Clean Architecture in Flutter

Introduction to Clean Architecture

Clean Architecture is a **software design pattern** introduced by **Robert C. Martin (Uncle Bob)** that enforces the **separation of concerns** by organizing the codebase into distinct layers. It helps create **scalable, maintainable, and testable** applications by ensuring that each layer has a well-defined responsibility.

Core Principles of Clean Architecture

1. **Separation of Concerns**: Each layer handles a specific responsibility, ensuring modularity and maintainability.
2. **Dependency Rule**: Inner layers do not depend on outer layers, while outer layers depend on inner layers.
3. **Testability**: Each layer is independent, making unit testing easier.
4. **Scalability**: A well-structured project can be easily extended without affecting unrelated parts.
5. **Reusability**: Business logic can be reused across different parts of the application.

Layers in Clean Architecture

Clean Architecture is divided into three core layers:

1. **Domain Layer**: Contains the business logic and is independent of UI, APIs, or databases.
2. **Data Layer**: Handles data operations such as API calls and database access.
3. **Application Layer**: Manages state and orchestrates data flow between the Domain and Presentation layers.
4. **Presentation Layer**: Displays data to users and interacts with state management.

Each layer has a specific role and communicates in a structured manner to ensure maintainability.

Folder Structure in Flutter

A well-structured Flutter application following Clean Architecture should be organized as follows:

```
lib/
├── _shared/                        # Common utilities and services
│   ├── _core/                      # App-wide core utilities (errors, failures, value object)
│   ├── config/                     # Configuration files (API keys, environment, theme config)
│   ├── services/                   # Shared services (network, logging)
│   ├── utilities/                   # Helper functions (extension methods, validators, constants)
├── application/                     # Application (State Management Layer)
│   ├── {feature1}/                   # Each feature has its own BLoC
│   ├── {feature2}/
├── domain/                           # Business Logic Layer
│   ├── shared/                       # Shared domain logic across features
│   │   ├── _core/                    # Core domain utilities (failure, value objects)
│   │   ├── entities/                  # Shared business objects
│   │   ├── repositories/              # Shared repository contracts
│   │   ├── usecases/                  # Shared use cases
│   ├── {feature1}/
│   │   ├── entities/                  # Feature-specific business objects
│   │   ├── repositories/              # Feature repository contracts (Abstract Classes)
│   │   │   ├── {feature1}_repository.dart
│   │   ├── usecases/                  # Feature-specific business logic
│   ├── {featureN}/
├── data/                              # Data Layer (Implements Domain)
│   ├── shared/
│   │   ├── dtos/                      # Shared DTOs across features
│   │   ├── repositories/              # Shared repository implementations
│   ├── {feature1}/
│   │   ├── dtos/                      # Feature-specific DTOs
│   │   ├── {feature1}_repository_impl.dart  # Implements Domain Repository
│   ├── {featureN}/
├── presentation/
│   ├── shared/
│   │   ├── constants/                 # Common constant widgets
│   │   ├── widgets/                   # Widgets shared across features
│   ├── {feature1}/
│   │   ├── widgets/                   # Feature-specific widgets
│   │   ├── {feature1}_page.dart       # Implements UI
│   ├── {featureN}/
│   │   ├── widgets/                   # Feature-specific widgets
│   │   ├── {featureN}_page.dart       # Implements UI
```

Explanation of Each Folder

_shared/ (App-Wide Utilities)

This folder contains utilities, configurations, and services that are used across all features.

- **_core/**: Contains global utilities such as failure handling and theme configuration.
- **config/**: Stores application-wide configuration settings like API keys and environment variables.
- **services/**: Includes shared services like authentication, logging, and networking.
- **utilities/**: Contains helper functions such as date formatters and validators.

This ensures reusability and prevents redundancy in different features.

application/ (State Management)

This folder manages application state using a state management solution like BLoC.

Each feature has its own state management layer:

```
application/
├── feature1/
│   │   ├── feature1_event.dart
│   │   ├── feature1_state.dart
│   │   ├── feature1_bloc.dart
├── feature2/
│   │   ├── feature2_event.dart
│   │   ├── feature2_state.dart
│   │   ├── feature2_bloc.dart
├── featureN/
│   │   ├── featureN_event.dart
│   │   ├── featureN_state.dart
│   │   ├── featureN_bloc.dart
```

By keeping state logic separate from UI, this structure makes state management more scalable.

---

domain/ (Business Logic Layer)

The domain layer contains **pure business logic** and remains independent of UI, APIs, and databases.

Shared Domain Layer (`domain/shared/`)

- **_core/**: Stores common domain utilities like failure handling and value objects.
- **entities/**: Contains business objects used across multiple features.
- **repositories/**: Defines shared repository contracts for multiple features.
- **usecases/**: Stores shared use cases that can be reused across features.

Feature-Specific Domain Layer (`domain/{featureX}/`)

Each feature has its own domain layer:

```
domain/feature1/
├── entities/
│   ├── feature1_entity.dart
├── repositories/
│   ├── feature1_repository.dart  # Abstract class
├── usecases/
│   ├── get_feature1_data.dart
│   ├── update_feature1_data.dart
```

This ensures that the business logic is well-structured and reusable.

---

data/ (Data Layer)

The data layer is responsible for fetching and storing data from APIs, databases, or caches.

Shared Data Layer (`data/shared/`)

- **dtos/**: Stores shared **Data Transfer Objects (DTOs)** for API communication.
- **repositories/**: Contains shared repository implementations.

Feature-Specific Data Layer (`data/{featureX}/`)

Each feature's data layer contains its own DTOs and repository implementations.

```
data/feature1/
├── dtos/
│   ├── feature1_dto.dart
├── feature1_repository_impl.dart
```

Repositories act as a bridge between **Domain Layer and Data Layer** and ensure that the application works with clean data models.

How Data Flows in Clean Architecture

```
UI → BLoC (Application Layer) → Use Case (Domain Layer) → Repository (Data Layer) → API/Database
```

Example: Fetching User Data

1. The **UI Layer** calls `UserBloc.add(GetUser(userId))`.
2. The **UserBloc** calls `GetUserUseCase(userId)`.
3. The **Use Case** calls `UserRepository.getUser(userId)`.
4. The **Repository Implementation** calls `UserRemoteDataSource.fetchUserFromApi(userId)`.
5. The **API returns JSON**, which is converted into a `UserDTO`.
6. The **UserDTO is converted into an Entity** and returned to the Use Case.
7. The **Use Case returns the Entity to the Bloc**, which updates the UI.

This ensures that each layer has a **single responsibility** and remains independent of other layers.

Conclusion

A well-structured Flutter application using **Clean Architecture** ensures that:

- Each layer has a **clear responsibility**, making the code **maintainable**.
- The **Domain Layer is independent** of external dependencies.
- The **Application Layer** manages state effectively using **BLoC**.
- The **Data Layer handles external APIs, databases, and DTOs**.

Following this approach helps in building **scalable, testable, and maintainable Flutter applications**.