Module 3: Navigation and Routing

An app with only one screen is just a poster. To build a real, interactive application, we need to move between different screens, pass data between them, and manage the user's journey. This process is called **navigation** or **routing**. In this module, we will explore how Flutter handles this, select a modern, powerful routing solution, and implement it in both our projects.

1. Learning Objectives

- Understand the "navigation stack" metaphor (push/pop) for screen management.
- Compare and contrast the two main navigation philosophies in Flutter: imperative vs. declarative.
- Analyze the pros and cons of Flutter's built-in `Navigator` versus a package-based solution like `go_router`.
- Implement a complete, URL-based routing solution using the `go_router` package.
- Define application routes in a centralized, easy-to-manage file.
- Navigate from a list item to a detail screen when the user taps it.
- Pass data (like a unique product or task ID) to the new screen as part of the navigation path.

2. Core Concepts

A. Fundamental Theory: The Navigation Stack

Imagine a stack of physical playing cards on a table. The first card you put down is the bottom of the stack. This is your app's home screen.

- **Push:** To show a new screen (e.g., a details page), you **push** a new card onto the top of the stack. The user now only sees this top card.
- **Pop:** To go back, you **pop** the top card off the stack, revealing the card that was underneath it.

Flutter's `Navigator` works exactly like this. It maintains a stack of "routes" (your screens). When you navigate forward, you `push` a new route. When you press the back button, Flutter automatically `pop`s the current route. Understanding this simple push/pop stack model is the key to all navigation in Flutter.

B. Exploration of Options (Comparative Analysis): How to Manage the Stack

There are two primary ways to tell the `Navigator` what to do.

- **Option 1: Imperative Navigation (`Navigator 1.0`)**
    - **What it is:** The traditional, built-in method where you give direct, step-by-step commands to the navigator.
    - **Analogy:** You are a backseat driver giving turn-by-turn directions. "Push this screen now." (`Navigator.push(...)`). "Okay, now go back." (`Navigator.pop(...)`).
    - **How it works:** You call methods directly from your UI code, usually inside an `onPressed` callback. `Navigator.of(context).push(MaterialPageRoute(builder: (context) => DetailsScreen()))`.
    - **Pros:**
        - **Conceptually Simple:** For basic apps, `push` and `pop` are very easy to understand.
        - **Built-in:** No external packages required.
    - **Cons:**
        - **Not URL-Friendly:** This is a major drawback. You cannot handle web URLs (`myapp.com/products/123`) or deep links from notifications easily.
        - **Tightly Coupled:** Navigation logic gets scattered throughout your UI widgets' `onPressed` handlers. This makes your code messy and hard to reason about.
        - **Poor for Complex Logic:** Advanced scenarios like redirecting a user if they aren't logged in become very difficult to manage.
- **Option 2: Declarative Navigation (with `go_router`)**
    - **What it is:** A modern approach where you *declare* all your app's possible destinations (routes) upfront, like a site map. To navigate, you simply change the current state (the URL path), and the UI *reacts* by displaying the correct screen.
    - **Analogy:** You are using a GPS. You don't give it turn-by-turn directions. You just declare your destination: "I want to go to 123 Main Street." The GPS then figures out the entire path and handles all the turns for you.
    - **How it works:** You define a list of `GoRoute` objects with paths like `/` and `/product/:id`. To navigate, you simply call `context.go('/product/123')`.
    - **Pros:**
        - **URL-Based & Scalable:** The holy grail of routing. Deep linking, web URLs, and a clear app structure are built-in from day one.
        - **Decoupled & Centralized:** All your routes are defined in one place, providing a clear map of your entire app. UI widgets just say "go here" without needing to know *how* to build the destination screen.

        - **Handles Complex Scenarios:** Redirects (e.g., login walls) and nested navigation are handled elegantly.
    - **Cons:**
        - **Requires a Package:** You need to add `go_router` to your `pubspec.yaml`.
        - **Slightly More Upfront Setup:** You have to create the router configuration file before you can navigate.

C. Our Chosen Approach & Rationale

For both **ShopSphere** and **TaskFlow**, we will use **`go_router`**, the declarative routing package officially supported by the Flutter team.

Rationale: Starting with `go_router` is an investment in our projects' futures. Imperative routing is a dead end for any app that might one day run on the web or need to handle notifications that link to specific content.

By choosing `go_router`, we are:

1. **Building Professional-Grade Apps:** Our architecture will be clean, scalable, and ready for real-world features from the very beginning.
2. **Writing Maintainable Code:** Centralizing our routes makes it easy for anyone (including our future selves) to understand the app's entire navigation flow at a glance.
3. **Learning the Modern Standard:** The Flutter ecosystem is rapidly standardizing on declarative, URL-based routing. Mastering `go_router` is a key skill for any modern Flutter developer.

The minimal cost of the initial setup is vastly outweighed by the long-term benefits in code quality and capability.

3. Practical Application: ShopSphere

Goal: Make each product card in our `ListView` tappable. When tapped, navigate to a new "Product Details" screen, passing the specific ID of the product that was tapped.

Step-by-Step Implementation:

1. **Add Dependency:** In `shop_sphere/pubspec.yaml`, add `go_router: ^12.1.1` (or the latest version) under `dependencies`. Run `flutter pub get`.
2. **Create Router File:** In `lib/`, create `app_router.dart`. Here, we'll define a `GoRouter` instance with two routes: `/` for the `HomePage` and `/product/:id` for the details page. The `:id` is a dynamic path parameter.
3. **Integrate Router:** In `main.dart`, change `MaterialApp` to `MaterialApp.router` and provide it with our router's configuration.

4. **Create Details Page:** Create a new folder `lib/pages/`. Inside, create `product_details_page.dart`. This `StatelessWidget` will accept the `productId` from the route and display it.
5. **Make `ProductCard` Tappable:** In `widgets/product_card.dart`, wrap the `Card` in a `GestureDetector` widget. In its `onTap` property, call `context.go('/product/${product.id}')` to perform the navigation.

4. Practical Application: TaskFlow

Goal: Make each task tappable to navigate to an "Edit Task" screen. We will also add a Floating Action Button (FAB) to navigate to the same screen but for creating a *new* task.

Step-by-Step Implementation:

1. **Add Dependency:** Add `go_router` to `task_flow/pubspec.yaml` and run `flutter pub get`.
2. **Create Router File:** Create `lib/app_router.dart`. We'll define routes for `/` (our `HomePage`) and `/task/:id`. We'll treat the special `id` value of "new" as the trigger for creating a new task.
3. **Integrate Router:** Update `main.dart` to use `MaterialApp.router`.
4. **Create Edit Page:** Create `lib/pages/edit_task_page.dart`. This page will be a `StatefulWidget` so it can handle a `TextEditingController` for a form field. It will accept an optional `taskId`.
5. **Update `TaskItem`:** The `ListTile` in `task_item.dart` already has an `onTap`. We will change it to navigate to `/task/${task.id}`.
6. **Add FAB:** In `home_page.dart`, add a `FloatingActionButton` to the `Scaffold`. Its `onPressed` will call `context.go('/task/new')`.

5. Full Code for New/Updated Files

`shop_sphere/pubspec.yaml` (Partial change)

```
dependencies:
  flutter:
    sdk: flutter
  go_router: ^12.1.1 # Add this line
```

`shop_sphere/lib/app_router.dart` (New)

```
import 'package:go_router/go_router.dart';
import 'package:shop_sphere/home_page.dart';
import 'package:shop_sphere/pages/product_details_page.dart';
```

```
final goRouter = GoRouter(
  initialLocation: '/',
  routes: [
    GoRoute(
      path: '/',
      builder: (context, state) => const HomePage(),
    ),
    GoRoute(
      path: '/product/:id',
      builder: (context, state) {
        final productId = state.pathParameters['id']!;
        return ProductDetailsPage(productId: productId);
      },
    ),
  ],
);
```

`shop_sphere/lib/main.dart` (Refactored)

```
import 'package:flutter/material.dart';
import 'package:shop_sphere/app_router.dart';

void main() {
  runApp(const ShopSphereApp());
}

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

  @override
  Widget build(BuildContext context) {
    // Change MaterialApp to MaterialApp.router
    return MaterialApp.router(
      routerConfig: goRouter,
      debugShowCheckedModeBanner: false,
      title: 'ShopSphere',
      theme: ThemeData(
        useMaterial3: true,
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),
      ),
    );
```

```
  }
}
```

shop_sphere/lib/pages/product_details_page.dart (New)

```
import 'package:flutter/material.dart';
import 'package:shop_sphere/data/mock_data.dart';
import 'package:shop_sphere/models/product.dart';

class ProductDetailsPage extends StatelessWidget {
  final String productId;

  const ProductDetailsPage({super.key, required this.productId});

  @override
  Widget build(BuildContext context) {
    // In a real app, you'd fetch the product from your state management
    // solution or API using the productId. Here, we'll find it in our mock
list.
    final Product product =
        mockProducts.firstWhere((p) => p.id == productId, orElse: () {
      // Return a dummy product or handle error if not found
      return const Product(id: 'error', name: 'Product Not Found', price: 0,
imageUrl: '');
    });

    return Scaffold(
      appBar: AppBar(
        title: Text(product.name),
      ),
      body: SingleChildScrollView(
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            Image.network(
              product.imageUrl,
              width: double.infinity,
              height: 300,
              fit: BoxFit.cover,
            ),
            Padding(
```

```
              padding: const EdgeInsets.all(16.0),
              child: Text(
                'Product Details for ID: $productId',
                style: const TextStyle(fontSize: 22, fontWeight:
FontWeight.bold),
              ),
            ),
            // More product details would go here
          ],
        ),
      ),
    );
  }
}
```

`shop_sphere/lib/widgets/product_card.dart` (Updated)

```
//... imports
import 'package:go_router/go_router.dart';

class ProductCard extends StatelessWidget {
//... constructor

  @override
  Widget build(BuildContext context) {
    // Wrap the Card with a GestureDetector to make it tappable
    return GestureDetector(
      onTap: () {
        // Use go_router to navigate to the product details page,
        // passing the product's ID in the URL path.
        context.go('/product/${product.id}');
      },
      child: Card(
        // ... all the previous Card content remains the same ...
      ),
    );
  }
}
```

`task_flow/pubspec.yaml` (Partial change)

```
dependencies:
```

```
  flutter:
    sdk: flutter
  go_router: ^12.1.1 # Add this line
```

`task_flow/lib/app_router.dart` (New)

```
import 'package:go_router/go_router.dart';
import 'package:task_flow/home_page.dart';
import 'package:task_flow/pages/edit_task_page.dart';

final goRouter = GoRouter(
  initialLocation: '/',
  routes: [
    GoRoute(
      path: '/',
      builder: (context, state) => const HomePage(),
    ),
    GoRoute(
      // This route handles both creating a new task and editing an existing
one.
      path: '/task/:id',
      builder: (context, state) {
        final taskId = state.pathParameters['id']!;
        return EditTaskPage(taskId: taskId);
      },
    ),
  ],
);
```

`task_flow/lib/main.dart` (Refactored)

(This will be identical in structure to ShopSphere's new `main.dart`, just using the TaskFlow theme and router)

`task_flow/lib/pages/edit_task_page.dart` (New)

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

class EditTaskPage extends StatefulWidget {
  final String taskId;
  const EditTaskPage({super.key, required this.taskId});

  @override
```

```
  State<EditTaskPage> createState() => _EditTaskPageState();
}

class _EditTaskPageState extends State<EditTaskPage> {
  // A controller to manage the text in a TextField.
  late final TextEditingController _titleController;

  bool get _isCreatingNew => widget.taskId == 'new';

  @override
  void initState() {
    super.initState();
    _titleController = TextEditingController();

    if (!_isCreatingNew) {
      // In a real app, you'd fetch the task data here and populate the
controller.
      _titleController.text = "Editing Task ID: ${widget.taskId}";
    }
  }

  @override
  void dispose() {
    _titleController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(_isCreatingNew ? 'Add New Task' : 'Edit Task'),
      ),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: [
            TextField(
              controller: _titleController,
```

```
              decoration: const InputDecoration(
                labelText: 'Task Title',
                border: OutlineInputBorder(),
              ),
            ),
            // More form fields would go here...
          ],
        ),
      ),
    );
  }
}
```

`task_flow/lib/home_page.dart` (Updated)

```
//... imports
import 'package:go_router/go_router.dart';

class HomePage extends StatelessWidget {
  // ... constructor
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('My Tasks')),
      // Add the FloatingActionButton
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          // Navigate to the edit screen with a special 'new' ID.
          context.go('/task/new');
        },
        child: const Icon(Icons.add),
      ),
      body: ListView.builder(
        // ... ListView.builder code is the same
      ),
    );
  }
}
```

`task_flow/lib/task_item.dart` (Updated `onTap`)

```
//... imports
import 'package:go_router/go_router.dart';
```

```
// ... inside _TaskItemState
// The existing `_handleTap` function needs to be replaced with navigation
logic.

  @override
  Widget build(BuildContext context) {
    return Card(
      // ...
      child: ListTile(
        // ... leading, title, subtitle, trailing
        onTap: () {
          // The primary action is now navigation. Toggling the checkbox
          // will be handled on the edit page itself.
          context.go('/task/${widget.task.id}');
        },
      ),
    );
  }
```

6. Assignment/Challenge

1. **For ShopSphere:** In `product_details_page.dart`, the `AppBar` automatically gets a back button from `go_router`. Override this. Add `leading: IconButton(...)` to the `AppBar` and in its `onPressed`, call `context.pop()` to manually perform the "pop" navigation action.
2. **For TaskFlow:** In `edit_task_page.dart`, add a "Save" `ElevatedButton`. If the user is creating a new task (`_isCreatingNew` is true), its `onPressed` should navigate them back to the home screen using `context.go('/')`. If they are editing, it should use `context.pop()` to go back to the previous screen. This teaches conditional navigation logic.
3. **Bonus - Type-Safe Routes:** As it stands, we are passing IDs as strings: `/product/123`. A more advanced (but safer) technique is to pass the entire `Product` object. In your `GoRouter` definition, you can use the `extra` parameter: `context.go('/product/${product.id}', extra: product)`. Then, on the details page, you can retrieve it with `final product = state.extra as Product;`. Try to refactor `ShopSphere` to use this method.