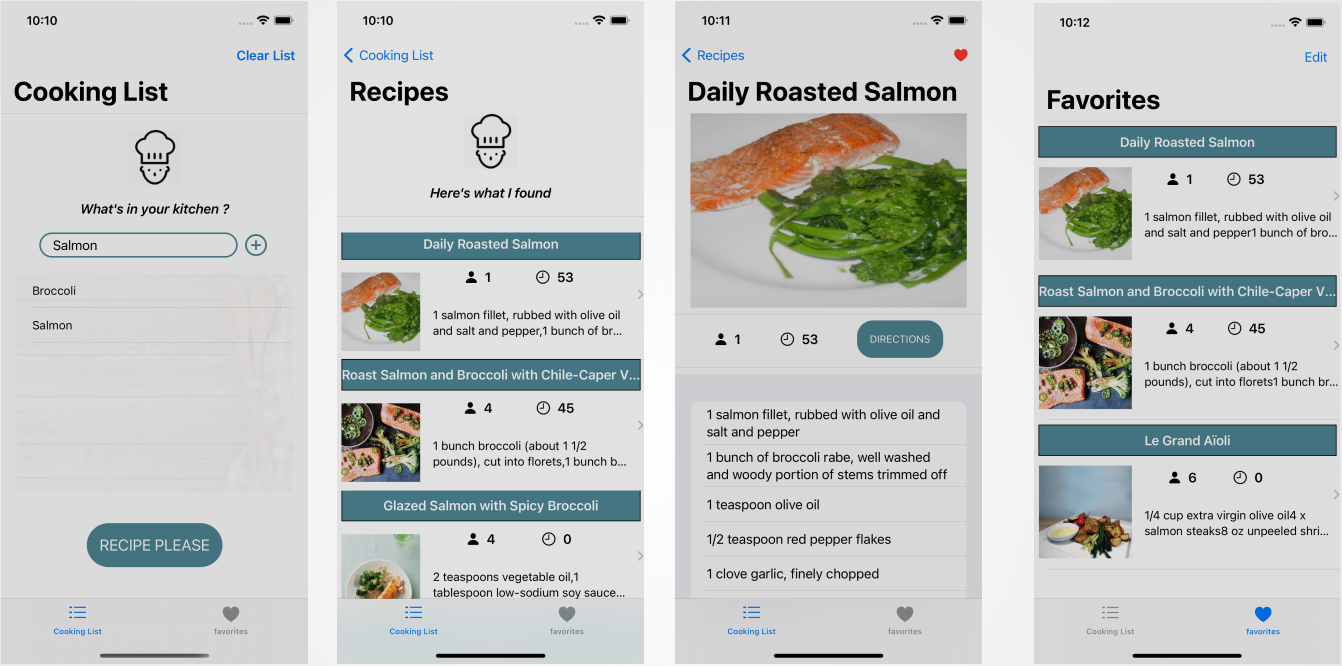
Reciplease 2.0 SwiftUI

Présentation de l’application:

Reciplease 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 une recherche de recettes de cuisine sur la base d’une liste d’ingrédients saisie par l’utilisateur.
- Le premier écran à gauche est l’écran d’accueil de l’application, il permet la saisie des ingrédients sous forme de liste éditable, le bouton en bas permet de lancer la recherche de recettes.
- Le second écran affiche la liste des résultats de la recherche sous forme de blocs contenant des informations sur chaque recette: nom, photo, nombre de couverts, temps de préparation, ingrédients. Chaque recette est cliquable et envoie vers un écran détaillé.

- Le troisème écran offre donc une vue détaillée de la recette et fournit des éléments supplémentaires: une liste exhaustive et scrollable des ingrédients nécessaires, un bouton « Directions » ouvrant la page web source de la recette et un bouton d'ajout aux favoris (coeur).

- Le dernier écran affiche la liste éditable des favoris, sauvegardés dans l'iPhone avec CoreData.

Architecture générale:

L'application s'organise autour d'une tabBar de 2 onglets.
L'onglet cookingList offre la navigation hiérarchique suivante:
Liste d'ingrédients —> Resultats de la recherche —> vue détaillée d'une recette

L'onglet Favoris ne contient qu'un seul écran: la liste des favoris.
L'architecture globale est de type MVVM, la logique de chaque vue étant implémentée dans un fichier séparé viewModel.

```
@main
struct Reciplease2App: App {

    var body: some Scene {
        WindowGroup {
            TabView {
                CookingListView()
                    .tabItem {
                        Image(systemName: "list.dash")
                        Text("Cooking List")
                    }

                FavoritesView()
                    .tabItem {
                        Image(systemName: "heart.fill")
                        Text("favorites")
                    }
```

Ecrans et Fonctionnement:

CookingListView:

Il s'agit de l'écran d'accueil qui contient un TextField permettant de saisir le nom d'un ingrédient, celui ci est ensuite aujouté à la liste en dessous soit en cliquant sur le bouton + à droite, soit en tapant « entrée » sur le clavier.
Pour ce faire, le TextField dispose d'un paramètre « OnCommit » qui nous permet d'exécuter du code et d'exploiter l'input du TextField lorsque l'utilisateur tape « entrée »:

```
TextField(" New Ingredient", text: $newIngredient) { isEditing in
    self.isEditing = isEditing
    if isEditing == true {
        newIngredient = "" }
}
onCommit: {
    self.isEditing = false
    hideKeyboard()
    recipeService.addIngredientToList(input: newIngredient)
}
```

La vue repose sur un viewModel nommé RecipeService qui contient la liste des ingrédients (cookingList) et les fonctions associées comme « addIngredientToList ».

Le bouton en bas de l'écran lance la fonction getRecipes() de notre viewModel qui déclenche l'appel réseau avec l'API Edamam pour trouver des recettes associées.
Le bouton déclenche également la navigation vers l'écran suivant qui affiche les résultats. Pour ce faire, le bouton est englobé dans un NavigationLink:

```
NavigationLink(destination: SearchResultsView(recipesData: recipeService.recipesData),
               isActive: $isShowingResults) { EmptyView() }
Button(action: { didTapSearchRecipeButton() })
```

Ici le NavigationLink est représenté par une EmptyView qui se superpose au bouton. L'appui sur le bouton lance la fonction didTapSearchRecipeButton():

```
func didTapSearchRecipeButton() {
    guard !recipeService.cookingList.isEmpty else {
        recipeService.showAlert.toggle()
        return }
    recipeService.getRecipes()
    sleep(2)
    isShowingResults = true
}
```

Dans cette fonction on s'assure tout d'abord que la liste d'ingrédients n'est pas vide, puis on lance l'appel réseau avec la fonction getRecipes().

Afin de laisser le temps au réseau de charger les résultats de la recherche, on crée un temps d'attente de 2 secondes, puis on switch le booléen isShowingResults qui déclenche la navigation vers l'écran SearchResultsView()

SearchResultsView:

Cette vue affiche les résultats de la recherche sous forme d'une liste de RecipeRow() qui est une sous vue représentant une recette.
Les données sont mappées depuis la data reçue de l'appel réseau vers une structure adaptée à nos besoins nommée RecipeRepresentable.
Chaque RecipeRow() est englobée dans une NavigationLink et donc cliquable, et nous emmeène vers la vue de la recette détaillée: RecipeInfo

RecipeInfo:

Cette vue représente la recette choisie de façon détaillée. On peut y accéder depuis les résultats de recherche mais également depuis l'onglet listant les favoris.
Elle dispose d'un bouton « Directions » qui permet d'ouvrir une « modal », page superposée affichant la page web source de la recette.

Pour ce faire, il suffit d'importer le framework Safari Services dans notre fichier, puis de créer une vue de type SFSafariViewController qui prend en paramètre l'url de la page à afficher.
SFSafariViewController fait partie de UIKit et nous avons donc dû l'englober dans une vue SwiftUI à l'aide du protocole UIViewControllerRepresentable.

```
struct SafariView: UIViewControllerRepresentable {
    let url: URL
    func makeUIViewController(context: UIViewControllerRepresentableContext<SafariView>)
    -> SFSafariViewController {
        return SFSafariViewController(url: url)
    }
    func updateUIViewController(_ uiViewController: SFSafariViewController,
                                context: UIViewControllerRepresentableContext<SafariView>) {
    }
}
```

La vue RecipeInfo contient également un bouton en haut à droite permettant d'ajouter la recette à nos favoris, sauvegardés dans CoreData.
Si la recette fait déjà partie des favoris (le coeur est rempli en rouge) l'appui sur le coeur enlèvera la recette des favoris et déclenchera une navigation directement à la page d'origine, liste des favoris, mise à jour.

Pour ce faire nous utilisons une variable d'environnement, la methode dismiss() déclenche la navigation automatique vers la vue parente dans la hiérarchie:

```
@Environment(\.presentationMode) var presentationMode
```

```
  self.presentationMode.wrappedValue.dismiss()
```

FavoritesView:

Cette vue affiche nos favoris sous forme de liste éditable. Chaque recette est cliquable et déclenche la navigation vers la vue détaillée de la recette: RecipeInfo. Les données sont récupérées depuis CoreData et affichées sous la forme de RecipeRepresentable.

Notes relatives à CoreData en SwiftUI:

Implémentation automatique:

Avec SwiftUI et les dernières versions de Xcode, lorsque l'on coche la case « use CoreData » à la création du projet, Xcode implémente CoreData pour nous dans un fichier nommé Persistence.swift.

Ce fichier définit une structure nommée PersistenceController dans laquelle est déclarée tout d'abord une instance partagée de type singleton, notre objet de type NSPersistenceContainer instancié avec le modèle de données (de type xcdatamodeld), et l'exécution de la méthode loadPersistentStore.

En plus du singleton, le fichier déclare une variable prête à utiliser pour nos previews initialisée dans un store sauvegardé dans la mémoire.
Un store « inMemory » pourra également ête utilisé pour des tests unitaires.
J'ai volontairement enlevé le preview de la capture d'écran ci dessous pour simplification.

```
struct PersistenceController {

    static let shared = PersistenceController()

    let container: NSPersistentContainer

    init(inMemory: Bool = false) {
        container = NSPersistentContainer(name: "testCoreData")
        if inMemory {
            container.persistentStoreDescriptions.first!.url = URL(fileURLWithPath: "/dev/null")
        }
        container.loadPersistentStores(completionHandler: { (storeDescription, error) in
            if let error = error as NSError? {

                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
```

Dans notre structure App, le singleton est déclaré et utilisé pour placer le managedObjectContext dans l'environnement:

```
@main
struct CoreDataHackingApp: App {

    let persistenceController = PersistenceController.shared

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(\.managedObjectContext, persistenceController.container.viewContext)
        }
```

On peut ainsi accéder aux données de CoreData grâce au propertyWrapper @FetchRequest et @Environment pour accéder au contexte et interagir avec les données.

```
@Environment(\.managedObjectContext) var moc
@FetchRequest(entity: Country.entity(), sortDescriptors: []) var countries: FetchedResults<Country>
```

Implémentation personnalisée:

Pour notre application Reciplease et ses favoris, nous avons voulu tenter une autre approche en implémentant nous même notre CoreData Stack dans un fichier que nous avons nommé CoreDataManager, qui contient nos méthodes d'interaction CRUD et que nous utilisons comme ViewModel pour les vues exploitant les favoris. CoreDataManager est une classe conforme au protocole ObservableObject dans laquelle nous avons déclaré un tableau contenant nos entitées CoreData publiées:

```
@Published var favorites = [FavoriteEntity]()
```

L'intialisation du CoreDataManager instancie notre container, charge le persistentStore et exécute la méthode fetchFavorites() qui va mettre à jour notre tableau de favoris par une requête de type NSFetchRequest.

```
func fetchFavorites() {
    let request = NSFetchRequest<FavoriteEntity>(entityName: "FavoriteEntity")
    do {
        favorites = try moc.fetch(request)
        saveData()
    } catch let error {
        print("error fetching. \(error)")
    }
}
```

Nous avons ensuite crée nos méthodes CRUD pour ajouter, supprimer et vérifier si un objet est déjà présent dans les favoris.
Il ne nous reste ensuite qu'à apeller notre CoreDataManager dans les vues concernées:

```
@StateObject private var viewModel = CoreDataManager.shared
```