iOS app architecture

Designing a robust and scalable iOS app architecture is essential for building efficient, maintainable, and extendable applications. An effective architecture not only enables clear separation of concerns but also enhances the app's performance and ease of testing. This document explores the key components of modern iOS app architecture, with a focus on splitting responsibilities across various layers such as the UI, data, and domain (feature). The core patterns for building the app are: MVVM and dependency injection.

TABLE OF CONTENTS

- Principles of Dependency Injection
- Protocol-Based Approach

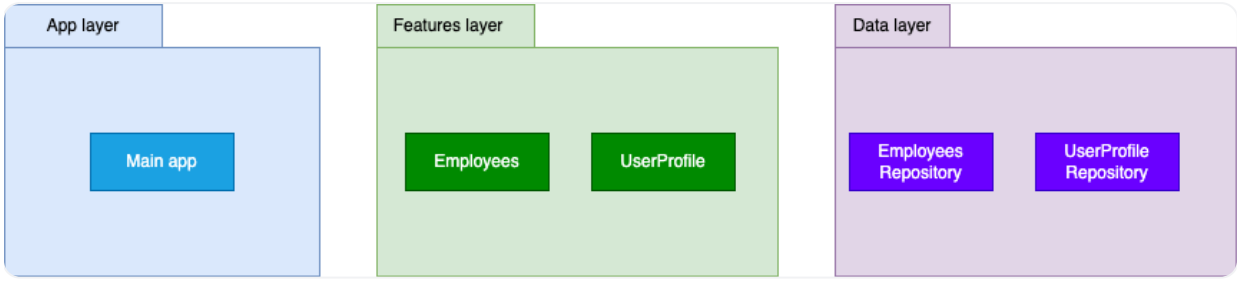
In app navigation

- Key Concepts of Flow Coordination
  - Centralized Navigation Management:
  - Decoupling Navigation from Views:
  - Stateful Navigation Control:
  - View Composition and Routing:
  - Navigation example
- Relationship Between Screens and the Coordinator
- Project example:

References

High overview

Understanding the architecture of an iOS application is crucial for building well-organized and scalable applications. The architecture is divided into three main layers: App Layer, Features Layer and Data Layer. Each layer encapsulates specific responsibilities to ensure the application's maintainability, reusability, and flexibility.



1. App Layer

The App Layer serves as the backbone of the application, where the initial setup and configuration of various modules occur. It is responsible for orchestrating the app's flow and presenting the domain use case screens. Key responsibilities include:

- **Module Configuration:** Integrates and configures different modules required by the application.

- **Entry Point:** Acts as the starting point of the application and coordinates the initialization of the Domain Layer.
- **Navigation and Routing:** Manages navigation logic and screen transitions between different domain modules.

2. Features Layer

The Features Layer is the core of the application, where various features are encapsulated as modules. Each module handles a specific feature, encapsulating both UI and business logic. This design ensures that the application can evolve feature-by-feature. Key components include:

- **Feature Modules:** Each module represents a feature such as user profiles, items lists, and item purchases.
- **UI Logic:** Manages the presentation logic and user interactions for specific features.
- **Business Logic:** Contains the rules and operations that manipulate data and control how the application behaves.
- **Navigation and Routing:** Manages navigation logic and screen transitions between screens of the feature.

3. Data Layer

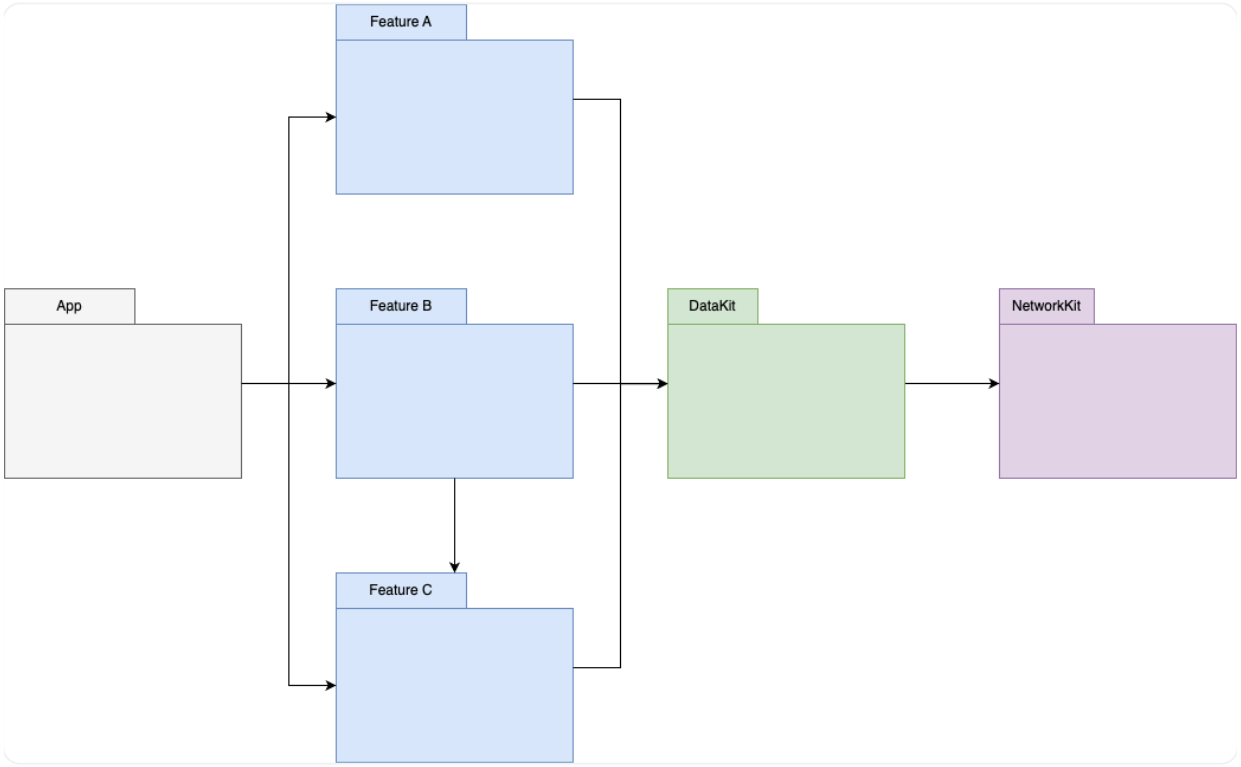
The Data Layer operates behind the scenes to handle data retrieval, caching, and transformations. It acts as an abstraction between the application and data sources, ensuring data integrity and consistency. Key responsibilities include:

- **Repositories:** Serve as the primary interface to fetch data, abstracting the complexities of different data sources.
- **Data Sources:** Include network services for fetching remote data and local persistence layers for cached or stored data.
- **Data Transformation:** Applies necessary transformations to raw data, presenting it in the format expected by the Domain Layer.

Data layer break down

The Data Layer plays a pivotal role in managing the flow of data within an iOS application. It acts as a conduit between the underlying data sources and the various application features, ensuring data consistency, transformation, and availability. This

section delves into the structure and responsibilities of the Data Layer, highlighting its components and interactions with other layers.



The Data layer consists of tow big components (modules): DataKit and NetworkKit.

NetworkKit

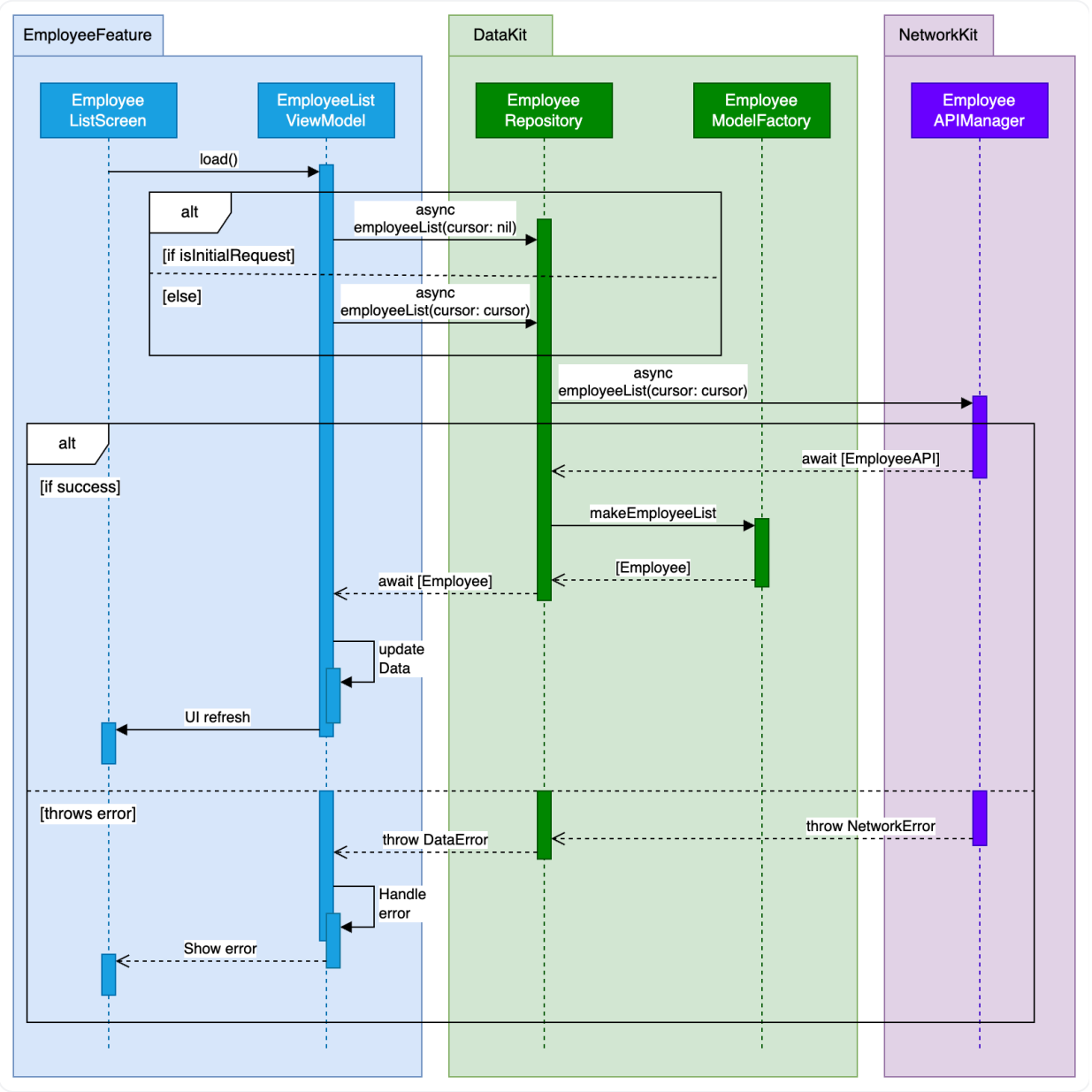
- Responsible for executing network calls and retrieving data from remote servers.
- Manages network configurations, request building, response handling.
- Acts as the bridge between the application's internal logic and external data APIs, ensuring secure and efficient communication.

DataKit

- Consists of repositories that serve as intermediaries between NetworkKit and the Feature Layers.
- Handles data retrieval from NetworkKit, applying necessary transformations and mappings to convert raw API data into a format suitable for the application.
- Provides a unified interface for the features to access and manipulate data without concerning themselves with the underlying network operations.
- this layer can be extended to support handling data coming from persistence layer.

Dive into implementation

Looking deeper into the architecture, below we have an example of how data flow will look for an `Employees list` feature. To keep things simple, think of this feature as a list of employees to be displayed.



The flow diagram performs the next actions:

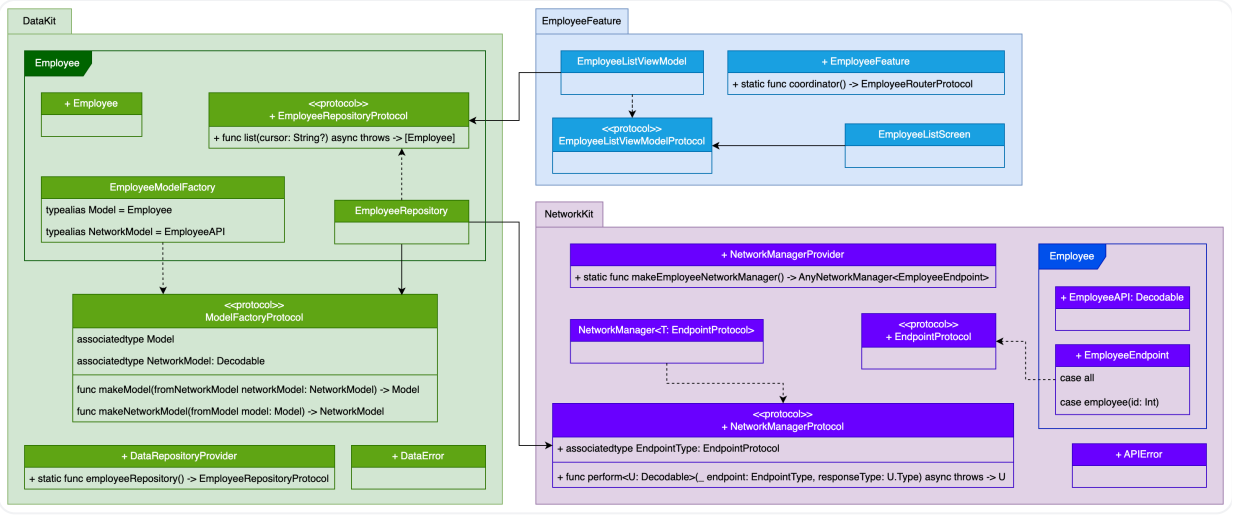
1. `UI` triggers a load data to `ViewModel;`
2. The `ViewModel` checks what page of employees should load then request it from `EmployeesRepository` from DataKit;
3. The `EmployeesRepository` fetches the remote employees via `EmployeesAPIManager` , the las is responsible to perform the actual network call. Important, the api manager returns api objects
4. The `EmployeesRepository` transforms `EmployeeApi` objects in `Employee` objects using the `EmployeesModelFactory`;

5. After the transformation is completed the repository is sending the `[Employee]` objects back to ViewModel;
6. Once the `EmployeesListViewModel` receive the new list its updates is state, which will trigger an UI update;

This sequence effectively demonstrates the layered approach employed to safeguard the separation of concerns, ensuring that network interactions and data transformations are confined within their designated components.

Classes architecture

Looking more deeply into architecture, we have next classes diagram:



In the above diagram we have a ground sniped for the classes architecture, starting from the feature layer going through data layer till network layer. The feature layer uses a basic MVVM architecture, we have `EmployeeListScreen` with an `EmployeeListViewModel` , then the view model has a reference to the `EmployeeRepositoryProtocol` to fetch the employees.

In the `DataKit` we have the next architecture:

- core implementation:
  - a public `DataRepositoryProvider` which is responsible to return the right repository;
  - an internal `ModelFactoryProtocol` that is responsible to transform objects;
- specific implementation for employee feature:
  - a public `EmployeeRepositoryProtocol` that represent the interface for the `EmployeeRepository` itself;
  - an `EmployeeModelFactory` that is responsible for specific mapping network employee objects and data layer objects;

    - a public `Employee` model, which is the actual model that will be passed to feature layer;

For the `NetworkKit` the class architecture is:

- core implementation:
    - a public `NetworkManagerProvider` responsible to return the right manager;
    - a public `NetworkManagerProtocol` that represents and interface for the `NetworkManager` ;
    - an internal generic `NetworkManager` that contains the actual implementation for networking calls, it builds the URLRequest and also send it to the service;
    - an internal `EndpointProtocol` defining an abstract layer of constrains of how an endpoint should behave;
- specific implementation for the feature:
    - an `EmployeeApi` that represents the network mode, that will be passed to data layer;
    - an `EmployeeEndpoint` where the actual implementation for the endpoint leaves;

Design Patterns

Design patterns are essential tools that guide the architecture of iOS applications, providing structured solutions to common design problems. The core patters proposed for this architecture are: Model-View-ViewModel (MVVM) and Dependency Injection (DI) that enhance the scalability, maintainability, and testability of applications.

MVVM patern

The Model-View-ViewModel (MVVM) pattern is a central architectural pattern adopted in the Domain Layer of the application to facilitate a clean separation of concerns and promote testability and maintainability.

Components of MVVM

Model:

- Represents the underlying data required by the application.
- Acts as the data source for the ViewModel.
- Typically includes data structures and classes obtained from the Data Layer.

View:

- The presentation layer with which the user interacts.
- Responsible only for displaying data provided by the ViewModel and reflecting user inputs.
- Should not have direct access to the Data Layer to preserve separation of concerns.

ViewModel:

- Acts as an intermediary between the View and the Model.
- Fetches data from the Data Layer and performs necessary transformations to prepare the data for presentation in the View.
- Handles user interactions, such as button taps or input changes, and updates the Model or triggers navigation actions in response.
- Maintains the state and logic required to present the View, allowing the View to remain as simple as possible.

Navigation Handling

Flow Coordinator:

- Handles the navigation logic separately from the MVVM structure.
- Coordinates the transition between Views and manages the presentation of new screens in a centralized manner.
- Ensures that ViewModels focus solely on responding to UI and business logic without becoming cluttered with navigation responsibilities.

Benefits of MVVM

- **Testability:** By keeping the business logic in the ViewModel, it becomes easier to write unit tests for the application's core functionalities without dealing with UI complexities.
- **Maintainability:** Clear separation of UI, business logic, and data handling facilitates smoother maintenance and updates as the application evolves.
- **Scalability:** As features grow, the modular nature of MVVM allows for the addition of new functionalities with minimal impact on existing code.

Dependency injection

Dependency Injection (DI) is a design pattern that enhances modularity and testability by decoupling the creation and management of object dependencies from their usage.

```swift
protocol ItemsRepositoryProtocol {
    func getItems() async throws -> [Item]
}

final class ItemsRepository: ItemsRepositoryProtocol {
    func getItems() async throws -> [Item] {
        []
    }
}

final class ItemsListViewModel {
    private let dependencies: Dependencies

    init(dependencies: Dependencies = .defaultOption) {
        self.dependencies = dependencies
    }
}

// MARK: - Dependencies
extension ItemsListViewModel {
    struct Dependencies {
        let itemsRepository: ItemsRepositoryProtocol

        static var defaultOption: Dependencies {
            .init(itemsRepository: ItemsRepository())
        }
    }
}
```

Principles of Dependency Injection

1. **Injection at Initialization:**

    - Dependencies are provided as inputs to an object's initializer, ensuring that all necessary collaborators are available when the object is created.
    - This approach promotes immutability and clarity, as dependencies are set once and for all during object creation.
2. **Default Values for Dependencies:**
    - Each dependency should have a default implementation or value, simplifying usage across the project.
    - By providing sensible defaults, developers can instantiate objects easily without manually providing each dependency, streamlining the development process.

Protocol-Based Approach

- **Protocol-Oriented Design:**
    - Emphasize using protocols to define the required interfaces for dependencies.
    - Protocols allow for flexible implementations and make it easier to swap out dependencies, especially useful when testing.
- **Ease of Injection and Substitution:**
    - By depending on protocols rather than concrete implementations, objects can work with any object conforming to the protocol, enhancing flexibility.
    - This makes it straightforward to inject mock or stub objects during testing without touching production code.

In app navigation

The Flow Coordinator pattern is a powerful tool for managing navigation within iOS applications, promoting a clean separation of concerns between navigation logic and view presentation. This approach is particularly well-suited for SwiftUI, facilitating scalable and maintainable code architectures.

Key Concepts of Flow Coordination

Centralized Navigation Management:

- The Flow Coordinator pattern encapsulates all navigation logic within specialized coordinator classes.

- Coordinators act as centralized controllers that manage the transitions and presentation of views, preventing the spread of navigation logic across view files.

Decoupling Navigation from Views:

- By separating navigation concerns, views become more focused on rendering UI elements and handling user interactions.
- This decoupling enhances maintainability, as view logic is unaffected by changes in navigation flows or transitions.

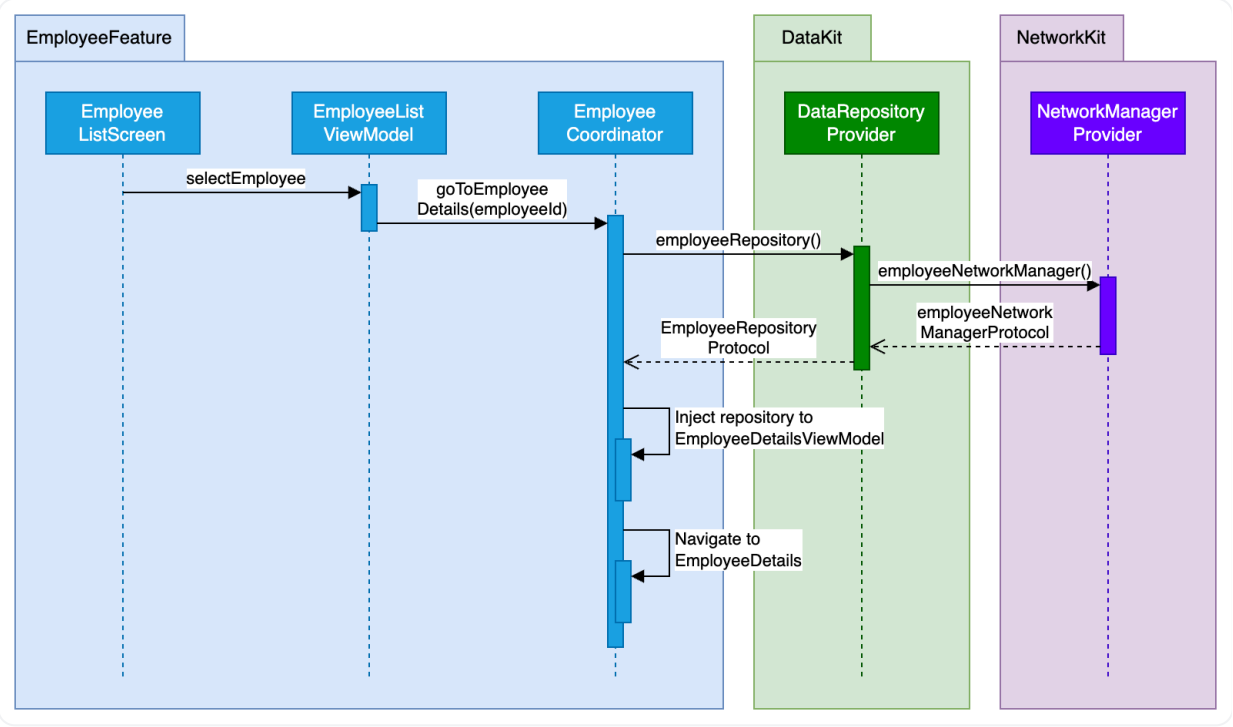
Stateful Navigation Control:

- Coordinators can maintain the state of navigation paths, ensuring consistency in navigation experiences and enabling advanced features like deep-linking or restoring previous states.

View Composition and Routing:

- Coordinators handle the logic for mapping navigation actions or routes to specific SwiftUI views.
- This approach allows for dynamic and flexible view compositions, adapting to various navigation contexts and enabling reusable components across different parts of the application.

Navigation example

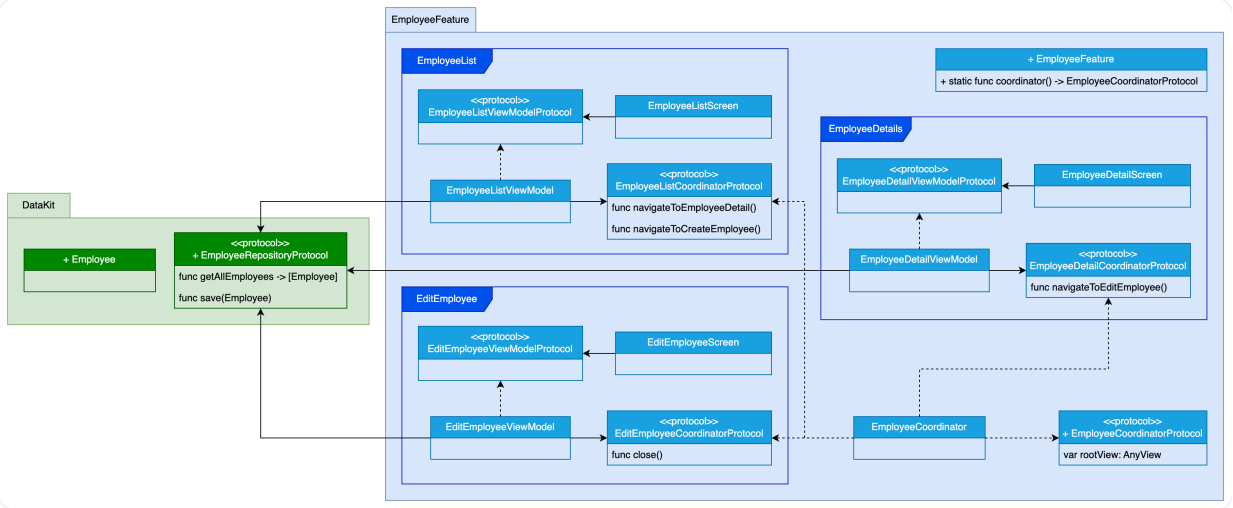


The flow diagram performs the next actions:

1. The user selects an employee from the **EmployeeListScreen**.
2. The **EmployeeListViewModel** processes the selection and calls goToEmployeeDetails(employeeId) on the **EmployeeCoordinator**.
3. The **EmployeeCoordinator** requests the **EmployeeRepository** from the **DataRepositoryProvider**.
4. The **DataRepositoryProvider** requests the **EmployeeNetworkManager** from the **NetworkManagerProvider**.
5. The **NetworkManagerProvider** returns the **EmployeeNetworkManager** as **EmployeeNetworkManagerProtocol**.
6. The **DataRepositoryProvider** returns the **EmployeeRepository** as **EmployeeRepositoryProtocol.**
7. After the provider is available, the **EmployeeCoordinator** inject the provider into the **EmployeeDetailsViewModel**.
8. The **EmployeeCoordinator** navigates to the **EmployeeDetails** screen.

Relationship Between Screens and the Coordinator

The following class diagram shows the relationship between each **ScreenViewModel** and the main coordinator, which in this case is the **EmployeeCoordinator**.



As shown in the diagram, the **EmployeeFeature** has three screens: **EmployeeList**, **EmployeeDetails**, and **EditEmployee**. Each screen has its own view model, and each view model has a corresponding coordinator, which is abstracted through a protocol. The feature coordinator (**EmployeeCoordinator**) implements the coordinator protocols for all three screens, as well as the public **EmployeeCoordinatorProtocol**.

Project example:

A project example for in app navigation, here

References

- Flow coordinator;
- MVVM pattern;
- Dependency injection;
- **Clean Architecture and Software Design Principles:**
  Concepts popularized by Robert C. Martin in *Clean Architecture: A Craftsman's Guide to Software Structure and Design*.