Clean Architecture Implementation guide

First add following folder structure

```
-> core
        -> error
        -> usecases
-> features
        -> <name of feature>
                -> data
                        -> datasources
                        -> models
                        -> repositories
                -> domain
                        -> entities
                        -> repositories
                        -> usecases
                -> presentation
                        -> bloc
                        -> pages
                        -> widgets
```

This will be the basic structure to work with TDD. For every new feature in the app there should be a folder with the name of that feature & the compelete data, domain & presentation structure.

Core folder will be used to create classes which can be used anywhere in any feature as different feature may reuse some common code.

Add following packages in application

1. get_it
2. flutter_bloc
3. equatable
4. http

For TDD in dev_dependencies "mockito" also needed.

SETTING UP TDD

In this section we will setup a sample project for TDD first approach.

First create your entities inside **domain->entities** folder according to your json response model. This dart class should extend ***Equatable***. It will help in comparing the models whenever we need it.

```
// trivia.dart
class Trivia extends Equatable {
  final String text;
  final num number;
  Trivia({required this.text, required this.number});

  @override
  List<Object?> get props => [number, text];
}
```

This will be our sample model for trivia with a number & some text about that number.

Now in the **domain->repositories**, we will create our repositories. This will mainly an abstract class with the contract of this repository.

Here we will define the return types of function with future as it will be a network call.

```
// trivia_repository.dart
abstract class TriviaRepository {
  Future<Trivia?> getRandomTrivia();
}
```

We put a question mark (?) there to allow it to be null for test cases.

Now create basic usecase of this class inside **domain->usecases**. Create a class with basic structure like this.

```
// random_trivia.dart
class RandomTrivia{
  final TriviaRepository repository;

  RandomTrivia(this.repository);
  Future<Trivia?> execute() async {
    return await repository.getRandomTrivia();
  }
}
```

Here we marked **Trivia** as nullable because we need to use this method in out mock tests.
Even if we are sure that our response will never be null. This is required so mock test will not show a non-nullable value related error.

Testing

As this is a **Test Driven Development**, we will now create the same folder structure as defined at the start inside the **test** folder.

Inside our **test/features/<project>/domains/usecase** create new test class

```
// get_random_number_trivia_test.dart
class MockTriviaRepository extends Mock implements TriviaRepository {}

void main() {
  late RandomTrivia usecase;
  late MockTriviaRepository mockTriviaRepository;

  setUp(() {
    mockTriviaRepository = MockTriviaRepository();
    usecase = RandomTrivia(mockTriviaRepository);
  });

  const tNumberTrivia = Trivia(number: 0, text: "No data for this");

  test("random number trivia is working", () async {
    when(mockTriviaRepository.getRandomTrivia())
        .thenAnswer((_) async => tNumberTrivia);

    final result = await usecase.execute();
    expect(result, tNumberTrivia);
    verify(mockTriviaRepository.getRandomTrivia());
    verifyNoMoreInteractions(mockTriviaRepository);
  });
}
```

Here we create **mockTriviaRepository** & use that in our **usecase**. We also create one dummy trivia response in ***tNumberTrivia***.

In test, we use ***when*** method of Mockito library. Using this method we create a mocking condition as given below.

```
when(mockTriviaRepository.getRandomTrivia())
        .thenAnswer((_) async => tNumberTrivia);
```

*This means whenever we call the **getRandomTrivia**() of the our **mockTriviaRepository**, we want to get **tNumberTrivia** as a result.*

Now when we call the execute method of our ***usecase*** (***RandomTrivia***), we will get the ***tNumberTrivia*** from the result. Which we can confirm using the “expect” function.

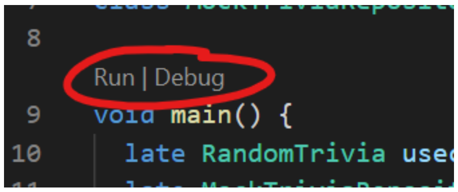
In the end, we need to call ***verify*** to confirm that we executed the right method to get the response. We also call ***verifyNoMoreInteractions***, to confirm that we can safely executed all the defined methods of our repository. This is important so that we can mark the test as completed.

```
verify(mockTriviaRepository.getRandomTrivia());
verifyNoMoreInteractions(mockTriviaRepository);
```

Run Test

There are two ways to run this test one

1. You will see a **Run|Debug** option on top of main function. Press Run to execute the test.



```
Run | Debug
void main() {
  late RandomTrivia use
```

2. You can also run test using the "***flutter test***" command. It will show you "**All tests passed!**" message if there is no issue found. Otherwise, it will show you the error of fail test.

Adding business logic

In this section we will start adding business logic to our app.

First we create one method in repository to get the data from API.

Create one file in **data\models** to implement API code.

```dart
// trivia_model.dart
class TriviaModel extends Trivia {
  TriviaModel({required String text, required num number})
      : super(text: text, number: number);

  factory TriviaModel.fromJson(Map<String, dynamic> json) => TriviaModel(
        text: json["text"],
        number: json["number"],
      );

  Map<String, dynamic> toJson() => {
        "text": text,
        "number": number,
      };
}
```

We create our model using the entity & add two more method in that for json conversion.

Create one file in **data\datasources** to implement API code.

```dart
// remote_data_source.dart
abstract class RemoteDataSource {
  Future<Trivia> getRandomTrivia();
}

class RemoteDataSourceImpl extends RemoteDataSource {
  http.Client client;

  RemoteDataSourceImpl({required this.client});

  @override
  Future<Trivia> getRandomTrivia() async {
    final response = await client.get(
        Uri.parse("http://numbersapi.com/random?json"),
        headers: {'Content-type': 'application/json'});

    if (response.statusCode == 200) {
      return TriviaModel.fromJson(jsonDecode(response.body));
    } else {
      throw Exception("No data found");
    }
  }
}
```

In this data source, we first create the abstract class for data source. Then using that data source class, we create our implementation class & add the code in that.

We throw an exception if the status code is not 200.

The **client** in constructor will be provided by dependency injection.

Create one file named **trivia_repository_impl** which will extend **trivia_repository**.

data\repositories

```
// trivia_repository_impl.dart
class TriviaRepositoryImpl extends TriviaRepository {
  RemoteDataSource remoteDataSource;
  TriviaRepositoryImpl({required this.remoteDataSource});

  @override
  Future<Trivia>? getRandomTrivia() async {
    try {
      final remoteTrivia = await remoteDataSource.getRandomTrivia();
      return remoteTrivia;
    } catch (err) {
      rethrow;
    }
  }
}
```

Here we add code to call ***getRandomTrivia()*** from our data source. If we get an error we will rethrow that error further.

The **remoteDataSource** in constructor will be provided by dependency injection.

Now we need to setup our trivia bloc class with events & state.

We create three files trivia_event, trivia_state & trivia_bloc.

presentation\bloc\

```
// trivia_state.dart
abstract class TriviaState extends Equatable {
  @override
  List<Object?> get props => [];
}
```

```
class Initial extends TriviaState {
  @override
  List<Object?> get props => [];
}

class Loading extends TriviaState {
  @override
  List<Object?> get props => [];
}

class Loaded extends TriviaState {
  final Trivia? trivia;
  Loaded(this.trivia);

  @override
  List<Object?> get props => [];
}

class Error extends TriviaState {
  final Exception exception;
  Error(this.exception);

  @override
  List<Object?> get props => [];
}
```

In here we create an abstract class for TriviaState & then use that class to create 4 different state classes which are Initial, Loading, Loaded & Error.

In the same way create class for events.

presentation\bloc\

```
// trivia_event.dart
class TriviaEvent extends Equatable {
  @override
  List<Object?> get props => [];
}

class GetRandomTrivia extends TriviaEvent {}
```

After these two files we need our bloc file.

presentation\bloc\

```
// trivia_bloc.dart
class TriviaBloc extends Bloc<TriviaEvent, TriviaState> {
  RandomTrivia randomTrivia;
  TriviaBloc({required this.randomTrivia}) : super(Initial());

  @override
  Stream<TriviaState> mapEventToState(TriviaEvent event) async* {
    if (event is GetRandomTrivia) {
      yield Loading();
      try {
        final result = await randomTrivia.execute();
        yield Loaded(result);
      } on Exception catch (err) {
        yield Error(err);
      }
    }
  }
}
```

Here we create a bloc with **TriviaEvent** & **TriviaState**. We override the ***mapEventToState*** method. This method will get the event we want to execute & by checking the event type we can run our business logic accordingly. Our **randomTrivia** already have an ***execute()*** in it, which we already used in our test case.

Initially the state of bloc will be ***Initial().***

If we get the response, we will then return ***Loaded(result).*** This will update the state as loaded.

If we get any exception from the call, we will return ***Error(err).*** This will update the state with the error.

The **randomTrivia** in constructor will be provided by dependency injection.

If project is using **flutter_bloc** version **7.2.0 & above**, then we need a different code as ***mapEventToState*** is deprecated since **v7.2.0.** Instead of ***mapEventToState*** new ***on<Event>*** API had been introduced.
With this new API above code will change like following.

```
class TriviaBloc extends Bloc<TriviaEvent, TriviaState> {
  RandomTrivia randomTrivia;
  TriviaBloc({required this.randomTrivia}) : super(Initial()){
        on<GetRandomTrivia>((event, emit) async{
          emit(Loading());
          try {
            final result = await randomTrivia.execute();
            emit(Loaded(result));
          } on Exception catch (err) {
            emit(Error(err));
          }
        });
  }
}
```

Dependency Injection

Now as our business logic is done, we can start implementing the dependency injection setup as most of the classes need some other class to work.

We create injection file to add our dependency injection setup using **get** package.

```
// lib/injection.dart
final inj = GetIt.instance;

void init() async{
}
```

In this ***init()*** we will add our dependencies as given below. All following lines will get added to ***init()***

First we need our **TriviaBloc** object so we add it.

```
inj.registerFactory<TriviaBloc>(() => TriviaBloc(randomTrivia: inj()));
```

If you go to **TriviaBloc**, it needs **RandomTrivia** object in constructor, so we now add that.

```
inj.registerLazySingleton(() => RandomTrivia(inj()));
```

RandomTrivia needs **TriviaRepository** in constructor.

```
inj.registerLazySingleton<TriviaRepository>(
      () => TriviaRepositoryImpl(remoteDataSource: inj()));
```

TriviaRepository needs **RemoteDataSource** in constructor.

```
inj.registerLazySingleton<RemoteDataSource>(
      () => RemoteDataSourceImpl(client: inj()));
```

RemoteDataSource needs **http client** in constructor. For that add following lines.

```
import 'package:http/http.dart' as http;
inj.registerLazySingleton(() => http.Client());
```

Notice here we are passing ***inj()*** in all the constructor params. This is to tell the system that these params needs to be fetched from dependency injection.

Finally to setup dependency injection & call this ***init()***. We add it in our ***main()*** of **main.dart** file.

```
import 'injection.dart' as di;

void main() {
  di.init();
  runApp(const MyApp());
}
```

Now the only thing left is adding bloc provider in our widget tree & create the UI.

Adding UI

Add the **BlocProvider** in the **mail.dart** file's ***build()***. This **BlocProvider** can be either inside or outside of **MaterialApp**.

```
@override
Widget build(BuildContext context) {
    return BlocProvider<TriviaBloc>(
      create: (_) => di.inj<TriviaBloc>(),
      child: MaterialApp(
        title: 'TDD Demo',
        theme: ThemeData(
          primarySwatch: Colors.blue,
        ),
        home: const TriviaPage(),
      ),
    );
}
```

Here we are using single **BlocProvider** as we have a single bloc file to implement. If we have more than one bloc file, we can use **MultiBlocProvider.**

On **TriviaPage**, we add some basic code to show some text in center & get some random trivia on **fab** click.

Here is the sample code for this.

```
class TriviaPage extends StatefulWidget {
  const TriviaPage({Key? key}) : super(key: key);

  @override
  _TriviaPageState createState() => _TriviaPageState();
}

class _TriviaPageState extends State<TriviaPage> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("Trivia Page"),
      ),
      body: BlocBuilder<TriviaBloc, TriviaState>(
        builder: (context, state) {
          if (state is Loaded) {
            return Column(
              mainAxisAlignment: MainAxisAlignment.center,
              crossAxisAlignment: CrossAxisAlignment.center,
              children: [
                Text(
                  state.trivia!.number.toString(),
                  style: Theme.of(context).textTheme.headline3,
                ),
                Padding(
                  padding: const EdgeInsets.all(16.0),
                  child: Text(state.trivia!.text),
                )
              ],
            );
          } else if (state is Loading) {
            return const Center(
              child: CircularProgressIndicator(),
            );
          } else if (state is Error) {
            return Center(
              child: Text(
                state.toString(),
                style: const TextStyle(color: Colors.red),
              ),
            );
          }
          return const Center(
            child: Text("Waiting.."),
          );
        },
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _getRandom,
        child: const Icon(Icons.refresh),
      ),
    );
  }
}
```

Here you can see that we wrapped our page body within **BlocBuilder.** This **BlocBuilder** will refresh its ***builder*** method as our bloc updates.

BlocBuilder needs the bloc class object we need to use in the builder & the bloc state class object.

We also setup separate UI conditions for Loaded, Loading & Error states. By default, the app will show text "Waiting" until user press the **FAB.**

To get the data from API we add ***_getRandom()*** to call the event related to this bloc.

```
_getRandom() {
    BlocProvider.of<TriviaBloc>(context).add(GetRandomTrivia());
}
```

That's all.
We can now run our project & see it live in action.

If we want, we can separate the UI elements in separate widget files in **widgets** folder.

Example code: https://github.com/kumar-aakash86/flutter_tdd_clean_arct