Angular Vertical Architecture Documentation

Project Architecture (Feature-Based)

Our project follows a feature-based vertical architecture, ensuring a clear separation of concerns while maintaining scalability and testability. Each feature is self-contained, with its own layers for API communication, business logic, state management, UI components, and shared utilities.

Folder Structure Overview

Inside the /app folder, the application consists only of the /features directory. Each feature contains the following subdirectories:

- **Client Folder**: Handles API requests and data retrieval from external sources.
- **Domain Folder**: Contains the core business logic, defining models and enforcing business rules.
- **Shared Folder**: Stores feature-specific utilities, services, guards, interceptors, and assets.
- **View Folder**: Contains UI components specific to this feature.
- **Store Folder**: Manages state using NgRx, Signals, or BehaviorSubjects.
- **feature.module.ts**: Defines the feature module and its dependencies.

1. Client Layer (Infrastructure Boundary)

Purpose: Acts as the boundary for communication between the Angular application and the backend.

Responsibilities:
- Uses Data Transfer Objects (DTOs) to standardize data exchange between the frontend and backend.
- Maps DTOs to domain models, ensuring the domain layer remains isolated from API-specific formats.
- The client layer can be modified without affecting the rest of the application.

Boundary: The Client Layer abstracts backend communication, preventing direct dependencies on

API details.

2. Domain Layer (Core Business Logic Boundary)

Purpose: This layer contains the essential rules and logic that define how the application operates, focusing only on business processes.

Responsibilities:
- Encapsulates business rules and logic.
- Defines domain models that represent the core data structures.
- Relies on repository interfaces for data access, ensuring flexibility.

Boundary: The Domain Layer is the heart of the application, kept isolated from API and UI concerns.

3. Shared Layer (Feature-Specific Utilities)

Purpose: Contains utilities, services, and assets specific to this feature.

Responsibilities:
- Provides reusable feature-specific utilities and helper functions.
- Stores feature-related assets such as images and JSON files.
- Includes guards, interceptors, and services relevant to this feature only.

Boundary: This layer is used only within its feature, preventing unnecessary global dependencies.

4. Presentation (View) Layer (UI Boundary)

Purpose: Manages the UI and user interactions for this feature.

Responsibilities:
- Displays data through Angular components and templates.
- Uses the Domain Layer to trigger business logic and update the UI.
- Handles user interactions and updates state accordingly.

Boundary: The Presentation Layer only interacts with the Domain Layer and Store, keeping UI concerns separate.

5. Store Layer (State Management)

Purpose: Manages application state specific to this feature.

Responsibilities:
- Stores the state of this feature and ensures reactive UI updates.
- Uses state management tools such as NgRx, Signals, or BehaviorSubjects.
- Handles side effects, such as API calls and caching.

Boundary: This layer manages the feature's state while keeping it isolated from other features.

Illustrative Example: Online Bakery System

- **Client Layer:** The order processing system receives customer orders, formats them correctly, and forwards them to the kitchen.
- **Domain Layer:** The kitchen follows recipes (business rules) to prepare the order, focusing purely on execution.
- **Shared Layer:** Contains feature-specific helpers, such as baking timers and ingredient lists.
- **Presentation Layer:** The serving staff (UI layer) displays the finished cakes and ensures the correct order is delivered.
- **Store Layer:** Manages inventory levels and order tracking for efficient state handling.