**CORE DATA**

**Uso de UserDefaults.** Para guardar datos, necesitamos 2 acciones

* Acceder a un valor guardado.

UserDefaults.standard.value(forKey:String)

Value puede ser sustituido por otros tipos de variable. Estamos sacando el valor almacenado con la clave que asignemos al parámetro forKey.

* Almacenar un valor.

UserDefaults.standard.set(value, forKey:String)A la función set le estamos pasando el valor que queremos almacenar y la clave donde queremos almacenar dicho valor.

import SwiftUI

struct ContentView: View {

@State private var tapCount = UserDefaults.standard.integer(forKey: "tap") //Ponemos que la variable acceda al valor guardado. Se pone con @State ya que es una propiedad reactiva que permite que la vista se actualice automáticamente cuando el valor cambia. Se pone private para que solo se pueda acceder desde ContentView con el State.

var body: some View {

VStack {

Text("\(tapCount)")

.font(.largeTitle)

Spacer()

Button("Aumentar contador") {

tapCount += 1

UserDefaults.standard.set(tapCount, forKey: "tap")

}

}

.padding()

}

}

#Preview {

ContentView()

}

**LOGIN**

struct ContentView: View {

@State private var username: String = "" // Variable de estado para almacenar el texto del campo "Username"

@State private var pwd: String = "" // Variable de estado para almacenar el texto del campo "Password"

var body: some View {

VStack {

Text("Welcome!")

.font(.title)

.bold()

// Campo de texto para ingresar el usuario

TextField("Username", text: $username) // Crea un campo de texto con placeholder "Username". Textfield es un componente que permite al usuario ingresar y editar texto. ‘’Username’’ es el placeholder, es decir el texto que aparece dentro de campo antes de que el usuario escriba algo. El $username significa que el campo de texto está vinculado a la variable username. Con $ indica que esta vinculado a un @State, lo que significa que cualquier cambio se reflejara en la variable username en tiempo real.

.padding()

.overlay() // Agrega una superposición alrededor del campo

RoundedRectangle(cornerRadius: 10)

.stroke(.gray) // Aplica un borde de color gris

)

// Campo de texto seguro para ingresar la contraseña

SecureField("Password", text: $pwd) // Campo de contraseña (oculta el texto ingresado)

.padding()

.overlay()

RoundedRectangle(cornerRadius: 10)

.stroke(.gray)

)

// Botón de login

Button {

// Aquí se podría agregar la lógica para manejar el inicio de sesión

} label: { // Define la apariencia del botón

Text("Login") // Texto dentro del botón

.padding(10) // Agrega padding interno

.frame(maxWidth: .infinity) // Hace que el botón ocupe todo el ancho disponible

}

.buttonStyle(.borderedProminent) // Aplica un estilo prominente al botón

}

.padding()

}

}

struct MainView: View {

var name: String // Variable que almacena un nombre, se debe pasar al crear esta vista

var body: some View {

HStack { // Organiza los elementos en una fila (horizontal)

Image(systemName: "fireworks") // Agrega una imagen del sistema con el ícono "fireworks"

.imageScale(.large) // Ajusta el tamaño de la imagen a grande

.bold()

Text("Estás dentro \(name)") // Texto que incluye la variable `name`

.font(.title)

.bold()

Image(systemName: "fireworks") // Agrega otra imagen idéntica en el lado derecho

.imageScale(.large)

.bold()

}

}

}

**MODELOS DE DATOS**

-Observable object. Se utilizan para notificar a las vistas cuando hay un cambio en los datos que están observando, así se actualiza.

Se utiliza @published. Almacena de forma automática un valor y proporciona un publicador que emite un comunicado antes de que el valor cambie, este comunicado es recibido por todas las vistas que esten suscritas a él. Ej. cuando cambia el estado de un pedido. Aunque ahora se ha modificado.

y @ObservedObject, son un tipo de dato que envuelven a la variable. Podemos reemplazarlo por @Observable delante de la propiedad.

@Published permite añadir un escuchador a la variable.

**CoreData**

**MVVM**

* Model. Modelamos los datos, es decir, donde creamos las estructuras y variables que representan la información de una aplicación.
* View. Donde están definidas todas las vistas de la aplicación. Es decir, la interfaz de usuario que el usuario utiliza para interactuar con los datos.
* ViewModel. Se asemeja al controlador, es el área donde se encuentra la lógica de la aplicación que maneja los datos para después mostrarlos en las diferentes vistas.

Carpetas creadas en el proyecto con el modelo CoreData.

* ContentView(Vista principal)

import SwiftUI

import CoreData // Importa CoreData para manejar almacenamiento de datos

struct ContentView: View {

@Environment(\.managedObjectContext) private var viewContext

// `viewContext` proporciona acceso al contexto de Core Data para manipular datos.

@Enviroment permite acceder a valores compartidos dentro de la aplicación, se utiliza en este caso para acceder a datos sin necesidad de pasarlos manualmente entre vistas.

\.managedObjectContext encargado de manejar los datos de la base de datos en memoria.

@FetchRequest(

sortDescriptors: [NSSortDescriptor(keyPath: \Item.timestamp, ascending: true)], animation: .default) //Aplica una animación cuando se actualizan los datos.

private var items: FetchedResults<Item> //FetchedResult<Item> maneja automáticamente los datos recuperados.

// `@FetchRequest` obtiene los datos almacenados en Core Data y los ordena por `timestamp`. Almacena los resultados en la variable ítems.

sortDescriptors: [NSSortDescriptor(…)] especifica cómo se ordenan los datos, con timestamp se ordenan por fecha en orden ascendente (true)

var body: some View {

NavigationView { // Envuelve la vista en una navegación

List { // Muestra una lista de elementos obtenidos de Core Data. Se usa para mostrar contenido en formato de tabla con desplazamiento automático.

ForEach(items) { item in //itera sobre itemd que son obtenidos anteriormente.

NavigationLink { // Crea un enlace a otra vista cuando se selecciona un elemento

Text("Item at \(item.timestamp!, formatter: itemFormatter)") //Muestra un texto con la fecha timestamp formateada. Esta vista se muestra al tocar el elemento de la lista. (Destino del enlace)

} label: {

Text(item.timestamp!, formatter: itemFormatter) // Muestra la fecha del ítem (contenido visible en la lista)

}

}

.onDelete(perform: deleteItems) // Permite eliminar elementos deslizando

}

.toolbar { // Agrega una barra de herramientas en la parte superior

ToolbarItem(placement: .navigationBarTrailing) //Placement indica donde se colocará el botón en la barra de herramientas y navigationBarTrailing significa que se colocará a la derecha. (Si se pone leading seria a la izquierda) {

EditButton() // Botón para habilitar la edición de la lista (eliminar elementos)

}

ToolbarItem { // Botón para agregar un nuevo ítem a la lista

Button(action: addItem) {

Label("Add Item", systemImage: "plus") // Muestra un ícono de "+" con texto

}

}

}

Text("Select an item") // Mensaje predeterminado cuando no se selecciona un ítem. Sirve como placeholder

}

}

// Función para agregar un nuevo ítem a Core Data

private func addItem() { //Declara una function privada, y solo se puede usar dentro de la estructura ContentView, no recibe ni devuelve valores.

withAnimation { //Añade una animación.a los cambios en la interfaz

let newItem = Item(context: viewContext) // Crea un nuevo objeto en Core Data

Item(context: viewContext) genera una nueva instancia de Item.

newItem.timestamp = Date() // Asigna la fecha actual como timestamp

//Si está todo bien intenta guardar los cambis en la base de datos con viewContext.save()

do {

try viewContext.save() // Guarda los cambios en Core Data

} catch {

// Manejo de errores: en caso de fallo, genera un error fatal con detalles

let nsError = error as NSError //Convierte el error en un formato que se puede leer mejor.

fatalError("Unresolved error \(nsError), \(nsError.userInfo)") //Detiene la app y muestra el error.

}

}

}

// Función para eliminar elementos de la lista

private func deleteItems(offsets: IndexSet) { //Recibe un parametron offsets: IndexSet que representa las posiciones de los elementos seleccionados para ser eliminados.

withAnimation {

offsets.map { items[$0] }.forEach(viewContext.delete) // Borra los elementos seleccionados. Convierte los índices (offsets) en elementos reales de la lista (ítems)

.map { ítems [$0] } toma cada índice de offsets y lo convierte en un Item de CoreData.

.forEach(viewContext.delete) borra cada elemento obtenido del contexto de Core Data.

Imagina que items contiene:

items = ["Nota 1", "Nota 2", "Nota 3"]

Si el usuario desliza y elimina "Nota 2", offsets contendrá {1}, lo que significa:

items[1] // "Nota 2"

Luego, viewContext.delete(items[1]) elimina "Nota 2" de la base de datos.

do {

try viewContext.save() // Guarda los cambios en Core Data

} catch {

// Manejo de errores en caso de fallo al eliminar

let nsError = error as NSError

fatalError("Unresolved error \(nsError), \(nsError.userInfo)")

}

}

}

}

// Formateador de fecha para mostrar los timestamps en un formato legible

private let itemFormatter: DateFormatter = {

let formatter = DateFormatter()

formatter.dateStyle = .short

formatter.timeStyle = .medium

return formatter

}()

import CoreData

// Controlador de persistencia que gestiona Core Data

struct PersistenceController {

static let shared = PersistenceController() // Singleton para usar Core Data en toda la app

// Configuración de un contenedor de datos en memoria para pruebas

static var preview: PersistenceController = {

let result = PersistenceController(inMemory: true) // Crea una instancia en memoria

let viewContext = result.container.viewContext // Obtiene el contexto de la base de datos

// Agrega un ítem de prueba para verificar la funcionalidad

let newItem = Item(context: viewContext)

newItem.timestamp = Date() // Asigna la fecha actual como dato

do {

try viewContext.save() // Intenta guardar el nuevo dato en Core Data

} catch {

// Manejo de errores si la operación de guardado falla

let nsError = error as NSError

fatalError("Unresolved error \(nsError), \(nsError.userInfo)") // Detiene la ejecución y muestra el error

}

return result // Retorna la instancia de prueba

}()

let container: NSPersistentContainer // Contenedor de almacenamiento de Core Data

// Inicializador del controlador de persistencia

init(inMemory: Bool = false) {

container = NSPersistentContainer(name: "Notes") // Define el nombre del modelo de datos

if inMemory {

// Si `inMemory` es true, se usa un almacenamiento temporal en memoria

container.persistentStoreDescriptions.first!.url = URL(fileURLWithPath: "/dev/null")

}

// Carga el almacén de datos de manera asíncrona

container.loadPersistentStores(completionHandler: { (storeDescription, error) in

if let error = error as NSError? {

// Si hay un error, se detiene la ejecución y se muestra el error

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

}

})

// Habilita la fusión automática de cambios entre contextos de Core Data

container.viewContext.automaticallyMergesChangesFromParent = true

}

}

En notes.xcdatamodeld es donde vamos a crear nuestra base como tal, nuestro modelado de datos. Estan las entities que es la representación de nuestras tablas y atributos serían los campos de esa tabla, cada columna indicade que tipo son.

**Estructura de carpetas con Core Data**

-Notes

**-ViewModel.** Aquí creamos nuestro modelo de datos ej.

Import Foundation

Class ViewModel: ObservableObject{

@Published var note = “”

@Published var date = Date ()

@Published var show = false //Lo utilizaremos para mostrar u ocultar una vista modal que muestre el contenido completo de la nota.

}

**-View**

--NotesApp.Swift.

--ContentView.swift

--Assets.xcassets

--Persitence.swift

--Notes.xcdatamodeld (model)

**-Preview Content.**

1º En View añadimos dos vistas.

HomeView donde mostraremos la lista de notas que iremos creando y un botón para añadir nuevas notas que abrirá una nueva vista modal.

import SwiftUI

import CoreData

struct HomeView: View {

@ObservedObject var model = ViewModel() //Instanciamos el modelo de datos definido anteriormente.

var body: some View {

NavigationStack {

Text("Demo")

.toolbar {

ToolbarItemGroup(placement: .primaryAction) { //Con ToolbarItemGroup permite agrupar multiples botones en una sección. Con primary action indicas que todos los botones de este grupo se colocarán en la parte derecha de la barra de navegación.

Button {

model.show.toggle() //Cambia el estado de show

} label: {

Image(systemName: "plus")

.font(.title)

}

.sheet(isPresented: $model.show, content: { //sheet se usa para presenter una vista emergente en pantalla. IsPresented controla con una variable booleana si la hoja se muestra o no. $ crea un binding, lo que significa que cuando model.show sea true la vista emergente aparecerá.

NewNoteView(model: model) //La nueva vista recibe model como parámetro.

}

}

}

}

.navigationTitle("Notas") //Se establece el título de la vista de navegación como notas.

}

}

2ºComo la vista anterior va a ser la vista principal, tendremos que cambiar la vista inicial que carga la aplicación en NotesApp.swift.

import SwiftUI

@main //Indica que esta es la entrada principal a la aplicación.

struct NotesApp: App {

let persistenceController = PersistenceController.shared //Se declara una constante persistenceController que obtiene una instancia compartida (shared) del PersistenceController. ¿Y que es PC? Una clase que maneja la base de datos usando CoreDate. Shared sugiere que esta clase sigue un patrón Singleton, lo que permite compartir una misma instancia en toda la aplicación.

var body: some Scene {

WindowGroup { //Crea la Ventana principal de la aplicación.

HomeView()

.environment(\.managedObjectContext, persistenceController.container.viewContext)

}

}

}

3º Añadimos la segunda vista en view que sería NewNoteView

import SwiftUI

struct NewNoteView: View {

@ObservedObject var model: ViewModel // ViewModel que maneja el estado de la vista

var body: some View {

VStack {

// Título de la vista

Text("Agregar nota")

.font(.largeTitle) // Aplica tamaño grande al texto

.bold()

Spacer() // Agrega un espacio flexible

// Editor de texto para escribir la nota

TextEditor(text: $model.note) // Se enlaza a `model.note` para guardar el texto ingresado

Divider() // Agrega una línea separadora entre los elementos

// Selector de fecha para la nota

DatePicker("Selecciona una fecha", selection: $model.date) // Permite elegir una fecha. DatePicker es un componente de interfaz que permite seleccionar una fecha y/u hora.

Spacer()

**// Botón para guardar la nota**

Button {

Model.saveData(context:context)

} label: {

Label(

title: { Text("Guardar nota") // Texto dentro del botón

.foregroundStyle(.white) // Color del texto en blanco

},

icon: { Image(systemName: "plus") } // Ícono "+" junto al texto

)

}

.padding() // Agrega espacio alrededor del botón

.frame(width: UIScreen.main.bounds.width - 50) // Ajusta el ancho del botón al 90% de la pantalla

.background(Color.blue) // Fondo azul del botón

.cornerRadius(8) // Bordes redondeados en el botón

}

.padding()

}

}

4º Creamos en el ViewModel una función que se encargue de guardar información en Coredata. Para ello lo primero es importar la librería de CoreData para poder utilizar sus elementos.

Función para guardar elementos

func saveData(context: NSManagedObjectContext) //Context es de tipo NSManagedObjectContext lo que significa que maneja la interacción con la base de datos de CoreData {

let newNote= Notes(context: context) //Crea una nueva instancia de Notes, newnote ahora representa un nuevo registro en la base de datos.

newNote.note= note

newNote.date = date

do {

try context.save()

print("Elemento guardado correctamente")

show.toggle()

} catch let error as NSError {

print("Error al guardar: ", error.localizedDescription)

}

}

En este caso a la instancia de la entidad Notes le estamos asignando los valores definidos en nuestro modelo de tal forma que cuando creemos un objeto del ViewModel, podremos llamar a esta función para persistir los datos.

5º. Una vez creada la funcionabilidad de guardado, tenemos que llamar a esta función desde el botón de guardar nota en la vista NewNoteView. Para poder hacer esto, debemos primero definir el contexto en la vista:

@Environmet(\.managedobjectContext) var context (se añade en NewNoteView debajo de observedObject)

6º Modificamos noteView para llamar a la función que hemos definido previamente.

7º Para mostrar estos datos guardados en local, debemos utilizar otro property wrapper:

@FetchRequest(entity: Notes.entity(), sortDescriptors: [NSSortDescriptor(key: "date", ascending: false)], animation: .smooth()) var notes: FetchedResults<Notes> (se añade en homeView despues de observedObject ya que obtenemos las notas almacenadas en CoreData)

//entity: Notes.entity() → Indica que se recuperan datos de la entidad Notes.

sortDescriptors: [NSSortDescriptor(key: "date", ascending: false)] → Ordena los resultados por la fecha (date) en orden descendente (los más recientes primero).

animation: .smooth() → Aplica una animación fluida cuando los datos cambian.

var notes: FetchedResults<Notes> → notes almacena los resultados obtenidos desde Core Data.

8º. Representarlo en la vista HomeView donde mostraremos todas las notas que vayamos añadiendo.

var body: some View {

NavigationStack { // Crea una navegación apilada para gestionar pantallas de manera estructurada

List { // Lista que muestra las notas almacenadas en Core Data

ForEach(notes) { item in // Itera sobre cada nota en la lista

VStack(alignment: .leading) { // Organiza el contenido en una columna alineada a la izquierda

Text(item.note ?? "Nota vacía") // Muestra el texto de la nota, si es nil muestra "Nota vacía"

.font(.title2)

.lineLimit(2) // Limita la cantidad de líneas a 2 para evitar textos muy largos

Text(item.date ?? Date(), style: .date) // Muestra la fecha de la nota, si es nil usa la fecha actual

.font(.subheadline) // Aplica un estilo de fuente más pequeño para la fecha

}

}

}

.toolbar {

ToolbarItemGroup(placement: .primaryAction) {

Button {

model.show.toggle() /

} label: {

Image(systemName: "plus")

.font(.title)

}

.sheet(isPresented: $model.show, content: { // Presenta una hoja modal cuando `show` es true

NewNoteView(model: model) // Muestra la vista para agregar una nueva nota

})

}

}

}

.navigationTitle("Notas")

}

**¿Cómo crear una entidad en un proyecto CoreData?**La entidad es la tabla que va almacenar los datos específicos de un mismo tipo. Lo hacemos en notex.xcdatamodelds y pulsamos en el botón de abajo add entity, si pulsamos podemos cambiar el nombre e ir añadiendo campos y atributos.

**Eliminar datos en CoreData.**

9º Definimos una nueva función en ViewModel.

func deleteData(item: Notes, context: NSManagedObjectContext) {

context.delete(item) // Elimina el objeto 'item' de Core Data

do {

try context.save() // Guarda los cambios en Core Data después de eliminar el elemento

print("Elemento eliminado correctamente") // Mensaje de confirmación en la consola

} catch let error as NSError {

print("Error al eliminar: ", error.localizedDescription) // Captura y muestra el error si ocurre

}

}

10º Vamos a implementar un menú contextual para poder eliminar notas (un conjunto de opciones que aparecen al mantener pulsado un elemento) Debemos vitaminar cada una de las notas que aparecen en la interfaz, en este caso debemos añadir esta funcionabilidad al VStack que contiene la información de cada nota.

VStack(alignment: .leading) {

Text(item.note ?? "Nota vacía")

.font(.title2)

.lineLimit(2)

Text(item.date ?? Date(), style: .date)

.font(.subheadline)

}

.contextMenu(ContextMenu(menuItems: { // Agrega un menú contextual (presionando prolongadamente)

Button(action: {

Model.deleteData(item: item, context:context)

}, label: {

Label(title: {

Text("Eliminar") // Texto del botón del menú

}, icon: {

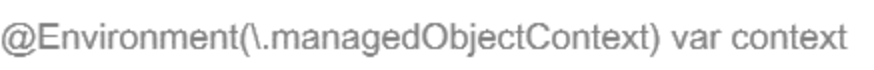
Image(systemName: "trash") // Ícono de papelera para eliminar

})

})

}))

11º Al igual que hicimos, debemos referenciar el contexto como una variable de la vista principal.



Después ya podemos llamar a la función de borrado desde la acción del botón.

**Editar datos en CoreData.** Tener en cuenta que es una acción en 2 pasos. Tenemos que abrir una pantalla de edición para después guardar los cambios.

12º Vamos a ViewModel y creamos una variable de tipo published.

@Published var updateItem: Notes! (puede dar error ya que notes puede ser opcional pero le ponemos la exclamación para confirmar que siempre vamos a tener un valor)

13º Creamos la función de enviar información

Func sendData(item:Notes) { //Recibe como parámetro un item que es la nota que queremos modificar.

updateItem = item //Referenciamos la variable updateItem con el valor que nos llegue.

note = item.note ? ? “”

date= item.date ? ? Date() //Asignamos valores por defecto si no tienen.

show.toggle () //Con esto entramos a la pantalla de detalles.

}

}

14º Vamos a Homeview y replicamos nuestro menú de acciones, creamos un botón para editar.

Button(action: {

model.sendData(item: item)

}, label: {

Label(title: {

Text("Editar")

}, icon: {

Image(systemName: "pencil")

})

15º Definimos la función de edición en el DataModel

func editData(context: NSManagedObjectContext) {

updateltem.note = note // Asigna el nuevo contenido de la nota

updateltem.date = date // Asigna la nueva fecha de la nota

do {

try context.save()

print("Elemento editado correctamente")

show.toggle()

} catch let error as NSError {

print("Error al editar: ", error.localizedDescription)

}

}

16º Modificamos NewNoteView para indicar si al pulsar el botón de guardar tenemos que crear un nuevo elemento o editarlo. Mediante el uso de un if podemos gestionar esta lógica.

struct NewNoteView: View {

@ObservedObject var model: ViewModel // Instancia del ViewModel para gestionar el estado y la lógica de la vista

@Environment(\.managedObjectContext) var context // Contexto de Core Data para guardar o editar notas

var body: some View {

VStack {

Text("Agregar nota")

.font(.largeTitle)

.bold()

Spacer()

TextEditor(text: $model.note) // Se enlaza con `model.note` para actualizar los datos

Divider()

DatePicker("Selecciona una fecha", selection: $model.date)

Spacer()

// Botón para guardar o editar la nota

Button {

if model.updateItem != nil { // Si `updateItem` no es nulo, significa que estamos editando

model.editData(context: context) // Llama a la función para editar la nota

} else {

model.saveData(context: context) // Llama a la función para guardar una nueva nota

}

} label: {

Label(

title: {

Text("Guardar nota")

.foregroundStyle(.white)

},

icon: {

Image(systemName: "plus")

}

)

}

.padding()

.frame(width: UIScreen.main.bounds.width - 50)

.background(Color.blue)

.cornerRadius(8)

}

.padding()

}

}

**TRANSFORMACIÓN DE JSON A UN MODELO DE DATOS**

-Hacemos uso del protocolo Codable que nos permite transformar la jerarquía de datos de un JSON a nuestro modelo de datos.

-Tenemos que modelar en SWIFTUI según los niveles de datos que haya en JSO, para que el protoco Codable sepa a que elemento hace referencia cada uno de los nivedes de datos.

{

"id": 1,

"name": "Leanne Graham",

"username": "Bret",

"email": "Sincere@april.biz",

"address": {

"street": "Kulas Light",

"suite": "Apt. 556",

"city": "Gwenborough",

"zipcode": "92998-3874",

"geo": {

"lat": "-37.3159",

"lng": "81.1496"

}

},

"phone": "1-770-736-8031 x56442",

"website": "hildegard.org",

"company": {

"name": "Romaguera-Crona",

"catchPhrase": "Multi-layered client-server neural-net",

"bs": "harness real-time e-markets"

}

}

Ej en swiftUI

import Foundation

struct User: Codable {

let id: Int

let name, username, email: String

let address: Address

let phone, website: String

let company: Company

}

struct Address: Codable {

let street, suite, city, zipcode: String

let geo: Geo

}

struct Geo: Codable {

let lat, lng: String

}

struct Company: Codable {

let name, catchPhrase, bs: String

}

Cada struct conforma Codable, lo que permite codificar y decodificar datos JSON fácilmente.

User contiene subestructuras Address, Geo y Company para representar los datos anidados correctamente.

**LLAMADAS DE GET Y POST**

-Llamadas GET realizan una operación de lectura de datos para obtener un recurso específico y realizar consultas.

-Llamadas POST se utilizan para realizar operaciones de escritura o modificación. Más adecuado para datos sensibles.

**Pasos para crear un proyecto que haga llamar a una API externa.**

-No es necesario implementar CoreData

-Ej con la siguiente API: <https://jsonplaceholder.typicode.com/users>

1º Creamos el ViewModel (se debe tener uno por cada vista que tenga nuestra aplicación) Definimos una clase que utilice ObservableObjects. Despues creamos una variable que vaya a contener la lista de usuarios que vamos a recibir desde la API. Para poder comunicar a la vista cualquier cambio que experimente esta variable usamos @Published

Import Foundation

Class MainViewModel: ObservableObject {

@Published var users: [User] = []

}

2ºGracias a init() Podemos relizar ciertas acciones en el momento justo en que se instancie la clase que estamos creando. En este caso es de utilidad para realizar una llamada a la API nada mas cargar la vista. Esta función llamará a una función que definiremos como fetchData().

init() {

fetchData() // Llama a la función fetchData() al inicializar la clase o estructura

}

// Esta función realiza una llamada asíncrona a la API para recibir un JSON con los resultados

func fetchData() {

print("Llamando a la API...") // Mensaje de depuración para indicar que se está iniciando la solicitud

// Verifica que la URL sea válida antes de continuar

guard let url = URL(string: "https://jsonplaceholder.typicode.com/users")

else {

return // Si la URL no es válida, detiene la ejecución de la función

}

// Crea una tarea de red para obtener los datos de la API

URLSession.shared.dataTask(with: url) { data, \_, \_ in

// Verifica que los datos recibidos no sean nil

guard let data = data else { return }

do {

// Decodifica el JSON recibido en un array de objetos `User`

let json = try JSONDecoder().decode([User].self, from: data)

// Actualiza la UI en el hilo principal

DispatchQueue.main.async {

self.users = json // Asigna los datos obtenidos a la variable `users`

print(self.users) // Imprime los datos obtenidos en la consola para verificar

}

} catch let error as NSError {

// Captura cualquier error en la decodificación y lo imprime en la consola

print("Error en la lectura del JSON: ", error.localizedDescription)

}

}

.resume() // Inicia la ejecución de la tarea de red

}

3º Desarrollo de vistas. Creamos una nueva vista donde vamos a crear una instancia del ViewModelo que acabmos de crear.

@StateObject var jsonData= MainViewmodel(). De esta forma al crear una instancia del ViewModel, cuando cargue esta vista, se llamará a la función init que a su vez llama a la función fetchData, encargada de conseguir los datos de la API remota.

4º Para mostrar los datos vamos a crear una lista iterable para mostrar cada uno de los elementos.

List(jsonData.users, id: \.id) { item in // Crea una lista con los datos de `users`, usando `id` como identificador único

VStack(alignment: .leading) {

Text(item.name) // Muestra el nombre del usuario

Text(item.address.city) // Muestra la ciudad del usuario

.font(.subheadline) // Aplica un estilo de fuente más pequeño para la ciudad

}

}

import SwiftUI

struct MainView: View {

@StateObject var jsonData = MainViewModel() // Instancia de ViewModel para manejar la data de la API

var body: some View {

VStack { // Organiza los elementos en una columna vertical

if jsonData.users.isEmpty { // Verifica si la lista de usuarios está vacía

ProgressView() // Muestra un indicador de carga mientras se obtienen los datos

} else {

List(jsonData.users, id: \.id) { item in // Crea una lista con los usuarios obtenidos

VStack(alignment: .leading) { // Alinea el contenido a la izquierda

Text(item.name) // Muestra el nombre del usuario

Text(item.address.city) // Muestra la ciudad del usuario

.font(.subheadline) // Usa un estilo de fuente más pequeño para la ciudad

}

}

}

}

}

}

**CREAR UN LOGIN**

Con llamadas tipo POST, las más utilizada para los inicios de sesión. Vamos a necesitar crear un nuevo modelo, un viewModel y una vista.

El modelo de datos lo utilizaremos para recibir la respuesta de la API una vez hagamos la llamada pOST. En el caso de la API de ejemplo que estamos utilizando, el endpoint de login (/api/login) recibe como parámetros un email y una contraseña, y como respuesta la API devuelve un token de la sesión

1º Al igual que hemos hecho en la llamada GET, el modelo que debemos crear debe asimilar el JSON que devuelve la API para convertir automáticamente la respuesta a datos más manejables dentro del código:

Import Foundation

Struct PostModel: Codable {

Var token: String

}

2º Una vez definido el modelo, es hora de crear el ViewModel donde definiremos toda la lógica del inicio de sesión

Import Foundation

Class LoginViewmodel: ObservableObject {

@Published var authenticated = false //Utilizamos una variable para controlar si estamos autenticados o no y la actualizamos a falso para cambiarlo a positivo cuando la API nos devuelva un token.

}

3º Se define la función encargada de hacer el login.

func login(email: String, password: String) {

// Verifica que la URL de la API es válida

guard let url = URL(string: "https://reqres.in/api/login") else { return }

// Crea un diccionario con los parámetros del cuerpo de la solicitud (email y contraseña)

let parametros = ["email": email, "password": password]

// Convierte los parámetros a formato JSON para enviarlos en la solicitud

let body = try! JSONSerialization.data(withJSONObject: parametros)

// Crea un objeto URLRequest con la URL de la API

var request = URLRequest(url: url)

request.httpMethod = "POST" // Especifica que la solicitud es de tipo POST

request.httpBody = body // Asigna el cuerpo de la solicitud con los datos en formato JSON

request.setValue("application/json", forHTTPHeaderField: "Content-Type") // Indica que se envían datos en formato JSON

// Inicia la solicitud de red de manera asíncrona

URLSession.shared.dataTask(with: request) { data, response, error in

// Si la respuesta del servidor es válida, la imprime en consola

if let response = response {

print(response)

}

// Verifica que los datos recibidos no sean nil

guard let data = data else { return }

do {

// Intenta decodificar los datos en un modelo de datos (PostModel)

let datos = try JSONDecoder().decode(PostModel.self, from: data)

// Verifica si el token recibido no está vacío

if !datos.token.isEmpty {

DispatchQueue.main.async {

print(datos.token) // Imprime el token recibido en consola

self.authenticated = 1 // Marca al usuario como autenticado

UserDefaults.standard.setValue(1, forKey: "sesion") // Guarda la sesión en UserDefaults

}

}

} catch let error as NSError {

// Captura cualquier error en la solicitud o decodificación de datos

print("Error al hacer post", error.localizedDescription)

DispatchQueue.main.async {

self.authenticated = 2 // Marca la autenticación como fallida

}

}

}

.resume() // Ejecuta la solicitud de red

}

**GESTION DE IMÁGENES DESDE INTERNET**

**Trabajar con imágenes mediante URLs**

Utilizando el componente AsyncImage estamos cargando de forma asíncrona la imagen almacenada en la url especificada.

AsyncImage(url: URL(string: "https://www.collegiate-ac.es/propeller/uploads/sites/3/2019/11/01-fachada-u-tad.jpg"))

Como estas imagenes pueden existir o no temenos que hacer una gestion de este recurso.

struct MainView: View {

// URL de la imagen a cargar

var imageURL = URL(string: "https://www.collegiate-ac.es/propeller/uploads/sites/3/2019/11/01-fachada-u-tad.jpg")

var body: some View {

VStack { // Organiza los elementos en una columna

VStack {

// Carga la imagen de forma asíncrona desde la URL

AsyncImage(url: imageURL) { phase in

switch phase {

case .success(let image): // Si la imagen se carga con éxito

image

.resizable() // Permite que la imagen sea escalable

.scaledToFit() // Ajusta la imagen manteniendo la relación de aspecto

case .failure(\_): // Si hay un error al cargar la imagen

Image(systemName: "photo") // Muestra un ícono de foto como fallback

.resizable()

.scaledToFit()

case .empty: // Mientras se carga la imagen, muestra un indicador de progreso

ProgressView()

@unknown default: // Maneja cualquier otro caso inesperado

EmptyView() // No muestra nada en pantalla

}

}

}

}

}

}