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

Theory:

a. Navigation: In Flutter, navigation involves moving between different screens or "routes" within an app. This can be accomplished using the Navigator widget, which manages a stack of routes. When navigating to a new screen, you push a new route onto the stack, and when navigating back, you pop the current route off the stack.
Flutter provides various navigation methods like
1. **Navigator.push()** to push a new route onto the stack
2. **Navigator.pop()** to remove the current route from the stack
3. **Navigator.pushNamed()** to navigate to a named route.

b. **Routing**: Routing in Flutter refers to the process of defining and managing named routes for different screens or pages in the app. Named routes provide a way to navigate to a specific screen using a unique identifier rather than directly referencing the widget class. This improves code readability, organization, and maintenance, especially in larger apps. Routes are typically defined in the **MaterialApp widget** using the routes parameter, where each route is associated with a corresponding widget.

c. Gestures: Gestures in Flutter enable users to interact with the app through touch-based actions like taps, swipes, pinches, and long-presses. Flutter provides a comprehensive set of gesture detection widgets such as **GestureDetector**, **InkWell**, and **LongPressGestureDetector**, which allow developers to detect and handle various user gestures. For example, you can use GestureDetector to detect taps, drags, and other gestures on any widget, and then respond to those gestures by invoking specific actions or navigation events.

Process:
1. Define Routes: Begin by defining routes for each screen in your app. This involves creating a
mapping between route names and corresponding widget classes.
2. Navigate Between Screens: Use navigation methods to move between screens based on user interactions, such as button clicks or gestures. You can push new routes onto the stack, pop routes off the stack, or replace routes as needed
3. Pass Data Between Screens :
Use route parameters to pass data between screens when navigating. This allows screens to communicate with each other and update their UI accordingly. For example, passing a user ID from a login screen to a profile screen to fetch and display user-specific data.
4. Handle Navigation Events :
Implement logic to handle navigation events and respond to user actions appropriately.

This may involve:

- Showing loading indicators while processing
- Validating user input before proceeding
- Performing background tasks (like saving data) before navigating to the next screen

5. Handle Navigation Back :

Consider how users will navigate back to previous screens using the system back button or gestures.

Ensure your app:

- Handles the back action correctly
- Preserves the expected state
- Provides a consistent and smooth user experience across all navigation paths

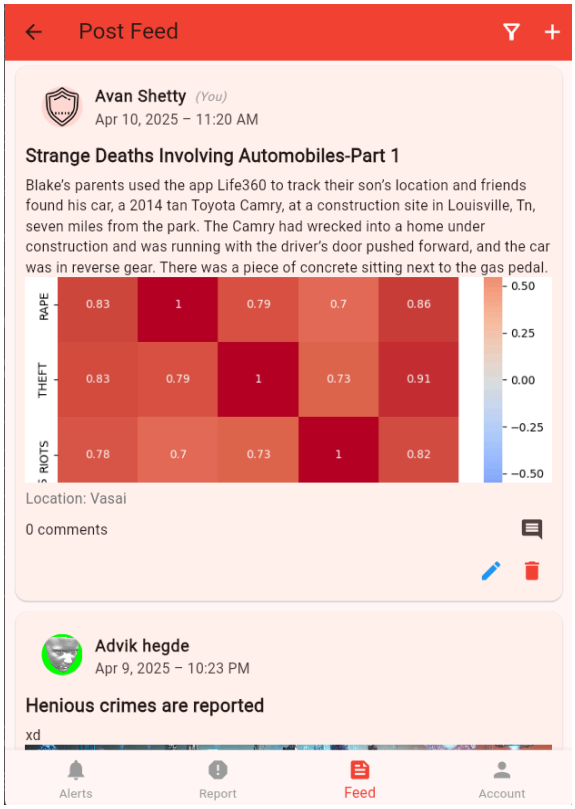
Code :

1. **Navigation and Routing**

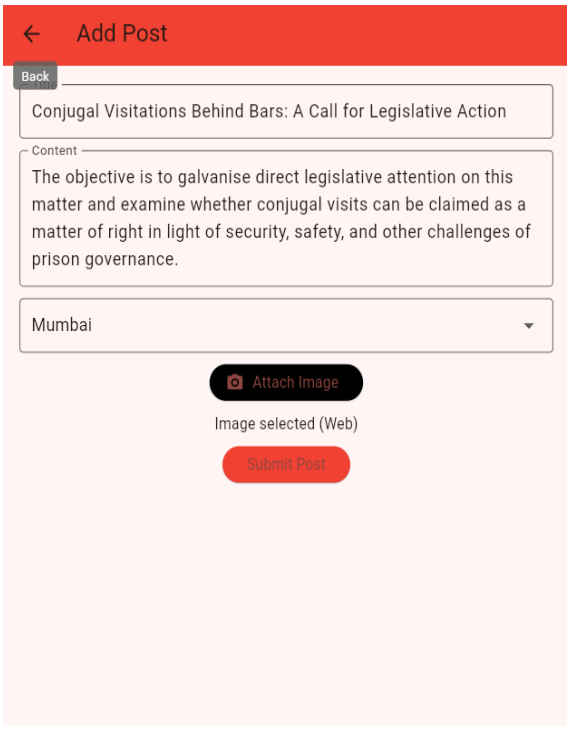
```
IconButton(
  icon: Icon(Icons.add, color: Colors.white),
  onPressed: () {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => AddPostScreen()),
    );
  },
),
```

This IconButton in the AppBar triggers navigation to the AddPostScreen when pressed.
Navigator.push adds a new route to the navigation stack, displaying the AddPostScreen.
The MaterialPageRoute defines the transition to the new screen.

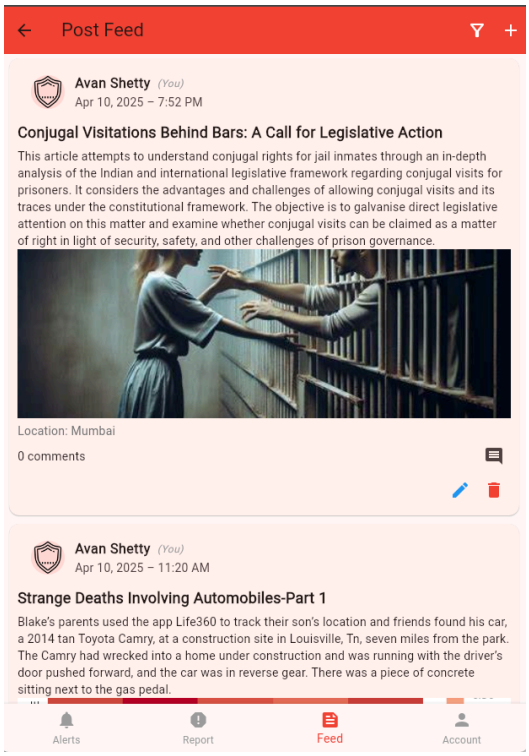
Post feed



After clicking add icon renders add_post



Once submitted then you can see the posts

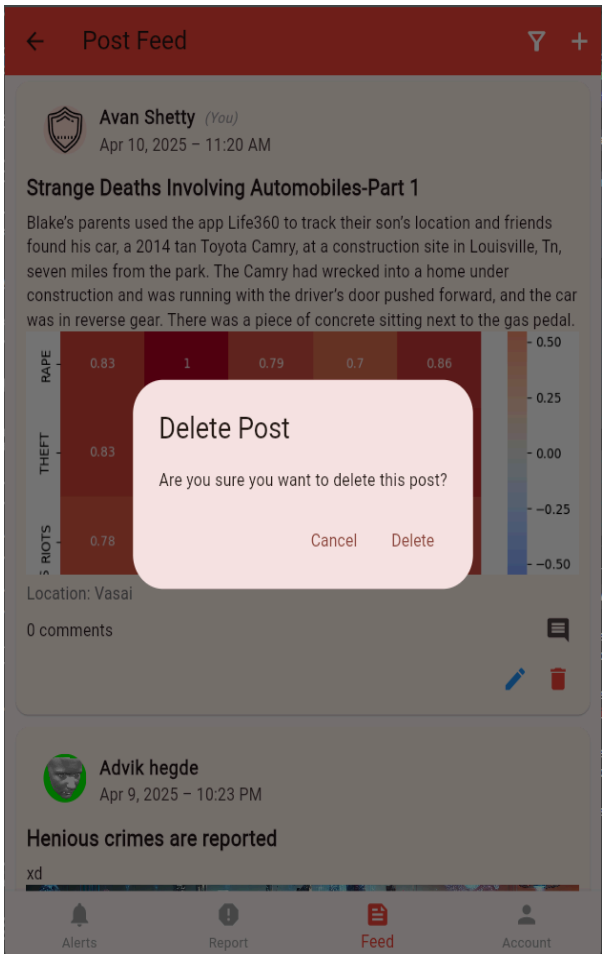


2. **Gestures :** Gestures are handled via interactive widgets like IconButton that respond to taps.

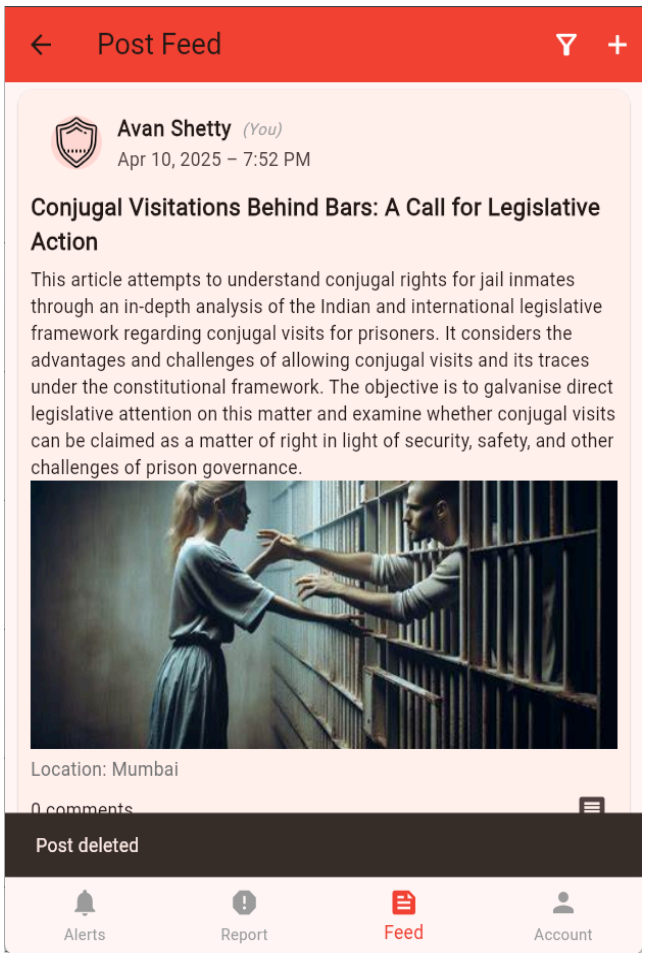
```
IconButton(
        icon: Icon(Icons.delete, color: Colors.red),
          onPressed: () async {
              bool? confirm = await showDialog(
                    context: context,
                    builder: (context) => AlertDialog(
                      title: Text("Delete Post"),
                      content: Text(
                          "Are you sure you want to delete this post?"),
                  actions: [
                        TextButton(
                          onPressed: () =>  Navigator.pop(context, false),
                          child: Text("Cancel"),
                        ),
                        TextButton(
                          onPressed: () =>
                              Navigator.pop(context, true),
                          child: Text("Delete"),
                        ),
                    ],
                ),
              );

                if (confirm == true) {
                    try {
                        await post.reference.delete();
                        ScaffoldMessenger.of(context).showSnackBar(SnackBar(
                                content: Text("Post deleted")));
                        } catch (error) {
                          ScaffoldMessenger.of(context)
                              .showSnackBar(SnackBar(
                                  content: Text(
                                      "Delete failed: $error")));
                    }
                }
            },
        ),
```

This IconButton detects a tap gesture and triggers a confirmation dialog using showDialog. The dialog itself contains TextButton widgets that also respond to taps (**onPressed**), allowing the user to confirm or cancel the deletion. This combines gestures with navigation (**dialog display**) and async operations.

On clicking the delete post button



Once clicked on delete the post gets deleted as u can see below



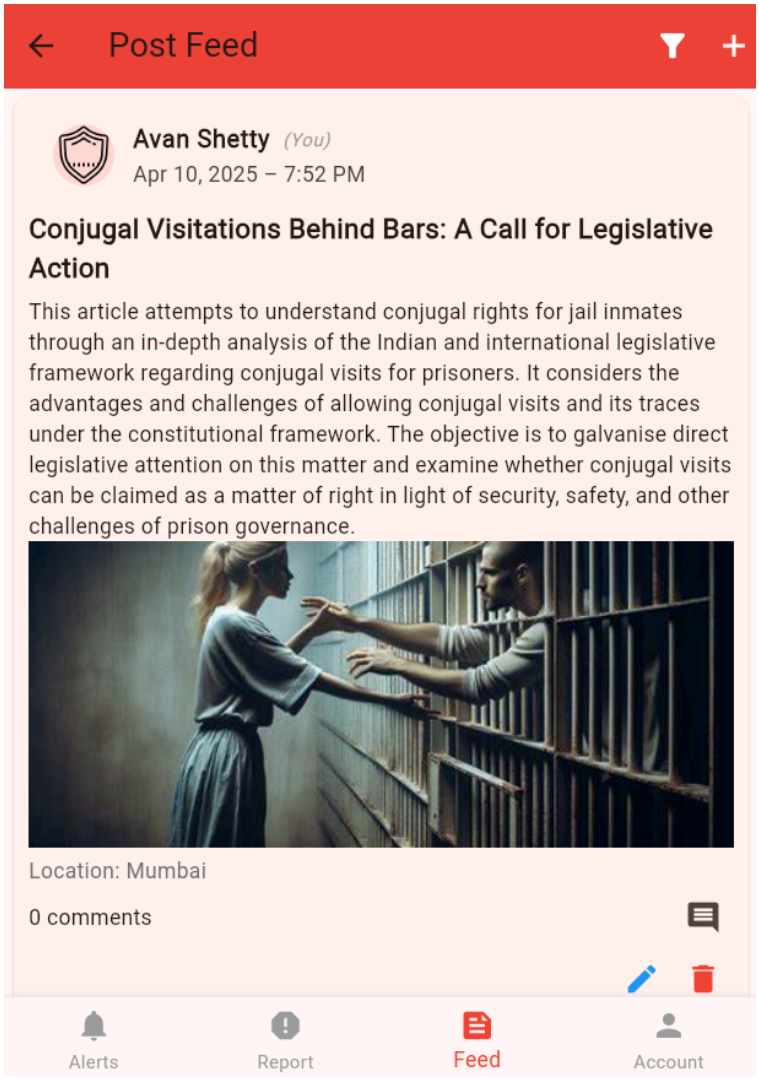
3. **Filter Button** (Toggle My Posts):

```
IconButton(
  icon: Icon(
    _showOnlyMyPosts ? Icons.filter_alt : Icons.filter_alt_outlined,
    color: Colors.white,
  ),
  onPressed: () {
    setState(() {
      _showOnlyMyPosts = !_showOnlyMyPosts;
    });
```

```
  },
  tooltip: _showOnlyMyPosts ? 'Show all posts' : 'Show only my posts',
),
```

This IconButton in the AppBar responds to a tap gesture to toggle between showing all posts or only the current user's posts. The setState call updates the UI, demonstrating a gesture-driven state change.

As i deleted one post i can see the remaining post using the filter button



Conclusion :

This experiment demonstrated how to implement navigation, routing, and gesture handling in a Flutter app. Using Navigator and named routes, we enabled smooth transitions between screens. Gesture-based interactions, like taps on IconButton, allowed users to perform actions such as adding, deleting, and filtering posts. These features together enhanced the app's interactivity and provided a seamless user experience.