Domain Layer

In **Flutter Clean Architecture**, the **Domain Layer** is a central part of the architecture that represents the core business logic of your application. This layer is highly abstracted and independent of both the data sources and the user interface, making it reusable and adaptable to different interfaces or data sources.

Key Responsibilities of the Domain Layer

The Domain Layer handles:

1. **Business Logic**: Contains the core rules and requirements of the application that define how data can be created, stored, and modified.
2. **Use Cases / Interactors**: Define specific actions the application can perform. For example, a `SignInUseCase` would contain the logic for signing in a user.
3. **Entities**: Represent the essential models or objects in the business domain with minimal data fields. Entities are usually simple classes that capture business concepts.
4. **Repositories (Interfaces)**: Define contracts for how the Domain Layer interacts with data sources. The Domain Layer doesn't interact with the data layer directly; instead, it depends on abstract repository interfaces.

Components of the Domain Layer

1. **Use Cases (Interactors)**:
   - Use cases encapsulate a single, focused action or operation, representing the actual purpose of a feature.
   - They are independent of the UI or specific data implementations and can be tested in isolation.
   - Example: `SignInUseCase`, `FetchProductsUseCase`.
2. **Repository Interfaces**:
   - These interfaces define the methods for accessing data, but they don't know where the data is stored (e.g., locally or in a remote database).
   - Repositories allow the Domain Layer to remain agnostic to data sources, enhancing testability and flexibility.
   - Example: An `AuthRepository` interface might define a `login` method but doesn't specify how or where the login is processed.

Example

Repository Interface - AuthRepository

The `AuthRepository` interface is defined in the Domain Layer, without any data-source-specific details.

```dart
import 'package:cs_ecommerce/features/auth/data/models/customer/signin_req.dart';
import 'package:cs_ecommerce/features/auth/data/models/customer/signup_req.dart';
import 'package:dartz/dartz.dart';

abstract class AuthRepository{

  Future<Either> signup(SignupReqParams signupReq);
  Future<Either> signin(SigninReqParams signinReq);

}
```

Either: `Either` is a powerful data type that represents a value of two possible types. It's commonly used to handle success and failure responses in a clean, predictable way. The `Either` type holds one of two values:

1. A **Left** value, which typically represents a failure or error.
2. A **Right** value, which typically represents a success or valid result.

Using `Either`, you can avoid throwing exceptions and instead return an instance that indicates whether the operation succeeded or failed, making error handling more structured and avoiding complex try-catch logic.

Structure of `Either`

An `Either` type in Flutter is usually structured as follows:

- `Left` for **Failure/Error** cases.
- `Right` for **Success** cases.

For instance, in a user authentication flow:

- If the login attempt succeeds, `Either` will contain a `Right` with the authenticated user data.
- If the login attempt fails, `Either` will contain a `Left` with error information.

Use Case - SignInUseCase

The `SignInUseCase` encapsulates the login operation.

```dart
import 'package:cs_ecommerce/core/usecase/usecase.dart';
import 'package:cs_ecommerce/features/auth/data/models/customer/signin_req.dart';
import 'package:cs_ecommerce/features/auth/domain/repository/auth.dart';
import 'package:dartz/dartz.dart';
import 'package:cs_ecommerce/service_locator.dart';

class SignInUseCase implements Usecase<Either,SigninReqParams>{
  @override
  Future<Either> call({SigninReqParams ? param}) async{
  return sl<AuthRepository>().signin(param!);
  }}
```