Entain Tech Test - Neds Next To Go

The following is designed to be a summary of my approach to the task.

Requirements:

1. Clone my public repository here:
2. Download the project and navigate to the project folder
3. Install SwiftLint: **brew install swiftlint**
   - Note: The project contains a build run script that requires SwiftLint to be installed. This method also requires the installation of homebrew
4. Open the project in Xcode and Cmd+R to build and run the project
   - Note: I recommend running the project on a physical device, there appears to be issues with the API call timing out on a simulated device. Possibly there is some config missing in the project to allow this.

Architecture - Clean Architecture

The project will be laid out into **UI**, **Domain** and **Data** layer components (and **Core** components) .

UI Layer

The UI layer will have a View-ViewModel structure. This layer will contain an abstracted Information type(s) that are supplied by the lower-level UseCasesProvider.

Domain Layer

The use cases is where all business logic resides. A use case contains a maximum of 1 publicly exposed function that supplies a abstract Model type from the data it receives from the lower-level RepositoryProvider, via a mapper.

Data Layer

The data layer is where the projects repositories can be found. The role of the repository is to fetch, update etc from/to some external source - an API, database, content management system, some localised strings file etc.

In the same way the Domain layer abstracts its data out to an Information Type, the repository will abstract out to a Domain layer Model type. More than this though, the Data layer will contain a special type of its own - what I call the **ResponseMapper** - to ensure separation of the Responses decoding logic and the data useable at the higher-level Domain layer.

Swift 6 Structured Concurrency

The project conforms to swift 6's concurrency, note the Repository has been defined as an actor and data is carried across non-MainActor-isolated boundaries until it reaches the viewModel, where everything is isolated to the @MainActor. All types in this part of the stack have been conformed to **Sendable** for this purpose.

Networking

Finally, I will define an EndPoint protocol with default properties that wraps the HTTPURLRequest data up for supply to the Networking 'module'. While my 'in-production' approach would be to write multiple different conformances in this way i.e other endpoints for caching, for different levels of auth, endpoints configured to work explicitly with retrying - or combinations of each.

Limitation: For the purposes of this task I will stick to defining a base Endpoint. Also the Networking module is going to be incredibly simplistic. On top of this the Endpoint will have primitive types, in place of more robust custom types that make request processing code more expressive and readable.

Dependencies/Modules

Dependencies within the project should be frame-worked or packaged out. For the purposes of this project, I will write modularised components access and the like as though they are a separated library, where necessary. The modules defined in this way can be found in the 'Modules' folder

Unit tests

This is where the benefits of CA really come out. While there is a bit of overhead in writing the mocks for the various layer components, the providers and the DependencyGraph itself, once done, writing further unit tests that use these same components becomes trivial in terms of both complexity and unit test development hours overhead.

Scope: The test suites and their capacity to scale is proportionate to the quality of the mocks that are written. For the purposes of this task I will write mocks that contain functionality only up to what I need to demonstrate the approach, in production the aim would be to cover every possible scenario with mocks that have greater flexibility.

Limitations: The testing suite I have written here is **not** sufficient for production, to properly mock and test all the iteration of the race filtering through the use case and repository would take considerable time, additionally I have run out of time to write the remaining tests in the viewModel. Hopefully, the recognition is already there that a comprehensive and exhaustive test suite is minimum for production, and I hope there is enough there for you to see that finishing.

Additionally – there is no UI Testing.

Linting

I will use SwiftLint here and it will be added to the project as a build phase (although this should also be a Git Hook). You can find the rules I've defined in the project directory: .swiftlint.yml

Notes: Force unwrapping is permitted in the test target, but its strictly forbidden in the primary app target.

SwiftUI

Splash Screen: I realise this is not part of the spec, however my old gaming design days prevent me from allowing someone into the application without a formal invitation to into the 'Magic Circle'.

RacesScreen: I have kept as closely to the spec as possible at this point. Also, the SwiftUI is as best-practice as I feel I could get it in the time I gave myself and to my understanding of SwiftUI so far (see limitations)

Limitations: My SwiftUI expertise! It has become clear that my knowledge of SwiftUI is lacking and certainly my mastery of it is something yet to come.

Localization

You can find the Localization.strings file in the Core folder.