Weather Keynote spitch

Slide 1 :

Introduction

This is a Weather application for the city of Paris for now but I willingly created an other screen view to work on the selection of other cities in the future.
I have used the OpenWeather API to display the weather data.
The data is saved in CoreData so the application is working off line too.
The architecture of this project is following the MVVM-C pattern rules.

Slide 2 :

Application simulation

Slide 3 :

What is MVVM-C pattern ?

The Model-View-ViewModel, is part of a large family of models known by the prefix MV, such as the MVC (Model View Controller) or MVP (Model View Presenter).
Each of these models aims to separate the logic from the user interface in order to make applications much more scalable, easier to maintain and above all to test.
For this project, I decided to choose the native reactive MVVM-C.

At first it's an MVC pattern with an additionnal layer.
The problem with MVC is that you can end up with gigantic ViewControllers.
The best way to test the model is to extract the information in intermediate layers.
This is where the MVVM appears using an additional layer called the ViewModel, which can be easily tested.

Slide 4 :

The approach used :

1 - Creation of two modules : "HomeWeather module" including a detail view "DetailWeatherDay" and "SelectCity module" using the Coordinator and the storyboard of each screen view.
2 - Implementation of a « UITabBar » with 2 items:
. home
. selectCity
3 - Creation of the network with an "HTTPClient", an "HTTPEngine" and a "Token"
4 - Creation of the model structure into WeatherItem.swift and adding of WeatherObject attributes into Core data.
5 - Writing of WeatherRepository.swift
6 - Injection of Core data in the context and the context in the appDelegate.
7 - Unit tests
8 - UI tests

Slide 5 :

How does logic communicate with the view ?

Each screen view is constructed using a ViewController and a ViewModel.

- **ViewControllers** only exist to control the view, they are clear, simple, and don't know data. ViewControllers subscribe to the reactive variables of ViewModels by the « bind() » function, called in the « viewDidLoad() ».

In this application we have 3 viewControllers:

- « WeatherViewController » for the HomeWeatherView
- « DetailWeatherDayViewController » for the DetailWeatherDayView
- « SelectCityViewController » for the SelectCityView

- **ViewModels** encapsulate the whole logic of each screen view that doesn't have to be in the ViewController, divided in two parts:
  - **Inputs**: Each event from the ViewController are implemented in the ViewModel, and added in the « viewDidLoad() » function.
  When an event is sent by the closure, the view is intrinsically updated.
  The view subscribes to reactive variables then sends the first input which often is the « viewDidLoad() » : the main input connection for reactive variables.
  - **Outputs**: The viewModel is providing reactive variables for each data/state and user interactions needed.
  They are receptacles of events from the corresponding inputs.
  After « viewDidLoad() », ViewController is listening for some changes or user interactions from the ViewModel.
  For each Viewmodel which need a Repository, it needs to inject a RepositoryType in order to mock it for the Unit Tests.

The UI logic of ViewControllers is separate from ViewModels logic.

In this application we have 3 ViewModels:

- « WeatherViewModel » for the HomeWeatherView
- « DetailWeatherDayViewModel » for the DetailWeatherDayView
- « SelectCityViewModel » for the SelectCityView

Slide 6 :

What does « Reactive Native » means ?

- Reactive = The view subscribes to closures, a closure is a variable whose type is a function. So we create reactive variables in the « ViewModel » and on the « ViewController » side, we subscribe to these closures so to a state.
- Native = It means that the application is coding without an external library. Different from MVVM RX for example.

+ Slide 7, 8, 9 : Inputs / Outputs of each view

Slide 10 :

The OpenWeather API

The OpenWeather API url is receiveing data of 5 days weather every 3 hours.
The model structure « WeatherItem » has been created to show the time, temperature, icon image and plus, the temperature min/max, pressure, humidity, feelsLike and sky description.

Slide 11 :

What is Core data ?

Core Data is a local database, provided with an object oriented API.

It stores data in a clear and organized way in a database, locally, inside the phone (and not on a remote server). It allows data organized by the entity-attribute relational model to be serialized in XML, binary or SQLite stores. Data can be manipulated using top-level objects representing entities and their relationships.
It allows developers to easily manipulate objects and not manipulate SQLite directly.

In this application, we just needed to create entity's name object in the « weather.xcdatamodeld » file and add its attributes.
A single entity comprising 9 attributes with « String » type has been created to save current weather data inside the phone.
WeatherObject includes: descriptionWeather, feelsLikeWeather, humidityWeather, iconWeather, pressureWeather, tempMaxWeather, tempMinWeather, tempWeather, timeWeather.

Slide 12 :

The application architecture diagramme and layers' description

I - Sources

- AppDelegate.swift

The appDelegate is the entry point of the app.
It instanciates a « UIWindow » optional, a Context and an « AppCoordinator » which managed the whole navigation.
Its « application() » function is implementing the « Context », the « HTTPClient » and the « CoreDataStack » to create the « AppCoordinator ». We need to create the « AppCoordinator » to call the main function « appCoordinator?.start() »
It starts the application, presenting the main view.

- Context.swift

The Context is the main object injected everywhere in the app. It has the responsability to provide dependencies injected first in the `AppDelegate`. Every new dependency should be injected through it. In this app, this layer instanciates the « HTTPClientType » and the « CoreDataStack ».

- CoreDataStack.swift

This layer instanciate an « NSPersistentContainer » and a public variable named « context » with « NSManagedObjectContext » type which return a « viewContext ».
CoreDataStack needs to initialize a name and a type. It needs an « NSPersistentStoreDescription».

If we want to be sure that the context writes the data to the production database, we will have to save the context. If the type of « CoreDataStackType » is **.test,** we use a custom description, if not

the default type is **.prod** and we use the default description. We will therefore save the context in the "persistentContainer" which is an instance of the production database.
The « saveContext() » function is here to save the data, there is no point in saving the context if there is no change.

II – Navigation

➢ Screens.swift

The responsibility of the « Screens » class is to show the ViewController. It needs to instanciate a « Context » and a « CoreDataStack ».
« Screens » uses the "Main" storyboard and instantiates a ViewController in the storyboard.
It is responsible for creating different views via the Coordinator of each screen view.
The advantage of having created a Coordinator for each item of the « tabBar » is to be able to develop the application with for example, the creation of a detail view as it was done for the WeatherView.

Coordinator

Coordinator is a separate entity responsible for navigation in application flows. With the help of the « Screens », it creates, presents and dismisses UIViewControllers, by keeping them separate and independent. Coordinator can create and run, child coordinators too. They can managed interraction using the delegate of each ViewModel.

Main Coordinator

➢ AppCoordinator.swift

In this app, we are using an AppCoordinator.swift which instantiate a « MainCoordinator », a « Context », an « AppDelegate » and a « CoreDataStack ».

It's « start() » function is initializing the rootViewController as « UIViewController().
It calls the « showMain() » function creating the « MainCoordinator » and calling the « mainCoordinator.start() ».
The « mainCoordinator.start() » is creating the « tabBarController » and the « WeatherCoordinator ». It's showing the home view calling the « showMainView() » function.

Child coordinators

Child Coordinators are responsible of the navigation of one screen view. They need to instanciate a « presenter » a « UINavigationController » and the « Screens » layer.

➢ WeatherCoordinator.swift

This Coordinator is responsible of the Home view navigation.
It uses 3 private functions.
The most important function is « start() » when start() is called, « showHome() » is called too and the corresponding Viewcontroller is created thanks to the « Screens » layer using « createMainViewController() ».
Then interractions are managed with the « WeatherViewModelDelegate » calling « showAlert() » into « displayWeatherAlert() » and « showWeatherDayDetail() » into « didSelect() ».

The detail view is created with « showWeatherDayDetail() » calling screens.createWeatherDetailViewController(selectedWeatherItem: weatherDay, delegate: self).

The weatherDay property is communicate by the « WeatherViewModelDelegate » with the « didSelect(item: WeatherItem) » function, where the « delegate » is **self** as « DetailWeatherDayViewModelDelegate » and has to be implemented inside an extension of « WeatherCoordinator » even if it doesn't call any function into it.

- SelectCityCoordinator.swift

This Coordinator is responsible of the SelectCity View navigation.
It uses 2 private functions.
The most important function is « start() » when start() is called, « showHome() » is called too and the corresponding Viewcontroller is created thanks to the « Screens » layer using « createSelectCityViewController () ».
There is no interraction created for now so we didn't created any extension.

III - Network

- HTTPEngine.swift

This class is created to make URL request.
The « send() » function is fetching **Json data** and parsing their response in `**Codable**` objects, handled by the « URLSessionEngine ».
Here, **(data, response or error)** is sent in callback.
It uses a « Token » RequestCancelationToken.swift. If the reactive token variable is called, at (willDealocate), the task is canceled. The reactive variable is called when the object is destroyed, so at « deinit() ».
The « Token » is a class which has a reactive variable called at the object « deinit() », it intervenes in the HTTPClient request.

- HTTPClient.swift

The role of the client is to decode the data from **Json** (or an other language) to an object format.
So the « HTTPClient» turns the **Json** into object.
In the « request<T>() » function, when « engine.send() » receives **(data, response, or error)**, the object is returned into the callback.

IV- Repository

The Repository corresponding to each view will make the data requests using the URL string.
It instantiates an « HTTPClient », a « Token » and a « CoredataStack ».

The « getWeather() » function calls « client.send()» returning a callback in two distinct cases.
If we pass in the **.success** case, the object is returned as a callback via a network request with the **.web** case.
If we pass in the **.error** case, either we pass in the **.dataBase** case, and the data is recovered from database or an error is returned.

Technical questions :

➢ What is the func « makeKeyAndVisible() » for ?

This is a convenience method to show the current window and position it in front of all other windows at the same level or lower.

➢ What is a reactive variable ?

A reactive variable is an optional function type variable that returns nothing. A function is something that can take an input parameter and that will have an output type.

➢ Why when we subscribe to a reactive variable we add the keyword [weak self] ?

Because our Viewcontroller instance is linked to an instance of the ViewModel.
When the Viewcontroller is built we instantiate a ViewModel so the **self** is the Viewcontroller's self of the instance.
We make sure that if **self** is destroyed, the link is destroyed too. Hence the usefulness of the optional self.
Hence the utility of using guard let to unpack it and make sure it exists.

➢ What is Expectation used for in tests ?

If the test is crashing, it allows us to check the logs and find the name of the expectation that does not work.

➢ What is DispachQueue.main.async used for?

When an event is asynchronous, so the result is not executed on the main thread.
The main thread is the thread for executing graphic elements.
So from the asynchronous thread which is automatically created when we make asynchronous requests, if we want to be able to display some items from the web for example, we ensures that its execution is well done on the main thread.
Everything that is asynchronous is therefore redefined on the same thread. And we call the function async.

➢ What is Swift Lint

It is a tool based on syntax guides, which can read the code to automatically and check its conformity. It will be able even to find the smallest space!

➢ How does subscript work her ?

Subscript rewrites the behavior of index of our "item" array, we pass an element of the ViewControllerItem (an enumeration) and return a UIViewController

```swift
extension TabBarSourceType {
    subscript(item: ViewControllerItem) -> UINavigationController {
        get {
            guard !items.isEmpty, item.rawValue < items.count, item.rawValue >= 0 else {
                fatalError("Item does not exists")
            }
            return items[item.rawValue]
        }
    }
}
```

- What could be the other way to create a tab bar without the subscript ?

I could create a TabBarViewController, which creates a UIViewController with a TaBarViewModel. Communicating with a TaBarDataSource.

- What is the new way to represent a Success or an Error ?

```
@frozen enum Result<Success, Failure> where Failure : Error
```

This declaration is a value that represents either a success or a failure, including an associated value in each case.

- What is a token for ?

A token is a class that is called only in certain cases.
When we scroll a collection, we launch a request too provide data.
Then, when the cell is no longer visible on the screen, we don't want to continue the request. It must therefore be canceled and we should save its data in cache.
We mustn't have lost requests.

So when we fetch data in a collectionview or tableView cell we make sure that the request is canceled thanks to the token.

In this app, the reactive variable is called when the object is destroyed, so at « deinit() » and the task is canceled.

- Why did you create a token ?

In this application, the token only intervene when the detailView is destroyed.

So when we select a weatherDay, the DetailWeatherDay is showing, then when we wanna come back to the HomeView, the DetailWeatherDay is destroyed.
So the data request will be cancel by the token at deinit().

- What is protocol equatable for ?

The Equatable protocol is what allows two objects to be compared using ==, and it's easy to implement because Swift does most of the work by default.

The == operator is defined by a protocol named « Equatable ».
So you can use the == operator between two objects only if they implement Equatable protocol.

This protocol put to use a function named « == ».
For example :

```
struct Item: Equatable {
   let name: String
   let description: String
}

 static func == (lhs: Item, rhs: Item) -> Bool {
          return lhs.name == rhs.name && lhs. description == rhs. description
  }
```

Here I should implement an other struct like VisibleWeatherItem on my ViewModel side to permit to use the equatable protocol in my unitTests.
In my test I will need to check if my WeatherItem is equal to my VisibleWeatherItem.

➢ What is Hashable protocol ?

This is a protocol that provides an integer hash value.
An Hash is an id from an algorythme which can generate a unique identifier from a mathematical rule.

We can use any type that conforms to the Hashable protocol in a set or as a dictionary key. Many types in the standard library conform to Hashable: strings, integers, floating-point and Boolean values.

For example :

```
struct iPad: Hashable {
   var serialNumber: String
   var capacity: Int
}
```

You can add more properties to your hash by calling combine()

```
func hash(into hasher: inout Hasher) {
   hasher.combine(serialNumber)
}
```

➢ Why are we using ?

```
protocol ViewModelDelegate: class { }
```

We use class as an accessor to reduce the scope of action. That means this protocol is only applies on classes.
This works like «final » or « private ».

➢ What is the difference between delegate and protocol ?

A delegate is a property with a protocol type.
A protocol is a contract