Project Architecture

We follow a layered architecture to maintain clear boundaries between different responsibilities within the application. Each layer is designed to handle specific concerns and interact with others in a well-defined manner. This structure ensures separation of concerns and provides a scalable, testable, and maintainable codebase.

1. **Data Layer** (Infrastructure Boundary)

- **Purpose** Acts as the boundary for communication between the application and external systems such as the backend, local storage, or IndexedDB.
- **Responsibilities**
  - Uses Data Transfer Objects (DTOs) to communicate with external systems.
  - Maps DTOs to Entities/Models used within the app, ensuring the domain layer stays isolated from the specifics of external data formats.
  - Contains swappable concrete implementations, allowing the data layer to be replaced or updated without affecting the rest of the application.
- **Boundary** The Data Layer directly interacts with external systems (e.g., APIs, databases) but never exposes these details to other layers. It ensures that the rest of the application remains agnostic to the specifics of data storage or retrieval.

2. **Domain Layer** (Core Business Logic Boundary)

- **Purpose** Represents the core business logic of the application. This layer is independent of the external world and encapsulates the rules that drive the application.
- **Responsibilities**
  - Contains Use Cases that define the business operations of the application. Defines Models used across the app, representing the core data structures.
  - Completely decoupled from the other layers and directly interacts with nothing outside itself. It only depends on abstractions (interfaces) defined and implemented in other layers.
- **Boundary** The Domain Layer is the most abstract layer and sits at the core of the application. It has no dependencies on infrastructure or presentation concerns. It communicates with other layers via interfaces, ensuring that the business logic is isolated from external changes or implementation details.

1. Presentation Layer (UI Boundary)
   - **Purpose** Manages the visual elements and user interactions. This layer is responsible for rendering data and responding to user inputs.

- **Responsibilities**
  - Displays data to the user through components and pages.
  - Uses Use Cases from the Domain Layer to trigger business logic and update the UI based on state changes.
  - Manages application state through stores.
  - Reponsible for application logic in combination with state/stores.
- **Boundary** The Presentation Layer only interacts with the Domain Layer through use cases and state stores. It does not directly handle business logic or manage data persistence. This ensures that UI concerns are kept separate from the core application logic.

Emphasized Boundaries:

- The Data Layer is entirely focused on external communication and ensures that no external system logic leaks into the Domain or Presentation layers.
- The Domain Layer defines all business logic in use cases and models, and is isolated from the details of data storage and user interface.
- The Presentation Layer is solely concerned with UI rendering and interacting with the domain logic via use cases and state management(application logic), without ever dealing with business rules directly.

System Advantages

The clear architectural boundaries provide several key benefits:

- **Separation of Concerns**: Each layer is responsible for one specific area of functionality, reducing complexity and improving maintainability.
- **Testability**: Isolated layers and clear boundaries make it easy to mock dependencies and test each layer independently.
- **Loose Coupling**: The core business logic (Domain Layer) is decoupled from the infrastructure (Data Layer) and presentation (UI Layer). Changes in one layer typically don't affect others.
- **Modularity**: Layers are independent, so individual components (like repositories or UI components) can be updated or replaced without affecting the entire system.
- **Scalability**: The system's modularity makes it easy to scale individual layers or features, enabling teams to work independently on different parts of the system without overlap.
- **Debuggability**: Isolated layers make it easier to trace issues to specific areas of the application.

State Management

For state management, we use the ngrx library. However, due to Angular's evolving approach towards Signals, we decided to use SignalStore from ngrx. This new state management system:

- Follows a functional approach rather than the traditional object-oriented model.
- Is easy to use, scalable, and integrates well with rxjs.
- Provides utilities to manage application state effectively without tightly coupling the state management to any specific layer.
- By using SignalStore, we maintain a clear boundary between state management and other layers, preserving the decoupling and separation of concerns that is central to our architecture.

Disadvantages

While the architecture offers numerous advantages, there are a few trade-offs and challenges to consider:

- **Increased Complexity**

Introducing multiple layers can increase the overall complexity of the project, especially for small applications. Developers need to be familiar with the architecture's boundaries and responsibilities to navigate the codebase effectively.

- **Longer Development Time (Initial Setup)**

Setting up the layers, especially with proper abstractions and dependency injection, can initially take more time compared to simpler architectures. The upfront effort to design and implement these boundaries may delay early development, but once it's setup is done developers can go with the flow with ease.

- **Potential Overhead**

For small or simple features, the strict separation into layers may introduce unnecessary overhead. Simple use cases could become over-engineered by adding multiple layers where a direct approach might suffice. This is why we allow custom few shortcuts to reduce the overhead in case of simple use cases.

- **Learning Curve**

Developers unfamiliar with this architecture may face a learning curve, especially in understanding the different layers' boundaries and how to interact with each layer appropriately.

Why We Should Adopt This Architecture

Despite the understandable disadvantages, the layered architecture offers substantial benefits, particularly for larger and more complex applications(our case). Here's why it makes sense to adopt this approach:

- **Scalability**

As the application grows, the need for a maintainable, scalable structure becomes more critical. The modular nature of this architecture allows different

teams to work on isolated layers without interference. This scalability is essential for long-term projects.

- **Maintainability**

With clearly defined boundaries, the system becomes easier to maintain. Each layer can evolve independently, and new features can be added without affecting the entire application. This reduces the risk of introducing bugs when modifying one part of the system.

- **Testability**

Isolating each layer makes testing easier. The separation allows us to mock dependencies, unit test use cases and repositories, and verify the system's behavior without complex setups. This is crucial for maintaining high code quality.

- **Loose Coupling**

The separation of concerns ensures that the business logic is independent of infrastructure and UI concerns. This decoupling makes it easier to change external systems or swap out libraries without affecting the core application.

- **Future-Proofing**

Adopting a layered architecture sets the foundation for handling future requirements and technologies. As new frameworks, tools, or patterns emerge, the clear boundaries between layers ensure that updates or changes can be made with minimal disruption to the overall system.