Name : Kunal Rajesh Kumbhare

Prn : 120A3024

Branch : IT

3nd Year (6th semester)

Experiment No: 8b

AIM: To design a responsive User Interface designing using jQuery Mobile/ Material UI/ Angular UI/ React UI for Ecommerce application.

THEORY:

As an Android developer, there are days spent on just creating XMLs for different screen sizes, because Android devices comes in different shapes and sizes, and it's important that your app looks just as fine in the pool of Android devices. The story isn't that different for iOS developers too, with the varying list of iPhone screen sizes from Apple. And sometimes, we have to support for the tablet or iPad devices as well.

So does that mean, we need to download a variety of simulators or emulators or buy different phones for my team, so we can test our app's UI on different devices?

In Flutter Interact 2019, Zoey Fan and Chris Sells talked about the Flutter Octopus where you could debug your app in multiple numbers of platforms and devices at the same time. But that's great mostly for observing performance of your apps in different devices. Would you really setup all that many devices just for checking the responsiveness of your UI?

Flutter Device Preview by Alois Daniel, is a tool that lets you preview your apps in different sized devices right from your single running emulator/device and platforms, from normal phone size to tablet to even watch heads sizes. Even it's a great way to check how your app looks with or without notches. Not only that, but there are other features like

★ Change the orientation of your app and preview how responsive your app is in different orientations.

★ Updating configurations like text scale factor, theming of your app, locale

★ Ability to take screenshots for sharing it with your team.

And all this, without affecting the state of the application!

Working with Device Preview Flutter Package

Approximate how your app looks and performs on another device. Main features are:

- Preview any device from any device
- Change the device orientation
- Dynamic system configuration (language, dark mode, text scaling factor, ...)
- Freeform device with adjustable resolution and safe areas
- Keep the application state
- Plugin system (Screenshot, File explorer, ...)
- Customizable plugins

Add dependency to your pubspec file : Since Device Preview is a simple Dart package, you have to declare it as any other dependency in your pubspec.yaml file.

```
dependencies:
  device_preview: ^1.0.0
```

Run this command to use this package as a library in Flutter:

```
$ flutter pub add device_preview
```

This will add a line like this to your package's pubspec.yaml (and run an implicit flutter pub get):

```
dependencies:
device_preview: ^1.0.0
```

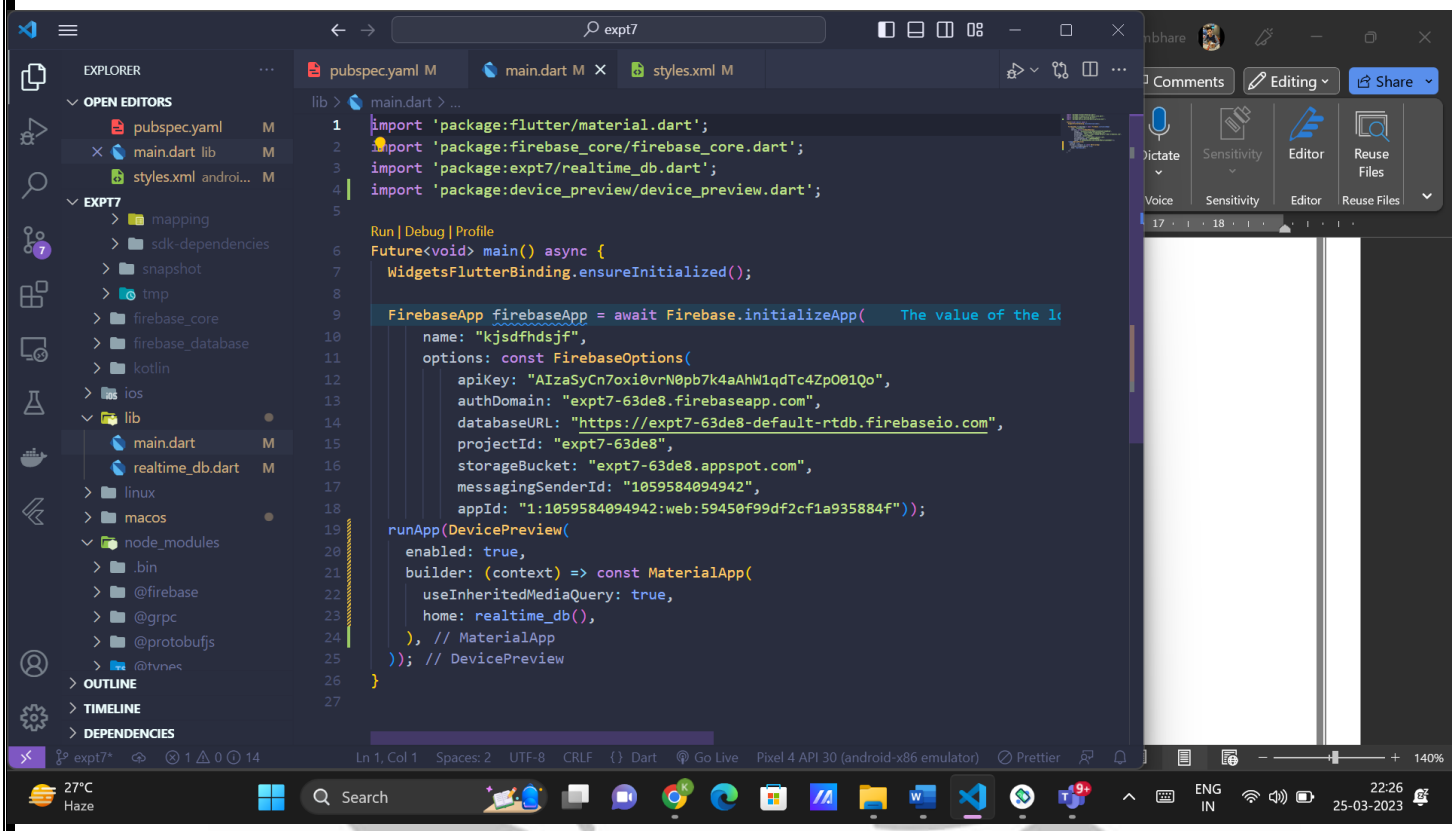
Alternatively, your editor might support flutter pub get. Check the docs for your editor to learn more.
Import it now in your Dart code, you can use:

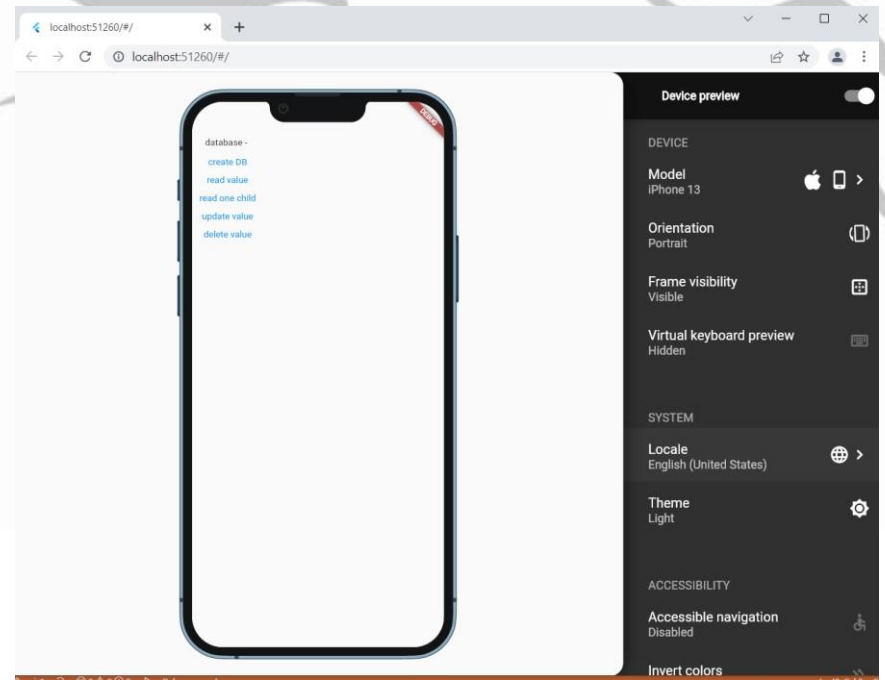
```
import 'package:device_preview/device_preview.dart';
```

pubspec.yaml

```yaml
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.2
  firebase_core: ^2.7.1
  firebase_database: ^10.0.15
  device_preview: ^1.0.0

dev_dependencies:
  flutter_test:
    sdk: flutter
```

main.dart





Using LayoutBuilder class

-Builds a widget tree that can depend on the parent widget's size.

Similar to the Builder widget except that the framework calls the builder function at layout time and provides the parent widget's constraints. This is useful when the parent constrains the child's size and doesn't depend on the child's intrinsic size. The LayoutBuilder's final size will match its child's size.

The builder function is called in the following situations:

- The first time the widget is laid out.
- When the parent widget passes different layout constraints.
- When the parent widget updates this widget.
- When the dependencies that the builder function subscribes to change.

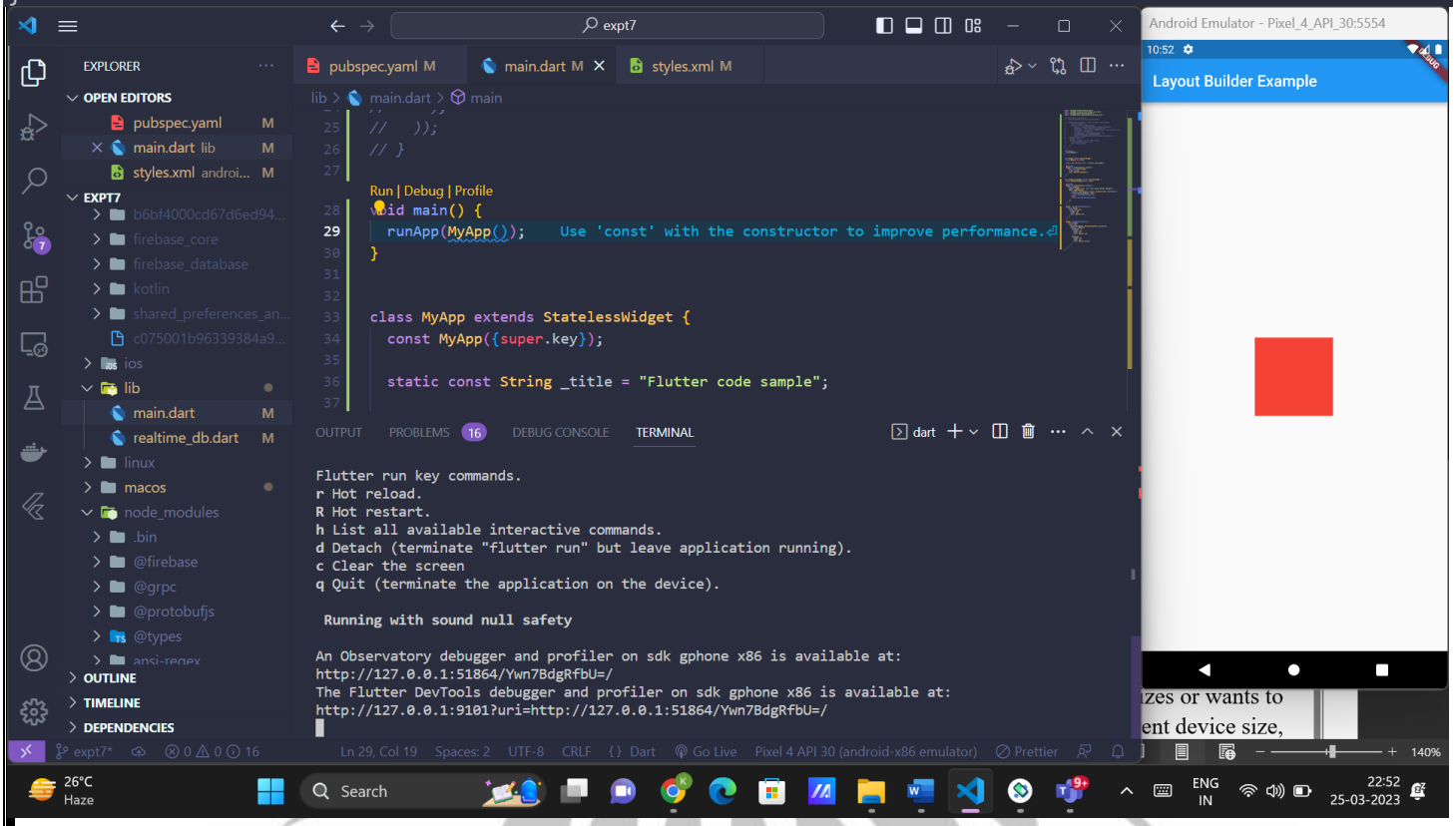
The builder function is *not* called during layout if the parent passes the same constraints repeatedly.

If the child should be smaller than the parent, consider wrapping the child in an Align widget. If the child might want to be bigger, consider wrapping it in a SingleChildScrollView or OverflowBox.

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  static const String _title = "Flutter code sample";

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
```

```
      title: _title,
      home: MyStatelessWidget(),
    );
  }
}

class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("Layout Builder Example")),
      body: LayoutBuilder(
          builder: (BuildContext context, BoxConstraints constraints) {
        if (constraints.maxWidth > 600) {
          return _buildWideContainers();
        } else {
          return _buildNormalContainers();
        }
      }),
    );
  }

  Widget _buildNormalContainers() {
    return Center(
      child: Container(
        height: 100,
        width: 100,
        color: Colors.red,
      ),
    );
  }

  Widget _buildWideContainers() {
    return Center(
      child: Row(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: <Widget>[
          Container(
            height: 100,
            width: 100,
            color: Colors.red,
          ),
          Container(
            height: 100,
```

```
            width: 100,
            color: Colors.yellow,
          )
        ],
      ),
    );
  }
}
```



Flutter – Managing the MediaQuery Object

During the process of developing an app for both phones and tablets, it is standard practice to have different UI layouts for different screen sizes for a better user experience. If the user has a preference set for different font sizes or wants to curtail animations. This is where MediaQuery comes into action, you can get information about the current device size, as well as user preferences, and design your layout accordingly. MediaQuery provides a higher-level view of the current app's screen size and can also give more detailed information about the device and its layout preferences. In practice, MediaQuery is always there. It can simply be accessed by calling MediaQuery.of in the build method.

From there you can look up all sorts of interesting information about the device you're running on, like the size of the screen, and build your layout accordingly. MediaQuery can also be used to check the current device's orientation or can be used to check if the user has modified the default font size. It can also be used to determine if parts of the screen are obscured by a system UI, similar to a safe area widget.

Flutter provides BaseWidget to build UI using custom screen information as per devices to achieve your goal but using MediaQuery widget you will able to manage responsive layouts, fonts as well as all widgets.

To get mediaQueryData just use following command:

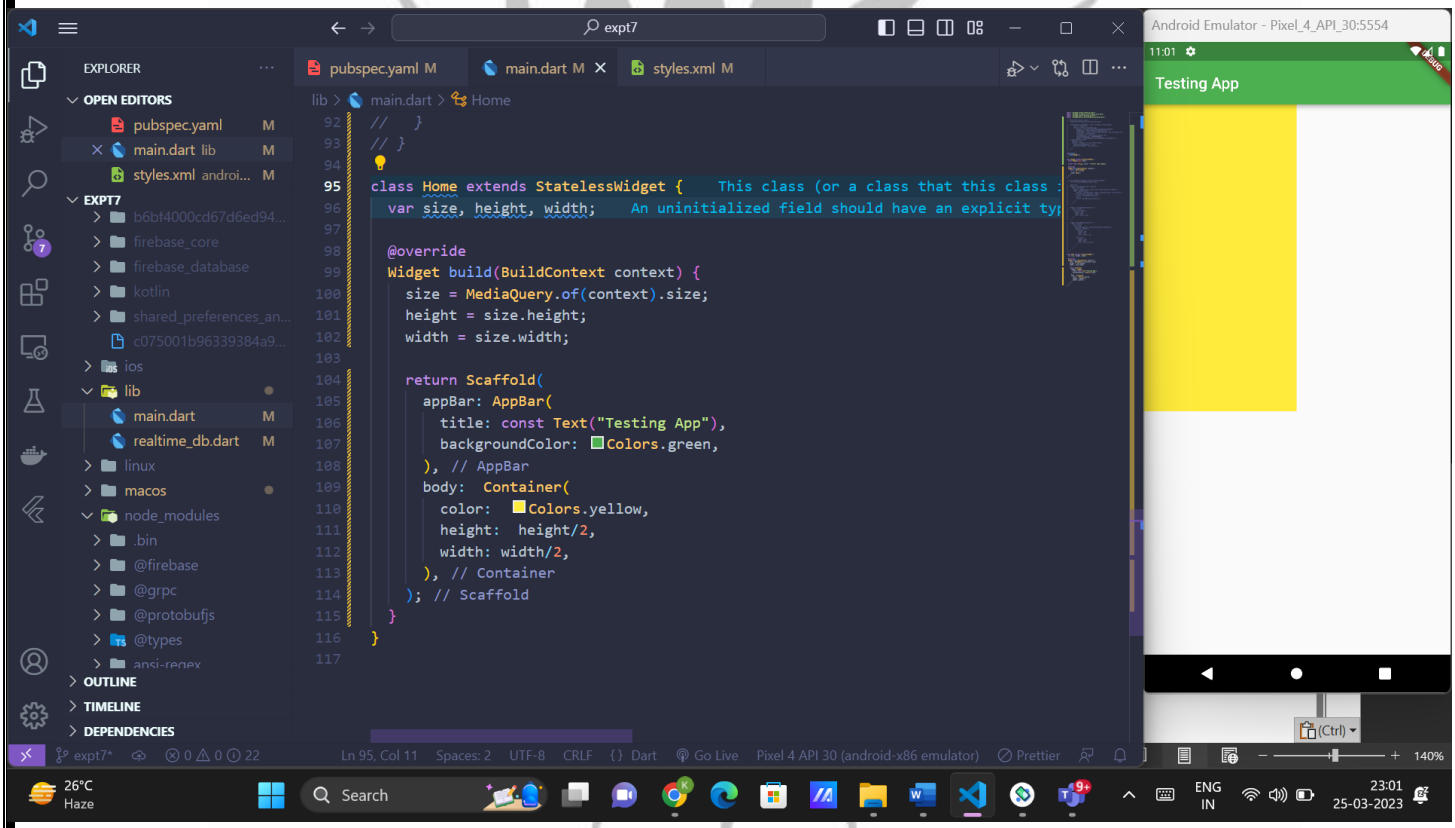
```
MediaQueryData mediaQueryData = MediaQuery.of(context);
```

On the basis of MediaQueryData you can get now device size, orientation, screen height, screen width, blockSize in horizontal, blockSize in vertical as well as safe area in vertical and horizontal both.

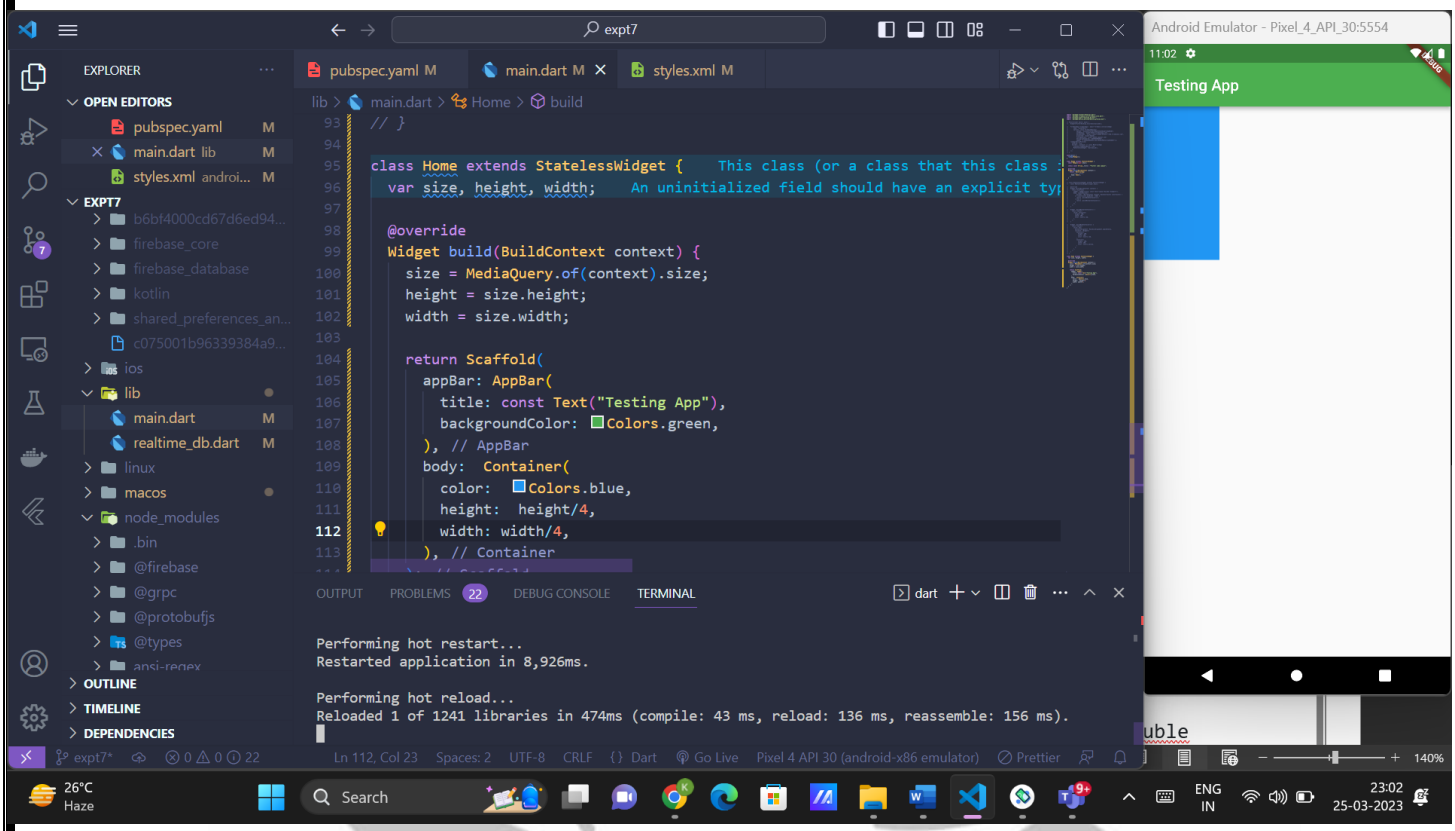
Safe area is like, if you want to create layout after notification bar then you have to get safe area with horizontal as well as if you want to manage device curve display and manage layouts vertical side then use safe area in with vertical.

Example:

Using mediaQuery.of automatically causes the widgets to rebuild themselves according to the current device sizes and layout preferences every time they change.



Example 2: Getting device orientation and rebuilding accordingly.



You can change the orientation by putting these code before the runApp() in your main.dart file:

```
void main() {
    WidgetsFlutterBinding.ensureInitialized();
    SystemChrome.setPreferredOrientations([DeviceOrientation.portraitUp]);
    runApp(MyHomePage());
}
```

Use mediaQuery for manage layout:

Create enum class for fetch screen size.

```
 enum ScreenSize { SMALL, MEDIUM_1, MEDIUM_2, LARGE
}
```

On the basis of device height decide currently app is run on which type of device.
First, apply condition for screen size like,

```
ScreenSize fetchScreenSizeInHeight(double height){
if(height <600){
   return ScreenSize.SMALL;
 }else if(height <700 && height > 600){
return ScreenSize.MEDIUM_1;
 }else if(height < 800 && height > 700){
return ScreenSize.MEDIUM_2;
 }
else{
   return ScreenSize.LARGE;
```

```
 }
}
```

On the bases of screen size fetch screen size in dart file where you want to implement responsive layout.

```
 double screenHeight =
MediaQuery.of(context).size.height;


ScreennSize screenSize = fetchScreenSizeInHeight(screenHeight);
```

Now, create function for manage layout or any widget size like,

```
double setWidgetHeight(ScreenSize screenSize, BuildContext context, double small,double
medium1,double medium2,double large){   switch(screenSize){     case ScreenSize.SMALL:
return  MediaQuery.of(context).size.height * small;
      break;     case ScreenSize.MEDIUM_1:       return
MediaQuery.of(context).size.height * medium1;       break;
case ScreenSize.MEDIUM_2:       return
MediaQuery.of(context).size.height * medium2;       break;
case ScreenSize.LARGE:       return
MediaQuery.of(context).size.height * large;
      break;
  } }
```

Same ways you can create function for manage different size width as well as manage font height for different devices. Use this function in your stateful/stateless class like below,

```
 class MyHomePage extends StatefulWidget {

 @override
 State<StatefulWidget> createState() {
return _MyHomePageState();
 }
}
class _MyHomePageState extends State<MyHomePage> {

 @override
 Widget build(BuildContext context) {
return Scaffold(       body: Center(
child:  Container(
            height: fetchSpaceSize(screenSize,context,0.35,0.3,0.4,0.3),
alignment: Alignment.center,
                          ),
           ],
         ),
     ),
   );
}
```

Conclusion: - Hence we have successfully designed a responsive User Interface designing using Material UI.