Experiment No. 5

Aim: To apply navigation, routing and gestures in Flutter App

Theory:

→ Navigation and routing –

Flutter provides a complete system for navigating between screens and handling deep links. Small applications without complex deep linking can use Navigator, while apps with specific deep linking and navigation requirements should also use the Router to correctly handle deep links on Android and iOS, and to stay in sync with the address bar when the app is running on the web.

→ Using the Navigator –

The Navigator widget displays screens as a stack using the correct transition animations for the target platform. To navigate to a new screen, access the Navigator through the route's BuildContext and call imperative methods such as push() or pop():

```
content_copy
onPressed: () {
 Navigator.of(context).push(
 MaterialPageRoute(
 builder: (context) => const SongScreen(song: song),
 ),
 );
},
child: Text(song.name),
```

Because Navigator keeps a stack of Route objects (representing the history stack), The push() method also takes a Route object. The MaterialPageRoute object is a subclass of Route that specifies the transition animations for Material Design. For more examples of how to use the Navigator, follow the navigation recipes from the Flutter Cookbook or visit the Navigator API documentation.

→ Using named routes –

Applications with simple navigation and deep linking requirements can use the Navigator for navigation and the MaterialApp.routes parameter for deep links:

```
content_copy
@override
Widget build(BuildContext context) {
```

```
 return MaterialApp(
 routes: {
 '/': (context) => HomeScreen(),
 '/details': (context) => DetailScreen(),
 },
 );
}
```

Routes specified here are called named routes.

→ Using the Router –

Flutter applications with advanced navigation and routing requirements (such as a web app that uses direct links to each screen, or an app with multiple Navigator widgets) should use a routing package such as go_router that can parse the route path and configure the Navigator whenever the app receives a new deep link. To use the Router, switch to the router constructor on MaterialApp or CupertinoApp and provide it with a Router configuration. Routing packages, such as go_router, typically provide a configuration for you. For example:

```
content_copy
MaterialApp.router(
 routerConfig: GoRouter(
 // …
 )
);
```

Because packages like go_router are declarative, they will always display the same screen(s) when a deep link is received.

→ Using Router and Navigator together –

The Router and Navigator are designed to work together. You can navigate using the Router API through a declarative routing package, such as go_router, or by calling imperative methods such as push() and pop() on the Navigator. When you navigate using the Router or a declarative routing package, each route on the Navigator is page-backed, meaning it was created from a Page using the pages argument on the Navigator constructor. Conversely, any Route created by calling Navigator.push or showDialog will add a pageless route to the Navigator. If you are using a routing package, Routes that are page-backed are always deep-linkable, whereas pageless routes are not. When a page-backed Route is removed from the Navigator, all of the pageless routes after it are also removed. For example, if a deep link navigates by removing a page-backed route

from the Navigator, all pageless _routes after (up until the next _page-backed route) are removed too.

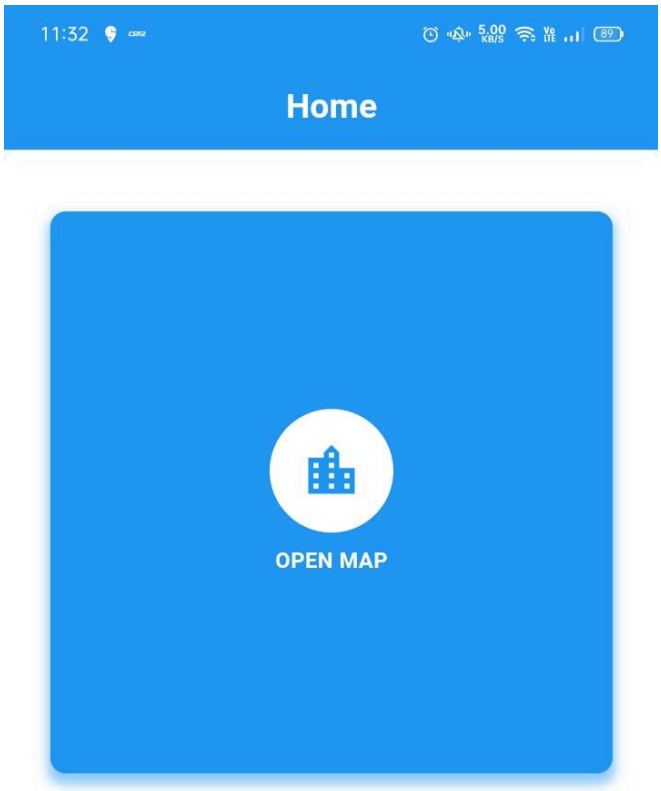
→ Gesture Detector –

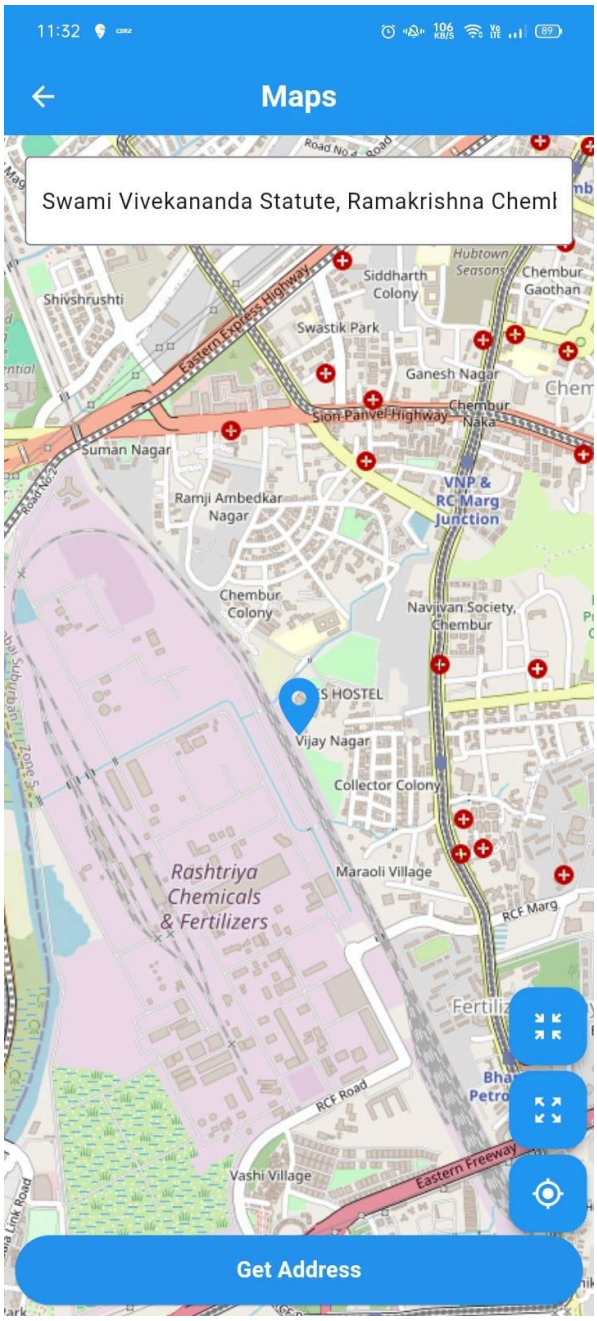
Gestures play a fascinating role in Flutter, enabling interaction with mobile apps and other touch-based devices. In essence, gestures encompass physical actions or movements made by users to control their devices. Examples of gestures include sliding a finger across the screen to unlock it, tapping a button, or holding an app icon to drag it across screens. To facilitate gesture handling, Flutter offers the GestureDetector widget, designed to provide robust support for various types of gestures. The GestureDetector is a non-visual widget, solely dedicated to detecting user gestures. Acting as a stateless widget, it incorporates parameters in its constructor to respond to different touch events.

```
Container(
     padding: const EdgeInsets.symmetric(
          horizontal: 32,
     ),
     child: GestureDetector(
          onTap: loginUser,
          child: Container(
          child: _isLoading
          ? const Center(
               child: CircularProgressIndicator(
                    color: primaryColor,
               ),
          )
          : const Text(
               'Log In',
               style: TextStyle(
                    fontSize: 16.0,
                    fontWeight: FontWeight.bold,
               ),
          ),
```

```
onPressed: () {
  Navigator.pushNamed(context, InitScreen.routeName);
},
child: const Text("Back to home"),
  // ElevatedButton
// Padding
t Spacer(),
```

Name: Aaqueeb Pinjari Roll No.: 56 Class: D15B





• Conclusion:

So, in conclusion, we've learned how to use navigation, routing, and gestures in the Login Screen, Sign Up Screen, and Home Screen of my Flutter app. It's like we've figured out how to make our app move smoothly between different screens and respond to cool finger movements on the screen. It's been a cool journey in understanding how to make our app more user-friendly and interactive. Excited to keep exploring and making Flutter app even better!