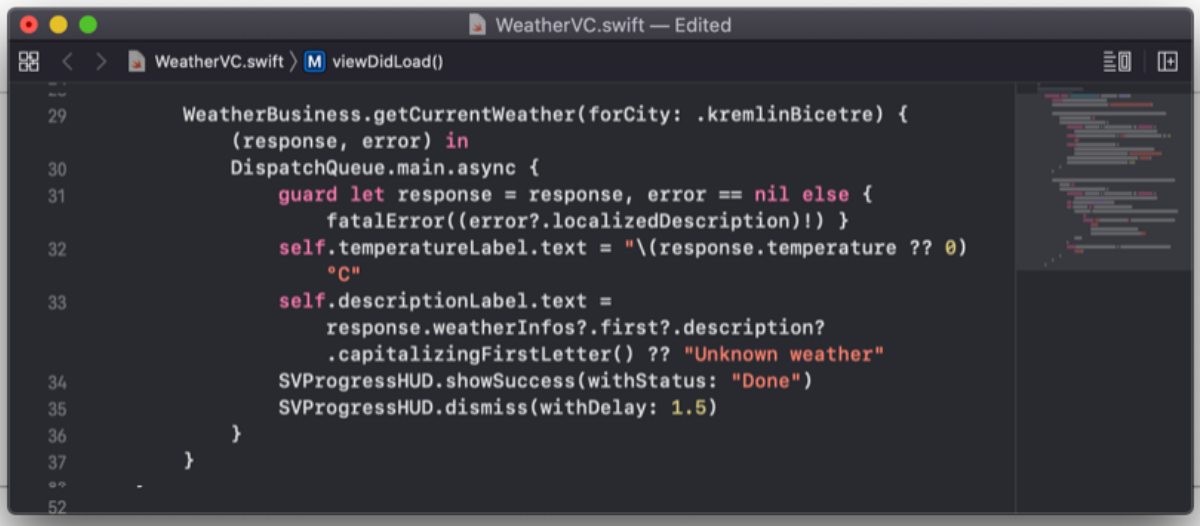
- *init* qui est la fonction d'initialisation de la structure
- *mapping qui est la fonction où nous allons attribuer nos propriétés à nos variables.*

La methode *mapping* est une méthode dite *mutating,* cet à dire que cette méthode peut modifier la valeur de la structure. (cf: https://docs.swift.org/swift-book/LanguageGuide/Methods.html).

Pour faire correspondre les propriétés de notre JSON à nos variables, nous avons besoins de renseigner comme clé la propriété que nous souhaitons. ObjectMapper permet également la notation par points dans les clés pour un mappage facile des objets imbriqués (nested objects).

ViewController

Lorsque nous voulons récupérer des datas dans notre ViewController, nous avons donc simplement à appeler la méthode que l'on souhaite de notre Business et la data que nous allons recevoir sera exactement celle que nous attendons et nous n'aurons pas de traitement à faire dessus si nécessaire car nous l'avons fait précédemment dans notre classe Business.



Lors de l'appel à la méthode voulue dans la *Business,* nous nous retrouvons avec une cl*osure* qui donne accès à nos datas (*response* et *erreur*) via notre *completionHandler.*

Nous pouvons enfin manipuler nos datas en accédant à leurs propriétés via la variable response.