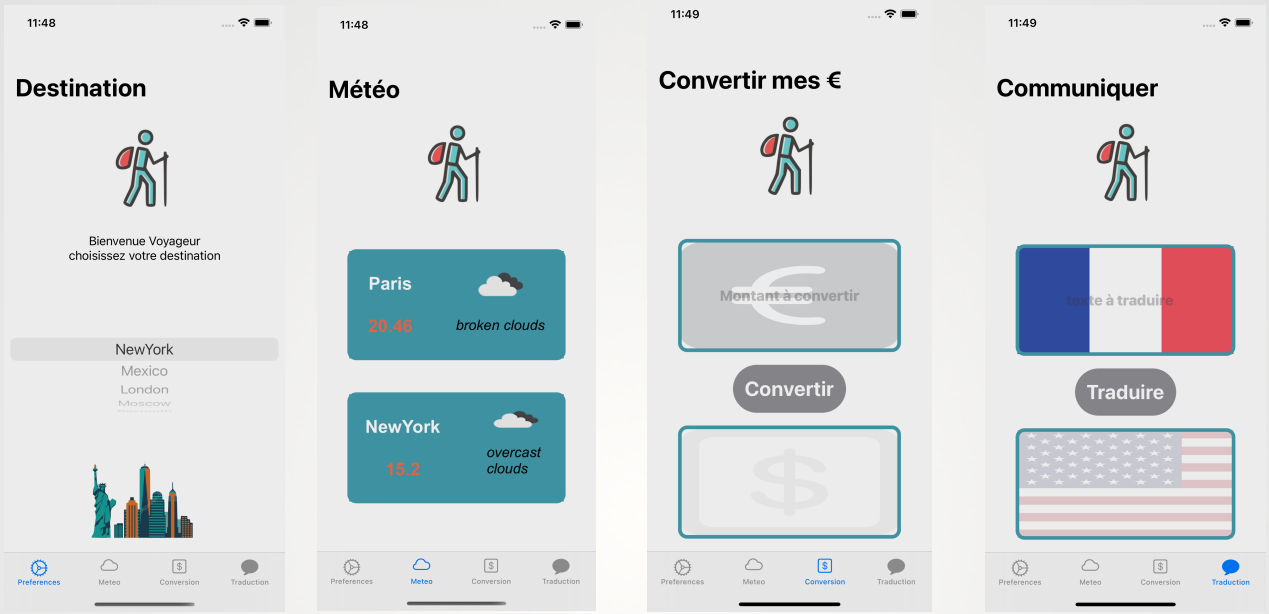
MyBaluchon 2.0 swiftUI

Présentation de l'application:

MyBaluchon est une Application iOS pour iPhone réalisée dans le cadre du parcours de développeur iOS OpenClassRooms.
Le présent document fait référence à la version SwiftUI de l'application développée dans le cadre du projet final de mon parcours d'apprentissage.



- Le concept de l'application est de proposer à un voyageur français une application multi fonctions lors de ses séjours à l'étranger.
- Le premier écran à gauche permet de sélectionner une destination dans un Picker parmi 6 possibilités: NewYork, Mexico, Londres, Moscou, Beyrouth, Tokyo.
- Le second écran permet une comparaison rapide des conditions météo dans la ville d'origine et la destination.
- Le troisème écran permet une conversion de monnaie depuis l'Euro.
- Le dernier écran à droite permet une traduction depuis le français.
- Chaque écran s'adapte automatiquement en fonction de la destination choisie, la sélection de la destination persiste après fermeture de l'application.

Architecture générale:

L'application s'organise autour d'une tab Bar de 4 onglets.

La DestinationView permet la sélection de la destination par le biais du Picker.

Cette valeur est synchronisée avec la variable d'état de l'application « destinationService ».

L'instance destinationService est placée dans l'environnement afin que chaque vue puisse y accéder pour son propre affichage mais aussi pour transmettre la variable à son ViewModel.

Nos vues s'adaptent ainsi automatiquement avec la destination choisie par l'utilisateur.

```
@main
struct myBaluchon2App: App {

    @StateObject var destinationService = Destination()

    var body: some Scene {
        WindowGroup {
                TabView {
                 DestinationView(destinationSelected: $destinationService.selection)
                     .tabItem {
                         Image(systemName: "gearshape")
                         Text("Preferences")
                     }.tag(0)
                 WeatherView()
                        .tabItem {
                            Image(systemName: "cloud.sun.fill")
                            Text("Meteo")
                        }.tag(1)
                 CurrencyView(viewModel: .init())
                        .tabItem {
                            Image(systemName: "eurosign.square")
                            Text("Conversion")
                        }.tag(2)
                 TranslationView(viewModel: .init())
                     .tabItem {
                         Image(systemName: "message.circle.fill")
                         Text("Traduction")
                     }.tag(3)

                }.environmentObject(destinationService)
        }
    }
}
```

Ecrans et Fonctionnement:

WeatherView:

La vue reposant sur 2 composants identiques, nous l'avons décomposée en 2 blocs apellés WeatherBlock, chacun permettant de charger les données d'une ville distincte.
Le viewModifier .onAppear nous permet de mettre à jour les données à l'apparition de l'écran:

```
.onAppear { viewModel.updateWeather() }
```

Les données des conditions météo proviennent de l'API OpenWeather : température, description des conditions, image représentant les conditions.
Nous avons ajouté les images directement dans les « assets » du projet pour de meilleures performances, elles sont nommées comme le String que nous renvoie l'API, permettant de les charger en local.

CurrencyView:

La vue se compose d'un TextField permettant de saisir le montant à convertir, d'un champ Text dans lequel sera affiché le résultat de la conversion et d'un bouton entre les deux pour effectuer la conversion selon l'input saisi.
Les données proviennent de l'API Fixer qui fournit des taux de change en temps réel.
Les données sont chargées depuis l'API à l'initialisation du ViewModel par la fonction getRates() qui charge les taux de change et les assigne à notre variable « exRates».

```
init(destination: Int) {
    self.destinationSelected = destination
    cancellable =
        getRate(baseURL: baseURL, parameters: parameters)
        .receive(on: DispatchQueue.main)
        .catch { _ in Just(self.rate) }
        .assign(to: \.rate, on: self)
}
```

La fonction convert() est déclenchée lorsque l'utilisateur appuie sur le bouton « Convertir », elle effectue la conversion, formate le résultat au centième, et l'assigne à notre variable « output » que l'on affiche dans le champ texte du bas.

TranslationView:

L'organisation de la vue est similaire à CurrencyView.
Les données proviennent de l'API Google Translation.
Dans le cas présent, la traduction est effectuée au moment de l'appui sur le bouton par le biais de la fonction translate().

Notes relatives à SwiftUI:

Property Wrappers:

@State:

Permet de modifier une propriété au sein d'une structure et donc de recréer notre vue chaque fois que cette valeur change.
Nous l'utilisons dans notre application (CurrencyView et TranslationView) avec une variable de type Booléen « isEditing » associée à un TextField.

```
@State private var isEditing: Bool = false
```

En switchant cette variable, nous indiquons à SwiftUI que le TextField est en cours de modification, ce qui nous permet par exemple de modifier l'opacité du champ texte ou encore de déclencher une animation.

```
.opacity(self.isEditing ? 0.8 : 0.2))
```

```
if isEditing == false {
    Image("myBaluchon")
        .position(x: 190, y: 80)
        .transition(.asymmetric(insertion: .scale, removal:
            .opacity)).animation(.easeInOut(duration: 1))
}
```

@Binding:

Permet de synchroniser une variable avec une variable d'état d'une vue parente.

```
@Binding var destinationSelected: Int
```

Dans notre application nous l'utilisons pour synchroniser la destination choisie par l'utilisateur avec notre variable d'état de l'application « destinationService ».

```
 Picker(selection: $destinationSelected,
```

```
TabView {
 DestinationView(destinationSelected: $destinationService.selection)
```

@StateObject:

Permet de créer une instance de classe dans une vue et de conserver une référence forte avec celle ci. Dans notre application, nous utilisons @StateObject pour créer l'instance de classe Destination au sommet de la hiérarchie de nos vues:

```
@main
struct myBaluchon2App: App {

    @StateObject var destinationService = Destination()
```

@ObservedObject:

Si la vue qui crée l'instance de classe utilise @StateObject comme indiqué précédemment, une sous vue à laquelle on passerait la même instance doit utiliser @ObservedObject.
C'est le cas de la plupart de nos vues dans lesquelles nous injectons l'objet destinationService.

@EnvironmentObject:

Il permet de placer une propriété au sommet de la hiérarchie et d'y accéder dans l'ensemble des vues et sous vues de cette même hiérarchie.
Notre instance destinationService est placée dans l'environnement au niveau de l'App, et ainsi y accéder directement dans la vue WeatherBlock sans passer la propriété par WeatherView.

```
@main
struct myBaluchon2App: App {

    @StateObject var destinationService = Destination()

    var body: some Scene {
        WindowGroup {
                TabView {

    }
    .environmentObject(destinationService)
```

Puis l'utiliser directement dans une sous vue sans la passer par les vues intermédiaires dans la hiérarchie, comme ceci:

```
struct WeatherBlock: View {

    @EnvironmentObject var destination: Destination
```

@Published:

Permet de créer une propriété qui annonce automatiquement lorsque des changements se produisent. Cette propriété doit appartenir à une Classe qui se conforme au protocole ObservableObject.
Toute vue qui fait référence à cet objet va automatiquement recréer sa propriété « body » lorsque sa valeur change.
Notre type Destination ainsi que les viewModel de notre application sont des Classes qui publient leurs valeurs pour modifier nos vues:

```
class Destination: ObservableObject {

    @Published var selection: Int {
```

```
struct CurrencyView: View {

    @ObservedObject var destination: Destination
```

Combine framework API Networking

Dans un but d'apprentissage et d'exploration, nous avons choisi pour effectuer nos appels réseau d'utiliser le framework Combine.
Combine fonctionne comme un flux de données, impliquant 3 types d'éléments: un Publisher qui va fournir des données, un suscriber qui va s'abonner à ce flux de données et des opérateurs qui vont éventuellement interagir en tant qu'intermédiaire entre les deux.
Nous allons ci dessous décrire le fonctionnement global d'un appel réseau avec Combine au travers d'un exemple concret d'utilisation, le CurrencyViewModel de notre application:

Publisher:

Le publisher est l'objet qui va fournir les données ainsi qu'une erreur si nécessaire. Il doit etre englobé dans un objet conforme au protocole « Cancellable » de Combine, ce qui signifie que l'activité (le flux de données) relatif à cet objet devra pouvoir être annulé à un moment donné par la fonction cancel().

```
private var cancellable: AnyCancellable?
```

Nous implémentons ensuite la fonction getRates() qui va renvoyer en sortie un objet de type AnyPublisher qui renvoie lui même un objet du type de notre choix, en l'occurence « Currency » ainsi qu'un objet de type « Error », également customizable.

```
func getRates(baseURL: String, parameters: [String : String]) -> AnyPublisher<Currency, Error> {
```

Nous vérifions tout d'abord que notre URL assemblé avec ses paramètres est correct, sinon la fonction renverra un objet de type AnyPublisher et une erreur associée:

```
guard var components = URLComponents(string: baseURL) else {
    return Fail(error: NetworkingError.URLError).eraseToAnyPublisher()
}
components.setQueryItems(with: parameters)
guard let url = components.url else {
    return Fail(error: NetworkingError.URLError).eraseToAnyPublisher()
}
```

Si l'URL est correct, la fonction renverra un objet de type dataTaskPublisher, un Publisher qui englobe un objet de type URLSessionDataTask.

```
return URLSession.shared
    .dataTaskPublisher(for: url)
```

Operateurs:

Les opérateurs vont intervenir pour préparer les données à fournir au suscriber. Notre data task publisher nous renvoie un objet de type data que nous allons mapper et decoder sur notre type Currency, puis reformater le type du publisher en sortie avec la fonction eraseToAnyPublisher():

```
.map { $0.data }
.decode(type: Currency.self, decoder: JSONDecoder())
.eraseToAnyPublisher()
```

Nous demandons ensuite à réceptionner les données sur la main thread dans la mesure où celles ci vont modifier notre interface.

```
.receive(on: DispatchQueue.main)
```

En cas d'erreur, le flux continue avec un nouveau Publisher de type Just qui va prendre le relais et publier la valeur de notre choix. Just ne peut pas renvoyer d'erreur, il renverra toujours une valeur, on renvoie la valeur actuelle de exRates dans le cas présent:

```
 .catch { _ in Just(self.exRates) }
```

Suscribers:

Enfin s'il n'y a pas d'erreur les données son assignées à notre variable exRates qui est donc mise à jour:

```
.assign(to: \.exRates, on: self)
```

Il existe 2 principaux suscribers fournis par le framework Combine sous la forme d'opérateurs et qui vont couvrir un large spectre de cas d'utilisation.

- assign(to: on:) que nous avons utilisé dans le cas présent et qui affecte chaque nouvelle valeur reçue à la propriété choisie, ici exRates.
- sink(receiveCompletion: receiveValue:) qui exécute une closure à chaque « completion signal » ou réception d'une nouvelle valeur.

Nous avons utilisé sink dans nos autres ViewModels mais assign était plus pertinent dans le cas du CurrencyViewModel car les données sont chargées uniquement à l'initialisation du modèle et non à chaque demande de conversion par l'utilisateur.