**Project Architecture**

We follow a layered approach for separation of concern, with each layer holding a single responsibility. The layers are as follows:

**Data Layer**

* Responsible for communication with the infrastructure (i.e backend, local storage, indexeddb...).
* Uses DTOs to communicate with the infra. and maps them to Entity/Models that are used across the app.
* Contains swapable concrete implementations.

**Domain Layer**

* Responsible for implementing pure business logic (UseCases) and describing Models used in the application.
* This is the layer with the highest level of abstraction in the system, so it communicates (or tightly coupled) with no other layer directly.
* To access data, it depends on dependency inversion using defined interfaces.

**Presentation Layer**

* Responsible for visible elements in the system like components and pages.
* Utilizes the `UseCase`s and state stores.

**Notes**

* The domain layer is the highest abstracted layer, and accesses or depends on no other layer.
* To reach this it depends on dependency inversion usually using dependency injection to incur an effect on the repository.

**Example**

```sh

src

+-- app

**|**

+--+-- core

+--+--+-- constants

+--+--+-- decorators

+--+--+-- directives

+--+--+-- pipes...

**|**

+--+-- features

+--+--+-- customers

+--+--+--+-- data *# the data leyer*

+--+--+--+--+-- dtos

+--+--+--+--+--+-- customer.dto.ts

+--+--+--+--+--+-- address.dto.ts

+--+--+--+--+-- implementation *# the repository*

+--+--+--+--+--+-- customer.repo.ts

+--+--+--+--+--+-- address.repo.ts

+--+--+--+--+-- mappers *# maps dtos to models*

+--+--+--+--+--+-- customer.mapper.ts

+--+--+--+--+--+-- address.mapper.ts

+--+--+--+-- domain

+--+--+--+--+-- models

+--+--+--+--+--+-- customer.model.ts

+--+--+--+--+--+-- address.model.ts

+--+--+--+--+-- repository

+--+--+--+--+--+-- customer.irepo.ts

+--+--+--+--+--+-- address.irepo.ts

+--+--+--+--+-- usecases *# containes all methods in the repository and other buisness logic functions*

+--+--+--+--+--+-- customer

+--+--+--+--+--+--+-- get-customers.usecase.ts

+--+--+--+--+--+--+-- get-customer-by-id.usecase.ts

+--+--+--+--+--+--+-- create-customer.usecase.ts

+--+--+--+--+--+--+-- update-customer.usecase.ts

+--+--+--+--+--+--+-- delete-customer.usecase.ts

+--+--+--+--+--+--+-- filter-customers.usecase.ts

+--+--+--+--+--+-- address...

**|**

+--+-- view

+--+--+-- pages

+--+--+--+-- shared

+--+--+--+-- invoice-page

+--+--+--+--+-- store

+--+--+--+--+-- components

+--+--+--+--+-- invoice-page.component.ts

+--+--+--+--+-- invoice-page.component.html

+--+--+--+--+-- invoice-page.component.scss

+--+--+--+-- invoice-list-page

+--+--+--+-- invoice-list-page

+--+--+-- components

+--+--+--+-- select-customer

+--+--+-- stores

+--+--+--+-- app-store

```

* The previous example embraces feature first folder layout.
* This enhances code base accessibility and maintainability.
* Also, we use simple abstractions across the system to unify the structure of its components like `UseCase` class

**System Advantages**

**Separation of concern**

* Because the system is composed of specialized layers and pure `UseCases`.

**Testability**

* It's super easy to mock the few isolated dependencies with business logic in the core of the application.

**Loosely Coupled**

* The core business logic is not tightly coupled with any library, peripheral implementation or even the framework itself.

**Modularity**

* The separate layers allow updating components without affecting the entire system and allows swapping and updating implementations with ease.

**Debuggability**

**Scalability**

* The system's modularity enhances scalability so much. You can scale features independently, and team members can work on different layers without interference or overlapping.

**State Management**

We use the famous `ngrx` library for state management. Due to Angular renaissance and the new direction to `signals` which is frequently referred to as the future of Angular, we decided to use the new library from `ngrx` which is `SignalStore`. Although it follows a different functional approach other than the OOP usual approach for `ngrx`, it proved itself for ease of use and scalability. Also, it provides a handful useful utilities to deal with `rxjs` and the state.