CMPS 312 Mobile Application Development
Lab 7 – State Management and Navigation

Objective

In this lab, you will practice building a Flutter application that uses modern state management and navigation techniques. Specifically, you will:

- Use the flutter_riverpod package to manage application state, making it easy to handle changes in data and UI across different screens.
- Implement navigation between app screens using go_router, a flexible and declarative routing library for Flutter.
- Add Navigation UI components, such as the BottomAppBar, to enhance user experience and facilitate easy navigation within the app.
- Pass and handle arguments between screens using both query parameters **(e.g., ?id=1)** and path parameters **(e.g., /details/1),** enabling the app to respond dynamically based on user input and navigation routes.

Overview

The lab is divided into two parts:

Part A: Navigation using go_router : Set up navigation between various screens using go_router, configure routes and pass arguments. Integrate navigation UI components like BottomAppBar for a better user experience.

Part B: State Management using flutter_riverpod : Use Riverpod for managing the app state. Create state notifiers to handle data such as banking accounts and transfers, and observe changes using providers.

By the end of the lab, you will have a clear understanding of how to integrate state management and navigation in a Flutter application using Riverpod and go_router.

Part A: Navigation using go_router

In this part, you will implement navigation between various screens in your Flutter application using the go_router package. The navigation structure will include different routes such as Home, Deposit, Transfer, Transactions, and Account screens, and you will configure routes using both path and query parameters. You will also learn how to structure nested routes and shell routes to maintain a consistent UI structure.

Task 1: Install the go_router Package

1) Open the **pubspec.yaml** file in the root directory of your project.

2) Add the **go_router** dependency under the dependencies section:

```
dependencies:
  go_router: ^14.3.0
```

Task 2: Create the Navigation Configuration in app_router.dart

1) **Create a new file** named app_router.dart inside the lib/router folder.

2) **Define Route Names and Paths**:
   a) Create a class `AppRouter` to hold the route names and paths for each screen.
   b) Define the following routes

Route Name	Path	Description
home	/home	Represents the Home screen.
deposit	/deposit/:accountNo	Represents the Deposit screen. Accepts an accountNo as a path parameter.
deposit	/withdraw/:accountNo	Represents the Withdraw screen. Accepts an accountNo as a path parameter.
transfer	/transfer	Represents the Transfer screen.
transactions	/transactions	Represents the Transactions screen.
account	/account	Represents the Account screen.
newTransfer	/transfer/new	A nested route under the Transfer route for initiating a new transfer.

3) **Set Up the GoRouter Configuration**:
   a) Set the initial route to the Home screen (/).
   b) Create a `GoRouter` instance and configure the routes defined above.
   c) Use a `ShellRoute` to wrap the routes that share a common UI structure (e.g., routes that share a **BottomNavigationBar** and **AppBar**).
   d) Use the builder function of the GoRoute object to render the appropriate screen for each route

Task 3: Update main.dart to Use the GoRouter Configuration

1) Open the main.dart file.
2) Update the MaterialApp Widget:
   a) Replace the existing MaterialApp configuration to use the GoRouter configuration created in app_router.dart.
   b) Set the **routerConfig** property to **AppRouter.router** to use the defined routes.

```
@override
  Widget build(BuildContext context) {
    return MaterialApp.router(
      routerConfig: AppRouter.router,     .....
```

Task 4 . Implement Navigation Logic

1) Navigating Between Screens:

a) Use the **context.goNamed()** function to navigate between screens by their route name. For example, to navigate to the **newTransfer** route:

```
context.goNamed(AppRouter.newTransfer.name);
```

2) Passing Parameters:

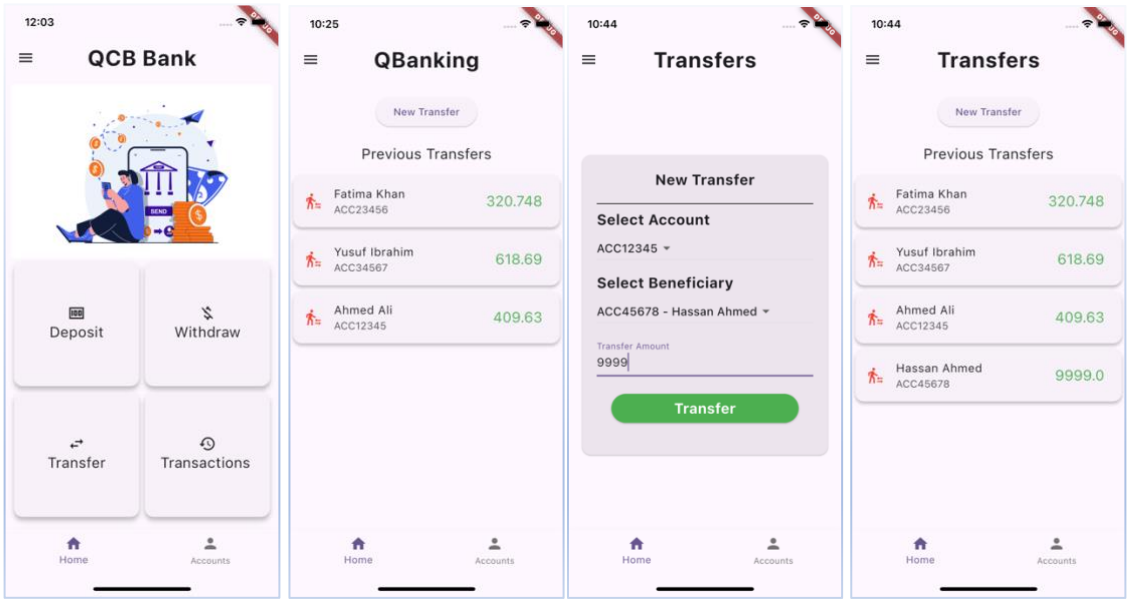
a) For routes that require parameters, pass the values using **pathParameters** or **queryParameters**.

b) To navigate to the Deposit screen with an account number:

```
context.goNamed(AppRouter.deposit.name, pathParameters:
{'accountNo': '12345'});
```

3) Handling Nested Routes:

- Use nested routes within the **ShellRoute** to handle sub-routes such as navigating from the **Transfer** screen to the **New Transfer screen**. Example scenario of transfer is shown below.



Part A: State Management using flutter_riverpod

In this part, you will implement state management for the banking application using the **flutter_riverpod** package. You will set up Providers and Notifiers to manage the state of the application, such as accounts, transfers, and beneficiaries. You will use these state management tools to control and observe changes in the app's data across various screens.

The project already includes model classes for Account, Transfer, and Beneficiary inside the model folder. You will create Providers and Notifiers to manage the state of these models and use them in different screens.

Task 1: Install the flutter_riverpod Package

1) **Open the pubspec.yaml file** in the root directory of your project.
2) **Add the flutter_riverpod dependency** under the dependencies section:

```
dependencies:
  flutter_riverpod: ^2.0.0
```

Task 2: Create State Notifiers and Providers

1) **Navigate to the `providers` folder** in your project directory.
2) **Create the following files** inside the `providers` folder to manage the state of the banking application:

   - `account_provider.dart`
   - `beneficiary_provider.dart`
   - `transfer_provider.dart`
   - `title_provider.dart`

3) **Define the Notifiers and Providers**:
   a) In each file, implement a `Notifier` class to manage the state of the respective model (e.g., `Account`, `Beneficiary`, `Transfer`).
   b) Create a corresponding `NotifierProvider` to expose each notifier to the rest of the app. Example Implemenation of the AccountNotifier and AccountNotifier Provider .
      i) Navigate **to `account_provider.dart`** and implement the following:
      ii) Create a `Notifier` class named `AccountNotifier` that extends `Notifier<List<Account>>`.
      iii) Define the following methods inside the class:
         (1) `build()`: Initializes the state and loads initial data.
         (2) `initializeAccounts()`: Loads data from `assets/data/accounts.json` and parses it into a list of `Account` objects.
         (3) `addAccount(Account account)`: Adds a new account to the state.
         (4) `updateAccount(Account account)`: Updates an existing account in the state based on the account number.
         (5) **Define the `NotifierProvider`**:
         (6) Use `NotifierProvider` to create and expose the `AccountNotifier`:

```
final accountNotifierProvider =
NotifierProvider<AccountNotifier, List<Account>>(() =>
AccountNotifier());
```

   c) Implement the remaining Notifiers and Providers.

```
providers
    ├── title_provider.dart
    ├── beneficiary_provider.dart
    ├── transfer_provider.dart
```

Task 3: Set Up Providers in the Application

a) **Open `main.dart`** and wrap the app with `ProviderScope` to enable Riverpod state management.
b) Update the `main` function in `main.dart`:

```
void main() { runApp(const ProviderScope(child:
QBankingApp()));}
```

c) **Navigate to the screens folder** and open a screen file (e.g., home_screen.dart).
d) **Access the providers** using the ConsumerWidget and ref.watch() to observe and display state changes.
For example, in home_screen.dart, access the account list using:

```
final accounts = ref.watch(accountNotifierProvider);
```

e) **Modify State**: Use the ref.read() function to interact with the provider's notifier and modify the state. For example, to add a new account:

```
ref.read(accountNotifierProvider.notifier).addAccount(newAccou
nt)
```