

Model: GPT-4



1 / 1

Please write example swift code for a class that consumes an external service on behalf of the rest of the application. Use async/await.

1 / 1

In this example, we create a `WeatherService` class that consumes an external weather API on behalf of the rest of the application. The class contains a function `fetchWeatherData(city: String)` that fetches the weather data for a given city using `async/await`.

First, you need to add support for the `URLSession async/await` API by importing the appropriate library:

swift

Copy code

```
import Foundation
import _Concurrency
```

Next, you can create the `WeatherService` class as follows:

swift

Copy code

```
class WeatherService {
    private let apiKey = "<YOUR_API_KEY>"
    private let baseURL = "https://api.openweathermap.org/data/2.5/weather"

    func buildRequestURL(city: String) -> URL? {
        var components = URLComponents(string: baseURL)
        components?.queryItems = [
            URLQueryItem(name: "q", value: city),
            URLQueryItem(name: "appid", value: apiKey),
            URLQueryItem(name: "units", value: "metric")
        ]
        return components?.url
    }
}
```

```

    }

    func fetchWeatherData(city: String) async throws -> WeatherData {
        guard let url = buildRequestURL(city: city) else {
            throw WeatherError.invalidURL
        }

        let (data, response) = try await URLSession.shared.data(from: url)
        guard let httpResponse = response as? HTTPURLResponse,
            httpResponse.statusCode == 200 else {
            throw WeatherError.invalidResponse
        }

        let weatherData = try JSONDecoder().decode(WeatherData.self, from:
data)
        return weatherData
    }
}

enum WeatherError: Error {
    case invalidURL
    case invalidResponse
}

struct WeatherData: Codable {
    let name: String
    let main: Main
    let weather: [Weather]

    struct Main: Codable {
        let temp: Double
    }

    struct Weather: Codable {
        let description: String
    }
}

```

To use the WeatherService class, you can fetch the weather data for a city like this:

swift

Copy code

```

Task {
    do {
        let weatherService = WeatherService()
        let weatherData = try await weatherService.fetchWeatherData(city:
"London")
        print("City: \(weatherData.name)")
    }
}

```

```

        print("Temperature: \(weatherData.main.temp)°C")
        print("Description: \(weatherData.weather.first?.description ?? "N/A")")
    } catch {
        print("Error fetching weather data: \(error)")
    }
}

```

Make sure to replace <YOUR_API_KEY> with your actual API key from the weather service you're using (e.g., OpenWeatherMap).



1 / 1

Thank you. Please write a second service.

1 / 1

Let's create a `CurrencyService` class that consumes an external currency exchange rate API on behalf of the rest of the application. The class contains a function `fetchExchangeRate(baseCurrency: String, targetCurrency: String)` that fetches the exchange rate between two currencies using `async/await`.

First, import the required libraries:

swift

Copy code

```

import Foundation
import _Concurrency

```

Next, create the `CurrencyService` class as follows:

swift

Copy code

```

class CurrencyService {
    private let apiKey = "<YOUR_API_KEY>"
    private let baseUrl = "https://api.exchangeratesapi.io/v1/latest"
}

```

```

    func buildRequestURL(baseCurrency: String, targetCurrency: String) ->
URL? {
    var components = URLComponents(string: baseURL)
    components?.queryItems = [
        URLQueryItem(name: "access_key", value: apiKey),
        URLQueryItem(name: "base", value: baseCurrency),
        URLQueryItem(name: "symbols", value: targetCurrency)
    ]
    return components?.url
}

    func fetchExchangeRate(baseCurrency: String, targetCurrency: String)
async throws -> Double {
    guard let url = buildRequestURL(baseCurrency: baseCurrency,
targetCurrency: targetCurrency) else {
        throw CurrencyError.invalidURL
    }

    let (data, response) = try await URLSession.shared.data(from: url)
    guard let httpResponse = response as? HTTPURLResponse,
httpResponse.statusCode == 200 else {
        throw CurrencyError.invalidResponse
    }

    let exchangeData = try JSONDecoder().decode(ExchangeData.self, from:
data)
    guard let rate = exchangeData.rates[targetCurrency] else {
        throw CurrencyError.rateNotFound
    }

    return rate
}
}

enum CurrencyError: Error {
    case invalidURL
    case invalidResponse
    case rateNotFound
}

struct ExchangeData: Codable {
    let base: String
    let rates: [String: Double]
}

```

To use the CurrencyService class, you can fetch the exchange rate between two currencies like this:

swift

Copy code

```
Task {  
    do {  
        let currencyService = CurrencyService()  
        let exchangeRate = try await  
currencyService.fetchExchangeRate(baseCurrency: "USD", targetCurrency: "EUR")  
        print("Exchange rate from USD to EUR: \(exchangeRate)")  
    } catch {  
        print("Error fetching exchange rate: \(error)")  
    }  
}
```

Make sure to replace <YOUR_API_KEY> with your actual API key from the currency exchange rate service you're using (e.g., exchangeratesapi.io).



1 / 1

Please integrate both of those example usages into a third, feature-level service that consumes the first two.

1 / 1

Let's create a `TripService` class that uses both the `WeatherService` and `CurrencyService` classes to fetch weather and currency exchange information for a specific city. This class will serve as a feature-level service that consumes the first two services.

First, import the required libraries:

swift

Copy code

```
import Foundation  
import _Concurrency
```

Now, create the `TripService` class as follows:

swift

Copy code

```
class TripService {
    private let weatherService = WeatherService()
    private let currencyService = CurrencyService()

    func fetchTripInfo(city: String, baseCurrency: String, targetCurrency:
String) async throws -> TripInfo {
        let weatherData = try await weatherService.fetchWeatherData(city:
city)
        let exchangeRate = try await
currencyService.fetchExchangeRate(baseCurrency: baseCurrency, targetCurrency:
targetCurrency)

        return TripInfo(
            city: weatherData.name,
            temperature: weatherData.main.temp,
            weatherDescription: weatherData.weather.first?.description ?? "N/
A",
            baseCurrency: baseCurrency,
            targetCurrency: targetCurrency,
            exchangeRate: exchangeRate
        )
    }
}

struct TripInfo {
    let city: String
    let temperature: Double
    let weatherDescription: String
    let baseCurrency: String
    let targetCurrency: String
    let exchangeRate: Double
}
```

You can use the TripService class to fetch trip information like this:

swift

Copy code

```
Task {
    do {
        let tripService = TripService()
        let tripInfo = try await tripService.fetchTripInfo(city: "London",
baseCurrency: "USD", targetCurrency: "EUR")
        print("City: \(tripInfo.city)")
        print("Temperature: \(tripInfo.temperature)°C")
        print("Weather description: \(tripInfo.weatherDescription)")
    }
}
```

```

        print("Exchange rate from \(tripInfo.baseCurrency) to \(
tripInfo.targetCurrency): \(tripInfo.exchangeRate)")
    } catch {
        print("Error fetching trip information: \(error)")
    }
}

```

The TripService class consumes the WeatherService and CurrencyService classes to provide a higher-level abstraction for fetching trip-related information, such as weather and currency exchange rate.



1 / 1

That's great. Now please add to TripService a method of saving a list of trips to disk and restoring it.

1 / 1

To save and restore a list of trips to disk, we will add two methods to the TripService class: saveTripsToDisk(trips: [TripInfo]) and restoreTripsFromDisk(). We will use the FileManager and JSONEncoder / JSONDecoder to save and load the trip information as JSON.

First, import the required libraries:

swift

Copy code

```

import Foundation
import _Concurrency

```

Next, update the TripInfo struct to conform to the Codable protocol:

swift

Copy code

```

struct TripInfo: Codable {
    let city: String
    let temperature: Double
}

```

```

    let weatherDescription: String
    let baseCurrency: String
    let targetCurrency: String
    let exchangeRate: Double
}

```

Now, add the `saveTripsToDisk(trips: [TripInfo])` and `restoreTripsFromDisk()` methods to the `TripService` class:

swift

Copy code

```

class TripService {
    private let weatherService = WeatherService()
    private let currencyService = CurrencyService()
    private let tripsFileName = "trips.json"

    // Other existing methods

    func saveTripsToDisk(trips: [TripInfo]) throws {
        let encoder = JSONEncoder()
        let data = try encoder.encode(trips)

        guard let documentsDirectory =
FileManager.default.urls(for: .documentDirectory, in: .userDomainMask).first
        else {
            throw TripError.documentsDirectoryNotFound
        }

        let fileURL =
documentsDirectory.appendingPathComponent(tripsFileName)
        try data.write(to: fileURL)
    }

    func restoreTripsFromDisk() async throws -> [TripInfo] {
        guard let documentsDirectory =
FileManager.default.urls(for: .documentDirectory, in: .userDomainMask).first
        else {
            throw TripError.documentsDirectoryNotFound
        }

        let fileURL =
documentsDirectory.appendingPathComponent(tripsFileName)
        let data = try Data(contentsOf: fileURL)

        let decoder = JSONDecoder()
        let trips = try decoder.decode([TripInfo].self, from: data)
        return trips
    }
}

```



```

    }
}

enum TripError: Error {
    case documentsDirectoryNotFound
}

```

You can now save and restore a list of trips to disk using the TripService class like this:

swift

Copy code

```

Task {
    do {
        let tripService = TripService()
        let tripInfo1 = try await tripService.fetchTripInfo(city: "London",
baseCurrency: "USD", targetCurrency: "EUR")
        let tripInfo2 = try await tripService.fetchTripInfo(city: "Paris",
baseCurrency: "USD", targetCurrency: "EUR")

        // Save the trips to disk
        try tripService.saveTripsToDisk(trips: [tripInfo1, tripInfo2])

        // Restore the trips from disk
        let restoredTrips = try await tripService.restoreTripsFromDisk()

        for trip in restoredTrips {
            print("City: \(trip.city)")
            print("Temperature: \(trip.temperature)°C")
            print("Weather description: \(trip.weatherDescription)")
            print("Exchange rate from \(trip.baseCurrency) to \(
(trip.targetCurrency): \(trip.exchangeRate)")
            print("-----")
        }
    } catch {
        print("Error: \(error)")
    }
}

```

The saveTripsToDisk(trips: [TripInfo]) and restoreTripsFromDisk() methods in the TripService class allow you to save a list of trips as a JSON file and restore the list from the file when needed.



1 / 1

Let's work on the three services' initializers. The first two services should take their API keys and base URLs as initializer arguments rather than baking them into the class. TripService should take the other two services and the trips filename as initializer arguments. Also, let's call it a filename, not a file name.

1 / 1

To update the initializers, we'll modify the WeatherService, CurrencyService, and TripService classes according to your suggestions. Here's the updated version of the classes:

swift

Copy code

```
import Foundation
import _Concurrency

class WeatherService {
    private let apiKey: String
    private let baseURL: String

    init(apiKey: String, baseURL: String = "https://api.openweathermap.org/data/2.5/weather") {
        self.apiKey = apiKey
        self.baseURL = baseURL
    }

    // Other existing methods
}

class CurrencyService {
    private let apiKey: String
    private let baseURL: String

    init(apiKey: String, baseURL: String = "https://api.exchangeratesapi.io/v1/latest") {
        self.apiKey = apiKey
        self.baseURL = baseURL
    }
}
```

```

    }

    // Other existing methods
}

class TripService {
    private let weatherService: WeatherService
    private let currencyService: CurrencyService
    private let tripsFilename: String

    init(weatherService: WeatherService, currencyService: CurrencyService,
tripsFilename: String = "trips.json") {
        self.weatherService = weatherService
        self.currencyService = currencyService
        self.tripsFilename = tripsFilename
    }

    // Other existing methods
}

```

Now you can initialize the services with the required parameters:

swift

Copy code

```

let weatherApiKey = "<YOUR_WEATHER_API_KEY>"
let currencyApiKey = "<YOUR_CURRENCY_API_KEY>"
let weatherService = WeatherService(apiKey: weatherApiKey)
let currencyService = CurrencyService(apiKey: currencyApiKey)
let tripService = TripService(weatherService: weatherService,
currencyService: currencyService)

```

This makes the services more flexible and allows you to easily configure the API keys and base URLs during initialization. The TripService class now takes the other two services and the trips filename as initializer arguments, allowing you to easily swap or configure services as needed.



1 / 1

That's great. Please organize the service initialization into an app delegate method to be called at launch. We will decide where to put the services later on.

1 / 1

To organize the service initialization into an AppDelegate method, you can create a new method named `initializeServices()`. This method will be called from the `application(_:didFinishLaunchingWithOptions:)` method, which is called at app launch.

First, import `UIKit` in your `AppDelegate.swift`:

swift

Copy code

```
import UIKit
```

Next, create the `initializeServices()` method and update the `application(_:didFinishLaunchingWithOptions:)` method in the AppDelegate class:

swift

Copy code

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?

    func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey:
Any]?) -> Bool {
        initializeServices()
        return true
    }

    func initializeServices() {
        let weatherApiKey = "<YOUR_WEATHER_API_KEY>"
        let currencyApiKey = "<YOUR_CURRENCY_API_KEY>"
        let weatherService = WeatherService(apiKey: weatherApiKey)
        let currencyService = CurrencyService(apiKey: currencyApiKey)
        let tripService = TripService(weatherService: weatherService,
currencyService: currencyService)

        // Store or use the initialized services as needed
    }
}
```

The `initializeServices()` method initializes the `WeatherService`, `CurrencyService`, and `TripService` classes with the required API keys. The method is called from the `application(_:didFinishLaunchingWithOptions:)` method at app launch. You can

decide how to store or use the services later on, based on your app's architecture and requirements.



1 / 1

Sorry, I misspoke. this is a SwiftUI app so let's put it in the App type.

1 / 1

In a SwiftUI app, you can organize the service initialization in the App type by using the `@StateObject` property wrapper to create and store instances of the services.

First, import SwiftUI in your @main file (e.g., `YourAppNameApp.swift`):

swift

Copy code

```
import SwiftUI
```

Next, update your app's main struct to initialize and store the service instances:

swift

Copy code

```
@main
struct YourAppNameApp: App {
    @StateObject private var weatherService = WeatherService(apiKey:
"<YOUR_WEATHER_API_KEY>")
    @StateObject private var currencyService = CurrencyService(apiKey:
"<YOUR_CURRENCY_API_KEY>")
    @StateObject private var tripService: TripService

    init() {
        tripService = TripService(weatherService: weatherService,
currencyService: currencyService)
    }

    var body: some Scene {
```

```

WindowGroup {
    ContentView()
        .environmentObject(weatherService)
        .environmentObject(currencyService)
        .environmentObject(tripService)
}
}

```

In the YourAppNameApp struct, the WeatherService, CurrencyService, and TripService instances are created and stored using the @StateObject property wrapper. The tripService instance is initialized in the init() method using the initialized weatherService and currencyService instances.

The services are then passed to the ContentView using the .environmentObject modifier. This makes them available for use throughout your app's view hierarchy.



1 / 1

I won't add any of these three services to the environment because views don't need to interact with them directly. So I have the following in my App type:

```

private var weatherService: WeatherService
private var currencyService: CurrencyService
private var tripService: TripService

init() {
    weatherService = WeatherService(apiKey: "<YOUR_WEATHER_API_KEY>")
    currencyService = CurrencyService(apiKey: "<YOUR_CURRENCY_API_KEY>")
    tripService = TripService(weatherService: weatherService, currencyService:
currencyService)
}

var body: some Scene {
    WindowGroup {
        ContentView()
    }
}

```

Now then, let's make a new type that consumes the TripService. It should conform to ObservableObject and store in memory the list of trips from disk. It should also expose ways to add a new trip to the list or clear the list. Then add an instance of this type to the environment at the app level.

1 / 1

You can create a new type called TripStore that conforms to ObservableObject and consumes the TripService. The TripStore will load the list of trips from disk and store them in memory. It will also expose methods to add a new trip to the list or clear the list. Let's add an instance of this type to the environment at the app level.

First, import SwiftUI and Combine:

swift

Copy code

```
import SwiftUI
import Combine
```

Next, create the TripStore class:

swift

Copy code

```
class TripStore: ObservableObject {
    @Published private(set) var trips: [TripInfo]
    private let tripService: TripService

    init(tripService: TripService) {
        self.tripService = tripService
        do {
            trips = try tripService.restoreTripsFromDisk()
        } catch {
            print("Error loading trips from disk: \(error)")
            trips = []
        }
    }

    func addTrip(_ trip: TripInfo) {
        trips.append(trip)
        saveTripsToDisk()
    }

    func clearTrips() {
        trips.removeAll()
    }
}
```

```

        saveTripsToDisk()
    }

    private func saveTripsToDisk() {
        do {
            try tripService.saveTripsToDisk(trips: trips)
        } catch {
            print("Error saving trips to disk: \(error)")
        }
    }
}

```

Now, update the YourAppNameApp struct to create an instance of the TripStore and add it to the environment:

swift

Copy code

```

@main
struct YourAppNameApp: App {
    private var weatherService: WeatherService
    private var currencyService: CurrencyService
    private var tripService: TripService
    @StateObject private var tripStore: TripStore

    init() {
        weatherService = WeatherService(apiKey: "<YOUR_WEATHER_API_KEY>")
        currencyService = CurrencyService(apiKey: "<YOUR_CURRENCY_API_KEY>")
        tripService = TripService(weatherService: weatherService,
currencyService: currencyService)
        _tripStore = StateObject(wrappedValue: TripStore(tripService:
tripService))
    }

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(tripStore)
        }
    }
}

```

The TripStore class consumes the TripService and manages the in-memory list of trips. It loads the list of trips from disk during initialization, and it provides `addTrip(_:)` and `clearTrips()` methods to add a new trip to the list or clear the list, respectively. These actions will also save the updated list of trips to disk. The TripStore instance is added to the environment at the app level, making it accessible to the views that need it.



1 / 1

On the `_tripStore` line, I have "Escaping autoclosure captures mutating 'self' parameter"

1 / 1

My apologies for the confusion. In this case, you can initialize the `TripStore` instance directly in the property declaration using the `TripService` instance. Here's how you can update the `YourAppNameApp` struct:

swift

Copy code

```
@main
struct YourAppNameApp: App {
    private var weatherService: WeatherService
    private var currencyService: CurrencyService
    private var tripService: TripService
    @StateObject private var tripStore: TripStore

    init() {
        weatherService = WeatherService(apiKey: "<YOUR_WEATHER_API_KEY>")
        currencyService = CurrencyService(apiKey: "<YOUR_CURRENCY_API_KEY>")
        tripService = TripService(weatherService: weatherService,
currencyService: currencyService)
    }

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environmentObject(TripStore(tripService: tripService))
        }
    }
}
```

By initializing the `TripStore` instance directly in the `.environmentObject` modifier, you can avoid the "Escaping autoclosure captures mutating 'self' parameter" error. The

TripStore instance will be created and added to the environment when the ContentView is presented.



1 / 1

I took this approach instead:

```
@StateObject private var tripStore: TripStore

init() {
    let weatherService = WeatherService(apiKey: "<YOUR_WEATHER_API_KEY>")
    let currencyService = CurrencyService(apiKey: "<YOUR_CURRENCY_API_KEY>")
    let tripService = TripService(weatherService: weatherService, currencyService:
currencyService)

    let tripStore = TripStore(tripService: tripService)
    _tripStore = StateObject(wrappedValue: tripStore)
}

var body: some Scene {
    WindowGroup {
        ContentView()
        .environmentObject(tripStore)
    }
}
```

Now let's visit the ContentView and add some UI. First, we need a navigation stack whose root view is a trips screen. The trips screen should display a list backed by the store's list of trips and offer a way to use all the functions exposed by the TripStore. In the case of adding a trip, we can use a cancellable modal screen with form fields.

1 / 1

To create a TripsView that displays a list of trips and offers a way to use all the functions exposed by the TripStore, you can follow these steps:

1 Create a new SwiftUI file called `TripsView.swift` and import SwiftUI:

swift

Copy code

```
import SwiftUI
```

2 Define the `TripsView` struct that conforms to the View protocol:

swift

Copy code

```
struct TripsView: View {
    @EnvironmentObject var tripStore: TripStore
    @State private var showAddTripModal = false

    var body: some View {
        NavigationView {
            List {
                ForEach(tripStore.trips) { trip in
                    VStack(alignment: .leading) {
                        Text(trip.destination)
                            .font(.headline)
                        Text("From: \(trip.startDate, formatter:
dateFormatter) To: \(trip.endDate, formatter: dateFormatter)")
                            .font(.subheadline)
                    }
                }
                .onDelete(perform: deleteTrip)
            }
            .navigationTitle("Trips")
            .toolbar {
                ToolbarItemGroup(placement: .navigationBarTrailing) {
                    Button(action: {
                        showAddTripModal.toggle()
                    }) {
                        Image(systemName: "plus")
                    }
                }
                .sheet(isPresented: $showAddTripModal) {
                    AddTripView()
                        .environmentObject(tripStore)
                }

                EditButton()
            }
        }
    }

    private func deleteTrip(at offsets: IndexSet) {
        tripStore.trips.remove(atOffsets: offsets)
    }
}
```

```

}

private let dateFormatter: DateFormatter = {
    let formatter = DateFormatter()
    formatter.dateStyle = .short
    formatter.timeStyle = .none
    return formatter
}()

```

This TripsView displays a list of trips backed by the TripStore. It includes an "Add" button and an "Edit" button in the navigation bar. The "Add" button presents a modal screen to add a new trip. You'll create the AddTripView next.

3 Create a new SwiftUI file called AddTripView.swift and import SwiftUI:

swift

Copy code

```
import SwiftUI
```

4 Define the AddTripView struct that conforms to the View protocol:

swift

Copy code

```

struct AddTripView: View {
    @EnvironmentObject var tripStore: TripStore
    @Environment(\.presentationMode) var presentationMode

    @State private var destination = ""
    @State private var startDate = Date()
    @State private var endDate = Date()

    var body: some View {
        NavigationView {
            Form {
                Section {
                    TextField("Destination", text: $destination)
                }
                Section {
                    DatePicker("Start Date", selection: $startDate,
displayedComponents: [.date])
                    DatePicker("End Date", selection: $endDate,
displayedComponents: [.date])
                }
            }
            .navigationTitle("Add Trip")
            .toolbar {
                ToolbarItem(placement: .navigationBarLeading) {
                    Button("Cancel") {
                        presentationMode.wrappedValue.dismiss()
                    }
                }
            }
        }
    }
}

```

The `AddTripView` includes form fields to enter the destination and start and end dates for a new trip. When the "Save" button is tapped, a new trip is added to the `TripStore` and the modal screen is



The `.onDelete` modifier in `TripsView` shouldn't call a private function of `TripsView`. We can delete that function. Instead let's have `.onDelete` call a new function in `TripStore` that does the right thing.

AddTripsView expects an id: UUID property to exist in TripInfo. Let's add that.

1 / 1

Copy code

```

struct TripsView: View {
    @EnvironmentObject var tripStore: TripStore
    @State private var showAddTripModal = false

    var body: some View {
        NavigationView {
            List {
                ForEach(tripStore.trips) { trip in
                    VStack(alignment: .leading) {
                        Text(trip.destination)
                            .font(.headline)
                        Text("From: \(trip.startDate, formatter:
Self.dateFormatter) To: \(trip.endDate, formatter: Self.dateFormatter)")
                            .font(.subheadline)
                    }
                }
                .onDelete(perform: tripStore.deleteTrip)
            }
            .navigationTitle("Trips")
            .toolbar {
                ToolbarItemGroup(placement: .navigationBarTrailing) {
                    Button(action: {
                        showAddTripModal.toggle()
                    }) {
                        Image(systemName: "plus")
                    }
                    .sheet(isPresented: $showAddTripModal) {
                        AddTripView()
                            .environmentObject(tripStore)
                    }

                    EditButton()
                }
            }
        }
    }

    static private let dateFormatter: DateFormatter = {
        let formatter = DateFormatter()
        formatter.dateStyle = .short
        formatter.timeStyle = .none
        return formatter
    }()
}

```

2 Add a new deleteTrip(at:) function in TripStore:

swift

Copy code

```

class TripStore: ObservableObject {
    // ...

    func deleteTrip(at offsets: IndexSet) {
        trips.remove(atOffsets: offsets)
        saveTripsToDisk()
    }

    // ...
}

```

3 Add the id: UUID property to the TripInfo struct:

swift

Copy code

```

struct TripInfo: Identifiable, Codable {
    let id: UUID
    let destination: String
    let startDate: Date
    let endDate: Date
}

```

4 Remove the default values for the @State properties in AddTripView:

swift

Copy code

```

struct AddTripView: View {
    @EnvironmentObject var tripStore: TripStore
    @Environment(\.presentationMode) var presentationMode

    @State private var destination: String = ""
    @State private var startDate: Date
    @State private var endDate: Date

    var body: some View {
        // ...
    }
}

```

With these changes, the `.onDelete` modifier in `TripsView` calls the `deleteTrip(at:)` function from `TripStore`, the date formatter is a static property of `TripsView`, the `TripInfo` struct has an `id: UUID` property, and the default values for the `@State` properties in `AddTripView` have been removed.



1 / 1

Here's the definition I have for TripInfo now:

```
struct TripInfo: Codable {
    let id: UUID
    let city: String
    let temperature: Double
    let weatherDescription: String
    let baseCurrency: String
    let targetCurrency: String
    let exchangeRate: Double
    let startDate: Date
    let endDate: Date
}
```

Please update AddTripView to handle all these fields.

1 / 1

To update the AddTripView to handle all the fields in the new TripInfo definition, you can follow these steps:

1 Add new @State properties in AddTripView for the new fields:

swift

Copy code

```
@State private var city: String = ""
@State private var temperature: String = ""
@State private var weatherDescription: String = ""
@State private var baseCurrency: String = ""
@State private var targetCurrency: String = ""
@State private var exchangeRate: String = ""
```

2 Update the form in AddTripView to include the new fields:

swift

Copy code

```
var body: some View {
    NavigationView {
        Form {
            Section {
                TextField("City", text: $city)
                TextField("Temperature", text: $temperature)
                .keyboardType(.decimalPad)
            }
        }
    }
}
```



```

        TextField("Weather Description", text: $weatherDescription)
    }
    Section {
        TextField("Base Currency", text: $baseCurrency)
        TextField("Target Currency", text: $targetCurrency)
        TextField("Exchange Rate", text: $exchangeRate)
            .keyboardType(.decimalPad)
    }
    Section {
        DatePicker("Start Date", selection: $startDate,
displayedComponents: [.date])
        DatePicker("End Date", selection: $endDate,
displayedComponents: [.date])
    }
}
.navigationTitle("Add Trip")
.toolbar {
    ToolbarItem(placement: .navigationBarLeading) {
        Button("Cancel") {
            presentationMode.wrappedValue.dismiss()
        }
    }
    ToolbarItem(placement: .navigationBarTrailing) {
        Button("Save") {
            let trip = TripInfo(
                id: UUID(),
                city: city,
                temperature: Double(temperature) ?? 0.0,
                weatherDescription: weatherDescription,
                baseCurrency: baseCurrency,
                targetCurrency: targetCurrency,
                exchangeRate: Double(exchangeRate) ?? 0.0,
                startDate: startDate,
                endDate: endDate
            )
            tripStore.addTrip(trip)
            presentationMode.wrappedValue.dismiss()
        }
        .disabled(city.isEmpty)
    }
}
}
}
}

```

With these changes, the AddTripView now includes form fields for all the properties in the TripInfo definition. The "Save" button is enabled only when the city field is not empty. The temperature and exchangeRate fields are converted from String to Double when creating a new TripInfo instance. If the conversion fails, the default value is set

to 0.0.



1 / 1

Here's TripService.fetchTripInfo.

```
func fetchTripInfo(city: String, baseCurrency: String, targetCurrency: String) async
throws -> TripInfo {
    let weatherData = try await weatherService.fetchWeatherData(city: city)
    let exchangeRate = try await currencyService.fetchExchangeRate(baseCurrency:
baseCurrency, targetCurrency: targetCurrency)

    return TripInfo(
        city: weatherData.name,
        temperature: weatherData.main.temp,
        weatherDescription: weatherData.weather.first?.description ?? "N/A",
        baseCurrency: baseCurrency,
        targetCurrency: targetCurrency,
        exchangeRate: exchangeRate
    )
}
```

Please update it to satisfy all fields of TripInfo.

1 / 1

To update TripService.fetchTripInfo to satisfy all fields of TripInfo, you need to add startDate and endDate parameters to the function and pass them when creating a TripInfo instance:

swift

Copy code

```
func fetchTripInfo(city: String, baseCurrency: String, targetCurrency:
String, startDate: Date, endDate: Date) async throws -> TripInfo {
    let weatherData = try await weatherService.fetchWeatherData(city: city)
```

```

    let exchangeRate = try await
currencyService.fetchExchangeRate(baseCurrency: baseCurrency, targetCurrency:
targetCurrency)

    return TripInfo(
        id: UUID(),
        city: weatherData.name,
        temperature: weatherData.main.temp,
        weatherDescription: weatherData.weather.first?.description ?? "N/A",
        baseCurrency: baseCurrency,
        targetCurrency: targetCurrency,
        exchangeRate: exchangeRate,
        startDate: startDate,
        endDate: endDate
    )
}

```

Now the `fetchTripInfo` function takes `startDate` and `endDate` parameters and passes them when creating the `TripInfo` instance, satisfying all the fields of the `TripInfo` struct.



1 / 1

Oh, now that you mention it, The inputs to this function are the fields we want to use in `AddTripsView`. Then when we complete the form we can call a new function in `TripStore` that uses `TripService.fetchTripInfo` and then adds the resulting `TripInfo` to the list.

While we're at it, let's remove `TripStore.clearTrips`.

1 / 1

I understand your requirements. Here are the updates you need to make:

- 1 Update `AddTripView` to only include the fields required for calling the `fetchTripInfo` function:

swift

Copy code

```
struct AddTripView: View {
    @EnvironmentObject var tripStore: TripStore
    @Environment(\.presentationMode) var presentationMode

    @State private var city: String = ""
    @State private var baseCurrency: String = ""
    @State private var targetCurrency: String = ""
    @State private var startDate: Date = Date()
    @State private var endDate: Date = Date()

    // ...
}
```

2 Update the form in AddTripView to only include the required fields:

swift

Copy code

```
var body: some View {
    // ...

    Form {
        Section {
            TextField("City", text: $city)
            TextField("Base Currency", text: $baseCurrency)
            TextField("Target Currency", text: $targetCurrency)
        }
        Section {
            DatePicker("Start Date", selection: $startDate,
displayedComponents: [.date])
            DatePicker("End Date", selection: $endDate, displayedComponents:
[.date])
        }
    }

    // ...
}
```

3 Add a new function in TripStore that uses TripService.fetchTripInfo and adds the resulting TripInfo to the list:

swift

Copy code

```
class TripStore: ObservableObject {
    // ...

    func addTrip(city: String, baseCurrency: String, targetCurrency: String,
startDate: Date, endDate: Date) async {
        do {
            let tripInfo = try await tripService.fetchTripInfo(city: city,
baseCurrency: baseCurrency, targetCurrency: targetCurrency, startDate:
```

```

startDate, endDate: endDate)
    DispatchQueue.main.async {
        self.trips.append(tripInfo)
        self.saveTripsToDisk()
    }
} catch {
    print("Error fetching trip info: \(error)")
}
}

// ...
}

```

- 4 Update the "Save" button action in AddTripView to call the new addTrip function from TripStore:

swift

Copy code

```

Button("Save") {
    Task {
        await tripStore.addTrip(city: city, baseCurrency: baseCurrency,
targetCurrency: targetCurrency, startDate: startDate, endDate: endDate)
    }
    presentationMode.wrappedValue.dismiss()
}
.disabled(city.isEmpty)

```

- 5 Remove TripStore.clearTrips function:

swift

Copy code

```

class TripStore: ObservableObject {
    // Remove the clearTrips function

    // ...
}

```

These updates change the form in AddTripView to only include the required fields, add a new function in TripStore to fetch the trip info and add it to the list, and remove the clearTrips function from TripStore.



This is great. Now we need to be able to mock the TripService for testing. First, let's create a protocol with all the functionality that TripStore uses and conform TripService to that. Then change TripStore to use an instance of the protocol type instead of specifically a TripService. Finally create a FakeTripStore that also conforms and that is easy to configure with test data.

1 / 1

I think you meant to create a FakeTripService instead of a FakeTripStore. Here's how you can create a protocol for the functionality used by TripStore, conform TripService to that protocol, and create a FakeTripService for testing:

1 Create a protocol named TripServiceProtocol:

swift

Copy code

```
protocol TripServiceProtocol {  
    func fetchTripInfo(city: String, baseCurrency: String, targetCurrency:  
String, startDate: Date, endDate: Date) async throws -> TripInfo  
}
```

2 Make TripService conform to the TripServiceProtocol:

swift

Copy code

```
class TripService: TripServiceProtocol {  
    // ...  
  
    // This function already exists in TripService, so no changes are needed  
    func fetchTripInfo(city: String, baseCurrency: String, targetCurrency:  
String, startDate: Date, endDate: Date) async throws -> TripInfo {  
        // ...  
    }  
  
    // ...  
}
```

3 Update TripStore to use an instance of TripServiceProtocol instead of TripService:

swift

Copy code

```
class TripStore: ObservableObject {  
    @Published private(set) var trips: [TripInfo] = []  
    private var tripService: TripServiceProtocol  
  
    init(tripService: TripServiceProtocol) {  
        self.tripService = tripService  
    }  
}
```

```

        loadTripsFromDisk()
    }

    // ...
}

```

- 4 Create a FakeTripService that conforms to the TripServiceProtocol and can be configured with test data:

swift

Copy code

```

class FakeTripService: TripServiceProtocol {
    private var testData: [TripInfo]

    init(testData: [TripInfo]) {
        self.testData = testData
    }

    func fetchTripInfo(city: String, baseCurrency: String, targetCurrency:
String, startDate: Date, endDate: Date) async throws -> TripInfo {
        return testData.first(where: { $0.city == city }) ?? TripInfo(
            id: UUID(),
            city: "Not found",
            temperature: 0.0,
            weatherDescription: "N/A",
            baseCurrency: baseCurrency,
            targetCurrency: targetCurrency,
            exchangeRate: 1.0,
            startDate: startDate,
            endDate: endDate
        )
    }
}

```

Now you can create an instance of FakeTripService with test data and use it when initializing TripStore for testing purposes.



1 / 1

Yes I did mean to create a FakeTripService. Thank you for catching that mistake.

I'm using a new piece of Swift syntax that decorates protocol existentials with "any".

Readers will appreciate this modern update:

```
@Published private(set) var trips: [TripInfo] = []
private var tripService: any TripServiceProtocol

init(tripService: any TripServiceProtocol) {
    self.tripService = tripService
    do {
        trips = try tripService.restoreTripsFromDisk()
    } catch {
        print("Error loading trips from disk: \(error)")
        trips = []
    }
}
```

I'm also adding saveTripsToDisk and restoreTripsFromDisk to the protocol:

```
protocol TripServiceProtocol {
    func fetchTripInfo(city: String, baseCurrency: String, targetCurrency: String,
        startDate: Date, endDate: Date) async throws -> TripInfo
    func saveTripsToDisk(trips: [TripInfo]) throws
    func restoreTripsFromDisk() throws -> [TripInfo]
}
```

Please include implementations of these in FakeTripService.

1 / 1

I understand your requirements. Here's the updated TripServiceProtocol with the new functions and the implementation of these functions in FakeTripService:

1 Update the TripServiceProtocol:

swift

Copy code

```
protocol TripServiceProtocol {
    func fetchTripInfo(city: String, baseCurrency: String, targetCurrency:
String, startDate: Date, endDate: Date) async throws -> TripInfo
    func saveTripsToDisk(trips: [TripInfo]) throws
    func restoreTripsFromDisk() throws -> [TripInfo]
}
```

2 Implement the new functions in FakeTripService:

swift

Copy code


```

class FakeTripService: TripServiceProtocol {
    private var testData: [TripInfo]

    init(testData: [TripInfo]) {
        self.testData = testData
    }

    func fetchTripInfo(city: String, baseCurrency: String, targetCurrency:
String, startDate: Date, endDate: Date) async throws -> TripInfo {
        // ...
    }

    func saveTripsToDisk(trips: [TripInfo]) throws {
        // You can either save the trips to a temporary file on disk for
testing purposes
        // or update the testData variable directly as per your testing
requirements
        testData = trips
    }

    func restoreTripsFromDisk() throws -> [TripInfo] {
        // You can either read the trips from the temporary file on disk
        // or return the testData variable directly as per your testing
requirements
        return testData
    }
}

```

Now, FakeTripService conforms to the TripServiceProtocol and provides the required implementations for saveTripsToDisk and restoreTripsFromDisk.

Additionally, you have updated the TripStore to use the any keyword for protocol existentials, which is a modern syntax improvement in Swift.



1 / 1

Awesome. Now let's configure the preview providers for both views with TripStore instances that take a FakeTripService. In AddTripView, we can leave the test data blank, but for TripsView let's populate it with a few different trips.

1 / 1

Here's how you can configure the preview providers for both views using a TripStore instance with a FakeTripService.

1 Update the preview provider for TripsView:

swift

Copy code

```
struct TripsView_Previews: PreviewProvider {
    static var previews: some View {
        let testData = [
            TripInfo(id: UUID(), city: "New York", temperature: 22.0,
weatherDescription: "Sunny", baseCurrency: "USD", targetCurrency: "EUR",
exchangeRate: 0.85, startDate: Date(), endDate: Date().addingTimeInterval(60
* 60 * 24)),
            TripInfo(id: UUID(), city: "Paris", temperature: 18.0,
weatherDescription: "Cloudy", baseCurrency: "EUR", targetCurrency: "USD",
exchangeRate: 1.18, startDate: Date(), endDate: Date().addingTimeInterval(60
* 60 * 24))
        ]
        let fakeTripService = FakeTripService(testData: testData)
        let tripStore = TripStore(tripService: any fakeTripService)

        return TripsView()
            .environmentObject(tripStore)
    }
}
```

This will populate the TripsView preview with two sample trips.

2 Update the preview provider for AddTripView:

swift

Copy code

```
struct AddTripView_Previews: PreviewProvider {
    static var previews: some View {
        let testData: [TripInfo] = []
        let fakeTripService = FakeTripService(testData: testData)
        let tripStore = TripStore(tripService: any fakeTripService)

        return AddTripView()
            .environmentObject(tripStore)
    }
}
```

This will configure the AddTripView preview with an empty FakeTripService.

Now, both views' previews are configured with TripStore instances using FakeTripService, allowing you to see sample data and test the views more effectively.



1 / 1

To be clear, you don't use `any` as a prefix on a value of type `any <protocol>`. It's just used when you define the type of a property or parameter or something like that, and only when it's an existential. I've fixed it in the preview providers.

OK this is really good. Let's add one more preview to the AddTripsView preview provider whose fields are already populated.

Next, please add a trip screen that's pushed on the navigation stack when the user taps a list item on the TripsView. It should display all the information about the trip, except the UUID which is just for bookkeeping.

1 / 1

I apologize for the confusion with the any keyword. I understand your clarification.

- 1 To add an additional preview to the AddTripView preview provider with pre-populated fields, you can provide a Binding to the @State properties with sample data:

swift

Copy code

```
struct AddTripView_Previews: PreviewProvider {
    static var previews: some View {
        let testData: [TripInfo] = []
        let fakeTripService = FakeTripService(testData: testData)
        let tripStore = TripStore(tripService: any fakeTripService)

        let sampleCity = "San Francisco"
        let sampleBaseCurrency = "USD"
```

```

        let sampleTargetCurrency = "EUR"

        return Group {
            AddTripView()
                .environmentObject(tripStore)

            AddTripView(city: .constant(sampleCity),
baseCurrency: .constant(sampleBaseCurrency),
targetCurrency: .constant(sampleTargetCurrency))
                .environmentObject(tripStore)
        }
    }
}

```

- 2 To create a trip screen that displays the trip information when a list item is tapped in TripsView, you can create a new SwiftUI view called TripDetailView. Then, in TripsView, use the `.navigationLink` modifier to navigate to the new view:

swift

Copy code

// TripDetailView.swift

```

import SwiftUI

struct TripDetailView: View {
    let trip: TripInfo

    var body: some View {
        VStack(alignment: .leading, spacing: 10) {
            Text("City: \(trip.city)")
            Text("Temperature: \(trip.temperature, specifier: "%.1f") °C")
            Text("Weather Description: \(trip.weatherDescription)")
            Text("Base Currency: \(trip.baseCurrency)")
            Text("Target Currency: \(trip.targetCurrency)")
            Text("Exchange Rate: \(trip.exchangeRate, specifier: "%.4f")")
            Text("Start Date: \(trip.startDate, formatter: dateFormatter)")
            Text("End Date: \(trip.endDate, formatter: dateFormatter)")
        }
        .padding()
        .navigationTitle("Trip Details")
    }

    private var dateFormatter: DateFormatter {
        let formatter = DateFormatter()
        formatter.dateStyle = .medium
        formatter.timeStyle = .none
        return formatter
    }
}

```

```
}
```

Then, update the TripsView:

swift

Copy code

```
// TripsView.swift

// ...

ForEach(tripStore.trips) { trip in
    NavigationLink(destination: TripDetailView(trip: trip)) {
        Text(trip.city)
    }
}
.onDelete(perform: tripStore.deleteTrip)

// ...
```

Now, when a user taps a list item on the TripsView, they will be navigated to the TripDetailView with the selected trip's information.



1 / 1

In TripDetailView, please wrap the content in a scroll view and align the content near the left edge of the screen. Group its content into neat sections.

Fill out the preview provider too.

1 / 1

To improve the TripDetailView, you can wrap the content in a ScrollView, align the content near the left edge of the screen, and group the content into sections. Here's the updated TripDetailView:

swift

Copy code

```

import SwiftUI

struct TripDetailView: View {
    let trip: TripInfo

    var body: some View {
        ScrollView {
            VStack(alignment: .leading, spacing: 20) {
                VStack(alignment: .leading, spacing: 10) {
                    Text("Location").font(.headline)
                    Text("City: \(trip.city)")
                    Text("Temperature: \(trip.temperature, specifier: "%.1f")
°C")
                    Text("Weather Description: \(trip.weatherDescription)")
                }

                VStack(alignment: .leading, spacing: 10) {
                    Text("Currency").font(.headline)
                    Text("Base Currency: \(trip.baseCurrency)")
                    Text("Target Currency: \(trip.targetCurrency)")
                    Text("Exchange Rate: \(trip.exchangeRate, specifier:
("%.4f")")")
                }

                VStack(alignment: .leading, spacing: 10) {
                    Text("Dates").font(.headline)
                    Text("Start Date: \(trip.startDate, formatter:
dateFormatter)")
                    Text("End Date: \(trip.endDate, formatter:
dateFormatter)")
                }
            }
            .padding()
            .frame(maxWidth: .infinity, alignment: .leading)
        }
        .navigationTitle("Trip Details")
    }

    private var dateFormatter: DateFormatter {
        let formatter = DateFormatter()
        formatter.dateStyle = .medium
        formatter.timeStyle = .none
        return formatter
    }
}

```

For the TripDetailView preview provider, you can create a sample TripInfo instance

and use it to display the preview:

swift

Copy code

```
struct TripDetailView_Previews: PreviewProvider {
    static var previews: some View {
        let sampleTrip = TripInfo(
            id: UUID(),
            city: "New York",
            temperature: 22.0,
            weatherDescription: "Sunny",
            baseCurrency: "USD",
            targetCurrency: "EUR",
            exchangeRate: 0.85,
            startDate: Date(),
            endDate: Date().addingTimeInterval(60 * 60 * 24)
        )

        TripDetailView(trip: sampleTrip)
    }
}
```

Now, the TripDetailView is more organized, and the preview provider displays the view with a sample trip.



1 / 1

Now let's add a favoriting feature. On the Trip Details screen we want a heart button in the toolbar. It reflects a boolean in TripInfo and can be tapped to toggle it. Update all the code needed to make this happen.

1 / 1

To add the favoriting feature, you'll need to make a few updates:

- 1 Add a isFavorite property to TripInfo:

swift

Copy code

```
struct TripInfo: Codable, Identifiable {
    let id: UUID
    let city: String
    let temperature: Double
    let weatherDescription: String
    let baseCurrency: String
    let targetCurrency: String
    let exchangeRate: Double
    let startDate: Date
    let endDate: Date
    var isFavorite: Bool
}
```

2 Add a method to TripStore for toggling the isFavorite property of a trip:

swift

Copy code

```
class TripStore: ObservableObject {
    // ...

    func toggleFavorite(trip: TripInfo) {
        if let index = trips.firstIndex(where: { $0.id == trip.id }) {
            trips[index].isFavorite.toggle()
            do {
                try tripService.saveTripsToDisk(trips: trips)
            } catch {
                print("Error saving trips to disk: \(error)")
            }
        }
    }
}
```

3 Update the TripDetailView to display the favorite button in the toolbar:

swift

Copy code

```
import SwiftUI

struct TripDetailView: View {
    @EnvironmentObject var tripStore: TripStore
    let trip: TripInfo

    var body: some View {
        // ...
        .toolbar {
            ToolbarItem(placement: .navigationBarTrailing) {
                Button(action: {
                    tripStore.toggleFavorite(trip: trip)
                }) {

```


4 Don't forget to update the FakeTripService since the TripInfo structure has changed:

```
class FakeTripService: TripServiceProtocol {
    // ...

    func fetchTripInfo(city: String, baseCurrency: String, targetCurrency:
String, startDate: Date, endDate: Date) async throws -> TripInfo {
        return TripInfo(
            id: UUID(),
            city: city,
            temperature: 25.0,
            weatherDescription: "Clear sky",
            baseCurrency: baseCurrency,
            targetCurrency: targetCurrency,
            exchangeRate: 1.0,
            startDate: startDate,
            endDate: endDate,
            isFavorite: false
        )
    }

    // ...
}
```

swift

Copy code

```
struct TripDetailView_Previews: PreviewProvider {
    static var previews: some View {
        let sampleTrip = TripInfo(
            id: UUID(),
            city: "New York",
            temperature: 22.0,
            weatherDescription: "Sunny",
            baseCurrency: "USD",
            targetCurrency: "EUR",
            exchangeRate: 0.85,
```

```

        startDate: Date(),
        endDate: Date().addingTimeInterval(60 * 60 * 24),
        isFavorite: false
    )

    let testData: [TripInfo] = [sampleTrip]
    let fakeTripService = FakeTripService(testData: testData)
    let tripStore = TripStore(tripService: any fakeTripService)

    return TripDetailView(trip: sampleTrip)
        .environmentObject(tripStore)
}

```

Now, the Trip Details screen includes a heart button in the toolbar that reflects the `isFavorite` property of the trip and can be tapped to toggle its state.



1 / 1

OK, let's make the `TripsView` into the root view of the App's window.

1 / 1

To make the `TripsView` the root view of the App's window, you'll need to update your `TripsApp` code. Replace the `ContentView` with `TripsView` in the body property:

swift

Copy code

```

import SwiftUI

@main
struct TripsApp: App {
    @StateObject private var tripStore: TripStore

    init() {
        let weatherService = WeatherService(apiKey: "<YOUR_WEATHER_API_KEY>")
        let currencyService = CurrencyService(apiKey:

```

```

"<YOUR_CURRENCY_API_KEY>")
    let tripService = TripService(weatherService: weatherService,
currencyService: currencyService)

    let tripStore = TripStore(tripService: any tripService)
    _tripStore = StateObject(wrappedValue: tripStore)
}

var body: some Scene {
    WindowGroup {
        TripsView()
        .environmentObject(tripStore)
    }
}
}

```

Now, the TripsView is the root view of the App's window, and you can see the list of trips when the app starts.



1 / 1

When there's no saved trips file, let's use an empty list.

1 / 1

To use an empty list when there's no saved trips file, you can update the TripService's `restoreTripsFromDisk()` method to return an empty list when the file is not found. You can achieve this by catching the specific error that is thrown when the file is not found and returning an empty list in that case:

swift

Copy code

```

import Foundation

class TripService {
    // ...

```

```

func restoreTripsFromDisk() throws -> [TripInfo] {
    let fileURL = try getTripsFileURL()

    do {
        let data = try Data(contentsOf: fileURL)
        let decoder = JSONDecoder()
        let trips = try decoder.decode([TripInfo].self, from: data)
        return trips
    } catch {
        if let nsError = error as NSError?, nsError.domain ==
NSCocoaErrorDomain, nsError.code == NSFileReadNoSuchFileError {
            return []
        } else {
            throw error
        }
    }
}

```

Now, when there's no saved trips file, the `restoreTripsFromDisk()` method returns an empty list. The `TripStore` will then use this empty list as the initial state for the `trips` property.



1 / 1

I've renamed `TripStore` to `TripState` throughout the project.

I'd like to expose a `TripService` in the environment, but let's use `.environment`, not `.environmentObject`, since a `TripService` has no observable properties and is just a utility service. Then, let's move the `TripState` property from `App` to `TripsView` so it only exists in that subtree of the view hierarchy.

1 / 1

To expose the `TripService` in the environment using `.environment` and move the `TripState` property from the `App` to the `TripsView`, you can follow these steps:

1 Create a custom environment key for TripService:

swift

Copy code

```
import SwiftUI

struct TripServiceKey: EnvironmentKey {
    static let defaultValue: TripService? = nil
}

extension EnvironmentValues {
    var tripService: TripService? {
        get { self[TripServiceKey.self] }
        set { self[TripServiceKey.self] = newValue }
    }
}
```

2 Update the TripsApp to provide the TripService in the environment using the .environment modifier:

swift

Copy code

```
import SwiftUI

@main
struct TripsApp: App {
    private var tripService: TripService

    init() {
        let weatherService = WeatherService(apiKey: "<YOUR_WEATHER_API_KEY>")
        let currencyService = CurrencyService(apiKey:
"<YOUR_CURRENCY_API_KEY>")
        tripService = TripService(weatherService: weatherService,
currencyService: currencyService)
    }

    var body: some Scene {
        WindowGroup {
            TripsView()
                .environment(\.tripService, tripService)
        }
    }
}
```

3 Move the TripState property from the App to the TripsView and update the TripsView to initialize it with the tripService provided in the environment:

swift

Copy code

```
import SwiftUI
```

```

struct TripsView: View {
    @Environment(\.tripService) private var tripService
    @StateObject private var tripState: TripState

    init() {
        if let tripService = tripService {
            _tripState = StateObject(wrappedValue: TripState(tripService: any
tripService))
        } else {
            fatalError("TripsView must be initialized with a TripService
provided in the environment.")
        }
    }

    // ...
}

```

Now, the TripService is exposed in the environment using `.environment`, and the TripState property only exists in the TripsView subtree of the view hierarchy.



1 / 1

Hmm, I don't like that after all. Let's back up to before my last message.

I've renamed TripStore to TripState throughout the project.

Much like we created a trip service protocol and fake trip service, please make a similar protocol and second implementation for WeatherService and CurrencyService and their one fetch method each.

1 / 1

Sure, let's create protocols and fake implementations for WeatherService and CurrencyService. First, define the protocols:

swift

Copy code

```
// WeatherServiceProtocol.swift
import Foundation

protocol WeatherServiceProtocol {
    func fetchWeatherData(city: String) async throws -> WeatherData
}

// CurrencyServiceProtocol.swift
import Foundation

protocol CurrencyServiceProtocol {
    func fetchExchangeRate(baseCurrency: String, targetCurrency: String)
    async throws -> Double
}
```

Now, create the fake implementations:

swift

Copy code

```
// FakeWeatherService.swift
import Foundation

class FakeWeatherService: WeatherServiceProtocol {
    private var fakeWeatherData: WeatherData

    init(fakeWeatherData: WeatherData) {
        self.fakeWeatherData = fakeWeatherData
    }

    func fetchWeatherData(city: String) async throws -> WeatherData {
        return fakeWeatherData
    }
}

// FakeCurrencyService.swift
import Foundation

class FakeCurrencyService: CurrencyServiceProtocol {
    private var fakeExchangeRate: Double

    init(fakeExchangeRate: Double) {
        self.fakeExchangeRate = fakeExchangeRate
    }

    func fetchExchangeRate(baseCurrency: String, targetCurrency: String)
    async throws -> Double {
        return fakeExchangeRate
    }
}
```

```
}
```

Next, update the `WeatherService` and `CurrencyService` classes to conform to their respective protocols:

swift

Copy code

```
// WeatherService.swift
import Foundation

class WeatherService: WeatherServiceProtocol {
    // ...
}

// CurrencyService.swift
import Foundation

class CurrencyService: CurrencyServiceProtocol {
    // ...
}
```

Now you have protocols and fake implementations for both `WeatherService` and `CurrencyService`. You can use these fake services for testing or previewing your views.



1 / 1

OK, let's try the `.environment` thing again but with a new object. I'd like to make a class named `Services` that has a property for the `TripService` but using the protocol type. We'll create that and store it instead of the `TripState` in the `App`. Then we'll supply the `Services` instance in `.environment` (again, it won't be an `ObservableObject`). Finally, we'll use that in `TripsView` to create the `TripState` and vend it to subviews.

1 / 1

To create a `Services` class that has a property for the `TripService` using the protocol type, and then use it in your `App` and `TripsView`, you can follow these steps:

1 Create the Services class:

swift

Copy code

```
// Services.swift
import Foundation

class Services {
    let tripService: any TripServiceProtocol

    init(tripService: any TripServiceProtocol) {
        self.tripService = tripService
    }
}
```

2 Create a custom environment key for Services:

swift

Copy code

```
// ServicesEnvironmentKey.swift
import SwiftUI

struct ServicesEnvironmentKey: EnvironmentKey {
    static let defaultValue: Services? = nil
}

extension EnvironmentValues {
    var services: Services? {
        get { self[ServicesEnvironmentKey.self] }
        set { self[ServicesEnvironmentKey.self] = newValue }
    }
}
```

3 Update the TripsApp to provide the Services instance in the environment using the .environment modifier:

swift

Copy code

```
import SwiftUI

@main
struct TripsApp: App {
    private var services: Services

    init() {
        let weatherService = WeatherService(apiKey: "<YOUR_WEATHER_API_KEY>")
        let currencyService = CurrencyService(apiKey:
"<YOUR_CURRENCY_API_KEY>")
        let tripService = TripService(weatherService: any weatherService,
currencyService: any currencyService)
    }
}
```

```

        services = Services(tripService: any tripService)
    }

    var body: some Scene {
        WindowGroup {
            TripsView()
                .environment(\.services, services)
        }
    }
}

```

- 4 Update the TripsView to create the TripState using the Services instance provided in the environment:

swift

Copy code

```

import SwiftUI

struct TripsView: View {
    @Environment(\.services) private var services
    @StateObject private var tripState: TripState

    init() {
        if let services = services {
            _tripState = StateObject(wrappedValue: TripState(tripService:
services.tripService))
        } else {
            fatalError("TripsView must be initialized with a Services
instance provided in the environment.")
        }
    }

    // ...
}

```

Now, you have a Services class that stores the TripService using the protocol type, and it is provided in the environment to be used in the TripsView to create the TripState. The TripState is now vended only to the subviews of TripsView.