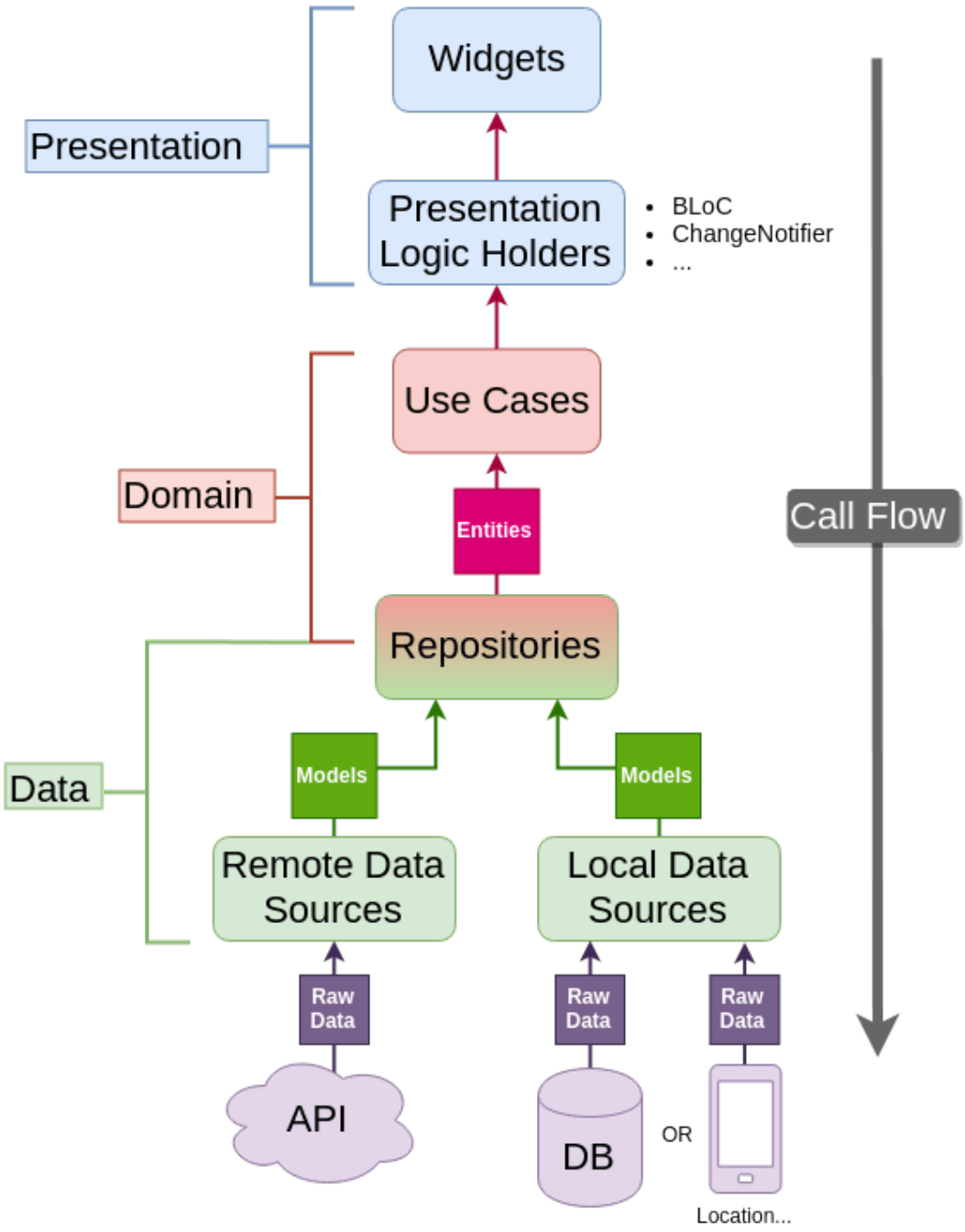
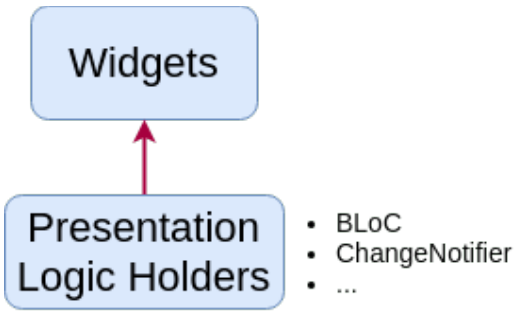
Clean Architecture & Flutter



Presentation

This is the stuff you're used to from "unclean" Flutter architecture.
You obviously need “widgets” to display something on the screen. These widgets then **dispatch events** to the **Bloc** and listen for **states** (or an equivalent if you don't use Bloc for state management).



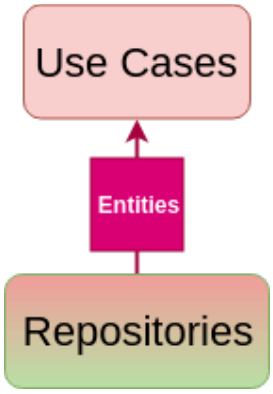
Note that the "Presentation Logic Holder" (e.g. Bloc) doesn't do much by itself. It delegates all its work to **use cases**. At most, the presentation layer handles basic input conversion and validation.

Domain

Domain is the inner layer which shouldn't be susceptible to the whims of changing data sources or porting our app to Angular Dart. It will contain only the core **business logic** (***use cases***) and **business objects** (***entities***). It should be totally *independent* of every other layer.

Use Cases are classes which encapsulate all the business logic of a particular use case of the app (e.g. Get*Concrete*NumberTrivia or Get*Random*NumberTrivia).

But... How is the **domain layer** completely independent when it gets data from a **Repository**, which is from the **data layer**? Do you see that fancy colorful gradient for the**Repository**? That signifies that it belongs to both layers at the same time. We can accomplish this with **dependency inversion**.

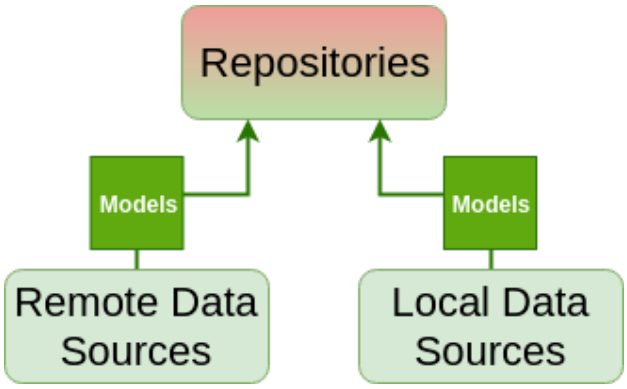


That's just a fancy way of saying that we create an abstract **Repository** class defining a contract of what the Repository must do - this goes into the domain layer. We then depend on the Repository "contract" defined in domain, knowing that the actual implementation of the **Repository** in the **data layer** will fullfill this contract.

Data

The **data layer** consists of a **Repository implementation** (the contract comes from the **domain layer**) and **data sources** - one is usually for getting remote (API) data and the other for caching that data. **Repository** is where you decide if you return fresh or cached data, when to cache it and so on.

You may notice that **data sources** don't return **Entities** but rather **Models**. The reason behind this is that transforming raw data (e.g JSON) into Dart objects requires some JSON conversion code. We don't want this JSON-specific code inside the **domain Entities** - what if we decide to switch to XML?



Therefore, we create **Model** classes which **extend Entities** and add some specific functionality (toJson, fromJson) or additional fields, like database ID, for example.