**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

# Experiment 1

**AIM:** To install and configure Flutter Environment.

## THEORY:

**Flutter Environment**

Flutter – a simple and high-performance framework based on Dart language, provides high performance by rendering the UI directly in the operating system’s canvas rather than through native framework.

Flutter also offers many ready to use widgets (UI) to create a modern application. These widgets are optimized for mobile environment and designing the application using widgets is as simple as designing HTML.

To be specific, Flutter application is itself a widget. Flutter widgets also supports animations and gestures. The application logic is based on reactive programming. Widget may optionally have a state. By changing the state of the widget, Flutter will automatically (reactive programming) compare the widget’s state (old and new) and render the widget with only the necessary changes instead of re-rendering the whole widget.

## Features of Flutter

Flutter framework offers the following features to developers −

* Modern and reactive framework.
* Uses Dart programming language and it is very easy to learn.
* Fast development.

## Advantages of Flutter

Flutter comes with beautiful and customizable widgets for high performance and outstanding mobile application. It fulfills all the custom needs and requirements. Besides these, Flutter offers many more advantages as mentioned below −

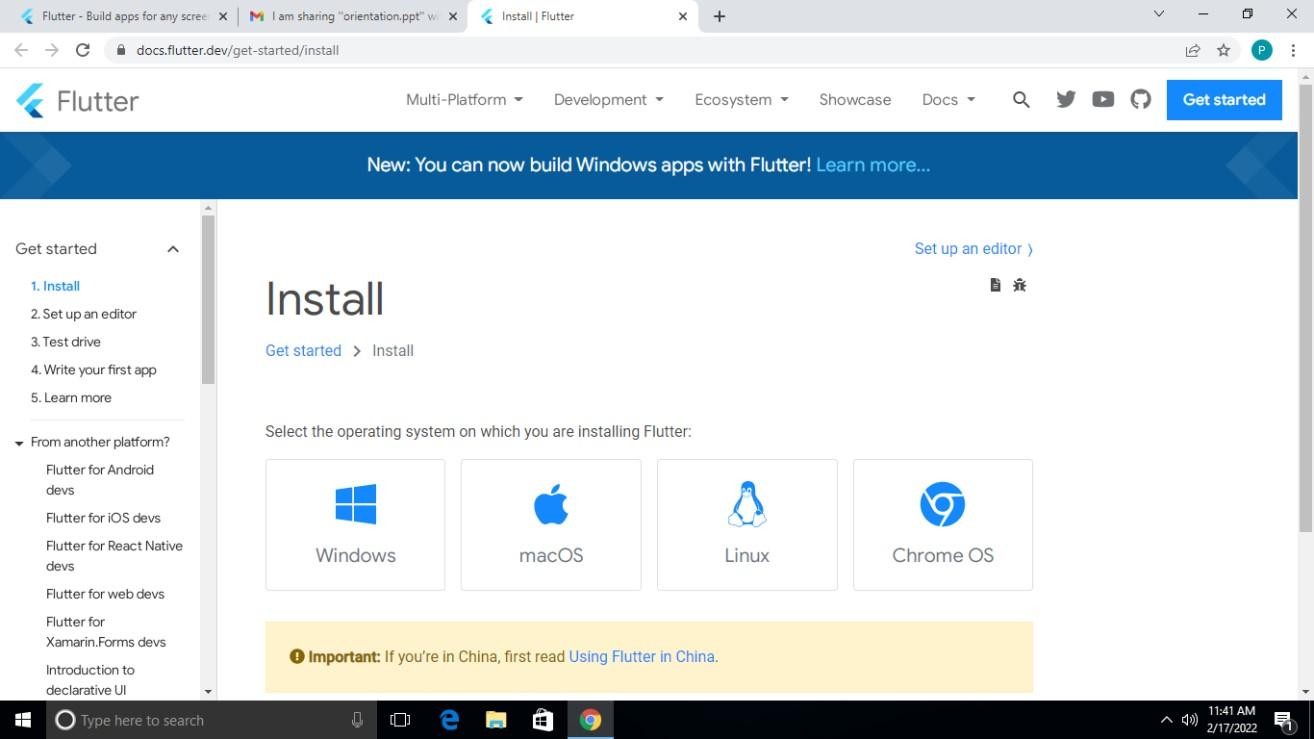
* Dart has a large repository of software packages which lets you to extend the capabilities of your application.
* Developers need to write just a single code base for both applications (both Android and iOS platforms). *Flutter* may to be extended to other platform as well in the future.

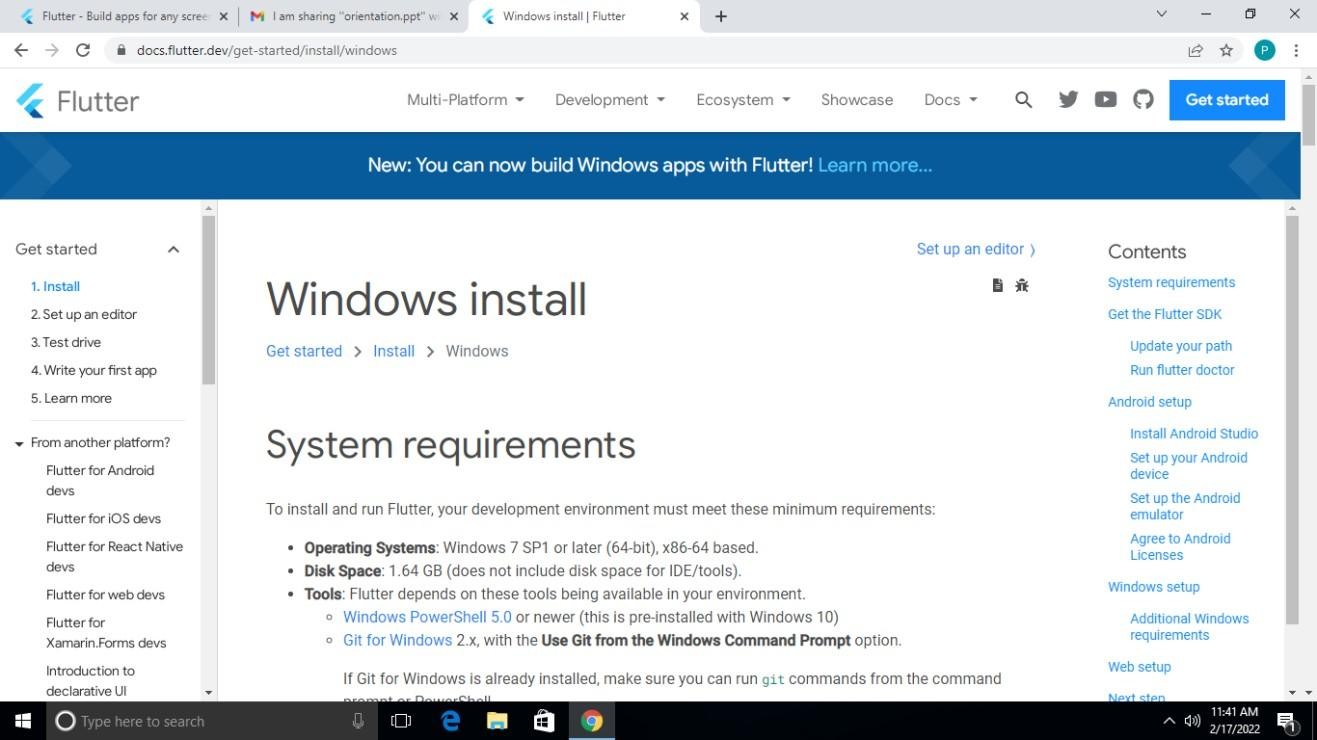
## Disadvantages of Flutter

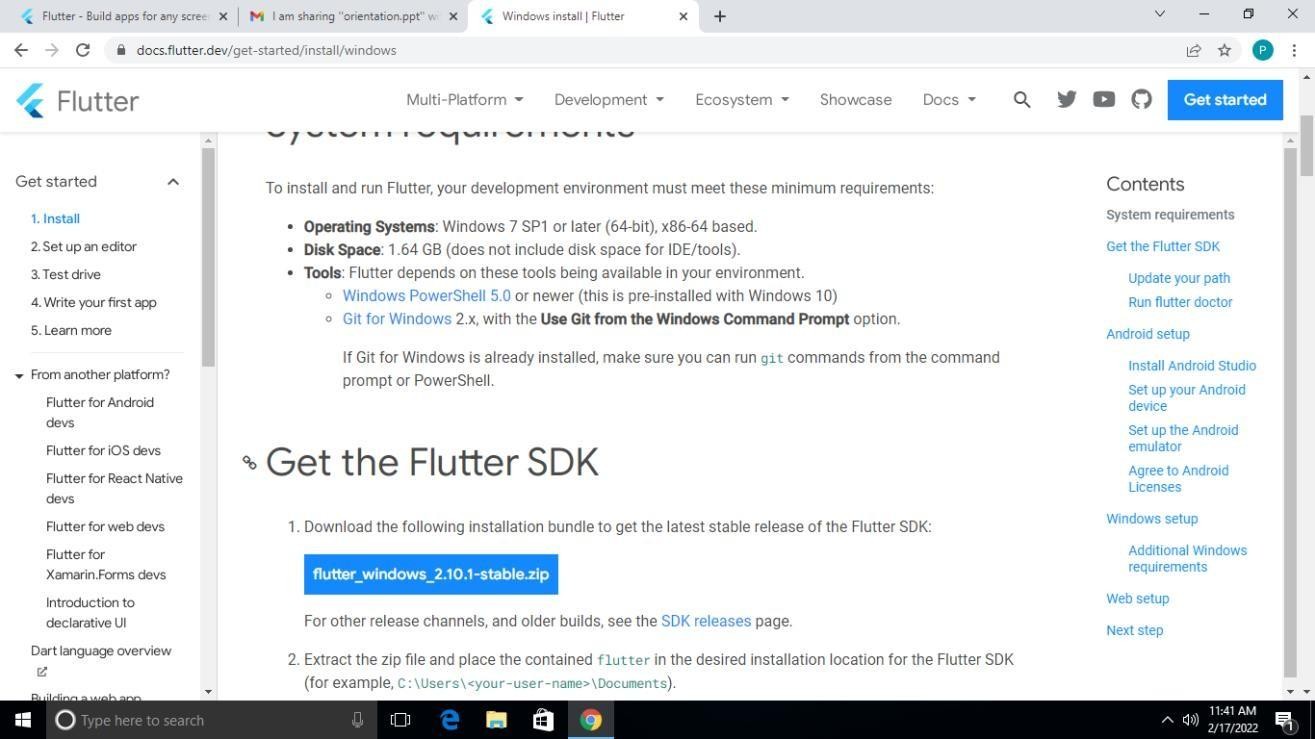
Despite its many advantages, flutter has the following drawbacks in it −

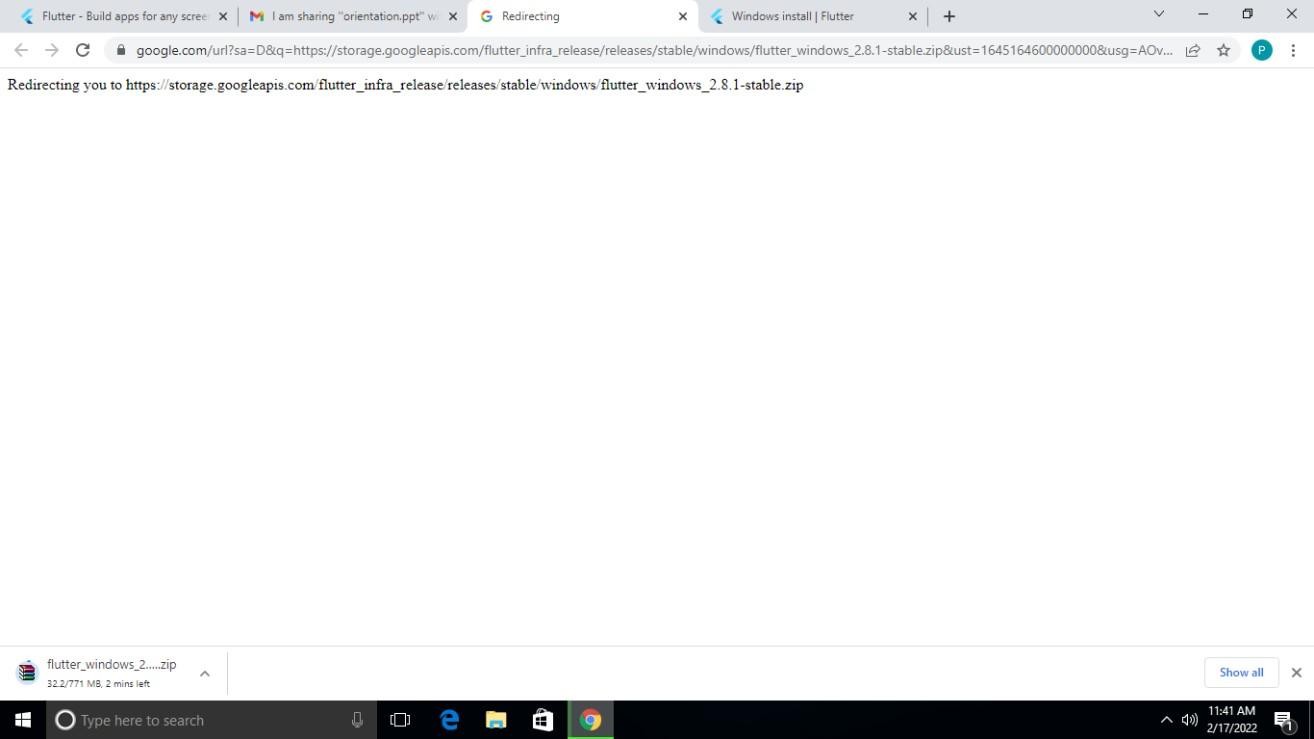
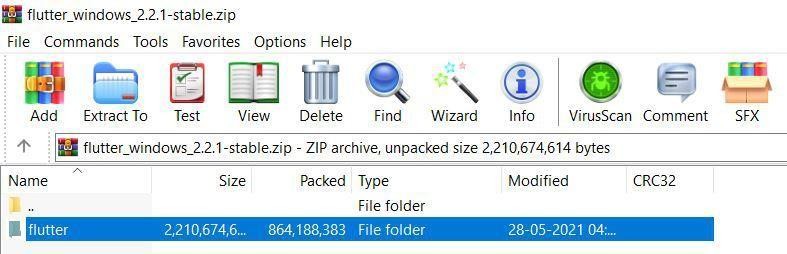
* Since it is coded in Dart language, a developer needs to learn new language (though it is easy to learn).
* Modern framework tries to separate logic and UI as much as possible but, in Flutter, user interface and logic is intermixed. We can overcome this using smart coding and using high level module to separate user interface and logic.

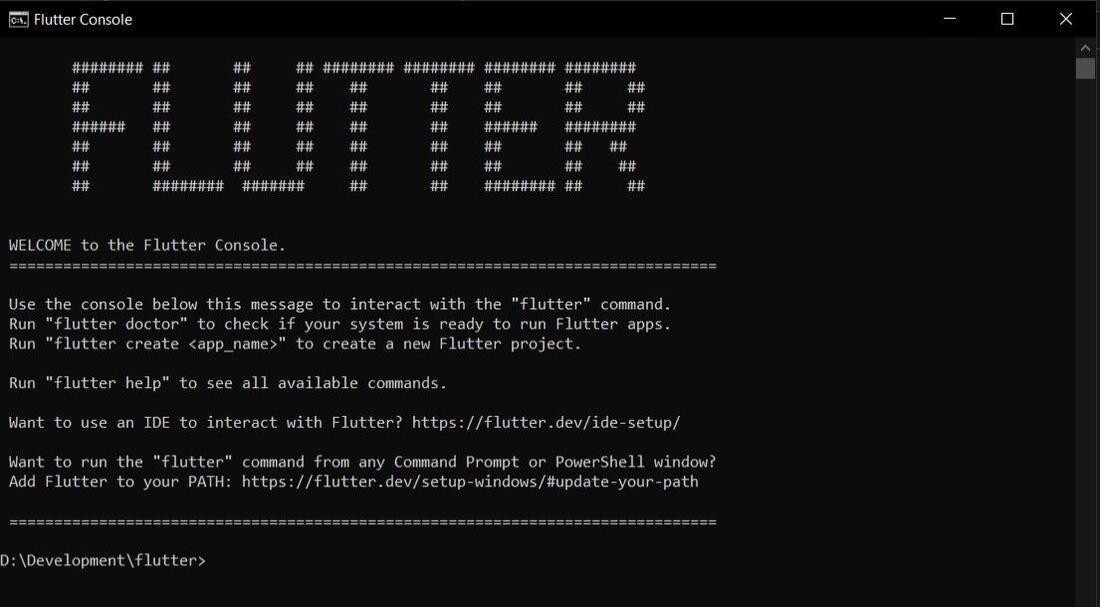
**Flutter Installation:**



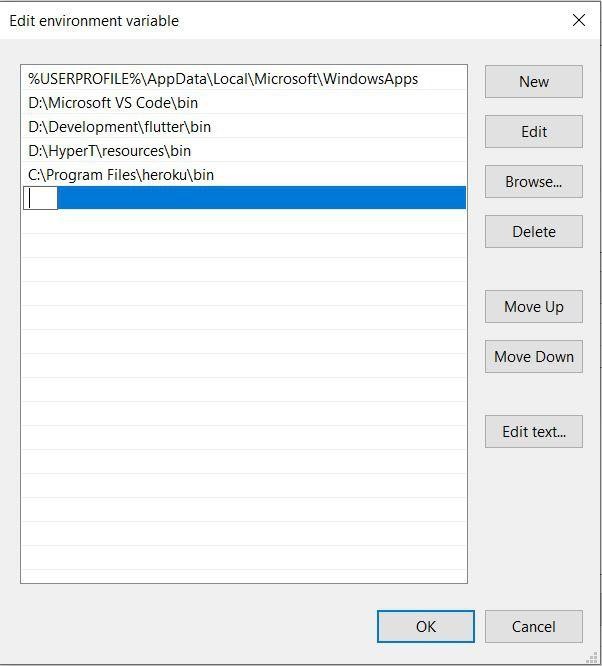
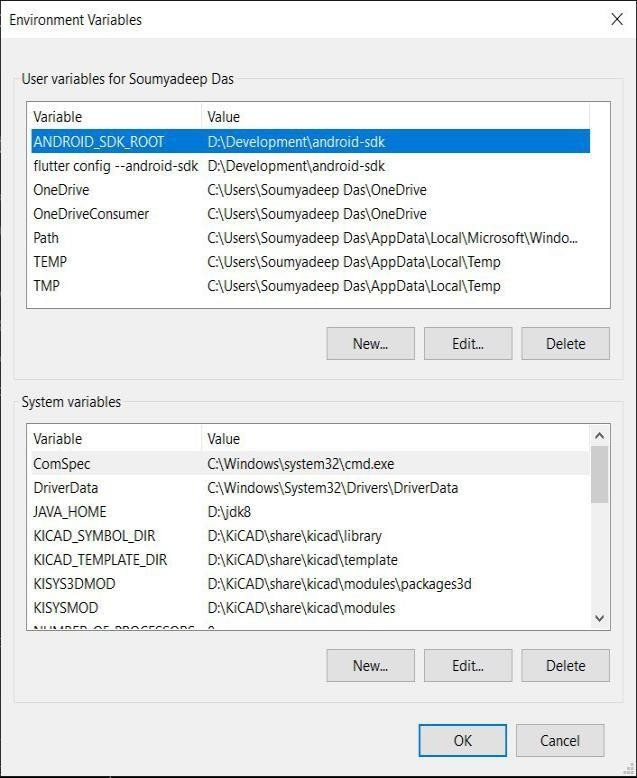


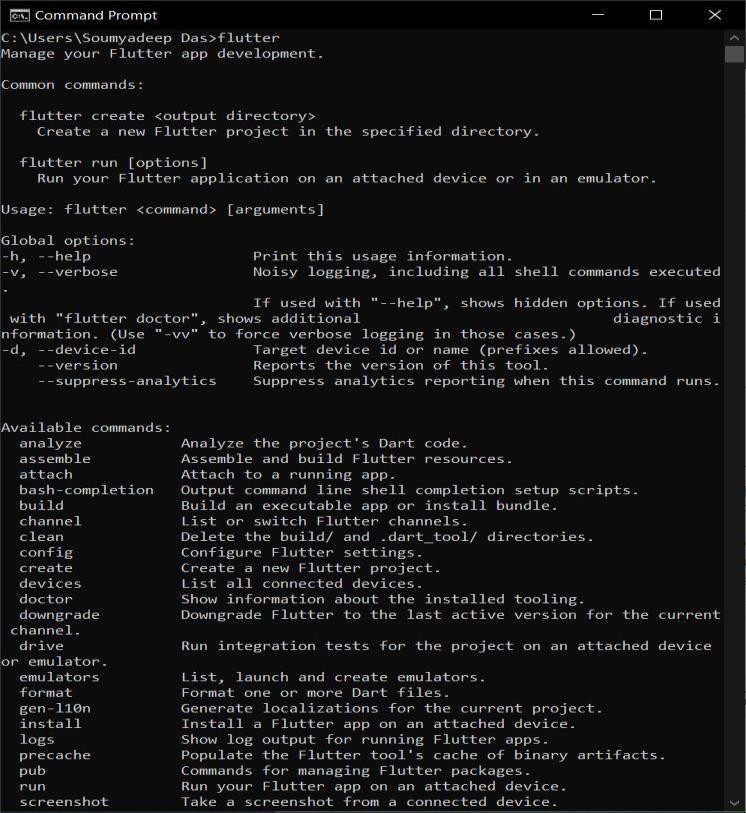












**CONCLUSION:** Hence we have successfully studied the installation and configuration of Flutter Environment.

**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

# Experiment 2

**AIM:** To design Flutter UI by including common widgets.

## THEORY:

**Flutter Widget**

Flutter widgets are built using a modern framework that takes inspiration from [React](https://reactjs.org/). The central idea is that you build your UI out of widgets. Widgets describe what their view should look like given their current configuration and state. When a widget’s state changes, the widget rebuilds its description, which the framework diffs against the previous description in order to determine the minimal changes needed in the underlying render tree to transition from one state to the next.

## Basic widgets

Flutter comes with a suite of powerful basic widgets, of which the following are commonly used:

## [Text](https://api.flutter.dev/flutter/widgets/Text-class.html)

The Text widget lets you create a run of styled text within your application.

## [Row](https://api.flutter.dev/flutter/widgets/Row-class.html), [Column](https://api.flutter.dev/flutter/widgets/Column-class.html)

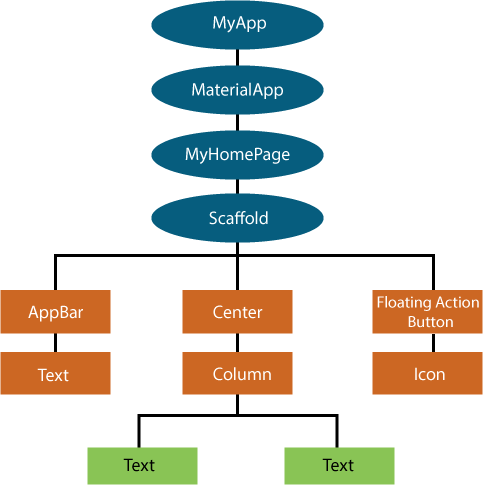
These flex widgets let you create flexible layouts in both the horizontal (Row) and vertical (Column) directions. The design of these objects is based on the web’s flexbox layout model.

## [Stack](https://api.flutter.dev/flutter/widgets/Stack-class.html)

Instead of being linearly oriented (either horizontally or vertically), a Stack widget lets you place widgets on top of each other in paint order. You can then use the [Positioned](https://api.flutter.dev/flutter/widgets/Positioned-class.html) widget on children of a Stack to position them relative to the top, right, bottom, or left edge of the stack. Stacks are based on the web’s absolute positioning layout model.

## [Container](https://api.flutter.dev/flutter/widgets/Container-class.html)

The Container widget lets you create a rectangular visual element. A container can be decorated with a [BoxDecoration](https://api.flutter.dev/flutter/painting/BoxDecoration-class.html), such as a background, a border, or a shadow. A Container can also have margins, padding, and constraints applied to its size. In addition, a Container can be transformed in three-dimensional space using a matrix.



Many Material Design widgets need to be inside of a [MaterialApp](https://api.flutter.dev/flutter/material/MaterialApp-class.html) to display properly, in order to inherit theme data. Therefore, run the application with a MaterialApp.

The MyAppBar widget creates a [Container](https://api.flutter.dev/flutter/widgets/Container-class.html) with a height of 56 device- independent pixels with an internal padding of 8 pixels, both on the left and the right. Inside the container, MyAppBar uses a [Row](https://api.flutter.dev/flutter/widgets/Row-class.html) layout to organize its children. The middle child, the title widget, is marked as [Expanded](https://api.flutter.dev/flutter/widgets/Expanded-class.html), which means it expands to fill any remaining available space that hasn’t been consumed by the other children. You can have multiple Expanded children and determine the ratio in which they consume the available space using the [flex](https://api.flutter.dev/flutter/widgets/Expanded-class.html#flex) argument to Expanded.

The MyScaffold widget organizes its children in a vertical column. At the top of the column, it places an instance of MyAppBar, passing the app bar a [Text](https://api.flutter.dev/flutter/widgets/Text-class.html) widget to use as its title. Passing widgets as arguments to other widgets is a powerful technique that lets you create generic widgets that can be reused in a wide variety of ways. Finally, MyScaffold uses an [Expanded](https://api.flutter.dev/flutter/widgets/Expanded-class.html) to fill the remaining space with its body, which consists of a centered message.

## CODE:

1. **Scaffold -**

import 'package:flutter/material.dart'; class MyAppBar extends StatelessWidget {

const MyAppBar({required this.title, Key? key}) : super(key: key);

// Fields in a Widget subclass are always marked "final". final Widget title;

@override

Widget build(BuildContext context) { return Container(

height: 56.0, // in logical pixels

padding: const EdgeInsets.symmetric(horizontal: 8.0), decoration: BoxDecoration(color: Colors.blue[500]),

// Row is a horizontal, linear layout. child: Row(

// <Widget> is the type of items in the list. children: [

const IconButton(

icon: Icon(Icons.menu), tooltip: 'Navigation menu',

onPressed: null, // null disables the button

),

// Expanded expands its child

// to fill the available space. Expanded(

child: title,

),

const IconButton(

icon: Icon(Icons.search), tooltip: 'Search', onPressed: null,

),

],

),

);

}

}

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

@override

Widget build(BuildContext context) {

// Material is a conceptual piece

// of paper on which the UI appears. return Material(

// Column is a vertical, linear layout. child: Column(

children: [ MyAppBar( title: Text(

'Example title',

style: Theme.of(context) //

.primaryTextTheme

.headline6,

),

),

const Expanded( child: Center(

child: Text('Hello, world!'),

),

),

],

),

);

}

}

void main() { runApp(

const MaterialApp(

title: 'My app', // used by the OS task switcher home: SafeArea(

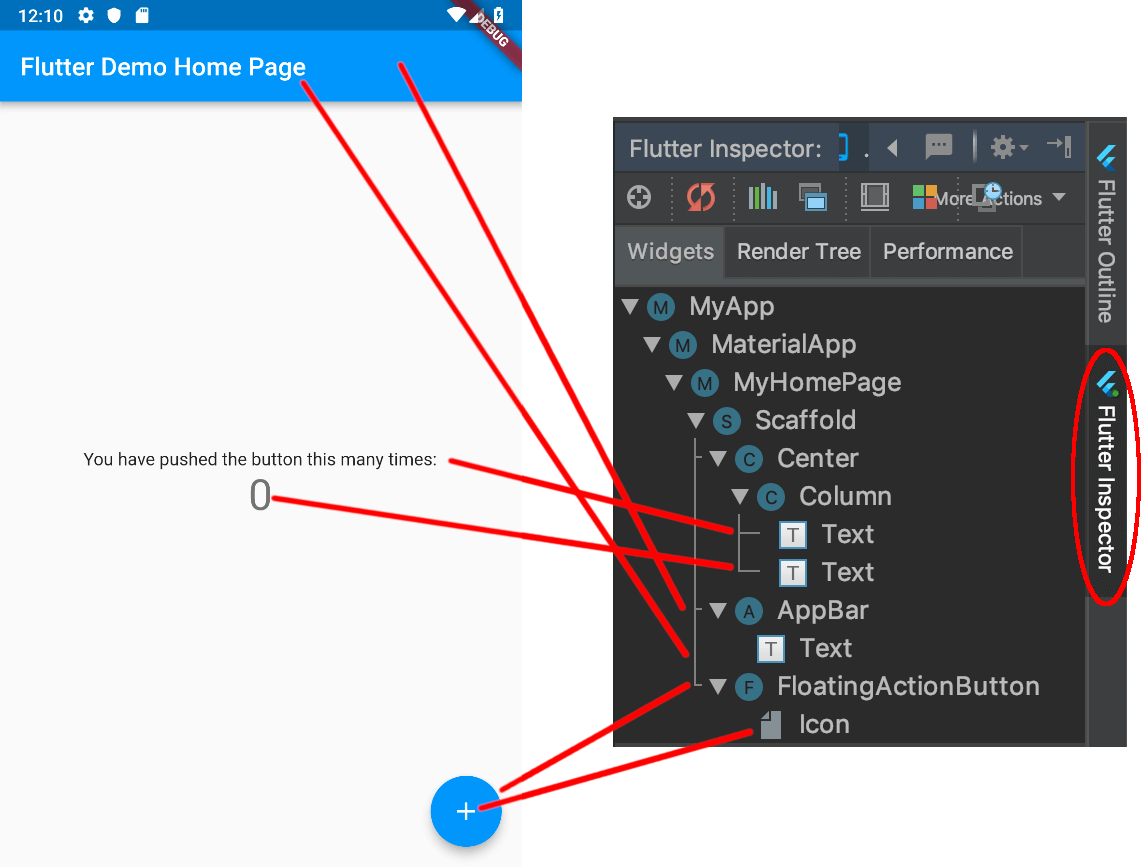
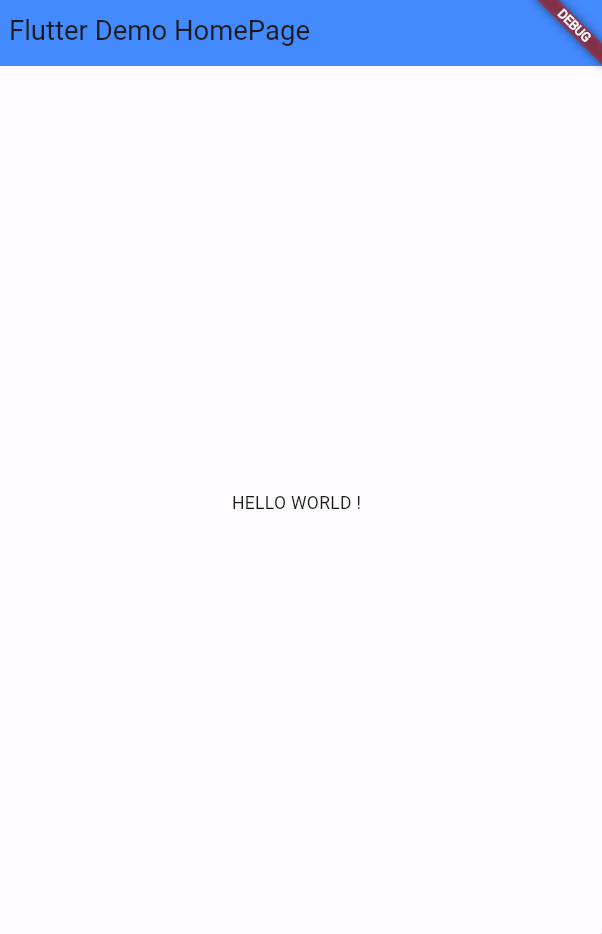
child: MyScaffold(),

),

),

);

## } OUTPUT:



1. **Text Widget -**

import 'package:flutter/material.dart'; void main() { runApp(MyApp()); }

class MyApp extends StatelessWidget { @override

Widget build(BuildContext context) { return MaterialApp(

home: MyHomePage()

);

}

}

class MyHomePage extends StatefulWidget { @override

\_MyHomePageState createState() => \_MyHomePageState();

}

// Text Widget

class \_MyHomePageState extends State<MyHomePage> { @override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: Text("Flutter Text Widget"),

),

body: Center(

child: Text("Hello World!", style: TextStyle( fontSize: 40,

color: Colors.blue, fontWeight: FontWeight.bold,

decoration: TextDecoration.combine([ TextDecoration.underline,

// TextDecoration.overline,

]),

decorationThickness: 2.0, decorationColor:

Colors.teal,

decorationStyle: TextDecorationStyle.wavy, letterSpacing: 1.0,

wordSpacing: 5.0, shadows: [ Shadow(

color: Colors.black, blurRadius: 2.0,

offset: Offset(4, 1)

)

],

fontFamily: "Times New Roman",

),),

)

);

}

}

## OUTPUT:



1. **Container Widget -**

class \_MyHomePageState extends State<MyHomePage> {

@override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: Text("Flutter Container Widget"),

),

body: Center( child:Container(

padding: EdgeInsets.all(8.0),

//constraints: BoxConstraints.expand(), alignment: Alignment.center,

width: 200,

height: 200,

// transform: Matrix4.rotationZ(0.5), decoration: BoxDecoration(

color:Colors.blue,

// border: Border.all(

// color: Colors.black,

// width: 2.0,

// style: BorderStyle.solid

// ),

// borderRadius: BorderRadius.all(Radius.circular(20.0)), boxShadow:[

BoxShadow(

color: Colors.grey, blurRadius: 4.0,

spreadRadius: 2.0,

)

],

gradient: LinearGradient( begin: Alignment.topCenter, end:Alignment.centerRight, colors: [

Colors.cyan, Colors.blue,

]),

shape: BoxShape.rectangle

),

child: Text("Flutter Container",style: TextStyle(fontSize: 20,color:Colors.black),))),

);

}

}

## OUTPUT:



1. **Stack and Positioned Widget -**

class \_MyHomePageState extends State<MyHomePage> {

@override

Widget build(BuildContext context) { return Scaffold(

// appBar: AppBar(

// title: Text("Flutter Stack & Positioned Widget"),

// ),

body: Center( child: Container( width: 400,

height: 700,

color: Colors.grey, child: Stack(

alignment: Alignment.topRight,

// fit: StackFit.expand, overflow: Overflow.visible, children: <Widget>[ Container(

width: 200,

height: 200, color: Colors.red,

),

Positioned( bottom: -20,

right: 0,

child: Container(

width: 100,

height: 100, color: Colors.blue,

),

),

],

),

)

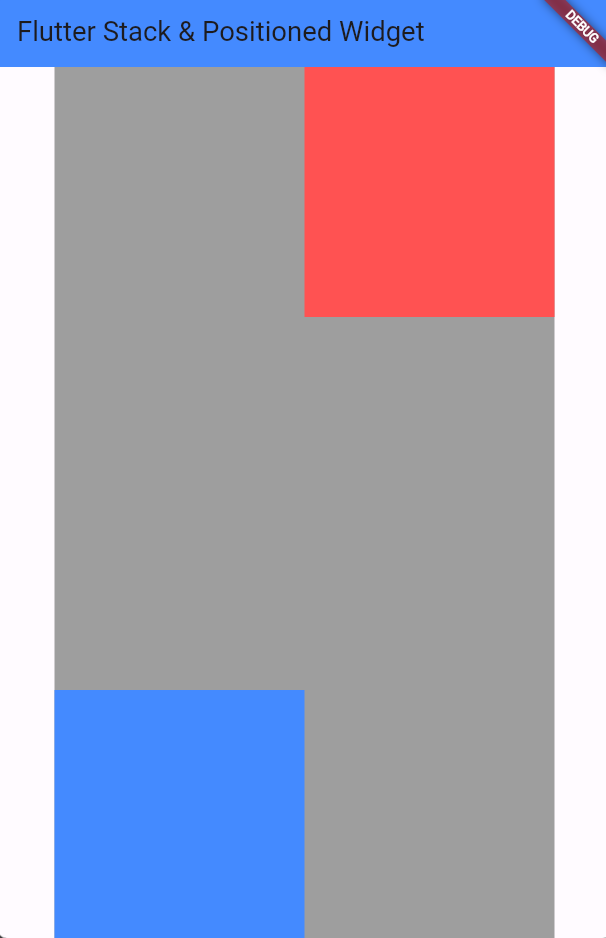
),

);

}

}

## OUTPUT:



1. **Row and Column Widget -**

class \_MyHomePageState extends State<MyHomePage> { @override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: Text("Flutter Column Example"),

),

body: Row(// For Column widget, replace Row with Column mainAxisAlignment: MainAxisAlignment.spaceEvenly, children:<Widget>[

Container(

margin: EdgeInsets.all(12.0), padding: EdgeInsets.all(8.0), decoration:BoxDecoration(

borderRadius:BorderRadius.circular(8), color:Colors.green

),

child: Text("React.js",style: TextStyle(color:Colors.yellowAccent,fontSize:25),),

),

Container(

margin: EdgeInsets.all(15.0), padding: EdgeInsets.all(8.0), decoration:BoxDecoration(

borderRadius:BorderRadius.circular(8), color:Colors.green

),

child: Text("Flutter",style: TextStyle(color:Colors.yellowAccent,fontSize:25),),

),

Container(

margin: EdgeInsets.all(12.0), padding: EdgeInsets.all(8.0), decoration:BoxDecoration(

borderRadius:BorderRadius.circular(8), color:Colors.green

),

child: Text("MySQL",style: TextStyle(color:Colors.yellowAccent,fontSize:25),),

)

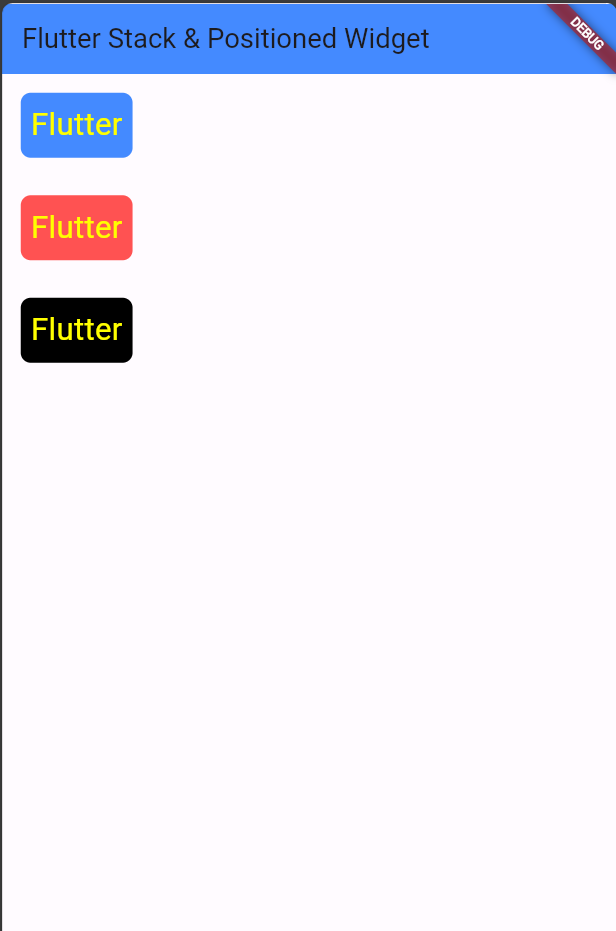
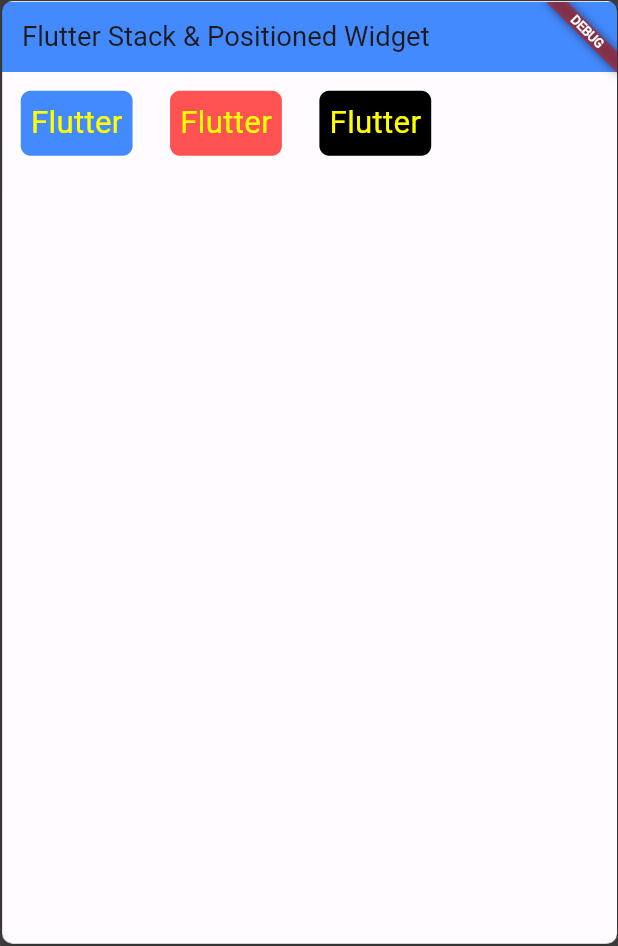
]

),

);

}}

## OUTPUT:



## Button Widget -

class \_MyHomePageState extends State<MyHomePage> { String msg = 'Flutter RaisedButton Example';

@override

Widget build(BuildContext context) { return MaterialApp(

home: Scaffold( appBar: AppBar(

title: Text('Flutter RaisedButton Example'),

),

body: Container( child: Center( child: Column(

mainAxisAlignment: MainAxisAlignment.center, children: [

Text(msg, style: TextStyle(fontSize: 25, fontStyle: FontStyle.italic),), RaisedButton(

child: Text("Click Here", style: TextStyle(fontSize: 15),), onPressed: \_changeText,

color: Colors.blue, textColor: Colors.white, padding: EdgeInsets.all(8.0), splashColor: Colors.grey,

)

],

),

),

),

),

);

}

\_changeText() { setState(() {

if (msg.startsWith('F')) {

msg = 'We have learned FlutterRaised button example.';

} else {

msg = 'Flutter RaisedButton Example';

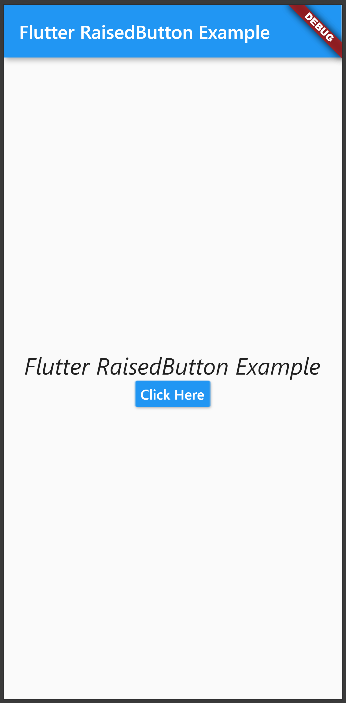
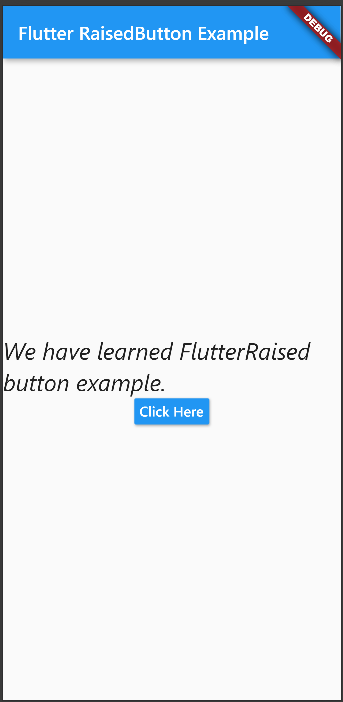
}

});

}

}

## OUTPUT:

**CONCLUSION:** Hence, we have design Flutter UI by including common widgets.

**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

# Experiment 3

**AIM:** To Create an Interactive Form using Form Widget.

## THEORY:

**Flutter Forms**

Forms are an integral part of all modern mobile and web applications. It is mainly used to interact with the app as well as gather information from the users. They can perform many tasks, which depend on the nature of your business requirements and logic, such as authentication of the user, adding user, searching, filtering, ordering, booking, etc. A form can contain text fields, buttons, checkboxes, radio buttons, etc.

## Creating Form

Flutter provides a Form widget to create a form. The form widget acts as a container, which allows us to group and validate the multiple form fields. When you create a form, it is necessary to provide the GlobalKey. This key uniquely identifies the form and allows you to do any validation in the form fields.

The form widget uses child widget TextFormField to provide the users to enter the text field. This widget renders a material design text field and also allows us to display validation errors when they occur.

First, create a Flutter project and replace the following code in the main.dart file. In this code snippet, we have created a custom class named MyCustomForm. Inside this class, we define a global key as \_formKey**.** This key holds a FormState and can use to retrieve the form widget. Inside the build method of this class, we have added some custom style and use the TextFormField widget to provide the form fields such as name, phone number, date of birth, or just a normal field. Inside the TextFormField, we have used InputDecoration that provides the look and feel of your form properties such as borders, labels, icons, hint, styles, etc. Finally, we have added a button to submit the form

## Form validation

Validation is a method, which allows us to correct or confirms a certain standard. It ensures the authentication of the entered data.

Validating forms is a common practice in all digital interactions. To validate a form in a flutter, we need to implement mainly three steps.

**Step 1:** Use the Form widget with a global key.

**Step 2:** Use TextFormField to give the input field with validator property.

**Step 3:** Create a button to validate form fields and display validation errors.

We have to use validator() function in the TextFormField to validate the input properties. If the user gives the wrong input, the validator function returns a string that contains an error message; otherwise, the validator function return null. In the validator function, make sure that the TextFormField is not empty. Otherwise, it returns an error message.

## CODE:

import 'package:flutter/material.dart'; void main() => runApp(MyApp());

class MyApp extends StatelessWidget { @override

Widget build(BuildContext context) { final appTitle = 'Flutter Form Demo'; return MaterialApp(

title: appTitle, home: Scaffold( appBar: AppBar(

title: Text(appTitle),

),

body: MyCustomForm(),

),

);

}

}

// Create a Form widget.

class MyCustomForm extends StatefulWidget { @override

MyCustomFormState createState() { return MyCustomFormState();

}

}

// Create a corresponding State class. This class holds data related to the form. class MyCustomFormState extends State<MyCustomForm> {

// Create a global key that uniquely identifies the Form widget

// and allows validation of the form.

final \_formKey = GlobalKey<FormState>();

@override

Widget build(BuildContext context) {

// Build a Form widget using the \_formKey created above. return Form(

key: \_formKey, child: Column(

crossAxisAlignment: CrossAxisAlignment.start, children: <Widget>[

TextFormField(

decoration: const InputDecoration(

icon: const Icon(Icons.person), hintText: 'Enter your name', labelText: 'Name',

),

),

TextFormField(

decoration: const InputDecoration( icon: const Icon(Icons.phone), hintText: 'Enter a phone number', labelText: 'Phone',

),

), TextFormField(

decoration: const InputDecoration( icon: const Icon(Icons.calendar\_today), hintText: 'Enter your date of birth', labelText: 'Dob',

),

),

new Container(

padding: const EdgeInsets.only(left: 150.0, top: 40.0), child: new RaisedButton(

child: const Text('Submit'), onPressed: null,

)),

],

),

);

}

validator: (value) {

if (value.isEmpty) {

return 'Please enter some text';

}

return null;

},

import 'package:flutter/material.dart'; void main() => runApp(MyApp());

class MyApp extends StatelessWidget { @override

Widget build(BuildContext context) { final appTitle = 'Flutter Form Demo'; return MaterialApp(

title: appTitle, home: Scaffold( appBar: AppBar(

title: Text(appTitle),

),

body: MyCustomForm(),

),

);

}

}

// Create a Form widget.

class MyCustomForm extends StatefulWidget { @override

MyCustomFormState createState() { return MyCustomFormState();

}}

// Create a corresponding State class, which holds data related to the form.

class MyCustomFormState extends State<MyCustomForm> {

// Create a global key that uniquely identifies the Form widget

// and allows validation of the form.

final \_formKey = GlobalKey<FormState>();

@override

Widget build(BuildContext context) {

// Build a Form widget using the \_formKey created above. return Form(

key: \_formKey, child: Column(

crossAxisAlignment: CrossAxisAlignment.start, children: <Widget>[

TextFormField(

decoration: const InputDecoration( icon: const Icon(Icons.person), hintText: 'Enter your full name', labelText: 'Name',

),

validator: (value) { if (value.isEmpty) {

return 'Please enter some text';

}return null;

},

),

TextFormField(

decoration: const InputDecoration( icon: const Icon(Icons.phone), hintText: 'Enter a phone number', labelText: 'Phone',

),

validator: (value) { if (value.isEmpty) {

return 'Please enter valid phone number';

}return null;

},

),

TextFormField(

decoration: const InputDecoration( icon: const Icon(Icons.calendar\_today),

hintText: 'Enter your date of birth', labelText: 'Dob',

),

validator: (value) { if (value.isEmpty) {

return 'Please enter valid date';

}

return null;

},

),

new Container(

padding: const EdgeInsets.only(left: 150.0, top: 40.0), child: new RaisedButton(

child: const Text('Submit'), onPressed: () {

// It returns true if the form is valid, otherwise returns false if (\_formKey.currentState.validate()) {

// If the form is valid, display a Snackbar. Scaffold.of(context)

.showSnackBar(SnackBar(content: Text('Data is in processing.')));

}

},

)),

],

),

);} }

## Flutter FormsFlutter FormsOUTPUT:

**CONCLUSION:** Hence we have Successfully Studied to Create Form using Form Widget.

**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

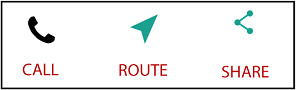
# Experiment 4

**AIM:** To design a layout of Flutter App using layout widgets.

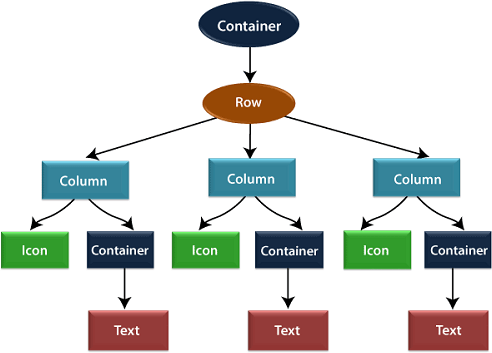
## THEORY:

**Flutter Layouts**

The main concept of the layout mechanism is the widget. We know that flutter assume everything as a widget. So, the image, icon, text, and even the layout of your app are all widgets. Here, some of the things you do not see on your app UI, such as rows, columns, and grids that arrange, constrain, and align the visible widgets are also the widgets.

Flutter allows us to create a layout by composing multiple widgets to build more complex widgets. **For example**, we can see the below image that shows three icons with a label under each one.

In the second image, we can see the visual layout of the above image. This image shows a row of three columns, and these columns contain an icon and label.



In the above image, the **container** is a widget class that allows us to customize the child widget. It is mainly used to add borders, padding, margins,

background color, and many more. Here, the text widget comes under the container for adding margins. The entire row is also placed in a container for adding margin and padding around the row. Also, the rest of the UI is controlled by properties such as color, text. Style, etc.

Layout a widget

Let us learn how we can create and display a simple widget. The following steps show how to layout a widget:

**Step 1:** First, you need to select a layout widget.

**Step 2:** Next, create a visible widget.

**Step 3:** Then, add the visible widget to the layout widget.

**Step 4:** Finally, add the layout widget to the page where you want to display. Types of Layout Widgets

We can categories the layout widget into two types:

* 1. Single Child Widget
  2. Multiple Child Widget

## List of some single child layout widgets.

* Align
* Container
* SizedBox
* Aspect Ratio
* Baseline
* ConstrainedBox

## Align

Using align class we can control the alignment of child widget. It allow us to place child widget at position you want.

With Align Class you will have more control over the exact position of the child widget. It allows place the child widget in the exact place you need.

A widget that aligns its child within itself and optionally sizes itself based on the child’s size.

Align Widget can change its size according to the size of its child.

## CODE:

import 'package:flutter/material.dart';

**void** main() { runApp(MyApp());

}

**class** MyApp **extends** StatefulWidget { @override

\_MyAppState createState() => \_MyAppState();

}

**class** \_MyAppState **extends** State<MyApp> { @override

Widget build(BuildContext context) {

**return** Container( height: 200, width:500,

color: Colors.grey,

**child**: Align(

alignment: Alignment.center,

**child**: Text(

"Text centre alignment",textDirection: TextDirection.ltr

),

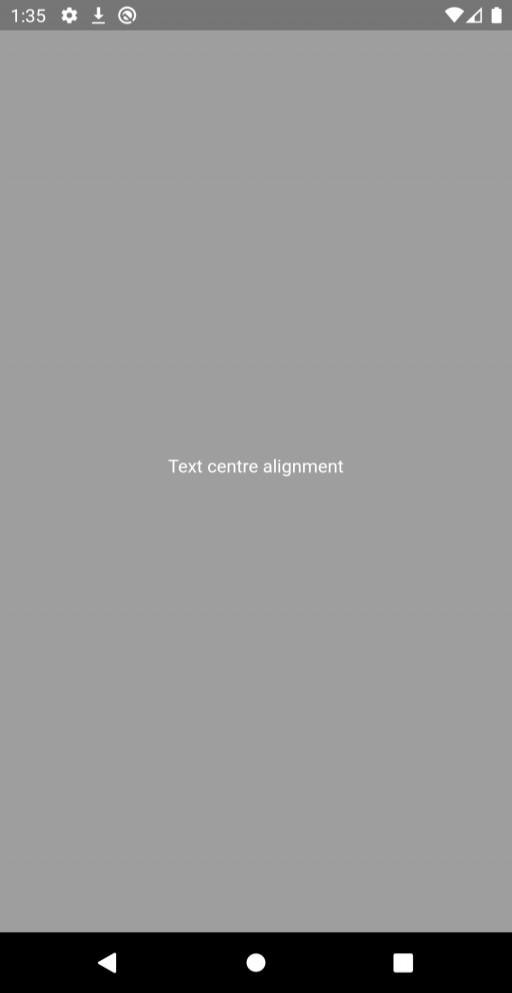
),

);

}

}

## OUTPUT:



**Container**

Like other, container class is also a widget.

A container can contain multiple child widgets.

It is a widget that combines common painting, positioning, and sizing of the child widgets.

A container is just like box which have any content inside it.

We can change background of container, can give margin to separate child widget. A container surrounds its child with padding.

## CODE:

import 'package:flutter/material.dart';

**void** main() { runApp(MyApp());

}

**class** MyApp **extends** StatefulWidget { @override

\_MyAppState createState() => \_MyAppState();

}

**class** \_MyAppState **extends** State<MyApp> { @override

Widget build(BuildContext context) {

**return** MaterialApp( home

: Scaffold(appBar

: AppBar(title

: Text("Example of container"),

),

body

: Container(

color: Colors.green, padding: EdgeInsets.all(30),

transform: Matrix4.rotationZ(0.1), margin: EdgeInsets.all(20),

**child**

: Text("this is a Contanier ", style

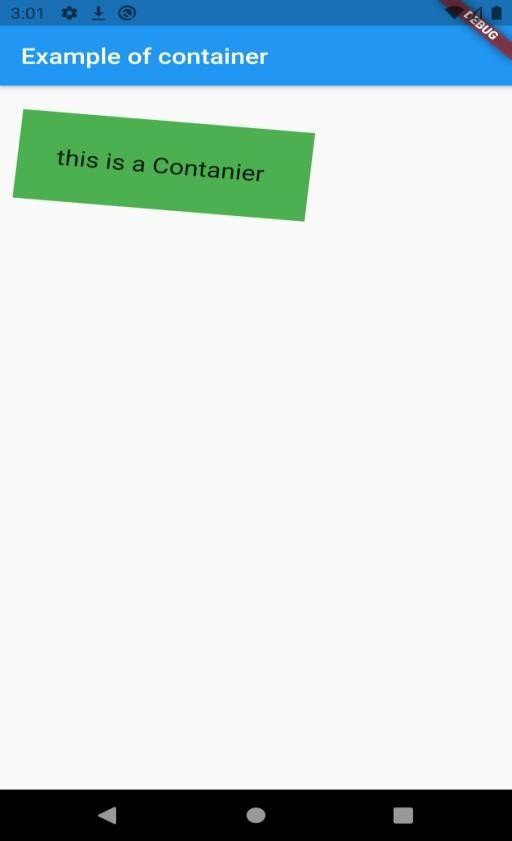
: TextStyle(fontSize : 20)),

), ), );

}

}

## OUTPUT:



**Multiple Child widgets**

The multiple child widgets are a type of widget, which contains **more than one child widget**, and the layout of these widgets are **unique**. For example, Row widget laying out of its child widget in a horizontal direction, and Column widget laying out of its child widget in a vertical direction. If we combine the Row and Column widget, then it can build any level of the complex widget.

Here, we are going to learn different types of multiple child widgets:

**Row:** It allows to arrange its child widgets in a horizontal direction.

## CODE:

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

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

**class** MyApp **extends** StatelessWidget {

// It is the root widget of your application. @override

Widget build(BuildContext context) {

**return** MaterialApp(

title: 'Multiple Layout Widget', debugShowCheckedModeBanner: **false**, theme: ThemeData(

// This is the theme of your application. primarySwatch: Colors.blue,

),

home: MyHomePage(),

);

}

}

**class** MyHomePage **extends** StatelessWidget { @override

Widget build(BuildContext context) {

**return** Center( child: Container(

alignment: Alignment.center, color: Colors.white,

child: Row(

children: <Widget>[ Expanded(

child: Text('Peter', textAlign: TextAlign.center),

),

Expanded(

child: Text('John', textAlign: TextAlign.center ),

),

Expanded(

child: FittedBox(

fit: BoxFit.contain, // otherwise the logo will be tiny child: **const** FlutterLogo(),

),

),

],

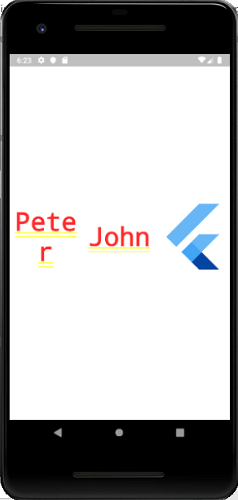
),

),

);

}}

## OUTPUT:



**Column:** It allows to arrange its child widgets in a vertical direction.

**ListView:** It is the most popular scrolling widget that allows us to arrange its child widgets one after another in scroll direction.

**GridView:** It allows us to arrange its child widgets as a scrollable, 2D array of widgets. It consists of a repeated pattern of cells arrayed in a horizontal and vertical layout.

**Expanded:** It allows to make the children of a Row and Column widget to occupy the maximum possible area.

**Table:** It is a widget that allows us to arrange its children in a table based widget.

**Flow:** It allows us to implements the flow-based widget.

**Stack:** It is an essential widget, which is mainly used for overlapping several children widgets. It allows you to put up the multiple layers onto the screen. The following example helps to understand it.

## CODE:

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

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

**class** MyApp **extends** StatelessWidget {

// It is the root widget of your application. @override

Widget build(BuildContext context) {

**return** MaterialApp(

title: 'Multiple Layout Widget', debugShowCheckedModeBanner: **false**, theme: ThemeData(

// This is the theme of your application. primarySwatch: Colors.blue,

),

home: MyHomePage(),

);

}

}

**class** MyHomePage **extends** StatelessWidget { @override

Widget build(BuildContext context) {

**return** Center( child: Container(

alignment: Alignment.center, color: Colors.white,

child: Stack( children: <Widget>[

// Max Size Container(

color: Colors.blue,

),

Container(

color: Colors.pink, height: 400.0,

width: 300.0,

),

Container(

color: Colors.yellow, height: 220.0,

width: 200.0,

)

],

),

),

);

}

}

## OUTPUT:



**CONCLUSION:** Hence, we have designed a layout of flutter app using layout widgets

**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

# Experiment 5

**AIM:** To apply navigation, routing and gesture in flutter app.

## THEORY:

**Flutter Navigation and Routing**

Navigation and routing are some of the core concepts of all mobile application, which allows the user to move between different pages. We know that every mobile application contains several screens for displaying different types of information. For example, an app can have a screen that contains various products. When the user taps on that product, immediately it will display detailed information about that product.

In Flutter, the screens and pages are known as routes, and these routes are just a widget. In Android, a route is similar to an Activity, whereas, in iOS, it is equivalent to a ViewController.

In any mobile app, navigating to different pages defines the workflow of the application, and the way to handle the navigation is known as routing. Flutter provides a basic routing class MaterialPageRoute and two methods Navigator.push() and Navigator.pop() that shows how to navigate between two routes. The following steps are required to start navigation in your application.

## Navigation with Named Routes

We have learned how to navigate to a new screen by creating a new route and manage it by using the Navigator. The Navigator maintains the stack-based history of routes. If there is a need to navigate to the same screen in many parts of the app, this approach is not beneficial because it results in code duplication. The solution to this problem can be removed by defining the named routes and can use the named routes for navigation.

We can work with named routes by using the Navigator.pushNamed() function. This function takes two required arguments (build context and string) and one optional argument. Also, we know about the MaterialPageRoute, which is responsible for page transition. If we do not use this, then it is difficult to change the page.

## Gestures

Gestures represent semantic actions (for example, tap, drag, and scale) that are recognized from multiple individual pointer events, potentially even multiple individual pointers. Gestures can dispatch multiple events, corresponding to the lifecycle of the gesture (for example, drag start, drag update, and drag end):

Tap onTapDown

A pointer that might cause a tap has contacted the screen at a particular location.

onTapUp

A pointer that will trigger a tap has stopped contacting the screen at a particular location.

onTap

The pointer that previously triggered the onTapDown has also triggered onTapUp which ends up causing a tap.

onTapCancel

The pointer that previously triggered the onTapDown will not end up causing a tap.

Double tap onDoubleTap

The user has tapped the screen at the same location twice in quick succession.

Long press onLongPress

A pointer has remained in contact with the screen at the same location for a long period of time.

Vertical drag onVerticalDragStart

A pointer has contacted the screen and might begin to move vertically. onVerticalDragUpdate

A pointer that is in contact with the screen and moving vertically has moved in the vertical direction.

onVerticalDragEnd

A pointer that was previously in contact with the screen and moving vertically is no longer in contact with the screen and was moving at a specific velocity when it stopped contacting the screen.

## CODE:

import 'package:flutter/material.dart'; void main() {

runApp(MaterialApp( title: 'Flutter Navigation', theme: ThemeData(

// This is the theme of your application. primarySwatch: Colors.green,

),

home: FirstRoute(),

));

}

class FirstRoute extends StatelessWidget { @override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: Text('First Screen'),

),

body: Center(

child: RaisedButton( child: Text('Click Here'),

color: Colors.orangeAccent, onPressed: () {

Navigator.push(

context,

MaterialPageRoute(builder: (context) =>SecondRoute()),

);

},

),

),

);

}

}

class SecondRoute extends StatelessWidget { @override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: Text("Second Screen"),

),

body: Center(

child: RaisedButton( color: Colors.blueGrey,

onPressed: () { Navigator.pop(context);

},

child: Text('Go back'),

),

),

);

}

}

## OUTPUT:

**CONCLUSION:** Hence, we have applied navigation, routing and gesture in flutter app.

**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

# Experiment 6

**AIM:** To connect flutter UI by including with firebase.

## THEORY:

Firebase is a Backend-as-a-Service (BaaS) app development platform that provides hosted backend services such as a real-time database, cloud storage, authentication, crash reporting, machine learning, remote configuration, and hosting for your static files.

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

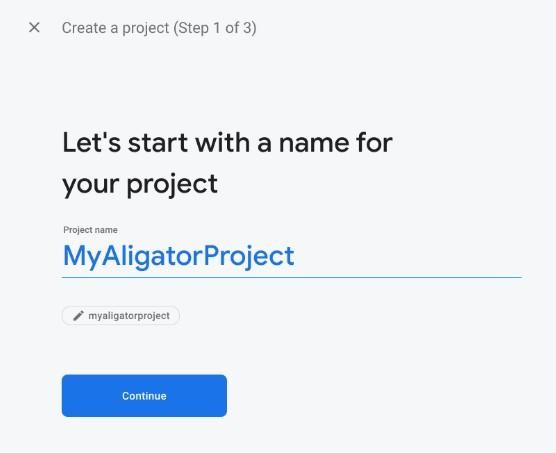
## STEPS TO CONNECT FLUTTER WITH FIREBASE:

1. Creating a New Flutter Project.

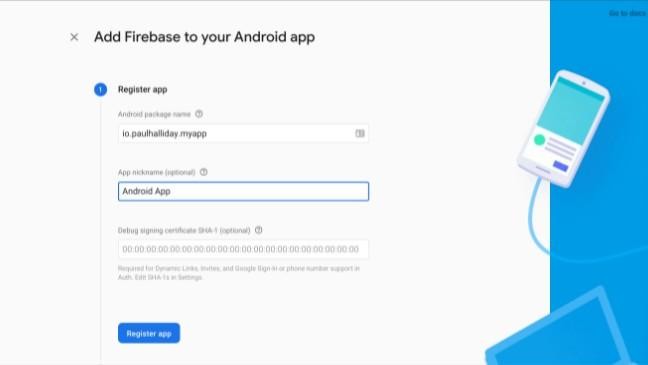
$ flutter create flutterfirebaseexample

$ cd flutterfirebaseexample

1. Creating a New Firebase Project.



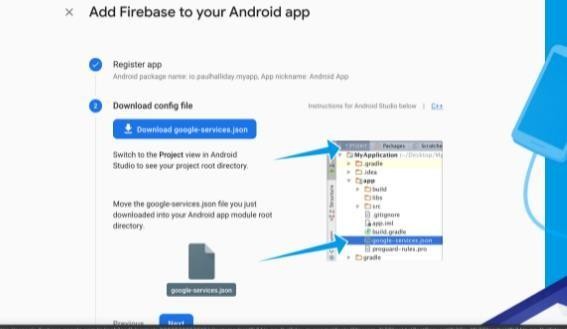
1. Adding Android support& Registering the App.



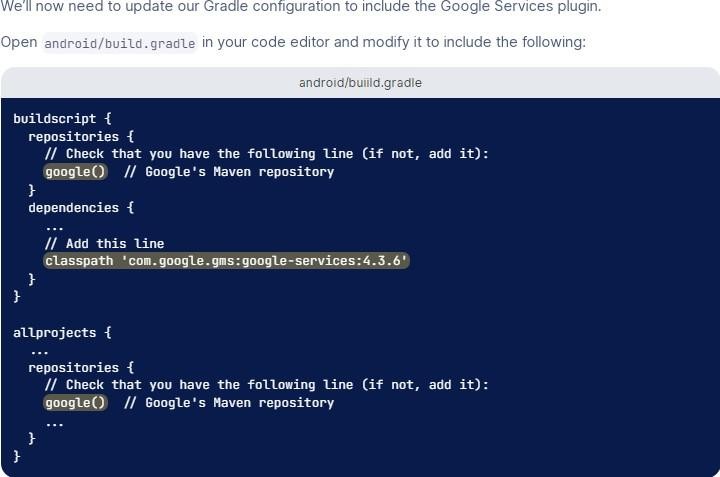
The most important thing here is to match up the **Android package name** that you choose here with the one inside of our application. Once you’ve decided on a name, open android/app/build.gradle in your code editor and update the applicationId to match the **Android package name**:

1. Downloading the Config File.

The next step is to add the Firebase configuration file into our Flutter project. This is important as it contains the API keys and other critical information for Firebase to use



1. Adding the Firebase SDK.



**CONCLUSION:** We have learned how to set up and ready our Flutter applications to be used with Firebase.

**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

# Experiment 7

**Aim:** To implement service worker events like fetch, sync and push for E- commerce PWA.

## Theory:

**Service worker**

A service worker is a script that your browser runs in the background, separate from a web page, opening the door to features that don't need a web page or user interaction. Today, they already include features like [push](https://developers.google.com/web/updates/2015/03/push-notifications-on-the-open-web) [notifications](https://developers.google.com/web/updates/2015/03/push-notifications-on-the-open-web) and [background sync](https://developers.google.com/web/updates/2015/12/background-sync). In the future, service workers might support other things like periodic sync or geofencing. The core feature discussed in this tutorial is the ability to intercept and handle network requests, including programmatically managing a cache of responses.

The reason this is such an exciting API is that it allows you to support offline experiences, giving developers complete control over the experience.

Before service worker, there was one other API that gave users an offline experience on the web called [AppCache](https://www.html5rocks.com/en/tutorials/appcache/beginner/). There are a number of issues with the AppCache API that service workers were designed to avoid.

Things to note about a service worker:

* It's a [JavaScript Worker](https://www.html5rocks.com/en/tutorials/workers/basics/), so it can't access the DOM directly. Instead, a service worker can communicate with the pages it controls by responding to messages sent via the [postMessage](https://html.spec.whatwg.org/multipage/workers.html#dom-worker-postmessage) interface, and those pages can manipulate the DOM if needed.
* Service worker is a programmable network proxy, allowing you to control how network requests from your page are handled.
* It's terminated when not in use, and restarted when it's next needed, so you cannot rely on global state within a service worker's onfetch and onmessage handlers. If there is information that you need to persist and reuse across restarts, service workers do have access to the [IndexedDB API](https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API).
* Service workers make extensive use of promises, so if you're new to promises, then you should stop reading this and check out [Promises, an](https://developers.google.com/web/fundamentals/getting-started/primers/promises) [introduction](https://developers.google.com/web/fundamentals/getting-started/primers/promises).

## The service worker life cycle

page.

A service worker has a lifecycle that is completely separate from your web

To install a service worker for your site, you need to register it, which you

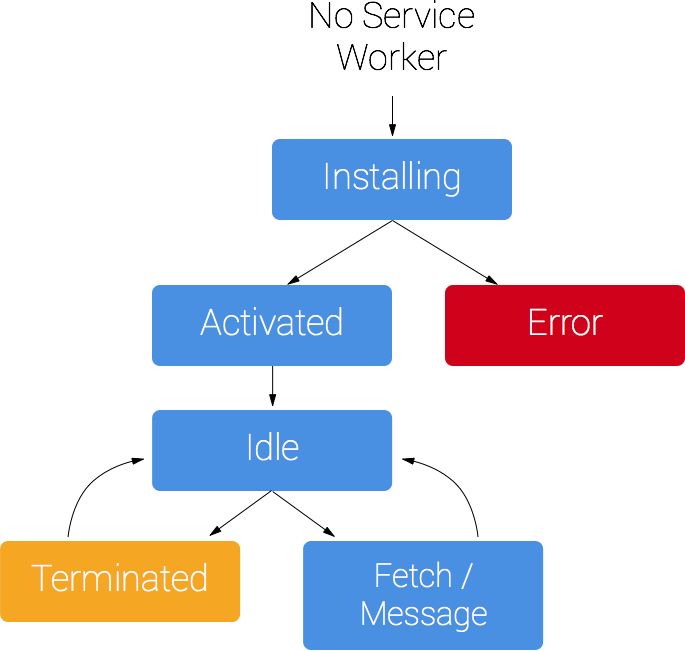
do in your page's JavaScript. Registering a service worker will cause the browser to start the service worker install step in the background.

Typically, during the install step, you'll want to cache some static assets. If all the files are cached successfully, then the service worker becomes installed. If any of the files fail to download and cache, then the install step will fail and the service worker won't activate (i.e., won't be installed). If that happens, don't worry, it'll try again next time. But that means if it does install, you know you've got those static assets in the cache.

When installed, the activation step will follow and this is a great opportunity for handling any management of old caches, which we'll cover during the service worker update section.

After the activation step, the service worker will control all pages that fall under its scope, though the page that registered the service worker for the first time won't be controlled until it's loaded again. Once a service worker is in control, it will be in one of two states: either the service worker will be terminated to save memory, or it will handle fetch and message events that occur when a network request or message is made from your page.

Below is an overly simplified version of the service worker lifecycle on its first installation.



## Prerequisites

Browser support

Browser options are growing. Service workers are supported by Chrome, Firefox and Opera. Microsoft Edge is now [showing public support](https://developer.microsoft.com/en-us/microsoft-edge/status/serviceworker/). Even Safari has dropped [hints of future development](https://trac.webkit.org/wiki/FiveYearPlanFall2015). You can follow the progress of all the browsers at Jake Archibald's [is Service worker ready](https://jakearchibald.github.io/isserviceworkerready/) site.

You need HTTPS

During development you'll be able to use service worker through localhost, but to deploy it on a site you'll need to have HTTPS setup on your server.

Using service worker you can hijack connections, fabricate, and filter responses. Powerful stuff. While you would use these powers for good, a man-in-the- middle might not. To avoid this, you can only register service workers on pages served over HTTPS, so we know the service worker the browser receives hasn't been tampered with during its journey through the network.

[GitHub Pages](https://pages.github.com/) are served over HTTPS, so they're a great place to host demos.

If you want to add HTTPS to your server then you'll need to get a TLS certificate and set it up for your server. This varies depending on your setup, so check your server's documentation and be sure to check out [Mozilla's SSL config](https://mozilla.github.io/server-side-tls/ssl-config-generator/) [generator](https://mozilla.github.io/server-side-tls/ssl-config-generator/) for best practices.

## Register a service worker

To install a service worker, you need to kick start the process by **registering** it in your page. This tells the browser where your service worker JavaScript file lives.

if('serviceWorker'in navigator){ window.addEventListener('load',function(){ navigator.serviceWorker.register('/sw.js').then(function(registration){

// Registration was successful

console.log('ServiceWorker registration successful with scope: ', registration.scope);

},function(err){

// registration failed :(

console.log('ServiceWorker registration failed: ', err);

});

});

}

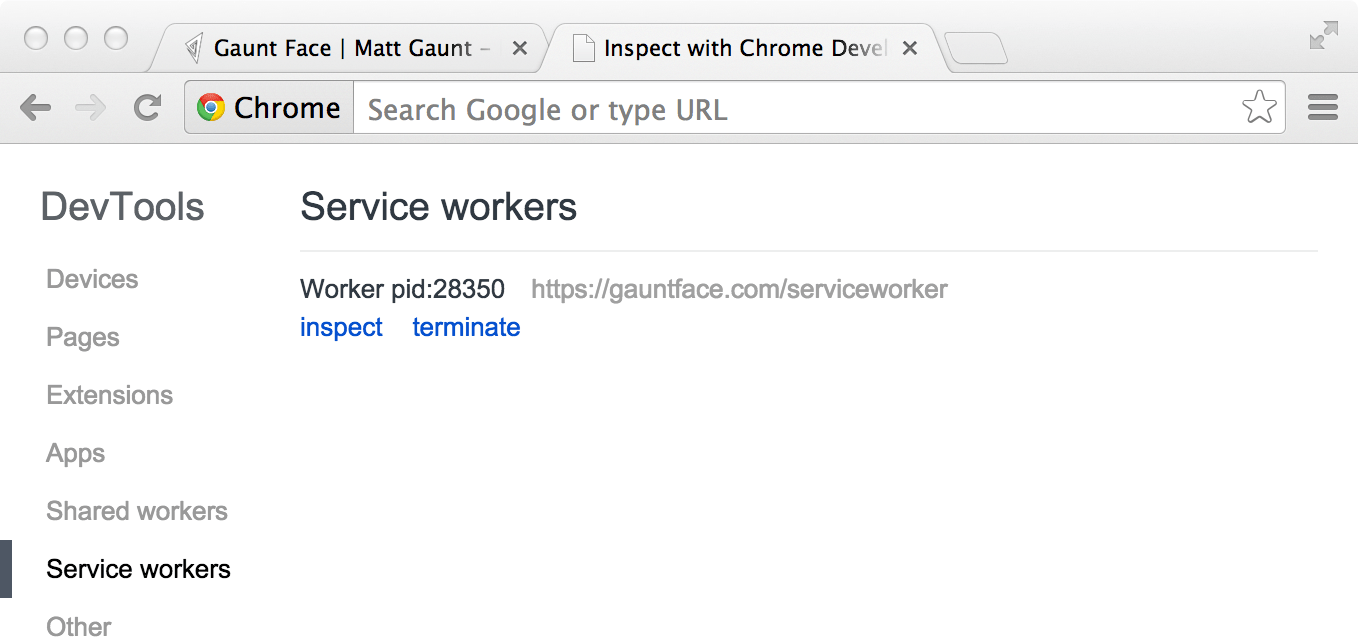
This code checks to see if the service worker API is available, and if it is, the service worker at /sw.js is registered [once the page is loaded](https://developers.google.com/web/fundamentals/instant-and-offline/service-worker/registration).

You can call register() every time a page loads without concern; the browser will figure out if the service worker is already registered or not and handle it accordingly.

One subtlety with the register() method is the location of the service worker file. You'll notice in this case that the service worker file is at the root of the domain. This means that the service worker's scope will be the entire origin. In other

words, this service worker will receive fetch events for everything on this domain. If we register the service worker file at /example/sw.js, then the service worker would only see fetch events for pages whose URL starts with /example/ (i.e., /example/page1/, /example/page2/).

Now you can check that a service worker is enabled by going to chrome://inspect/#service-workers and looking for your site.



When service worker was first being implemented, you could also view your service worker details through chrome://serviceworker-internals. This may still be useful, if for nothing more than learning about the life cycle of service workers, but don't be surprised if it gets replaced completely by chrome://inspect/#service-workers at a later date.

You may find it useful to test your service worker in an Incognito window so that you can close and reopen knowing that the previous service worker won't affect the new window. Any registrations and caches created from within an Incognito window will be cleared out once that window is closed.

## Install a service worker

After a controlled page kicks off the registration process, let's shift to the point of view of the service worker script, which handles the install event.

For the most basic example, you need to define a callback for the install event and decide which files you want to cache.

self.addEventListener('install',function(event){

// Perform install steps

});

Inside of our install callback, we need to take the following steps:

1. Open a cache.
2. Cache our files.
3. Confirm whether all the required assets are cached or not.

var CACHE\_NAME ='my-site-cache-v1'; var urlsToCache =[

'/',

'/styles/main.css', '/script/main.js'

];

self.addEventListener('install',function(event){

// Perform install steps event.waitUntil( caches.open(CACHE\_NAME)

.then(function(cache){ console.log('Opened cache'); return cache.addAll(urlsToCache);

})

);

});

Here you can see we call caches.open() with our desired cache name, after which we call cache.addAll() and pass in our array of files. This is a chain of promises (caches.open() and cache.addAll()). The event.waitUntil() method takes a promise and uses it to know how long installation takes, and whether it succeeded or not.

If all the files are successfully cached, then the service worker will be installed. If **any** of the files fail to download, then the install step will fail. This allows you to rely on having all the assets that you defined, but does mean you need to be careful with the list of files you decide to cache in the install step. Defining a long list of files will increase the chance that one file may fail to cache, leading to your service worker not getting installed.

This is just one example; you can perform other tasks in the install event or avoid setting an install event listener altogether.

## Cache and return requests

Now that you've installed a service worker, you probably want to return one of your cached responses, right?

After a service worker is installed and the user navigates to a different page or refreshes, the service worker will begin to receive fetch events, an example of which is below.

self.addEventListener('fetch',function(event){ event.respondWith( caches.match(event.request)

.then(function(response){

// Cache hit - return response if(response){

return response;

}

return fetch(event.request);

}

)

);

});

Here we've defined our fetch event and within event.respondWith(), we pass in a promise from caches.match(). This method looks at the request and finds any cached results from any of the caches your service worker created.

If we have a matching response, we return the cached value, otherwise we return the result of a call to fetch, which will make a network request and return the data if anything can be retrieved from the network. This is a simple example and uses any cached assets we cached during the install step.

If we want to cache new requests cumulatively, we can do so by handling the response of the fetch request and then adding it to the cache, like below.

self.addEventListener('fetch',function(event){ event.respondWith( caches.match(event.request)

.then(function(response){

// Cache hit - return response if(response){

return response;

}

return fetch(event.request).then( function(response){

// Check if we received a valid response

if(!response || response.status !==200|| response.type !=='basic'){ return response;

}

// IMPORTANT: Clone the response. A response is a stream

// and because we want the browser to consume the response

// as well as the cache consuming the response, we need

// to clone it so we have two streams. var responseToCache = response.clone();

caches.open(CACHE\_NAME)

.then(function(cache){ cache.put(event.request, responseToCache);

});

return response;

}

);

})

});

What we are doing is this:

1. Add a callback to .then() on the fetch request.
2. Once we get a response, we perform the following checks:
   1. Ensure the response is valid.
   2. Check the status is 200 on the response.
   3. Make sure the response type is **basic**, which indicates that it's a request from our origin. This means that requests to third party assets aren't cached as well.
3. If we pass the checks, we [clone](https://fetch.spec.whatwg.org/#dom-response-clone) the response. The reason for this is that because the response is a [Stream](https://streams.spec.whatwg.org/), the body can only be consumed once. Since we want to return the response for the browser to use, as well as pass it to the cache to use, we need to clone it so we can send one to the browser and one to the cache.

## Update a service worker

There will be a point in time where your service worker will need updating. When that time comes, you'll need to follow these steps:

1. Update your service worker JavaScript file. When the user navigates to your site, the browser tries to redownload the script file that defined the service worker in the background. If there is even a byte's difference in the service worker file compared to what it currently has, it considers it *new*.
2. Your new service worker will be started and the install event will be fired.
3. At this point the old service worker is still controlling the current pages so the new service worker will enter a waiting state.
4. When the currently open pages of your site are closed, the old service worker will be killed and the new service worker will take control.
5. Once your new service worker takes control, its activate event will be fired. One common task that will occur in the activate callback is cache management. The reason you'll want to do this in the activate callback is because if you were to wipe out any old caches in the install step, any old service worker, which keeps control of all the current pages, will suddenly stop being able to serve files from that cache.

Let's say we have one cache called 'my-site-cache-v1', and we find that we want to split this out into one cache for pages and one cache for blog posts. This means in the install step we'd create two caches, 'pages-cache-v1' and 'blog- posts-cache-v1' and in the activate step we'd want to delete our older 'my-site- cache-v1'.

The following code would do this by looping through all of the caches in the service worker and deleting any caches that aren't defined in the cache allowlist.

self.addEventListener('activate',function(event){

var cacheAllowlist =['pages-cache-v1','blog-posts-cache-v1'];

event.waitUntil( caches.keys().then(function(cacheNames){ returnPromise.all(

cacheNames.map(function(cacheName){ if(cacheAllowlist.indexOf(cacheName)===-1){ return caches.delete(cacheName);

}

})

);

})

);

});

## Rough edges and gotchas

This stuff is really new. Here's a collection of issues that get in the way. Hopefully this section can be deleted soon, but for now these are worth being mindful of. If installation fails, we're not so good at telling you about it

If a worker registers, but then doesn't appear in chrome://inspect/#service- workers or chrome://serviceworker-internals, it's likely failed to install due to an error being thrown, or a rejected promise being passed to event.waitUntil().

To work around this, go to chrome://serviceworker-internals and check "Open DevTools window and pause JavaScript execution on service worker startup for debugging", and put a debugger statement at the start of your install event. This, along with [Pause on uncaught exceptions](https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints), should reveal the issue.

The defaults of fetch()

### No credentials by default

When you use fetch, by default, requests won't contain credentials such as cookies. If you want credentials, instead call:

fetch(url,{ credentials:'include'

})

This behaviour is on purpose, and is arguably better than XHR's more complex default of sending credentials if the URL is same-origin, but omitting them otherwise. Fetch's behaviour is more like other CORS requests, such as <img crossorigin>, which never sends cookies unless you opt-in with <img crossorigin="use-credentials">.

### Non-CORS fail by default

By default, fetching a resource from a third-party URL will fail if it doesn't support CORS. You can add a no-CORS option to the Request to overcome this, although this will cause an 'opaque' response, which means you won't be able to tell if the response was successful or not.

cache.addAll(urlsToPrefetch.map(function(urlToPrefetch){ returnnewRequest(urlToPrefetch,{ mode:'no-cors'});

})).then(function(){

console.log('All resources have been fetched and cached.');

});

**CONCLUSION:** Hence, we successfully studied the Service Worker Events in PW

**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

# Experiment 8

**AIM:** To study and implement deployment of Ecommerce PWA to GitHub pages.

## THEORY:

*Continuous deployment* (CD) is the practice of using automation to publish and deploy software updates. As part of the typical CD process, the code is automatically built and tested before deployment.

Continuous deployment is often coupled with continuous integration.

You can set up a GitHub Actions workflow to deploy your software product. To verify that your product works as expected, your workflow can build the code in your repository and run your tests before deploying.

You can configure your CD workflow to run when a GitHub event occurs (for example, when new code is pushed to the default branch of your repository), on a set schedule, manually, or when an external event occurs using the repository dispatch webhook. GitHub Actions provides features that give you more control over deployments. For example, you can use environments to require approval for a job to proceed, restrict which branches can trigger a workflow, or limit access to secrets. You can use concurrency to limit your CD pipeline to a maximum of one in-progress deployment and one pending deployment.

## OUTPUT:

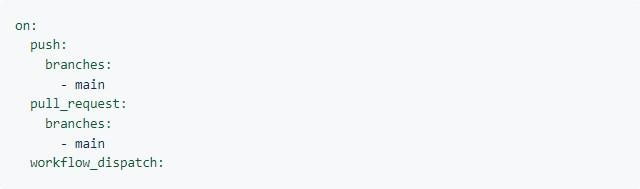
* **Triggering your deployment**

You can use a variety of events to trigger your deployment workflow. Some of the most common are: pull\_request, push,

and workflow\_dispatch.

For example, a workflow with the following triggers runs whenever:

* There is a push to the main branch.
* A pull request targeting the main branch is opened, synchronized, or reopened.
* Someone manually triggers it.



## Using concurrency

Concurrency ensures that only a single job or workflow using the same concurrency group will run at a time. You can use concurrency so that an environment has a maximum of one deployment in progress and one deployment pending at a time.

For example, when the following workflow runs, it will be paused with the status pending if any job or workflow that uses the production concurrency group is in progress. It will also cancel any job or workflow that uses the production concurrency group and has the status pending. This means that there will be a maximum of one running and one pending job or workflow in that uses the production concurrency group.



You can also specify concurrency at the job level. This will allow other jobs in the workflow to proceed even if the concurrent job is pending.



You can also use cancel-in-progress to cancel any currently running job or workflow in the same concurrency group.



**CONCLUSION:** Thus, we have studied and implemented deployment on GitHub pages.

**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

# Experiment 9

**AIM:** To use Google Lighthouse PWA analysis tool to set the PWA Functioning.

## THEORY:

Lighthouse is **an open-source tool from Google that audits a web app for PWA features**. It provides a set of metrics to help guide you in building a PWA with a full application like experience for your users. Lighthouse tests if your app: Can load in offline or flaky network conditions.

**Widgets:** Each element on a screen of the Flutter app is a widget. The view of the screen completely depends upon the choice and sequence of the widgets used to build the app. And the structure of the code of an app is a tree of widgets**.**

## PWA score?

Lighthouse measures the potential of a website using five categories: Performance, Progressive Web App (PWA), Accessibility, Best Practices, and SEO. It reviews each of these categories separately and by providing a performance score **between 0 and 100**. 0 being the lowest possible score and 100 being the best.

Lighthouse tests if your app:

* Can load in offline or flaky network conditions
* Is relatively fast
* Is served from a secure origin
* Uses certain accessibility best practices

Lighthouse is available as a Chrome extension for Chrome 52 (and later) and a command line tool.

## Running Lighthouse as a Chrome extension

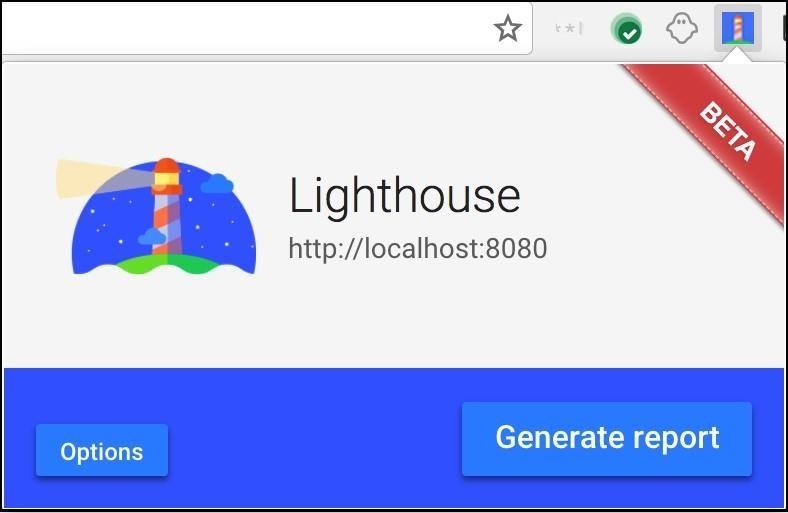
Download the Lighthouse Chrome extension from the [Chrome Web Store.](http://chrome.google.com/webstore/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk)



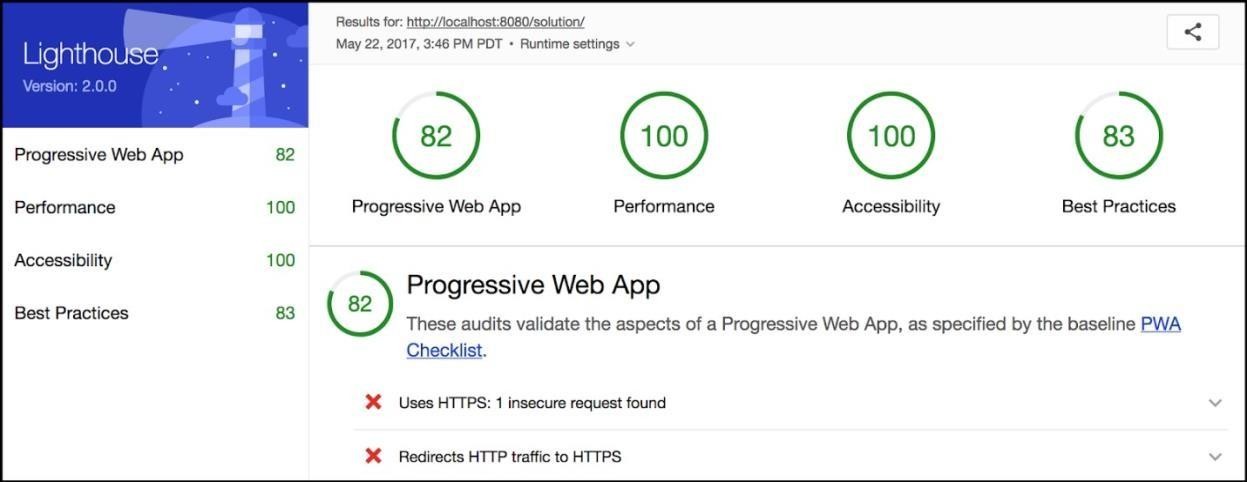
When installed it places an icon in your taskbar.

Run Lighthouse on your application by selecting the icon and choosing

**Generate report** (with your app open in the browser page).



Lighthouse generates an HTML page with the results. An example page is shown below.



## Running Lighthouse from the command line

If you want to run Lighthouse from the command line (for example, to integrate it with a build process) it is available as a [Node](https://nodejs.org/en/) module. You can download Node from[nodejs.org](https://nodejs.org/en/)(select the version that best suits your environment and operating system).To install Lighthouse's Node module from the command line, use the following command:

npm install -g lighthouse

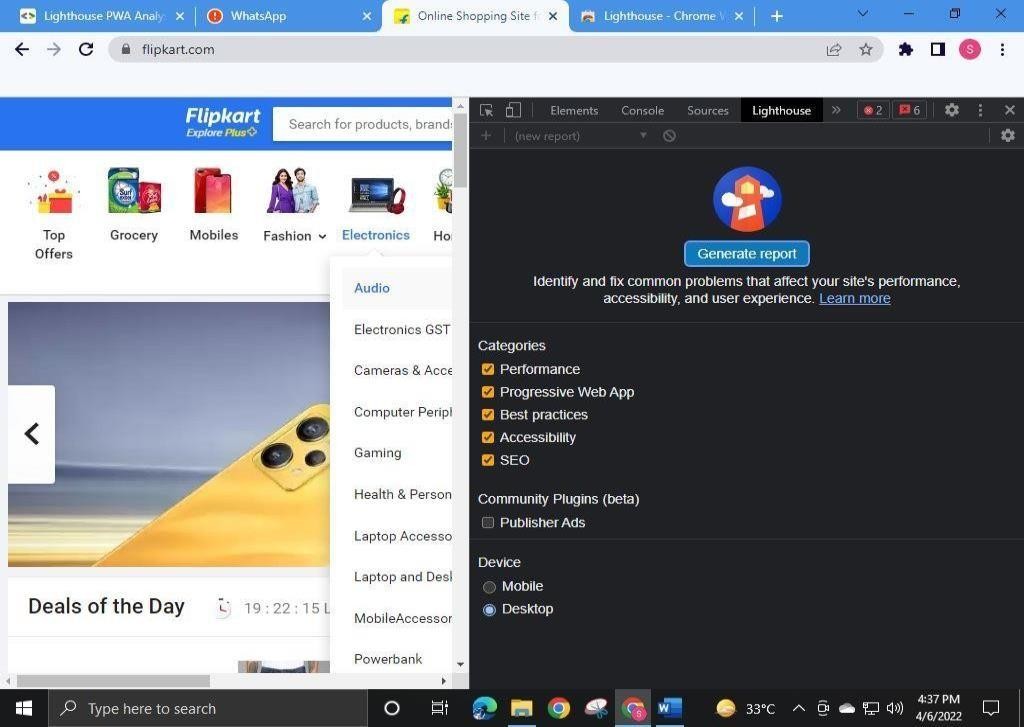
This installs the tool globally. You can then run Lighthouse from the command line (where<https://airhorner.com/>is your app): lighthouse https://airhorner.com/

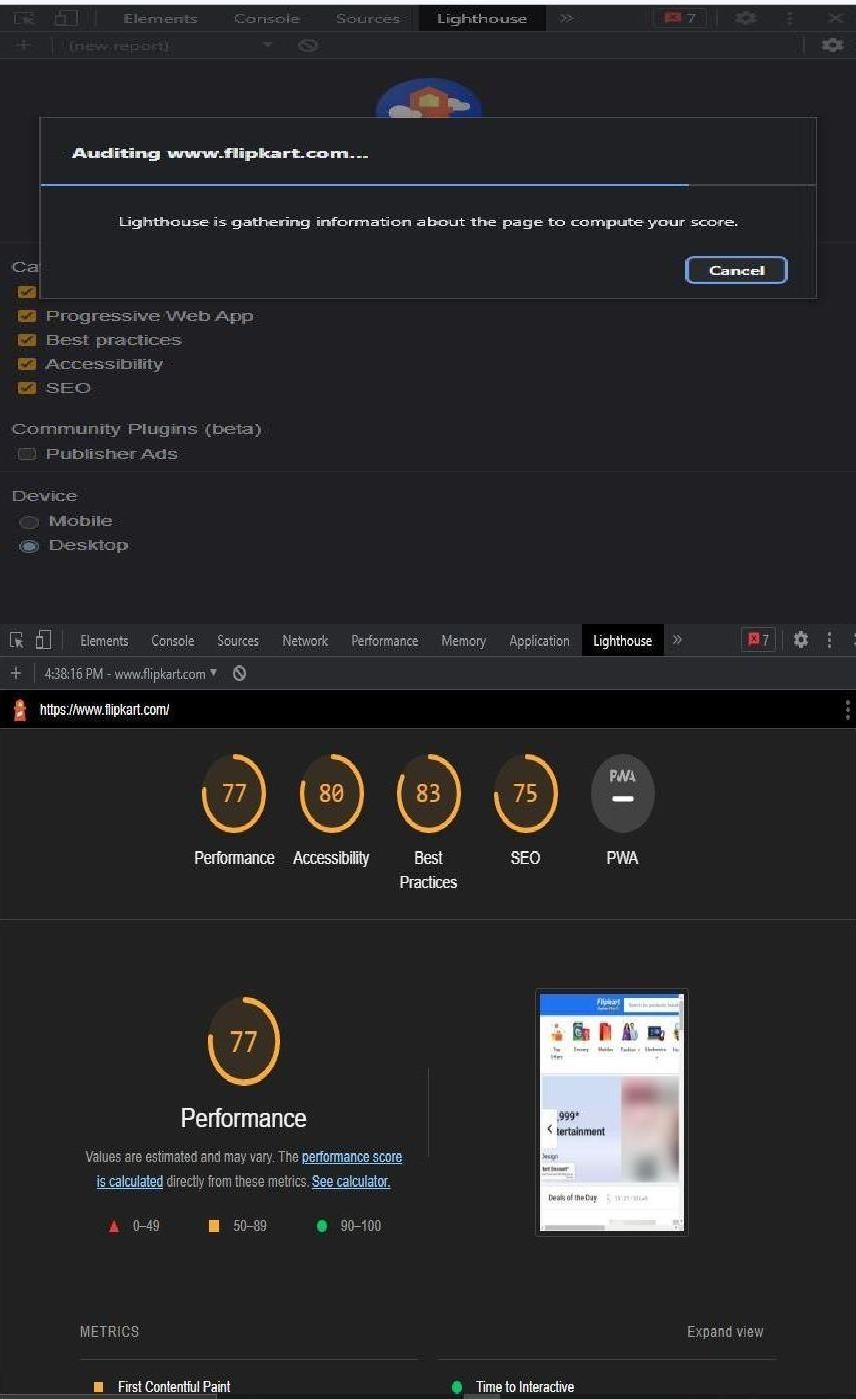
You can check Lighthouse flags and options with the following command: lighthouse –help

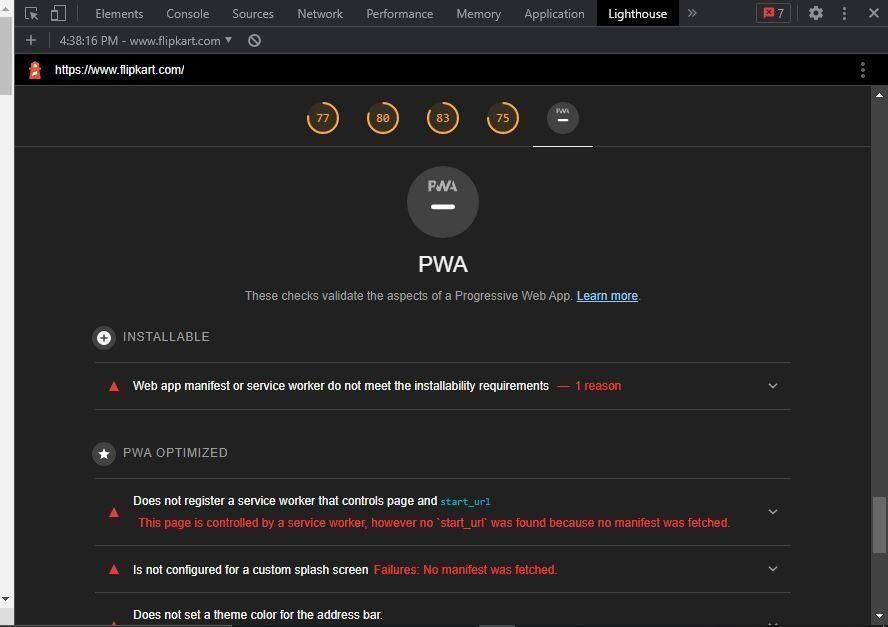
PWA features are advantages making them the clear choice over native and classic websites. PWAs function even when the device is offline. They make full use of modern web features including push notification, cache, and secure connections to provide rich Web based experiences.

## PWA Functioning Example:

We test the website score: Take random website and open lighthouse and click generate PWA.







PWA features are advantages making them the clear choice over native and classic websites. PWAs function even when the device is offline. They make full use of modern web features including push notification, cache, and secure connections to provide rich Web based experiences

**CONCLUSION:** Thus, we have successfully performed a Google Lighthouse PWA analysis tool to set the PWA Functioning.

**Name:** Kamal Agrahari **ID No:** VU4F2223028

**Division:** A **Batch:** B

# Experiment 10

**AIM:** To deploy an Ecommerce PWA using SSL enabled static hosting solution.

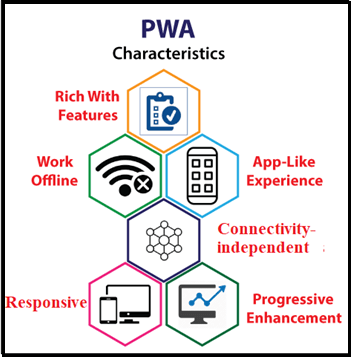
## THEORY:

The full form of PWA is Progressive Web Applications. PWA is HTML 5 webpage. It is a web application like a mobile application that is stored directly on the mobile from the website. In other words, it is a website that runs on the mobile browser of the user. PWA is a new technology. PWA allows the website to be stored in your device. It creates an icon in the form of a website app, and that icon feels like a mobile application upon opening.

After the PWA icon of the website is created in the device, all the posts that are open with the help of the internet, are automatically stored in your device which does not need the internet to read the second time. It is a methodology where we combine native app experience with the Browser feature.

Generally, when you have to build applications, you have to develop that application for different platforms, such as IOS, Android, and Windows. But in the case of PWA, you do not need to develop separate applications for different platforms. You only need to create HTML5 based webpages that can run on any mobile browser.

## Characteristics of PWA

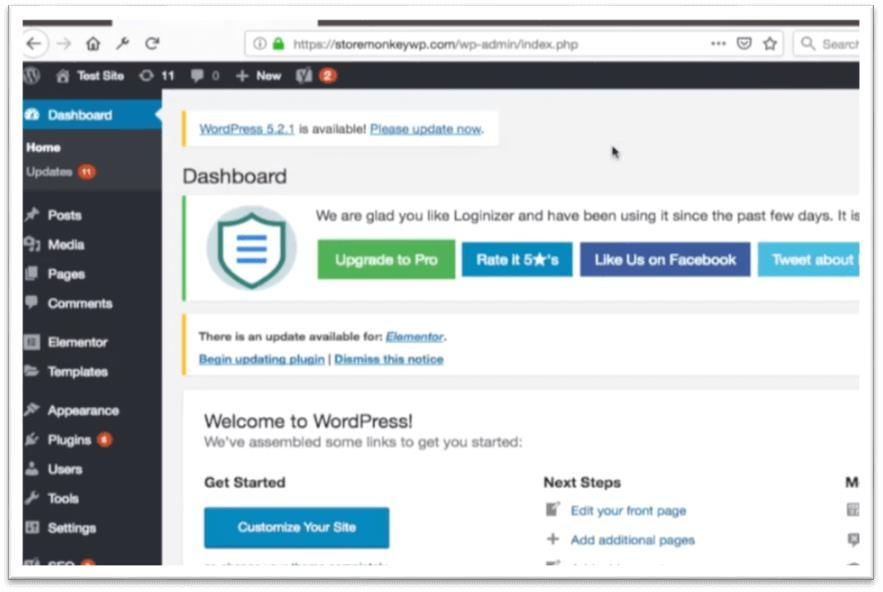


Progressive web applications have the following characteristics that are important to understand: Java

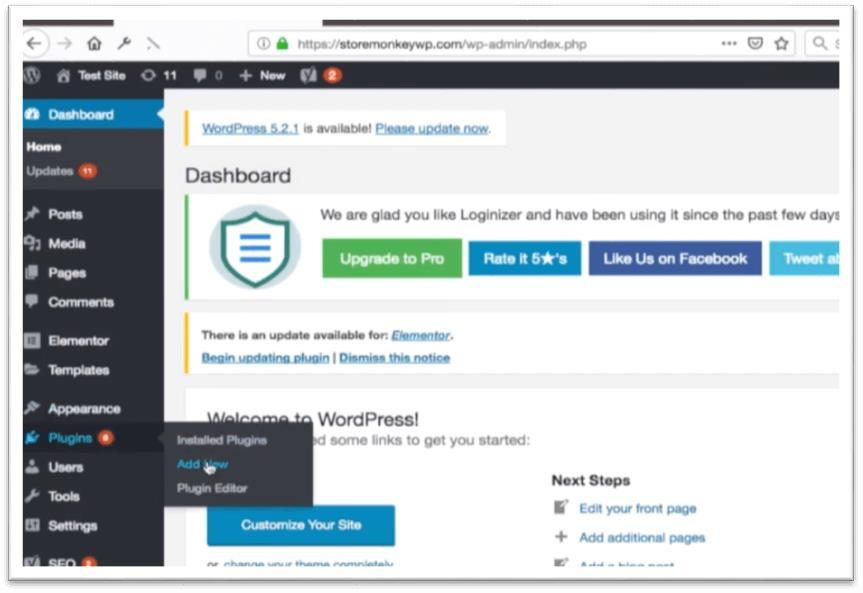
* 1. **Progressive:** The term progressive means, a PWA application must work on any device and improve the performance of the user's mobile browser and design.
  2. **Discoverable:** A PWA is a website with some extra features. It can be searched via mobile searching applications like Google Chrome. App Store or Play Store is not required for this.
  3. **Responsive:** The UI of a progressive web app should fit the form factor and screen size of the device.
  4. **App-like:** A PWA application should look like a native application. Although the methods for creating, sharing, launching, and updating of the PWA are completely different from the original application.
  5. **Connectivity-independent:** It works even when connectivity is very low.

How to convert the website into PWA using WordPress? There are seven simple steps to convert a website to PWA.

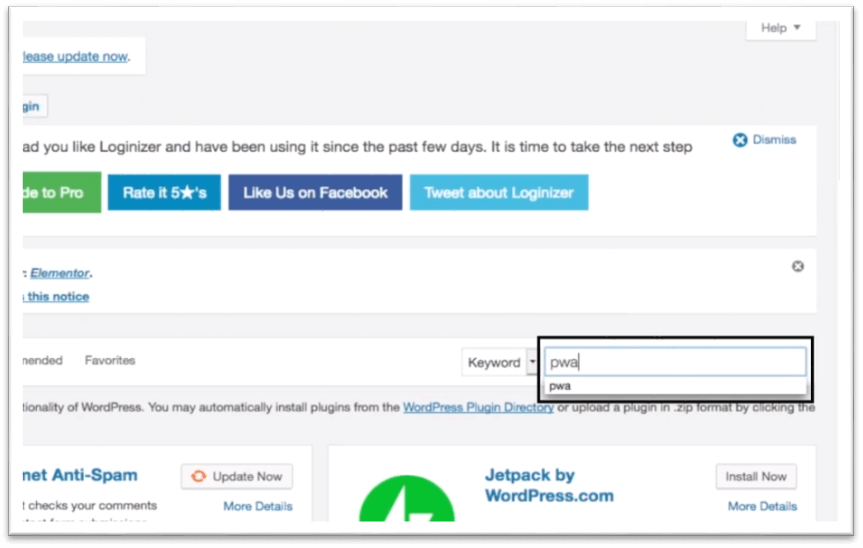
The first thing to do is to log in to the WordPress admin of your website.



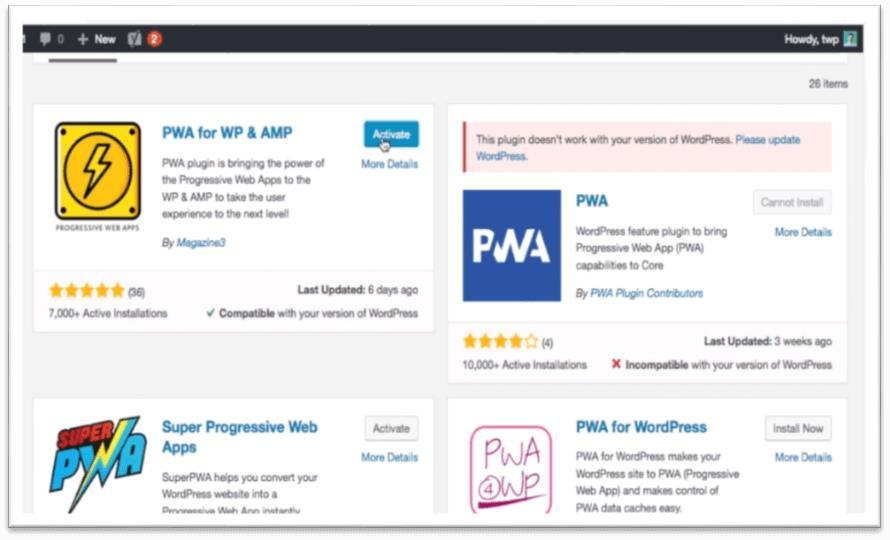
**Step 2**: After that, you have to go to the plugin option and click on the add new option.



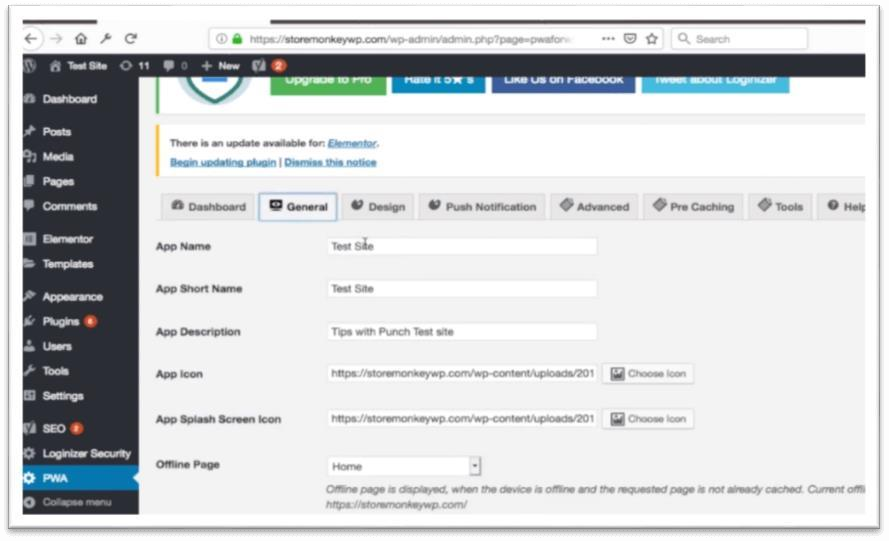
**Step 3:** After this, click on the search button and search the plugin of the progressive web app and install it in WordPress.



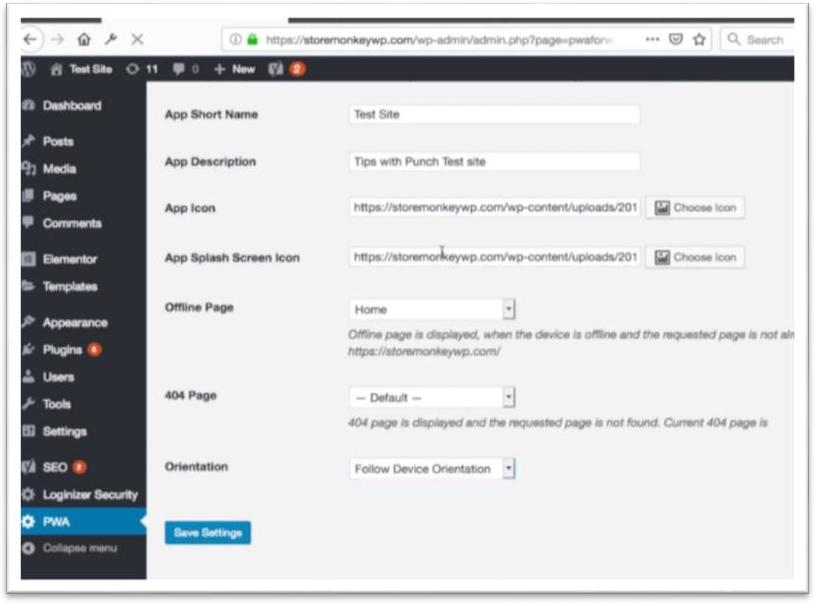
**Step 5:** Then, activate the plugin of the progressive web app.



**Step 6:** Now, you will see many options in your dashboard (such as Application Name, Description, Icon, Color), which can customize as per your choice.



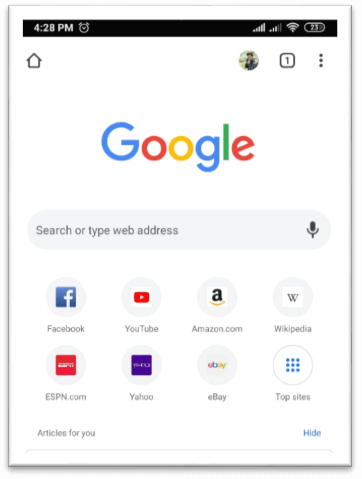
**Step 7:** Then click on the Save Settings button.



How do you install a PWA?

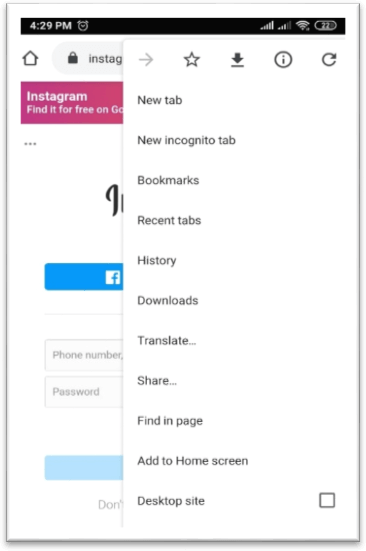
On Android phones, installing a PWA is pretty simple.

**Step 1:** The first thing to do is open google chrome on your mobile phone.

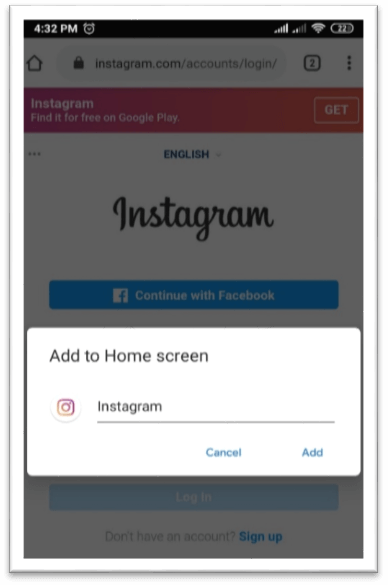


**Step 2:** Then, open the website, for example, Instagram.com.



**Step 3:** Now, click on the three dots of the right corner in the google chrome.

**Step 4:** Then, click on the Add to Home screen option and add it.



On iOS, you visit the website with the Safari browser and click the share icon. This opens a sub-menu with more icons and an option to save to the home screen.

**CONCLUSION:** Hence, we studied to deploy an Ecommerce PWA using SSL enabled static hosting solution.