flutter tutorial

(Recipie App project)

① to create a project type command:- → create command is used to create the flutter.

```
flutter create appname
```

② to check the version:-

```
flutter --version
```

③ `flutter doctor` → this checks that if there is any problem or error in the file & program.

④ In the lib folder we have the `main.dart` file which is the entry point of the application, in this we have code.

→
```
void main(){
    runApp(const MyApp())
}
```

→ basic entry point for our application.

runApp → actually runs our application and then attach it to the screen.

→ widgets:- Key component of the user interface in flutter (root)
↳ used describe the structure and layout of our application. such as buttons, icons, text elm etc --containers.

widget tree → top (material App)

→ there are two types of widgets in flutter

statefulwidget

in this data changes within it during lifecycle of our project.

stateless widget

→ they do not change their state. do not change data that within them

→ the top widget will be stateless widget and widget between them can be stateless or statefull widget.

→ in MyApp which is a statelses i am creating build fn and within that iam defining material app

MaterialApp() is a class in flutter that is top level widget for a flutter application & uses Material Design visual layout structure.
MaterialApp() -> constructor which is called

Date:- / /
Page No.:-

In our material App we are defining ttle for our application, we are describing theme using creating instance of ThemeData and displaying color using color.fromseed (seed color).

useMaterial 3 : true,

↳ after this we are defining AppBar -> ~~setting its fontstyle, color,~~ ~~fontsize~~ etc.

-> createState() -> defines how states is created for this widget.

_loginpagestate -> _underscore defines that the class is private.

-> scaffold() -> an empty canvas.

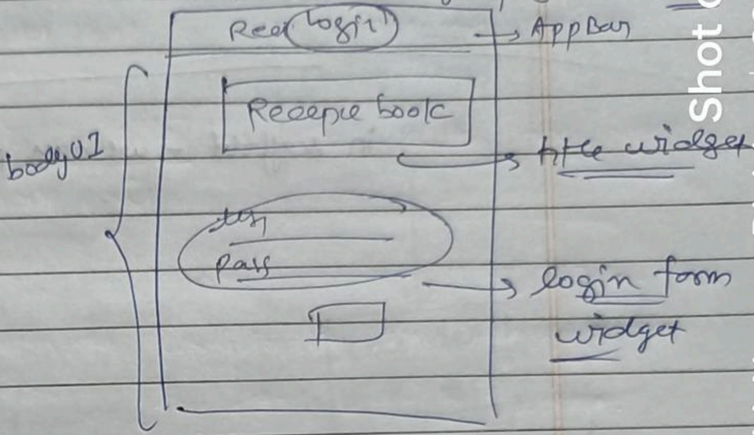
```
class Loginpage entends statefulWidget {
@override
  state<statefulwidget> createstate() {      } this my main
       return _Login Page state();           } stateful widget
                                             } for login page
    }
  }
                    ↪ Its a private class.

class _Loginpagestate entends state<LoginPage>{    } to manage stat of
 @override                                         } login page widget
   Widget build (BuildContent content){            } we have created
       return scaffold (,);                        } this state
                        ↳
                      bag color

   }
 }
```

some basic properties

```
in AppBar(
    title: Text('  ');
→ to make it center   [centerTitle: true]
```

```
scaffold
  appBar: AppBar()
    body: _buildUI(),
  );
```

login page body ui



```
→ Widget _buildUI(){
    return Column(
        children: [
            _title(),
          _loginForm(),
        ],
      ));
  }
```

this is title widget

```
→ Widget _title() {
    return const Text( "Reep book", style = TextStyle (fontSize: 30,
            fontWeight: FontWeight.w500, ),); ,}
```

```
→ Widget _loginform() {
    return SizedBox (
            width :
            height:
            child : ~~TextFormField~~(), Form(
                   child: Column(
                    children: [
                       TextFormField(
                          decoration: InputDecoration(
                                hintText:"   ",
                     ),
                     ),
          ),7 /
    / / (
```

so basically we have a widget ~~_buildUI and we are creating~~ our small other widget and calling in it.

Form(), TextFormField
↓
decoration

```
SizedBox
  ↓
child ~~form~~
  ↓column
   ↓
   children
   ↓
   TextFormField
```

Project name:- Blog App

flutter CLEAN Architecture

Date:- / /
Page No.:-

- flutter clean architecture is an approach to structuring flutter applications to enhance maintainability, scalability & testability.
- it devides the app into distinct layers, each with a responsibility.
- main idea is to decouple the code into layers that communicate through well defined interfaces, making it easier to modify & extend application without affecting other parts.

→ Three layers of Clean Architecture

1) Presentation layer :- responsible for UI and handling user interactions. It contains flutter widgets, state management & pages.

- 1) widgets :- components that user interacts with.
- 2) state management :- handles the state of UI, Bloc.
- 3) Pages :- logic related to UI.

2) Domain layer :- core layer where business logic resides, defines entities, usecases & business rules.

- 1) Entities :- objects, data types & simple data classes. (how data are stored & retrieve)
- 2) usecases :- application specific operation involving multiple entites.
- 3) Repository (Interfaces) :- define interfaces & contracts for data acess, abstracting underlying data sources.

3) Data layer :- handles data management, including fetching data from API, database and other sources.

- implements interface actual impl^n in this layer.

1) Models (structuring of data).

2) Data sources :- remote, APIs, local database etc.

3) Repositories (Implementation) :- implement repository interface from domain layer, managing data sources and providing data to use cases.

* **flow** :-

1) **Pl to Dl** :- Ul interacts with domain layer through use cases, Pl triggers uses cases to perform actions or fetch data.

2) **Dl to Da layer**: uses cases request data through interfaces. Dl has no knowledge of actual data source implementation, a/c to dependency inversion principle.

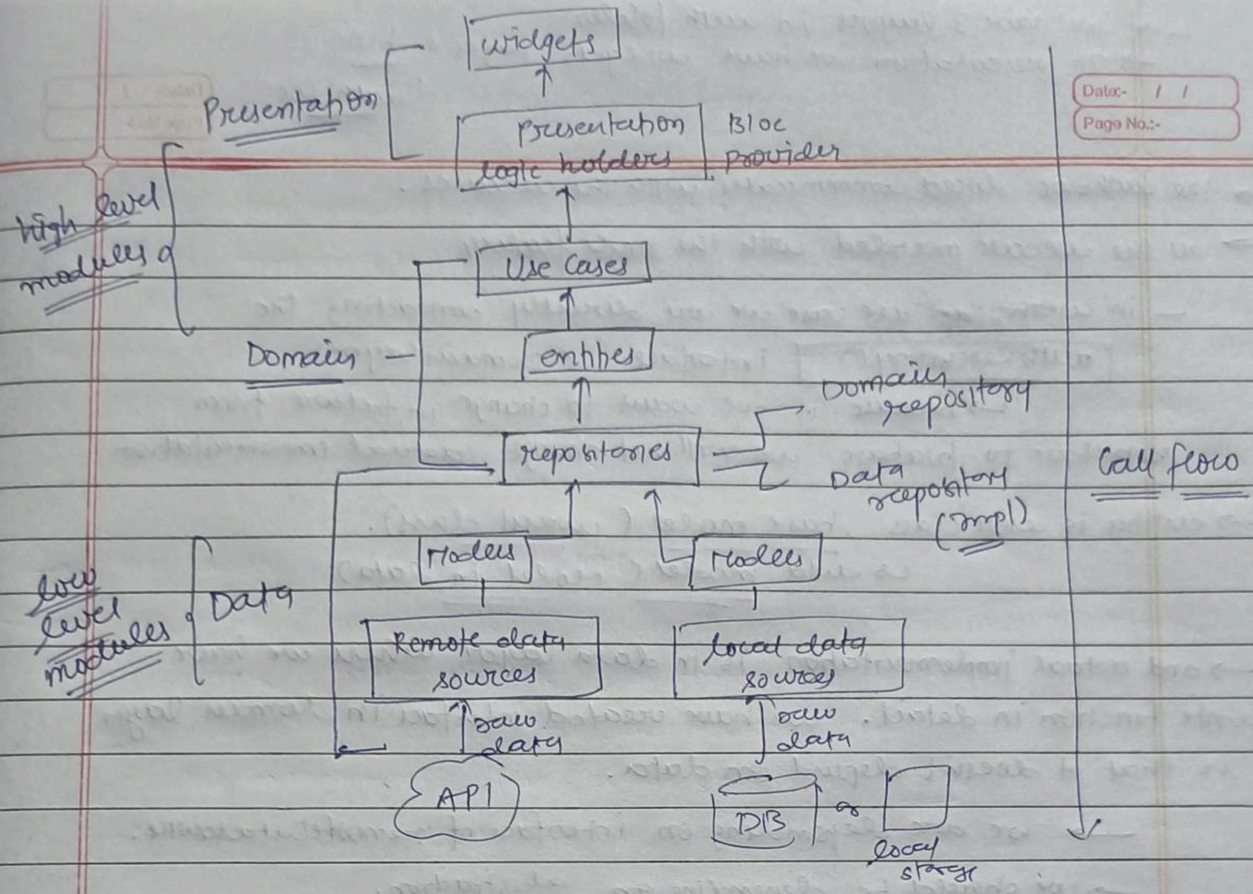
3) **Data layer**: fullfill request from domain layer by interacting with data sources, converting raw data into domain entities. (Models → json format).

→ usecase represents a single action that application can perform. ensures that all business rules are executed in consistent manner. & application logic is centralized.

→ interface, ~~depends~~ defines a contract that specifies what operations can be performed to access data. It doesn't specify how these operations are implemented.

imp

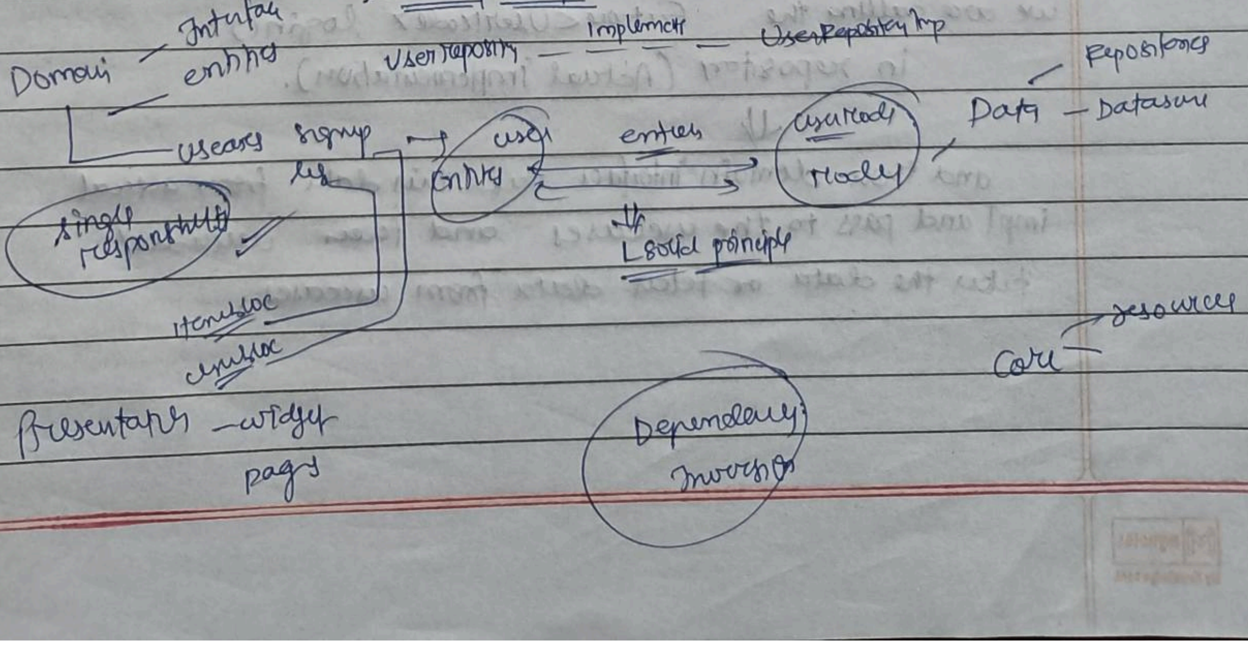
- it ensures that high level modules (use cases) are not dependent on low level modules (data), instead both should depend on abstraction (Interfaces).
- interfaces can change their internal workings without affecting uses cases.
- according to DIP, domain and presentation layers should not depend directly on concrete implementation of data layer.
- abstraction should not depend on details, details should depends on abstraction.



- for each feature in our project we have to create three folders:-

```
-features
  └ auth
      └ presentation
      └ Domain
      └ data
  └ Blog
      └ Presentation
      └ Domain
      └ data
```

solid principles



→ we have 3 layers in auth folder
→ In presentation we have widgets, pages & bloc

↓ authbloc

Date:- / /
Page No.:-

→ the authbloc direct communicates with the usecases.

→ all the usecase provided with the single features

— in userSignUp use case we are directly contacting the

[authRepository()] interface. (Domain layer)

↳ because if we want to change in future from supabase to firebase we will not change actual implementation.

→ euthha is user class ,base model (parent class).

↳ child model (model in data)

→ and actual implementation is in data layer where we have write function in detail. we have created interface in domain layer so that it doesn't depend on data.

→ we are depending on interface of remoteDatasource.

→ we should be depending on abstraction.

→ in actual implementation we are getting raw data from our database.

→ using (.fromJSON) → we are converting raw data into model and then send it to the data repository.

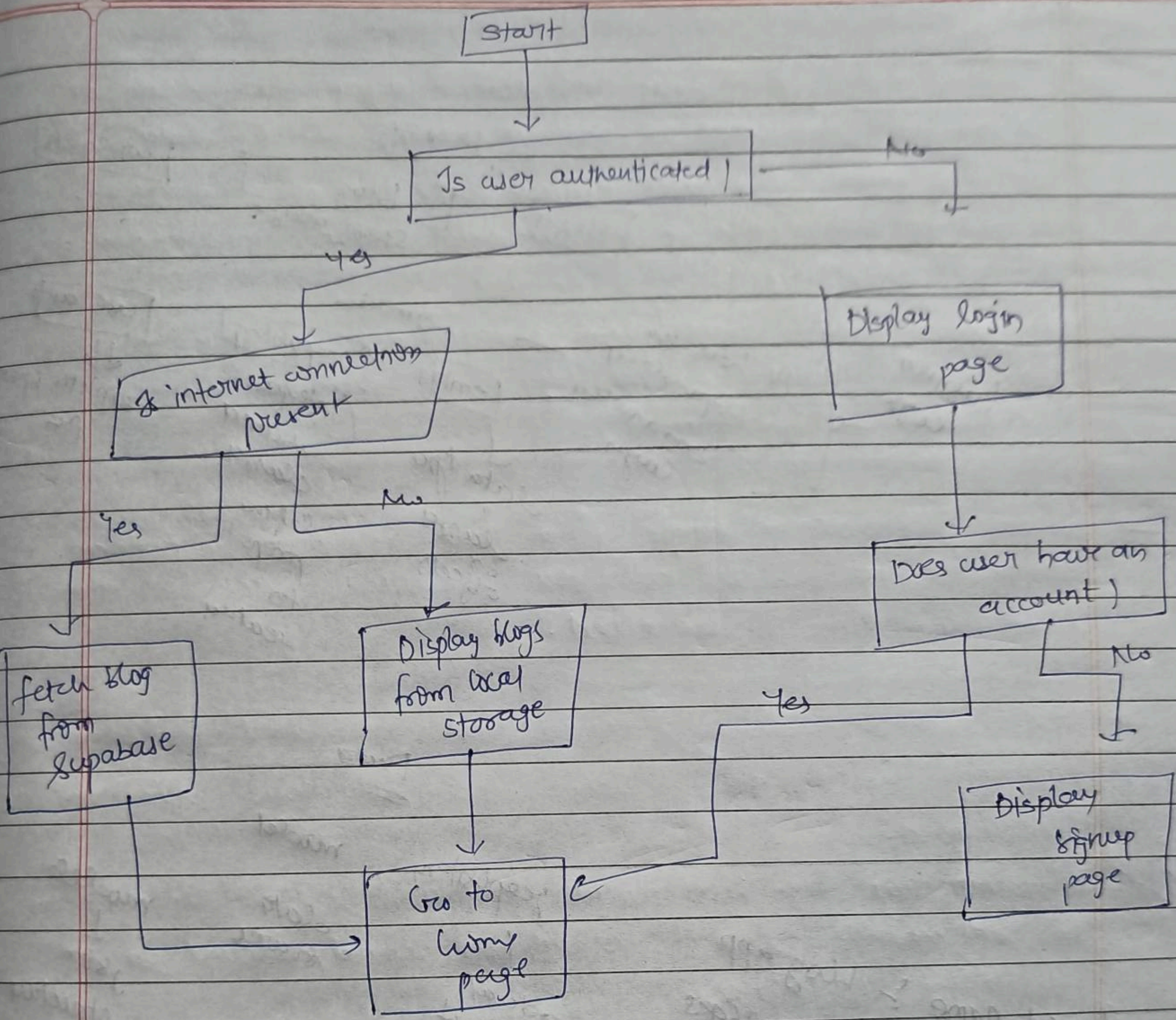
- this fn is define in our UserModel Class.

- and then when we return usermodel from authremotedatasource we are calling the Future<UserModel> login() in repository (Actual implementation).

⇓

and then domain interface take this data from actual impl and pass to the usecases and then authblock takes the data or fetch data from usecases, Clean architecture (blogApp)

Date: / /
Page No.:



only two features:
- authentication
- blogs
  - ↳ creation
  - ↳ reading

- Clean architecture folder structure.
  - Presentation - widgets
  - Domain - uses cases, Entities
  - Data
    - ↳ DB, APIs, local data source, models,

features first approach
based on the features each feature have its own folder.