SHREE H. N. SHUKLA COLLEGE OF I.T. & MGMT.

(AFFILIATED TO SAURASHTRA UNIVERSITY)

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Lt. Shree Chimanbhai Shukla

MSCIT SEM-3 – FLUTTER APP DEVELOPMENT

Shree H.N.Shukla College
Street No. 2, Vaishali Nagar,
Nr. Amrapali Under Bridge,
Raiya Road, Rajkot.
Ph. (0281)2440478, 2472590

Shree H.N.Shukla College
Street No. 2, Vaishali Nagar,
Nr. Amrapali Under Bridge,
Raiya Road, Rajkot.
Ph. (0281)2440478, 2472590

SHREE H. N. SHUKLA COLLEGE OF I.T. & MGMT.

(AFFILIATED TO SAURASHTRA UNIVERSITY)

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

CS – 13 : Flutter App Development		
SR.NO.	**TOPIC**	**DETAIL**
1	**Introduction to Flutter and Dart**	• Overview of Flutter and Dart • Overview of the Flutter architecture and how it works • Setting up the development environment • Dart syntax and data types • Basic Flutter widgets and layout • Basic Flutter widget properties and methods • Dart libraries and packages • Control flow and loops in Dart
2	**Building User Interfaces**	• Stateful vs. Stateless widgets • Layout and widgets hierarchy • Navigation and routing • Responsive design and media queries • Advanced Flutter widgets (e.g., • SliverAppBar, AnimatedContainer) • Custom widget creation in Flutter • Debugging UI issues in Flutter • Advanced layout techniques (e.g., Flexbox, GridView)
3	**Managing App State**	• State management in Flutter • InheritedWidget and InheritedModel • ScopedModel and Provider • BLoC pattern for state management • Stream-based state management with

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

		RxDart
		• Redux architecture for Flutter • Firebase integration for state management • Using the Flutter DevTools for debugging
4	**Networking and Persistence**	• RESTful APIs and HTTP requests • JSON serialization and deserialization • SQLite and local storage • Shared preferences and secure storage • WebSockets for real-time • communication in Flutter • Firebase integration for data storage • Caching data in Flutter • Using third-party libraries for networking and data storage (e.g., Dio, Hive)
5	**Animation, Integration, Testing, Debugging & Accessibility**	• Animations and motion • Advanced animation techniques (e.g., • Flare, Lottie) • Internationalisation and localization • Native platform integration • Push notifications in Flutter (Firebase) • Integration with other native features • (e.g., camera, location) • Testing and debugging • Accessibility in Flutter apps

SHREE H. N. SHUKLA COLLEGE OF I.T. & MGMT.
(AFFILIATED TO SAURASHTRA UNIVERSITY)

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

CHAPTER-3
Managing App State

- State management in Flutter
- InheritedWidget and InheritedModel
- ScopedModel and Provider
- BLoC pattern for state management
- Stream-based state management with RxDart
- Redux architecture for Flutter
- Firebase integration for state management
- Using the Flutter DevTools for debugging

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Q-1 Explain State Management in Flutter.

Answer:

- Managing state in an application is one of the most important and necessary process in the life cycle of an application.
- Let us consider a simple shopping cart application.
  - User will login using their credentials into the application.
  - Once user is logged in, the application should persist the logged in user detail in all the screen.
  - Again, when the user selects a product and saved into a cart, the cart information should persist between the pages until the user checked out the cart.
  - User and their cart information at any instance is called the state of the application at that instance.
- A state management can be divided into two categories based on the duration the particular state lasts in an application.
  - ✓ **Ephemeral** − Last for a few seconds like the current state of an animation or a single page like current rating of a product. *Flutter* supports its through StatefulWidget.
  - ✓ **app state** − Last for entire application like logged in user details, cart information, etc., *Flutter* supports its through scoped_model.

Ephemeral State Management

- Flutter application is composed of widgets, the state management is also done by widgets.
- The entry point of the state management is Statefulwidget.
- Widget can be inherited from Statefulwidget to maintain its state and its

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

children state.

- Statefulwidget provides an option for a widget to create a state, State (where T is the inherited widget) when the widget is created for the first time through createState method and then a method, setState to change the state whenever needed.
- The state change will be done through gestures. For example, the rating of a product can be changed by tapping a star in the rating widget.

- Let us create a widget, RatingBox with state maintenance.
- The purpose of the widget is to show the current rating of a specific product.
- The step by step process for creating a RatingBox widget with state maintenance is as follows −

- Create the widget, RatingBox by inheriting StatefulWidget.
  class RatingBox extends StatefulWidget { }

- Create a state for RatingBox, _RatingBoxState by inheriting State<T>
  class _RatingBoxState extends State<RatingBox> { }

- Override the createState of StatefulWidget method to create the state, _RatingBoxState.

```
class RatingBox extends StatefulWidget {
  @override
  _RatingBoxState createState() => _RatingBoxState();
}
```

- Create the user interface of the RatingBox widget in build method of _RatingBoxState.
- Usually, the user interface will be done in the build method of RatingBox widget itself. But, when state maintenance is needed, we need to build the user interface in _RatingBoxState widget.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- This ensures the re-rendering of user interface whenever the state of the widget is changed.

App State Management

- Flutter provides an easy way to manage the state of the application using scoped_model package.
- Flutter package are simply library of reusable functionality.

Model

- Model encapsulates the state of an application.
- We can use as many Model (by inheriting Model class) as needed to maintain the application state.
- It has a single method, notifyListeners, which needs to be called whenever the Model state changes. notifyListeners will do necessary things to update the UI.

```
class Product extends Model {
  final String name;
  final String description;
  final int price;
  final String image;
  int rating;

  Product(this.name, this.description, this.price, this.image, this.rating);
  factory Product.fromMap(Map<String, dynamic> json) {
    return Product(
      json['name'],
      json['description'],
      json['price'],
      json['image'],
      json['rating'],
    );
  }
```

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

```
  void updateRating(int myRating) {
    rating = myRating; notifyListeners();
  }
}
```

ScopedModel

- ScopedModel is a widget, which holds the given model and then passes it to all the descendant widget if requested.
- If more than one model is needed, then we need to nest the ScopedModel.

- **Single model**

```
ScopedModel<Product>(
  model: item, child: AnyWidget()
)
```

- **Multiple model**

```
ScopedModel<Product>(
  model: item1,
  child: ScopedModel<Product>(
    model: item2, child: AnyWidget(),
  ),
)
```

Q-2 Explain InheritedWidget and InheritedModel in Flutter.

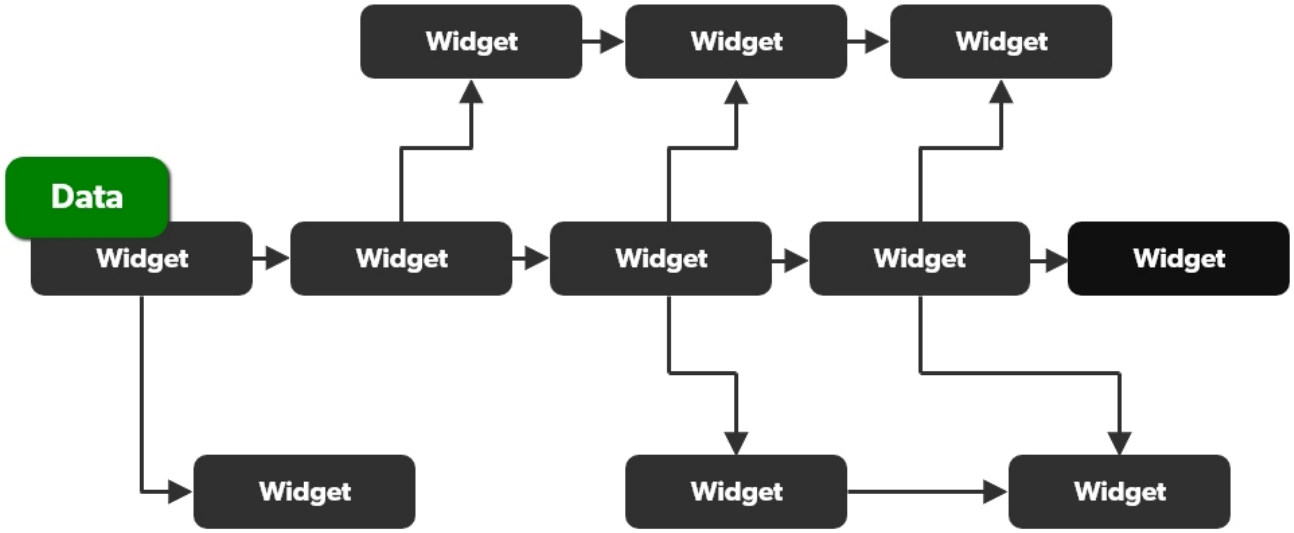
Answer:

- when you have lots of widgets in your app and you want to pass data from one widget to another widget, this will be a pain for many developers.
- this will be really hard for them to pass the data from a widget of widget tree to bottom widget.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- widgets there are generally two types of widgets Stateless and stateful widgets.
- But there is another type of widget called InheritedWidget.



✓ Inherited Widget :-

- The inherited widget provides us with a convenient way to pass data between widgets.
- While developing an app you will need some data from your parent's widgets or grant parent widgets or maybe beyond that.
- So if your app is small you can pass your data using the constructor of the widget class, but for a larger app, this is not an easy task.
- Unknowingly we have used inherited widgets in many places while developing the flutter app.Theme.of(context), MediQuery.of(context), and Navigator.of(context) (**if you check the codes of Navigation you will find that**).

```
const InheritedWidget({
Key? key, required Widget child
```

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

```
  })
      : super(key: key, child: child);
```

- The first parameter is the key, a key is an identifier for widgets, and elements in Flutter.
- A new widget will only be used to update an existing widget if its key is the same as the key of the current widget associated with the element.
- The child parameter is of type Widget so we can pass any widget as a parameter as a child widget. Let's create a simple example,

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: MyScreen(),
    );
  }
}

class MyInheritedWidget extends InheritedWidget {
  const MyInheritedWidget({Key? key,required this.child,required
this.message}) : super(key: key, child: child);

  final Widget child;
```

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

```
  // message of our inherited widget class
  final String message;

  static MyInheritedWidget of(BuildContext context) {
    return
context.dependOnInheritedWidgetOfExactType<MyInheritedWidget>()!;
  }

  @override
  bool updateShouldNotify(MyInheritedWidget oldWidget) {
    return oldWidget.message != message;
  }
}

class MyScreen extends StatelessWidget {
  const MyScreen({Key? key}) : super(key: key);

    @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text("GFG"),
        backgroundColor: Colors.green,
      ),
      body: MyInheritedWidget(
        // passing the message as string
        message:"Hey GEEKS",
        child: Builder(
          builder: (BuildContext innerContext) {

            return Center(
                    child: Text(
                  // using the message of our inherited widget using of()
                  MyInheritedWidget.of(innerContext).message,
```

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645


```
                style: const TextStyle(fontWeight: FontWeight.bold),
              )
            );
          },
        ),
      ),
    );
  }
}
```

✓ **Inherited Model :-**

- An InheritedWidget that's intended to be used as the base class for models whose dependents may only depend on one part or "aspect" of the overall model.
- An inherited widget's dependents are unconditionally rebuilt when the inherited widget changes per InheritedWidget.updateShouldNotify. This widget is similar except that dependents aren't rebuilt unconditionally.
- Widgets that depend on an InheritedModel qualify their dependence with a value that indicates what "aspect" of the model they depend on. When the model is rebuilt, dependents will also be rebuilt, but only if there was a change in the model that corresponds to the aspect they provided.

- The type parameter T is the type of the model aspect objects.

- Widgets create a dependency on an InheritedModel with a static method: InheritedModel.inheritFrom. This method's context parameter defines the subtree that will be rebuilt when the model changes. Typically the inheritFrom method is called from a model-specific static maybeOf or of methods, a convention that is present in many Flutter framework classes which look things up. For example:

```
class MyModel extends InheritedModel<String> {
  const MyModel({super.key, required super.child});

  // ...
```

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

```
  static MyModel? maybeOf(BuildContext context, [String? aspect]) {
    return InheritedModel.inheritFrom<MyModel>(context, aspect: aspect);
  }

  // ...
  static MyModel of(BuildContext context, [String? aspect]) {
    final MyModel? result = maybeOf(context, aspect);
    assert(result != null, 'Unable to find an instance of MyModel...');
    return result!;
  }
}
```

- Calling MyModel.of(context, 'foo') or MyModel.maybeOf(context, 'foo') means that context should only be rebuilt when the foo aspect of MyModel changes.
- If the aspect is null, then the model supports all aspects.
- When the inherited model is rebuilt
  the updateShouldNotify and updateShouldNotifyDependent methods are used to decide what should be rebuilt. If updateShouldNotify returns true, then the inherited model's updateShouldNotifyDependent method is tested for each dependent and the set of aspect objects it depends on.
  The updateShouldNotifyDependent method must compare the set of aspect dependencies with the changes in the model itself. For example:
- link
- content_copy

```
class ABModel extends InheritedModel<String> {
  const ABModel({
    super.key,
    this.a,
    this.b,
    required super.child,
  });

  final int? a;
  final int? b;
```

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

```
  @override
  bool updateShouldNotify(ABModel oldWidget) {
    return a != oldWidget.a || b != oldWidget.b;
  }

  @override
  bool updateShouldNotifyDependent(ABModel oldWidget, Set<String>
   dependencies) {
    return (a != oldWidget.a && dependencies.contains('a'))
      || (b != oldWidget.b && dependencies.contains('b'));
  }

  // ...
}
```

Q-3 Explain ScopedModel and Provider in Flutter.

Answer:

✓ **ScopedModel :-**

- ScopedModel is a widget, which holds the given model and then passes it to all the descendant widget if requested.
- If more than one model is needed, then we need to nest the ScopedModel.
- Scoped Model is somehow advanced compared to Inherited Widget and it's also better in performance as well.

✓ **Flutter Provider Package :-**

- The provider package is an easy to use package which is basically a wrapper around the InheritedWidgets that makes it easier to use and manage. It provides a

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

state management technique that is used for managing a piece of data around the app.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- These two might seem redundant because the big difference is that Provider gives you a lot more options, and it can use ChangeNotifier which is part of the Flutter framework. ScopedModel is a lot more simplified and you extend Model, which is a specific class within the ScopedModel package.
- ScopedModel is like a version of Provider for Dummies. Provider is a lot more flexible, but not quite as easy to use.

- The basic classes available in the provider package are –

- ChangeNotifierProvider<T extends ChangeNotifier>

  It listens to a ChangeNotifier extended by the model class, exposes it to its children and descendants, and rebuilds depends whenever notifyListeners is called.

```
ChangeNotifierProvider(
  create: (context) => DataModel(),
  child: ...
)
```

- Consumer<T> It obtains the provider from its ancestors and passes the value obtained to the builder.

```
@override
Widget build(BuildContext context) {
  return Consumer<DataModel>(
  builder: (context, data, child) => DataWidget(par1: par1, par2: par2),
    child: Text(data.first),
  );
}
```

- InheritedProvider<T> The InheritedProvider provides a general implementation of the InheritedWidget.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- MultiProvider A provider that is used to provide more than one class at the same time.

```
MultiProvider(
        providers: [
          Provider<Model1>(create: (context) => Model1()),
          StreamProvider<Model2>(create: (context) => Model2()),
          FutureProvider<Model3>(create: (context) => Model3()),
        ],
        child: someWidget,
      )
```

- Provider<T> It is the basic provider.

- ProxyProvider<T, R> This provider depends on other providers for value. The value can be used by create or update.

```
Constructor
ProxyProvider(
   {Key key,
   Create<R> create,
   @required ProxyProviderBuilder<T, R> update,
   UpdateShouldNotify<R> updateShouldNotify,
   Dispose<R> dispose, bool lazy,
   TransitionBuilder builder,
   Widget child}
)
```

This initializes key for subclasses.

- StreamProvider<T> This class listens for a Stream and then passes its values to its children and descendants. This can be used as

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

```
Constructors
StreamProvider<T>(
   {Key key,
   @required Create<Stream<T>> create,
   T initialData,
   ErrorBuilder<T> catchError,
   UpdateShouldNotify<T> updateShouldNotify,
   bool lazy,
   TransitionBuilder builder,
   Widget child}
)
```

- This creates a stream using create and subscribes to it.

```
StreamProvider.value(
   {Key key,
   @required Stream<T> value,
   T initialData,
   ErrorBuilder<T> catchError,
   UpdateShouldNotify<T> updateShouldNotify,
   bool lazy,
   TransitionBuilder builder,
   Widget child}
)
```

- This constructor notifies the changed values to the StreamProvider children.

Q-4 Explain BLoC pattern for state management in Flutter.

Answer:

- State management is an essential part of building any app, and it becomes especially important as your app grows in complexity.
- In Flutter, there are a number of approaches that you can take to manage state, including using global variables, Inherited Widgets, and more recently, the Provider package.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- One approach that has gained popularity in the Flutter community is the BLoC (Business Logic Component) pattern.

- The BLoC pattern is a reactive architecture that separates business logic from the user interface, making it easier to manage state and build modular, testable code.
- The **BLoC pattern** works by separating business logic from the user interface, allowing for more modular and testable code.
- At the core of the pattern is the BLoC itself, which is a standalone Dart class that contains all of the business logic for a particular feature of the app.
- To use the BLoC pattern in a Flutter app, you'll need to ,

1. Create a BLoC class: The first step in implementing the BLoC pattern is to create a standalone Dart class that contains all of the business logic for a particular feature of the app. This class should be responsible for managing the state of the app and handling any user interactions or updates.

2. Define streams: The BLoC class should define streams to emit events and receive updates. These streams can be used to communicate with the rest of the app and update the user interface.

3. Connect the BLoC to the user interface: To connect the BLoC to the user interface, use the StreamBuilder widget to listen to the streams defined in the BLoC class. The StreamBuilder will rebuild the user interface whenever a new event is emitted.

4. Handle user actions: The BLoC class should handle any user actions by updating its internal state and emitting events through the streams. This will allow the user interface to be updated in real-time.

5. Test the BLoC: Because the BLoC is a standalone class, it can be tested in isolation from the rest of the app. This makes it easier to ensure that the business logic is correct and that the BLoC is managing the state correctly.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

6. Keep the BLoC as simple as possible: To efficiently manage the state in Flutter with the BLoC pattern, it is important to keep the BLoC class as simple as possible. Avoid adding unnecessary logic or functionality to the BLoC class and keep the focus on managing state and handling user interactions.

7. Update the state of the BLoC: The BLoC class should handle any updates to its internal state in a consistent and predictable manner. This can be done by using a state management library such as Provider, which will make it easier to update the state of the BLoC and keep the user interface in sync.

Here's an example of a simple BLoC class that manages a counter:

- Dart

```
class CounterBloc {
 final _counterController = StreamController<int>();
 StreamSink<int> get _inCounter => _counterController.sink;
 Stream<int> get outCounter => _counterController.stream;

 void incrementCounter() {
  _inCounter.add(counter + 1);
 }

 void dispose() {
  _counterController.close();
 }
}
```

- To use this BLoC in the UI, you can use a StreamBuilder widget to listen to the outCounter stream and rebuild the UI whenever the counter value changes.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Q-5 Explain Stream-based state management with RxDart in Flutter.

Answer:

- if we have an application that displays a list of universities we need a screen with a list of widgets where each widget displays some information for a specific university.
- This screen should also display something when it loads the data (e.g. a loading spinner) and it also has to show something when it failed to load the data.
- We need to find a way to manage all 3 phases (Loading, Success, Error) within the app life-cycle in an easy-to-test and easy-to-use way.\
- These 3 phases, the moment when the screen is loading the data, the moment when the data is successfully loaded and the moment when there was something wrong and an error is displayed, build the screen state.
- What **we will achieve** here is **to manage the screen state** using Dart streams and **RxDart** library.

What is RxDart?

- **Rx** stands for **ReactiveX** and it comes from reactive programming.
- In Dart, **RxDart** comes with a bunch of extensions over `Streams` and `StreamControllers` and they introduce a lot of other specific Rx components such as `BehaviourSubject`, `ReplaySubjects`, `MergeStreams`, `CombineStreams` etc.
- Rx library is based on functional programming style, meaning we can have a chain of transformation functions applied to a stream:

- Since **RxDart** uses Streams it follows the Observer pattern.

- What you **must always know** when working with streams is that **they never work unless you subscribe** to them.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645


```
// Stream.fromIterable builds a stream which emits the values from the given list
Stream.fromIterable([1, 2, 3, 4, 5, 6, 7])
    // where function calls its body each time a value is emitted by the stream
    // and checks if it is an even number. This function acts like a filter.
                        .where((element) {
        // if the element is even this function returns true
        // and the element passes the filter
        // otherwise it returns false and the element is filtered out.
                    return element % 2 == 0;
            });
```

- The above code **does nothing** because there is no other code that subscribes/listens to it.
- So what we can do to make it work is to call listen on this stream.

```
var evenNumbers = Stream.fromIterable([1, 2, 3, 4, 5, 6, 7])
                        .where((element) {
                    return element % 2 == 0;
            });
evenNumbers.listen((element){
        print(element);
});
```

- Now, what `.listen` function does is to create a subscription to the `evenNumbers` stream and listen to its events.
- The moment `.listen` is called and registered as a subscription for `evenNumbers` stream, the stream code starts working.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- Firstly it takes the first element from the list then it checks it in the `.where` function and then if it passes the test (in our case is an even number) it is sent in the `listen` callback and then it's printed.
- So the above code prints `2 4 6`

Q-6 Explain how to manage state with RxDart.

Answer:

- Before jumping into the code I want to describe a little the application built using **RxDart** for state management.
- There will be an app that provides a list of universities that can be queried by their country.
- There we are using:

- RxDart for data streams and state management
- **MVVM** with **Clean Architecture** as a design pattern
- Retrofit with Dio for handling API calls
- Freezed for avoiding models boilerplate code
- Json Serializer for helping us with serializing and de-serializing JSON data
- Free API http://universities.hipolabs.com for getting universities' data

- Since we are following the MVVM with Clean Architecture it means we have the following flow of data:

1. Wiget/Screen is opened
2. When it is initialized, its ViewModel is initialized with it as well
3. The ViewModel calls the usecase for initializing the data
4. The usecase calls the repository to get the data
5. Based on where the data is and the logic behind it, the repository calls an endpoint to an API or queries the local database, or gets the data from shared preferences

   The data comes back exactly through the same classes in reverse order using a stream.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- E.g:

1. UniversitiesScreen is opened
2. UniversitiesScreen initializes the ViewModel
3. UniversitiesScreen subscribes to the stream with universities data from ViewModel
4. UniversitiesViewModel calls GetUniversitiesByCountryUseCase to get the universities.
5. The use-case calls UniversitiesRepository to get the universities' data
6. The repository calls the UniversityRemoteDataSource to get the universities' data
7. The data source using UniversityEndpoint does an API request to the API to fetch the data

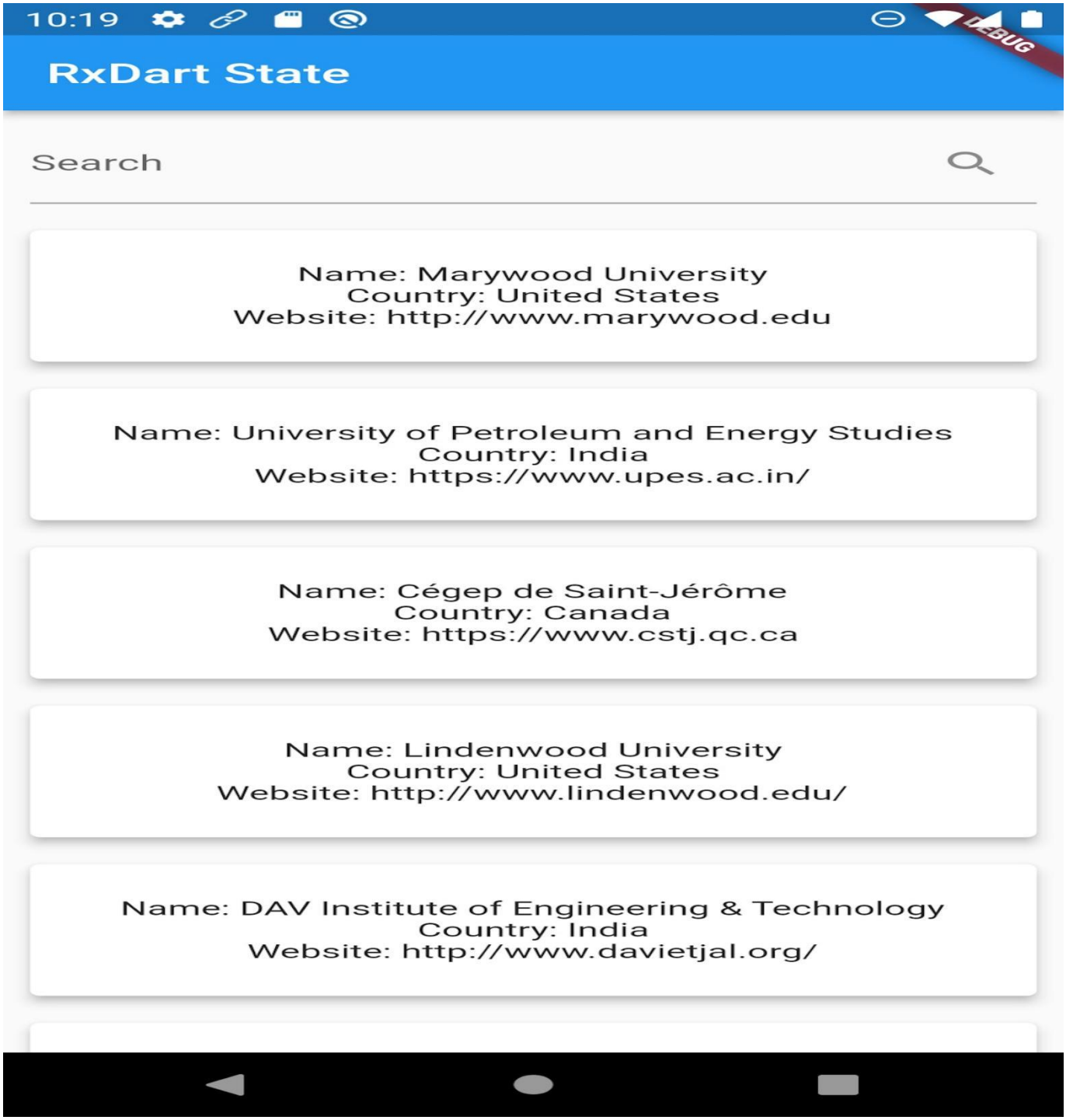
- When there is a result from the API call the data goes all the way back via the streams.

1. UniversityEndpoint gets the data (error, data, loading)
2. it sends it back to UniversityRemoteDataSource
3. sends it back to UniversitiesRepository
4. sends it back to GetUniversitiesByCountryUseCase
5. sends it back to UniversitiesViewModel
6. then it finally arrives on the UniversitiesScreen where it is handled by its state and displayed

- Since we are using streams, we can easily send how many values we want on the same stream.
- Here we have the UniversitiesScreen:

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645



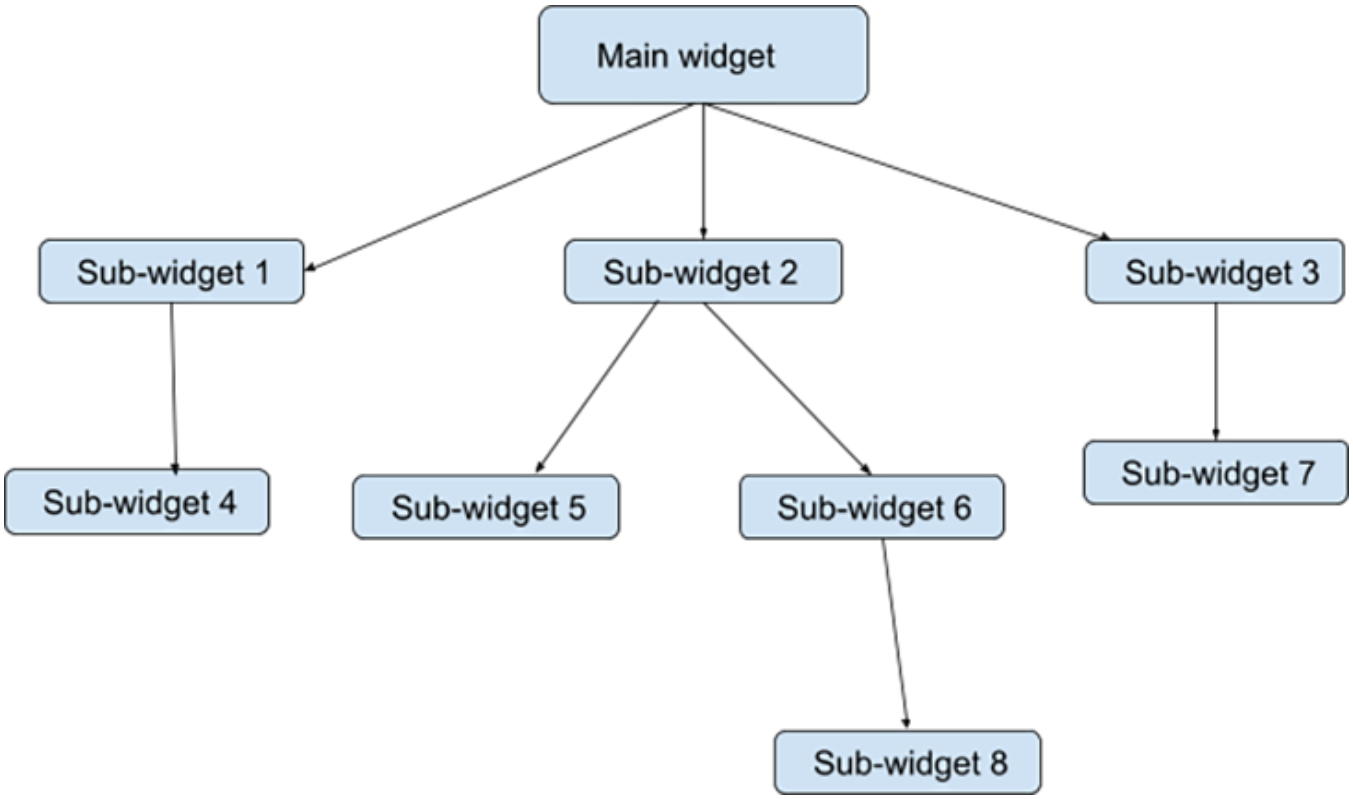
2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Q-7 Explain Redux architecture for Flutter.

Answer:

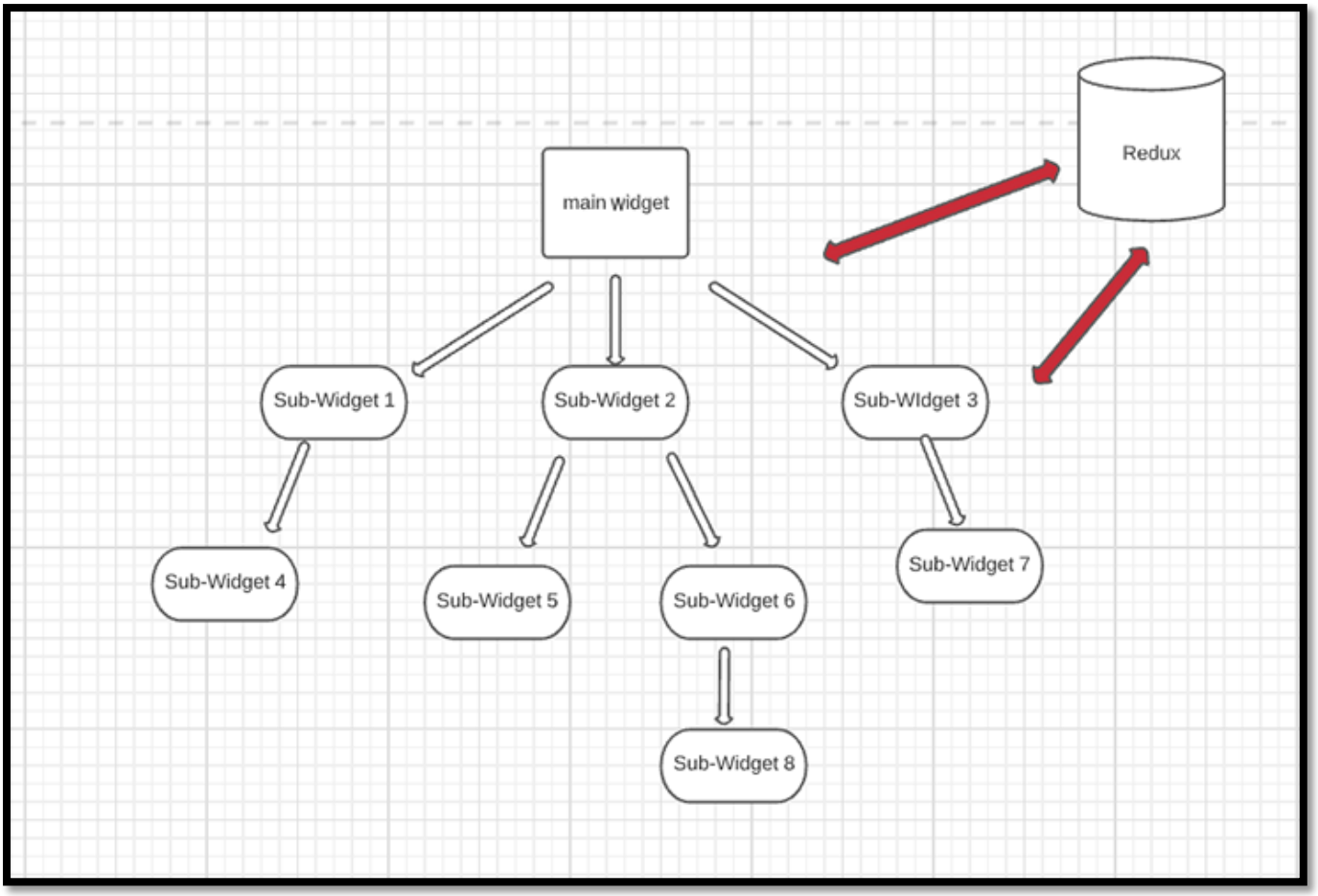
- Redux is a state management architecture library that successfully distributes data across widgets in a repetitive manner. It manages the state of an application through a unidirectional flow of data. Let's explore the diagram below:



- With Redux, you can structure your application so that the state is extracted in a centrally-located store.
- The data in this centralized store can be accessed by any widget that requires the data, without needing to pass through a chain of other widgets in the tree.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645



- Any widget that needs to add, modify, or retrieve the data in a state managed by the Redux store would have to request it with the appropriate arguments.
- Likewise, for every change made to the state, the dependent widgets respond to the change either through the user interface or any other configured means

Redux Setup:

Install Packages:

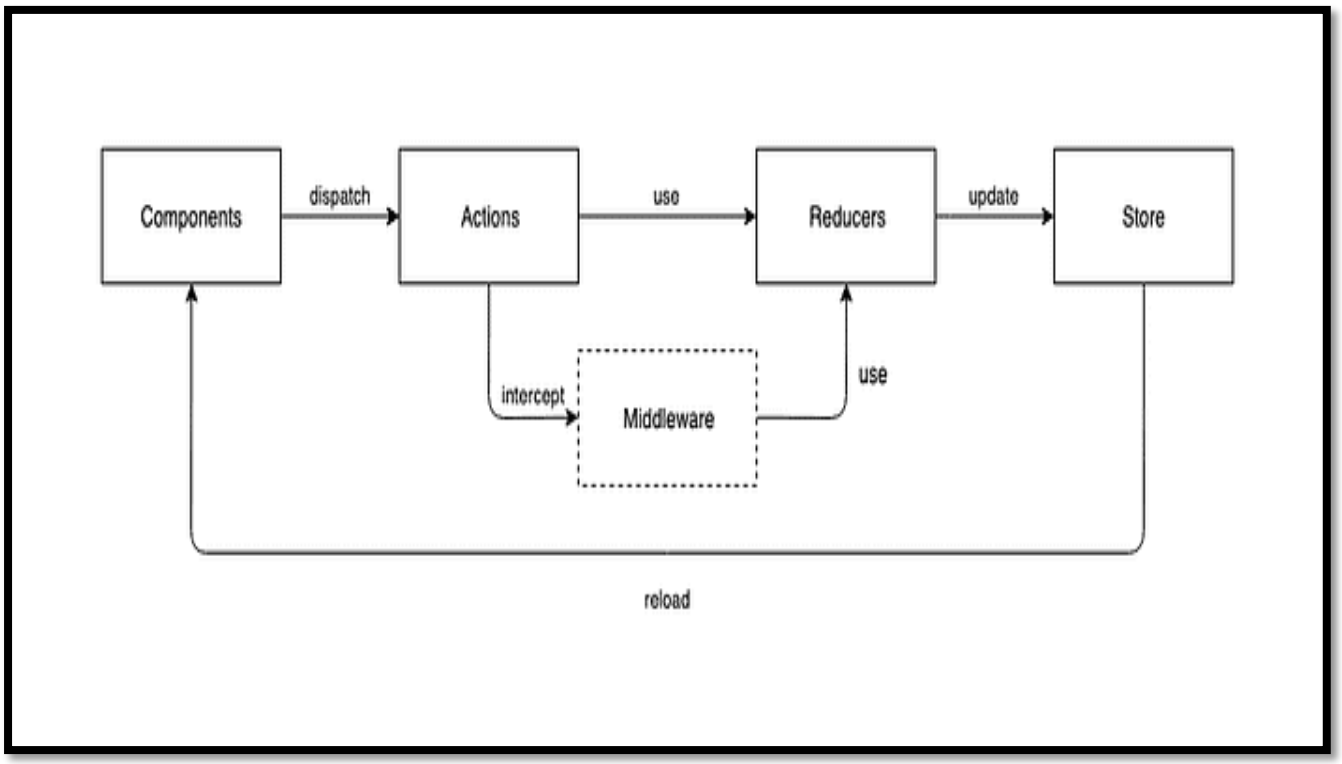
2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- Now we'll add the Redux dependencies to our app with the latest version of package and add them to the ***pubspec.yaml*** file.

redux: ^5.0.0**flutter_redux**: ^0.8.2**redux_thunk**: ^0.4.0

Redux Elements:



The Redux components

- we can See the brief explanation of each component in following with example code. Now we move to create modal class.
- We are taking example of login module with our flutter app. First we need to create login_model.dart

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Model:

- Here is our login model: login_model.dart

```
class Loginmodel {
  String token;

  Loginmodel({this.token});

  Loginmodel.fromJson(Map<String, dynamic> json) {
    token = json['token'];
  }

  Map<String, dynamic> toJson() {
    final Map<String, dynamic> data = new Map<String, dynamic>();
    data['token'] = this.token;
    return data;
  }
}
```

State:

- We would like to store the user data after a successful login and also we would like to present a loading indicator if an API call is in progress.
- Also, we'd like to display error message if an error happened. We have to compose the stored objects to match those requirements.

login_state.dart

```
class LoginState {
  bool loading;
  dynamic error;
  Loginmodel success;

  LoginState({
    this.loading,
```

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

```
  this.error,
  this.success,

});
 factory LoginState.initial() => LoginState(
       loading: false,
       error: null,
       success: null,

  );
}
```

- we'll create AppState this is our main state object, the one that holds entire applications state.

```
class AppState {final LoginState login;AppState({this.login,});factory AppState.initial() =>
AppState(login: LoginState.initial());AppState copyWith({LoginState login,}) {return
AppState(
login:login ?? this.login, );
}
}
```

Q-8 Explain Firebase integration for state management in Flutter.

Answer:

- Firebase is a Backend-as-a-Service (BaaS) app development platform that provides hosted backend services such as a realtime database, cloud storage, authentication, crash reporting, machine learning, remote configuration, and hosting for your static files.
- We seek a skilled Flutter developer to integrate our existing Flutter app with the Firebase database and handle state management using Bloc and freezed packages.
- The app's UI/UX design has been completed, and we require someone to connect the app to our Firebase database, execute queries, and manage the

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

received data to update app states. We need experts in layered/clean architecture.

- ✓ **Responsibilities:**
  1. Create and execute queries to fetch and update data from Firebase.
  2. Implement BLoC state management using the Freezed and JSON annotation packages.
  3. Handle form validation with Formz
  4. Implement Navigation with AutoRoute
  5. Collaborate with our UI/UX designers to ensure seamless integration of app functionality.
  6. Troubleshoot and debug any issues related to Firebase integration and Bloc state management.
  7. Implement Push Notifications using Firebase.

- ✓ Add Firebase to your Flutter app

Step 1: Install the required command line tools

1. If you haven't already, install the Firebase CLI.
2. Log into Firebase using your Google account by running the following command:

   firebase login

3. Install the FlutterFire CLI by running the following command from any directory:

   dart pub global activate flutterfire_cli

Step 2: Configure your apps to use Firebase

Use the FlutterFire CLI to configure your Flutter apps to connect to Firebase.

From your Flutter project directory, run the following command to start the app configuration workflow:

flutterfire configure

What does thisflutterfire configureworkflow do?

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

After this initial running of **flutterfire configure**, you need to re-run the command any time that you:

- Start supporting a new platform in your Flutter app.
- Start using a new Firebase service or product in your Flutter app, especially if you start using sign-in with Google, Crashlytics, Performance Monitoring, or Realtime Database.

Re-running the command ensures that your Flutter app's Firebase configuration is up-to-date and (for Android) automatically adds any required Gradle plugins to your app.

Step 3: Initialize Firebase in your app

1. From your Flutter project directory, run the following command to install the core plugin:

```
flutter pub add firebase_core
```

2. From your Flutter project directory, run the following command to ensure that your Flutter app's Firebase configuration is up-to-date:

```
flutterfire configure
```

3. In your lib/main.dart file, import the Firebase core plugin and the configuration file you generated earlier:

```
import 'package:firebase_core/firebase_core.dart';
import 'firebase_options.dart';
```

4. Also in your lib/main.dart file, initialize Firebase using the DefaultFirebaseOptions object exported by the configuration file:

```
await Firebase.initializeApp(
  options: DefaultFirebaseOptions.currentPlatform,
);
```

5. Rebuild your Flutter application:

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

flutter run

Step 4: Add Firebase plugins

You access Firebase in your Flutter app through the various Firebase Flutter plugins, one for each Firebase product (for example: Cloud Firestore, Authentication, Analytics, etc.).

Since Flutter is a multi-platform framework, each Firebase plugin is applicable for Apple, Android, and web platforms. So, if you add any Firebase plugin to your Flutter app, it will be used by the Apple, Android, and web versions of your app.

Here's how to add a Firebase Flutter plugin:

1. From your Flutter project directory, run the following command:

flutter pub add *PLUGIN_NAME*

2. From your Flutter project directory, run the following command:

flutterfire configure

Running this command ensures that your Flutter app's Firebase configuration is up-to-date and, for Crashlytics and Performance Monitoring on Android, adds the required Gradle plugins to your app.

3. Once complete, rebuild your Flutter project:

flutter run

- Like all packages, the firebase_analytics plugin comes with an example program.

1. Open a Flutter app that you've already configured to use Firebase (see instructions on this page).
2. Access the lib directory of the app, then delete the existing main.dart file.
3. From the Google Analytics example program repository, copy-paste the following two files into your app's lib directory:

- main.dart
- tabs_page.dart

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

4. Run your Flutter app.
5. Go to your app's Firebase project in the Firebase console, then click **Analytics** in the left-nav.

a. Click **Dashboard**. If Analytics is working properly, the dashboard shows an active user in the "Users active in the last 30 minutes" panel (this might take time to populate this panel).
b. Click **DebugView**. Enable the feature to see all the events generated by the example program.

Q-9 Explain How to use Flutter DevTools for debugging.

Answer:

- DevTools is a tooling suite for Flutter and Dart developers consisting of layout inspection tools, performance tools, memory tools basically all the debugging tools that you need to be an efficient and effective Flutter developer bundled into a single web suite.

Usage of DevTools:

- The Flutter DevTools can be used to perform a number of operations. Some of them are listed below:

1. UI inspection.
2. App state inspection.
3. Diagnose UI junk performance.
4. Diagnose issues with flutter apps.
5. DevTools use for CPU profiling.
6. Network profiling for an app.
7. Source-level debugging of an app.
8. Debug memory issues in a Flutter or Dart or command-line app.
9. View general log and diagnostics information of an app.
10. Analyze your code and app size of the flutter app.

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

Installing DevTools:

1. Install Flutter DevTool in Android Studio

Follow the below steps to install DevTools in your Android Studio:

- **Step 1:** Install the Flutter plugin in Android Studio, if you don't already have it installed. This can be done using the normal Plugins page in the Android Studio settings. Once *settings->plug-ins* page is open, you can search flutter in the marketplace and install the plugin.
- **Step 2:** You first run a Flutter app. Ensuring that you have a device connected to the project, and clicking the Run or Debug toolbar buttons.
- **Step 3:** Launch DevTools from the toolbar/menu in your flutter project. Once an app is running successfully, start DevTools implementing the following instruction one by one:

1. Open DevTools toolbar action from Run view.
2. DevTools toolbar action visible in the Debug view. (if debugging)
3. DevTools action from the More Actions menu in Inspector view in your flutter project.

2. Installing DevTools from VS Code

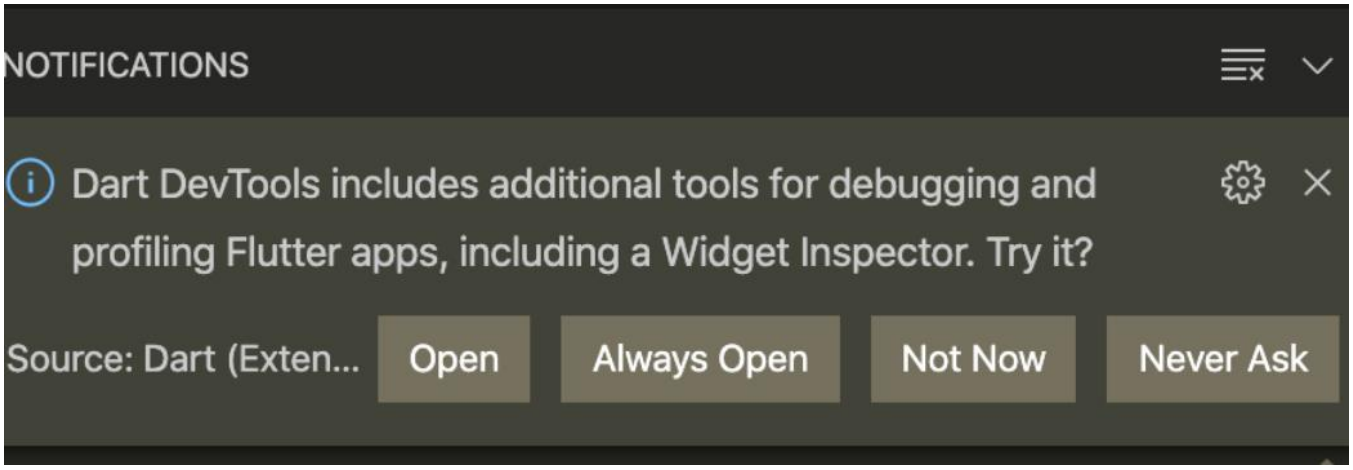
Follow the below steps to install DevTools from VS Code:

- **Step 1:** To use the DevTools from VS Code firstly install the Dart extension also you need to install the Flutter extension for debugging flutter applications.
- **Step 2:** Launch an application to debug your application. Start debugging in VS Code by clicking *Run > Start Debugging (F5)*.
- **Step 3:** If once debugging started, the Dart Opens DevTools command becomes available in the VS Code command palette:

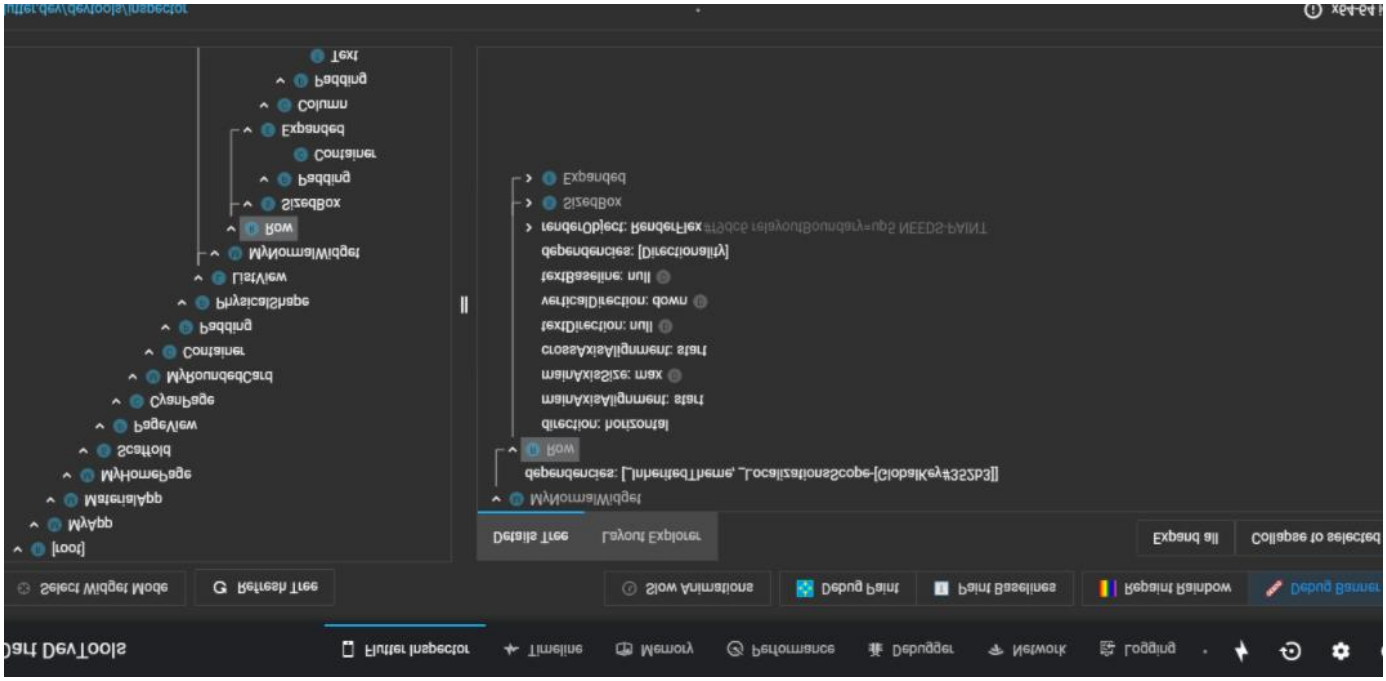
Screenshot showing Open DevTools command:

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645



- When you run your app the first time, you will be prompted to activate or upgrade DevTools. The below image shows an Active DevTools command:



- Clicking the Open button to activate the DevTools package for your application. After this, DevTools launches in your browser and automatically connects to your debug session as shown below:

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- When DevTools is in an active state, you'll see them in the status bar of VS Code.

3. Install DevTools from the command line

- If you have flutter on your computer, you can run:
- flutter pub global activate devtools
- The above command installs or updates DevTools on your machine or computer. Launch the DevTools from the application server. Run the local webserver. To do that runs the following commands:
- flutter pub global run devtools
- On the command line, the output looks something like this:
- Serving DevTools at http://127.0.0.1:9100

Start an application to debug:

- Start a Flutter application or a Dart command-line application. The command for flutter app:
- cd path/to/flutter/app
- flutter run
- You need to have a device connected for the flutter run to work. After the app starts, Following message in your terminal:

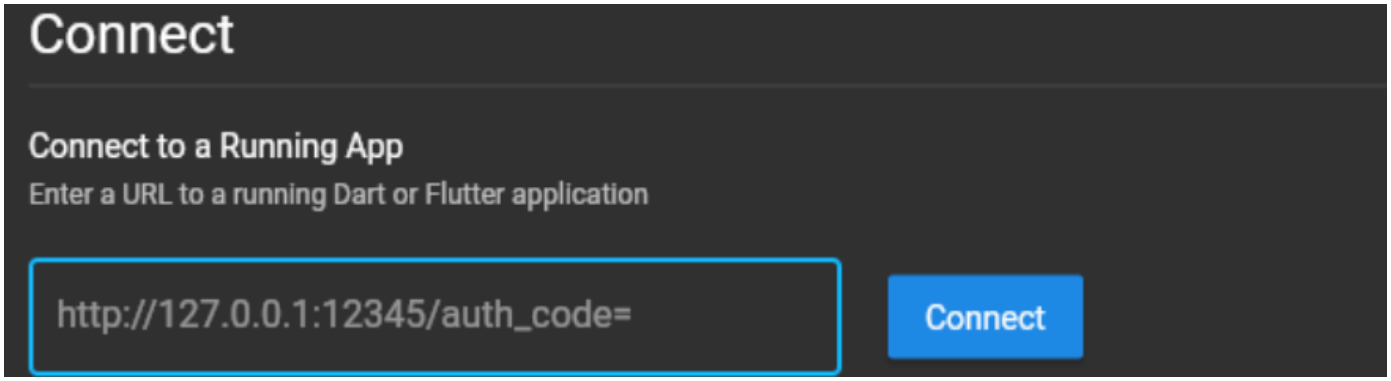
2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

- An Observatory debugger and profiler on Galaxys20 is available
- at: http://127.0.0.1:50976/Swm0bjIe0hj=/
- Note that this URL, we will use to connect our app to DevTools.

Connect DevTools to target app:

- Once All things set up, debugging through DevTools is very simple as opening a Chrome browser window and navigating to the below link:
- http://localhost:9100
- After DevTools opens, the dialog box pop-up you will get the below Logging view:



- Paste the URL, which you already note from running your app (example, http://127.0.0.1:50976/Swm0bjIe0hg=/) into the connect dialog to connect your app to DevTools to debugging.

The Flutter inspector:

- The Flutter widget inspector is a great tool for visualizing and monitoring the Flutter widget trees of your application.
- The Flutter framework uses widgets as the core building block for control anything (such as text, buttons, etc.), to layout (such as centering, padding, row,

2 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot – 360001
Ph.No–(0281)2440478

3 – Vaishalinagar
Nr. Amrapali Under Bridge
Raiya Road
Rajkot - 360001
Ph.No–(0281)2471645

and column. Inspector helps to visualize and explore widget trees, and can be used for the following:

- With the help of the inspector, you can easily understand the existing layouts of your app.
- Diagnosing layout issues in your application if anyone arises.
- *Debugging layout issues visually*
- The following is an instruction to the features available in the inspector's toolbar in flutter's DevTools:

- **Select widget mode:** Select a widget and click on the button on the device to inspect it.

- **Refresh tree:** This button used to reload the current widget info.

- **Slow Animations:** It is used to slow down animations to enable visual inspection.

- **Debug Paint:** Debug Paint button used to add visual debugging, Which hints to the rendering that displays borders, padding, alignment, and spacers, etc.

- **Paint Baselines:** Use of each RenderBox to paint a line at each of its text baselines in the project.

- **Repaint Rainbow:** Repaint Rainbow allows rotating colors on layers when repainting.

- **Debug Mode Banner:** It is used to display the debug banner even when running a debug build.