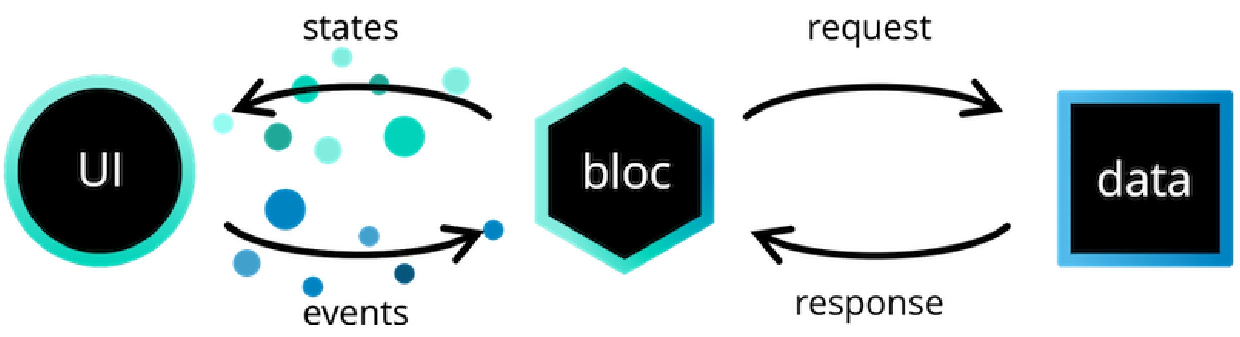
Architecture



Using the bloc library allows us to separate our application into three layers:

- Presentation
- Business Logic
- Data
  - Repository
  - Data Provider

We're going to start at the lowest level layer (farthest from the user interface) and work our way up to the presentation layer.

Data Layer

> The data layer's responsibility is to retrieve/manipulate data from one or more sources.

The data layer can be split into two parts:

- Repository
- Data Provider

This layer is the lowest level of the application and interacts with databases, network requests, and other asynchronous data sources.

Data Provider

> The data provider's responsibility is to provide raw data. The data provider should be generic and versatile.

The data provider will usually expose simple APIs to perform CRUD operations. We might have a `createData`, `readData`, `updateData`, and `deleteData` method as part of our data layer.

```dart
class DataProvider {
    Future<RawData> readData() async {
        // Read from DB or make network request etc...
    }
}
```

Repository

> The repository layer is a wrapper around one or more data providers with which the Bloc Layer communicates.

```dart
class Repository {
    final DataProviderA dataProviderA;
    final DataProviderB dataProviderB;

    Future<Data> getAllDataThatMeetsRequirements() async {
        final RawDataA dataSetA = await dataProviderA.readData();
        final RawDataB dataSetB = await dataProviderB.readData();

        final Data filteredData = _filterData(dataSetA, dataSetB);
        return filteredData;
    }
}
```

As you can see, our repository layer can interact with multiple data providers and perform transformations on the data before handing the result to the business logic Layer.

Business Logic Layer

> The business logic layer's responsibility is to respond to input from the presentation layer with new states. This layer can depend on one or more repositories to retrieve data needed to build up the application state.

Think of the business logic layer as the bridge between the user interface (presentation layer) and the data layer. The business logic layer is notified of events/actions from the presentation layer and then communicates with repository in order to build a new state for the presentation layer to consume.

```dart
class BusinessLogicComponent extends Bloc<MyEvent, MyState> {
    final Repository repository;

    Stream mapEventToState(event) async* {
        if (event is AppStarted) {
            try {
                final data = await repository.getAllDataThatMeetsRequirem
                yield Success(data);
            } catch (error) {
                yield Failure(error);
            }
        }
    }
}
```

Bloc-to-Bloc Communication

> Every bloc has a state stream which other blocs can subscribe to in order to react to changes within the bloc.

Blocs can have dependencies on other blocs in order to react to their state changes. In the following example, `MyBloc` has a dependency on `OtherBloc` and can `add` events in response to state changes in `OtherBloc`. The

`StreamSubscription` is closed in the `close` override in `MyBloc` in order to avoid memory leaks.

```dart
class MyBloc extends Bloc {
  final OtherBloc otherBloc;
  StreamSubscription otherBlocSubscription;

  MyBloc(this.otherBloc) {
    otherBlocSubscription = otherBloc.listen((state) {
        // React to state changes here.
        // Add events here to trigger changes in MyBloc.
    });
  }

  @override
  Future<void> close() {
    otherBlocSubscription.cancel();
    return super.close();
  }
}
```

Presentation Layer

> The presentation layer's responsibility is to figure out how to render itself based on one or more bloc states. In addition, it should handle user input and application lifecycle events.

Most applications flows will start with a `AppStart` event which triggers the application to fetch some data to present to the user.

In this scenario, the presentation layer would add an `AppStart` event.

In addition, the presentation layer will have to figure out what to render on the screen based on the state from the bloc layer.

```dart
class PresentationComponent {
    final Bloc bloc;
    PresentationComponent() {
        bloc.add(AppStarted());
    }

    build() {
        // render UI based on bloc state
    }
}
```

So far, even though we've had some code snippets, all of this has been fairly high level. In the tutorial section we're going to put all this together as we build several different example apps.

‹ PREVIOUS
package:bloc

NEXT ›
Testing