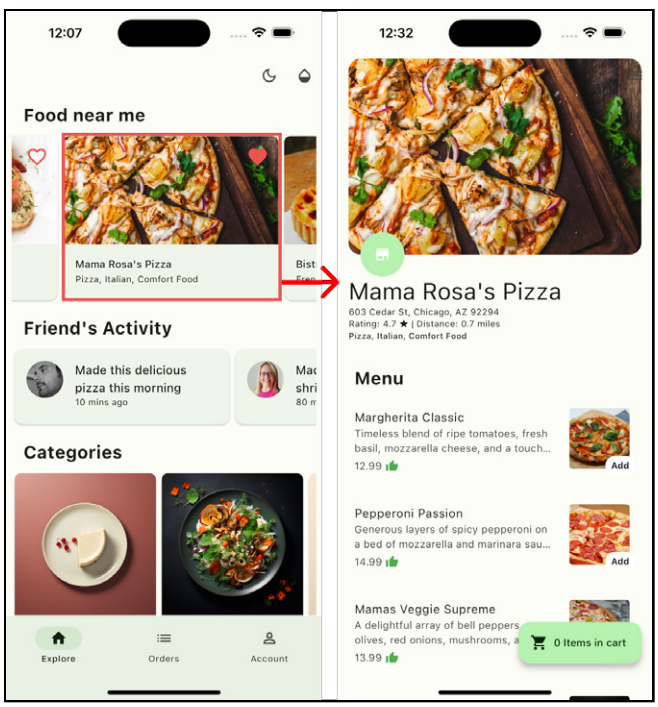
Chapter 8: Routes & Navigation

By Vincent Ngo

Navigation, or how users switch between screens, is an important concept to master. Good navigation keeps your app organized and helps users find their way around without getting frustrated.

In the previous chapter, you got a taste of navigation where users tapped on a restaurant to view its menu items as shown below:



But this uses the imperative style of navigation, known as **Navigator 1.0**. In this chapter, you'll learn to navigate between screens the **declarative** way.

You'll cover the following topics:

- Overview of `Navigator` 1.0.
- Overview of `Router API`.
- How to use **go_router** to handle **routes** and **navigation**.

By the end of this chapter, you'll have everything you need to navigate to different screens!

> **Note**: If you'd like to skip straight to the code, jump to **Getting Started**. If you'd like to learn the theory first, read on!

Introducing Navigation

If you come from an iOS background, you might be familiar with **UINavigationController** from UIKit, or **NavigationStack** from SwiftUI.

In Android, you use **Jetpack Navigation** to manage various fragments.

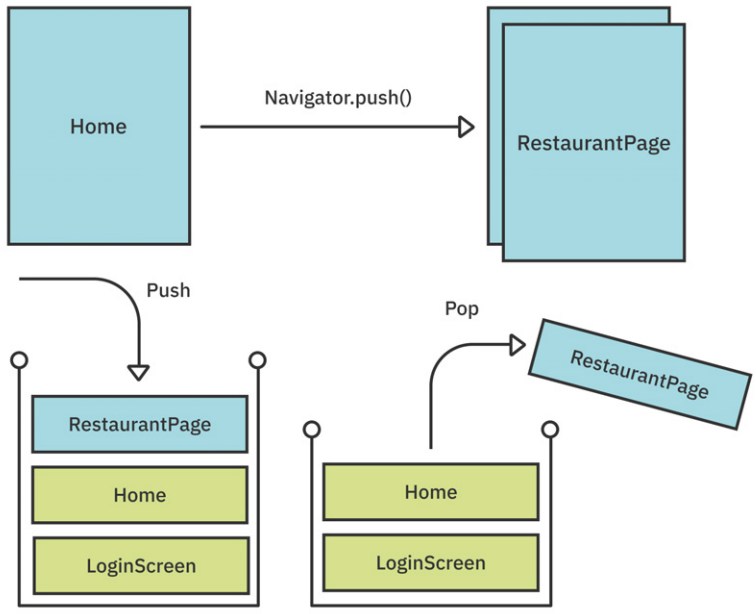
In Flutter, you use a **Navigator** widget to manage your screens or pages. Think of screens and pages as **routes**.

> **Note**: This chapter uses these terms interchangeably because they all mean the same thing.

A **stack** is a data structure that manages pages. You insert the elements **last-in**, **first-out** (LIFO), and only the element at the top of the stack is visible to the user.

For example, when a user views a list of restaurants, tapping a restaurant **pushes** `RestaurantPage` to the top of the stack. Once the user finishes making changes, you **pop** it off the stack.

Here's a top-level and a side-level view of the navigation stack:



Now, it's time for a quick overview of Navigator 1.0.

Navigator 1.0 Overview

Before Flutter 1.22, you could only shift between screens by issuing direct commands like "show this now" or "remove the current screen and go back to the previous one". Navigator 1.0 provides a simple set of APIs to navigate between screens. The most common ones are:

- `push()`: **Adds** a new route on the stack.
- `pop()`: **Removes** a route from the stack.

So, how do you add a navigator to your app?

Most Flutter apps start with `WidgetsApp` as the root widget.

> **Note**: So far, you've used `MaterialApp`, which extends `WidgetsApp`.

`WidgetsApp` wraps many other common widgets that your app requires. Among these wrapped widgets there's a top-level `Navigator` to manage the pages you **push** and **pop**.

Pushing and Popping Routes

To show the user another screen, you need to push a `Route` onto the `Navigator` stack using `Navigator.push(context)`. Here's an example:

```
bool result = await Navigator.push<bool>(
  context,
  MaterialPageRoute<bool>(
    builder: (BuildContext context) =>RestaurantPage(
      restaurant: restaurants[index],
      cartManager: cartManager,
      ordersManager: orderManager,
    )
  ),
);
```

Here, `MaterialPageRoute` returns an instance of your new screen widget. `Navigator` returns the result of the push whenever the screen pops off the stack.

Here's how you pop a route off the stack:

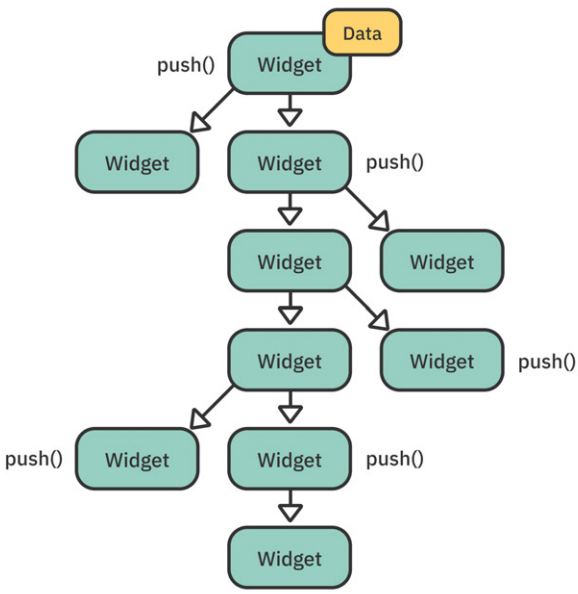
```
Navigator.pop(context);
```

This seems easy enough. So why not just use Navigator 1.0? Well, it has a few disadvantages.

Navigator 1.0's Disadvantages

The imperative API may seem natural and easy to use, but, in practice, it's hard to manage and scale.

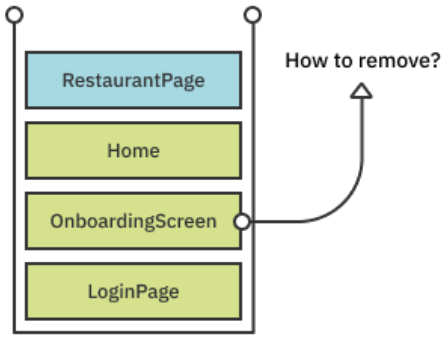
There's no good way to manage your pages without keeping a mental map of where you push and pop a screen.



Imagine a new developer joining your team. Where do they even start? They'd surely be confused.

Moreover, Navigator 1.0 doesn't expose the route stack to developers. It's difficult to handle complicated cases, like adding and removing a screen **between pages**.

For example, in Yummy, you only want to show the Onboarding screen if the user hasn't completed the onboarding yet. Handling that with Navigator 1.0 is complicated.



Another disadvantage is that Navigator 1.0 doesn't update the web **URL** path. When you go to a new page, you only see the base URL, like this: **www.localhost:8000/#/**. Additionally, the web browser's forward and backward buttons may not work as expected.

Finally, the **Back** button on Android devices might not work with Navigator 1.0 when you have nested navigators or add Flutter to your host Android app.

Wouldn't it be great to have a declarative API that solves most of these pain points? That's why **Router API** was designed!

> To learn more about Navigator 1.0, check out the Flutter documentation (https://flutter.dev/docs/cookbook/navigation).

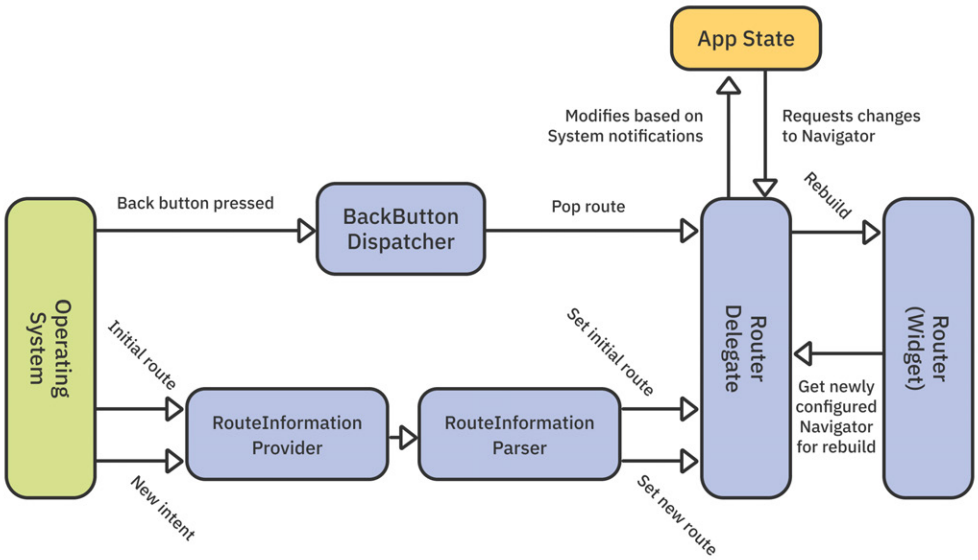
Router API Overview

Flutter 1.22 introduced the **Router API**, a new declarative API that lets you control your navigation stack completely. Also known as Navigator 2.0, the Router API aims to feel more Flutter-like while solving the pain points of Navigator 1.0.

Its main goals include:

- **Exposing the navigator's page stack**: You can now **manipulate** and manage your page routes. More power, more control!
- **Backward compatibility with imperative API**: You can use imperative and declarative styles in the same app.
- **Handling operating system events**: It works better with events like the Android and Web system's **Back button**.
- **Managing nested navigators**: It gives you control over which navigator has **priority**.
- **Managing navigation state**: You can **parse routes** and handle **web URLs** and **deep linking**.

Here are the new abstractions that make up Router's declarative API:



The new API includes the following key components:

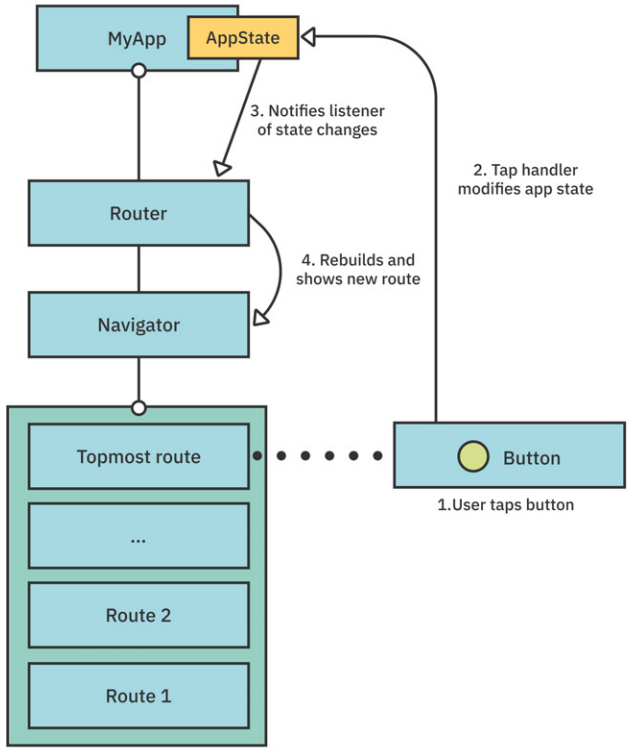
- **Page**: An abstract class that describes the **configuration** for a route.
- **Router**: Handles configuring the list of **pages** the Navigator displays.
- **RouterDelegate**: Defines how the router **listens** for changes to the app state to rebuild the navigator's configuration.
- **RouteInformationProvider**: Provides `RouteInformation` to the router. Route information contains the **location info** and **state** objects to configure your app.
- **RouteInformationParser**: Parses route information into a **user-defined** data type.
- **BackButtonDispatcher**: Reports presses on the platform system's Back button to the router.
- **TransitionDelegate**: Decides how pages transition into and out of the screen.

> **Note**: This chapter will leverage a routing package, **go_router**, to make the Router API easier to use.
>
> If you want to know how to use the vanilla version of the Router API, check out Edition 2.0 (https://www.kodeco.com/books/flutter-apprentice/v2.0/) of this book.

Navigation and Unidirectional Data Flow

As discussed with Navigator 1.0, the imperative API is very basic. It forces you to place `push()` and `pop()` functions all over your widget hierarchy which couples all your widgets! To present another screen, you must place callbacks up the widget hierarchy.

With the new declarative API, you can manage your navigation state **unidirectionally**. The widgets are **state-driven**, as shown below:



Here's how it works:

1. A user **taps** a button.
2. The button handler tells the **app state** to update.
3. The router is a **listener** of the state, so it receives a **notification** when the state changes.
4. Based on the new state changes, the router **reconfigures** the list of pages for the navigator.
5. The navigator detects if there's a new page in the list and handles the **transitions** to show the page.

That's it! Instead of having to build a mental mind map of how every screen presents and dismisses, the state drives which pages appear.

Is Declarative Always Better Than Imperative?

You don't have to migrate or convert your existing code to use the new API if you have an existing project.

Here are some tips to help you decide which is more beneficial for you:

- **For medium to large apps**: Consider using a **declarative API** and a router widget when managing a lot of your navigation state.
- **For small apps**: The **imperative API** is suitable for rapid prototyping or creating a small app for demos. Sometimes push and pop are all you need!

Next, you'll get some hands-on experience with declarative navigation.

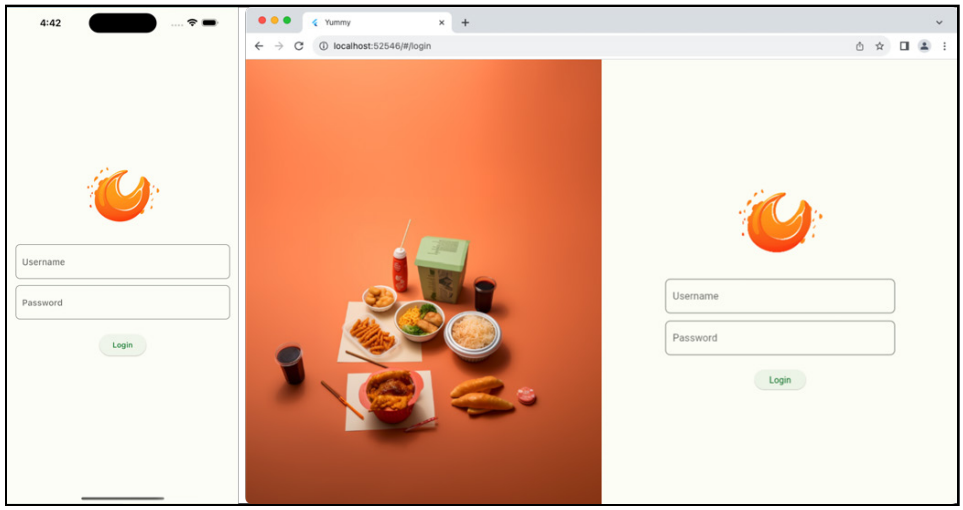
> **Note**: To learn more about Navigator 1.0, check:
>
> • Flutter's Dev Cookbook Tutorials (https://flutter.dev/docs/cookbook/navigation)
>
> • Flutter Navigation: Getting Started (https://www.kodeco.com/4562634-flutter-navigation-getting-started)

Getting Started

Open the **starter** project in Android Studio. Run `flutter pub get` and then run the app.

> **Note**: It's better to start with the starter project rather than continuing with the project from the last chapter because it contains some changes specific to this chapter.

You'll see that **Yummy** only shows the **Login** screen. Of course, it also supports responsive UI on different devices!



Don't worry. You'll connect all the screens soon. You'll build a simple flow that features a **login screen** and an **onboarding** widget before showing the existing tab-based app you've made so far. But first, take a look at some changes to the project files.

Changes to the Project Files

Before you dive into navigation, there are new files in this starter project to help you out.

What's New in the Screens Folder

There are new changes in **lib/** and **lib/screens/**:

- **home.dart**: Now includes a **Profile** button at the top-right for the user to view their profile.
- **screens.dart**: A **barrel** file that groups all the screens into a single import.
- **login_page.dart**: Lets the user log in.
- **account_page.dart**: Lets users check their **profile**, update settings and log out.

Later, you'll use these to construct your authentication UI flow.

What's New in the Models Folder

There are three new model objects in **lib/models/**.

- **models.dart**: A barrel file that groups all the models into a single import.
- **auth.dart**: Manages user **authentication** state, whether they are login in or out.
- **user.dart**: Describes a single **user** and includes information like the user's role, profile picture, full name and app settings.

What's New in the Components Folder

There is one change in **lib/components/**.

- **components.dart**: A barrel file that groups all the components into a single import.

New Packages

There are three new packages in **pubspec.yaml**:

```
  url_launcher: ^6.2.1
  go_router: ^13.0.1
  shared_preferences: ^2.2.2
```

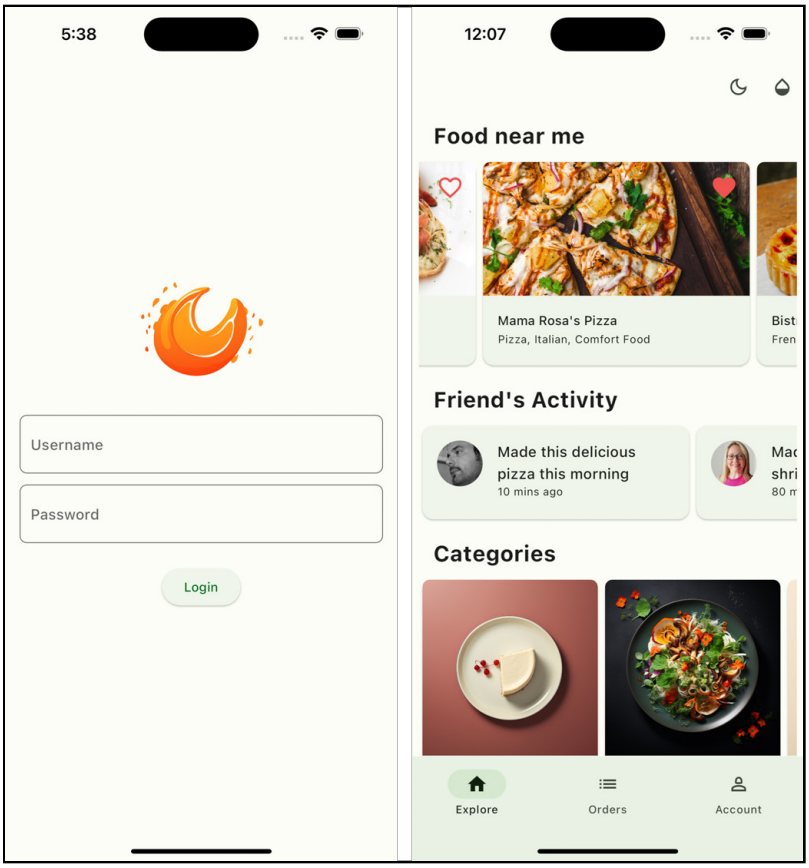
Here's what they do:

- **url_launcher**: A cross-platform library to help launch a **URL**.
- **go_router**: A package built to reduce the complexity of the Router API. It helps developers easily implement **declarative navigation**.
- **shared_preferences**: Wraps platform-specific **persistent storage** for simple data. `AppCache` uses this package to store the login state.

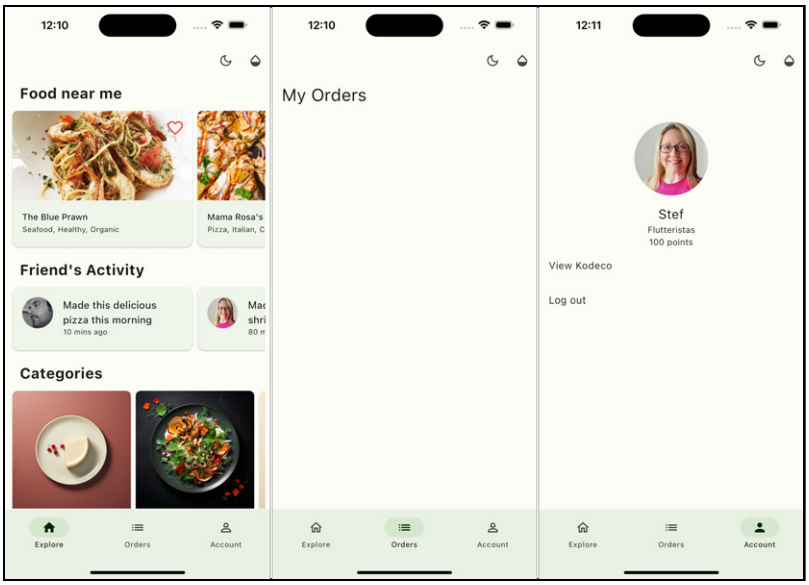
Now that you know what's changed, it's time for a quick overview of the UI flow you'll build in this chapter.

Looking Over the UI Flow

Here are the first two screens you show the user:



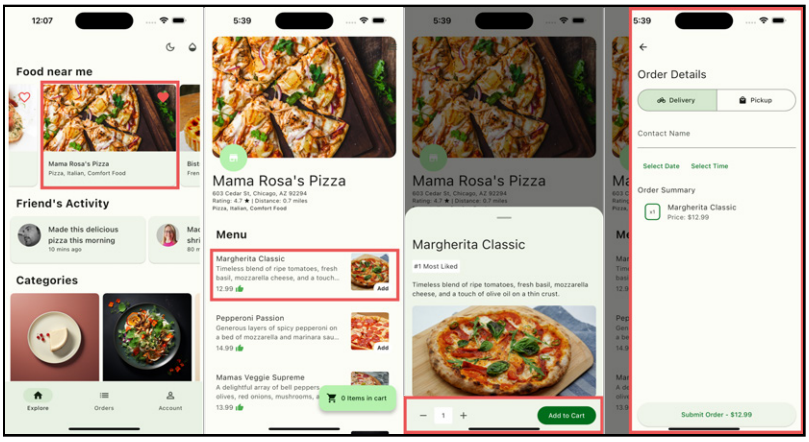
1. When the user launches the app, he must log in by entering their **username** and **password**, then tap **Login**.
2. Once the user logs in, the user goes to the app's **Home**. They can now start using the app.



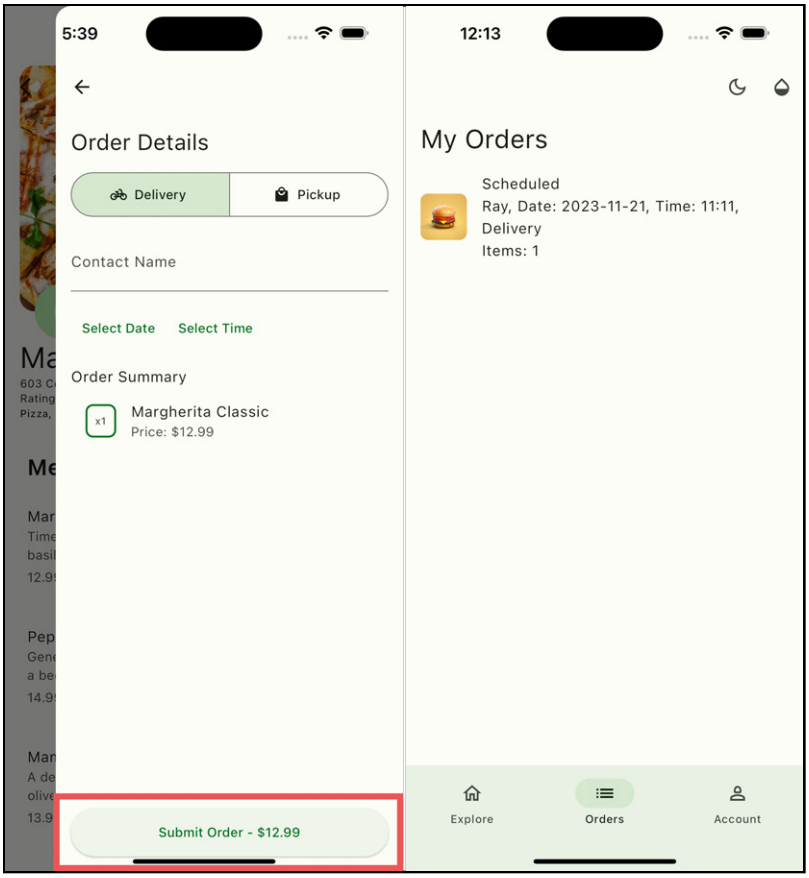
The app presents the user with three tabs with these options:

1. **Home**: View restaurants, friend posts, and food categories.
2. **Orders**: Track all orders submitted.
3. **Account**: View the user's profile and logout.

Next, the user can **tap** on a restaurant to view the menu to order food. They can **select** items to add to their cart and **submit** an order.



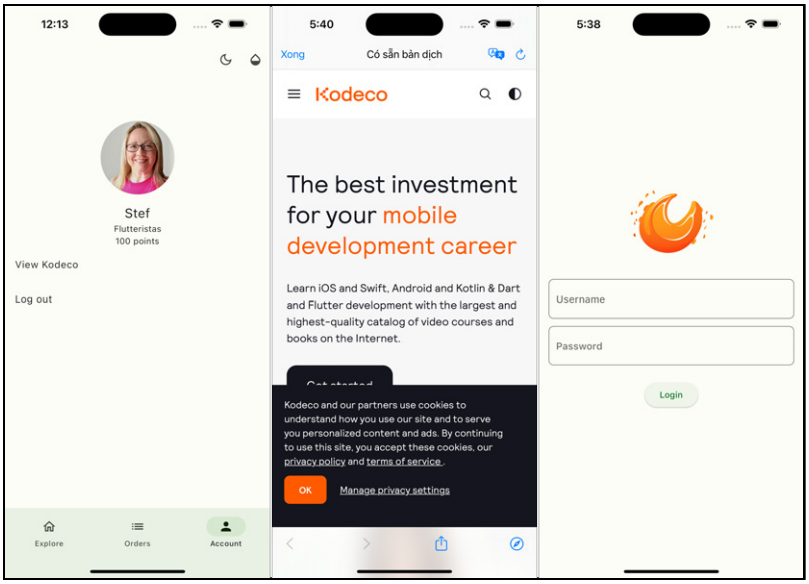
Once the order is submitted, the user is redirected to **Orders** tab:



On the **Account** screen, they can:

- View their profile and see how many points they've earned.
- Visit the Kodeco website.
- Log out of the app.

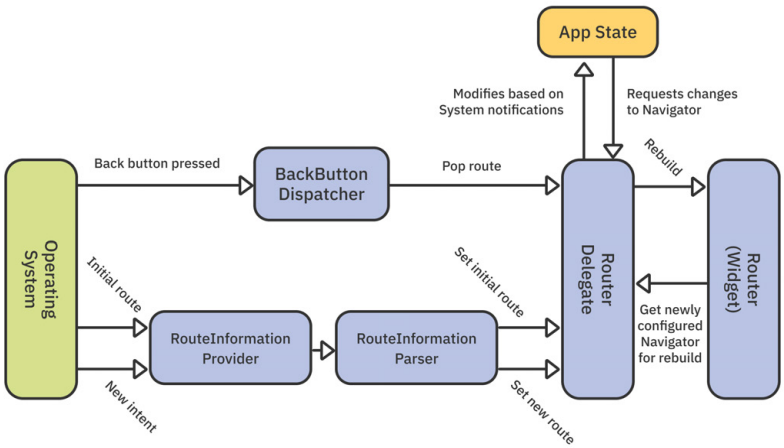
Below you'll see an example:



Your app is going to be awesome when it's finished. Now it's time to learn about **go_router**!

Introducing go_router

The **Router API** gives you more abstractions and control over your navigation stack. However, the API's complexity and usability hindered a bit the developer experience.



For example, you must create your `RouterDelegate`, bundle your app state logic with your navigator and configure when to show each route.

To support the web platform or handle **deep links**, you must implement `RouteInformationParser` to parse route information.

Eventually, developers and even Google realized the same thing: creating these components wasn't straightforward. As a result, developers wrote other routing packages to make the process easier.

> **Interesting Read**: Google's Flutter team came out with a research paper evaluating different routing packages. You can check it out here (https://github.com/flutter/uxr/blob/master/nav2-usability/Flutter%20routing%20packages%20usability%20research%20report.pdf).

Of the many packages available, you'll focus on `GoRouter`. Such a package, created by Chris Sells, is now fully maintained by the Flutter team. `GoRouter` aims to make it easier for developers to handle routing, letting them focus on building the best app they can.

In this chapter you'll focus on how to:

- Create routes.
- Handle errors.
- Redirect to another route.

Time to code!

Creating the go_router

Within **main.dart**, add the following import:

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

Next locate the comment `// TODO: Initialize GoRouter` and replace it with the following:

```
// 1
late final _router = GoRouter(
  // 2
```

```
    initialLocation: '/login',
    // TODO: Add App Redirect
    // 3
    routes: [
      // TODO: Add Login Route
      // TODO: Add Home Route
    ],
    // TODO: Add Error Handler
  );
```

Here's how it works:

1. **Initializes** an instance of `GoRouter`, a declarative router for Flutter.
2. Sets the **initial route** that the app will navigate to. When the user opens the app they will navigate to the login page.
3. `routes` contains a list of possible routes for the application. Each route will typically be defined with a path, builder or redirect function.

There are other configurations you can set such as **app redirect**, and **error handling**. For example, if the user is logged in it should redirect to home, or if the user enters a wrong path it should show an error or a **404 page**.

> **Note** on `late final` in **Router Declaration**: The `late final` keyword is used for the router to defer its **initialization** until necessary, such as waiting for user authentication. It ensures the router is **non-nullable** and remains constant once initialized, aligning with the needs of dependent states or objects in the app.

Using Your Router

Next, locate `// TODO: Replace with Router`. Replace it and the entire `return MaterialApp();` code with:

```
// 1
return MaterialApp.router(
  debugShowCheckedModeBanner: false,
  // 2
  routerConfig: _router,
  // TODO: Add Custom Scroll Behavior
  title: 'Yummy',
  scrollBehavior: CustomScrollBehavior(),
  themeMode: themeMode,
  theme: ThemeData(
    colorSchemeSeed: colorSelected.color,
    useMaterial3: true,
    brightness: Brightness.light,
  ),
  darkTheme: ThemeData(
    colorSchemeSeed: colorSelected.color,
    useMaterial3: true,
    brightness: Brightness.dark,
  ),
);
```

Here's how it works:

1. `MaterialApp.router`. This constructor is used for apps with a navigator that uses a declarative routing approach. It takes a **router configuration** rather than a set of routes.

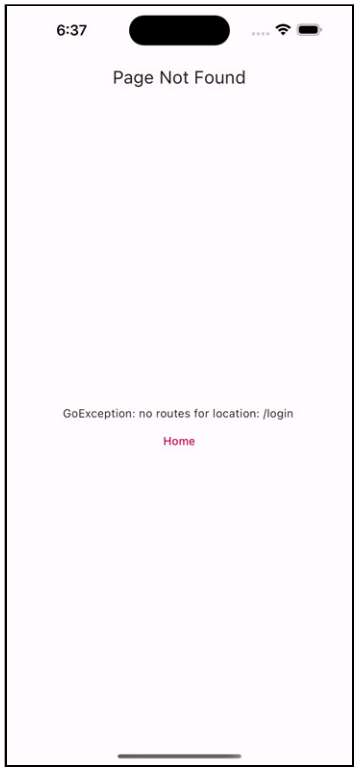
2. `routeConfig` reads `_router` to know about navigation properties. This will help the `MaterialApp` to set up the essential parts of a router. Under the hood, it will configure `routerDelegate`, `routeInformationParser`, and `routeInformationProvider`.

Your router is all set!

Adding Screens

With all the infrastructure in place, it's time to define which screen to display according to the route. But first, check out the current situation.

Build and run on iOS. You'll notice an error screen exception:



If the route isn't found, GoRouter provides a **Page Not Found** screen by default. That's because you haven't defined any routes yet!

Setting Up Your Error Handler

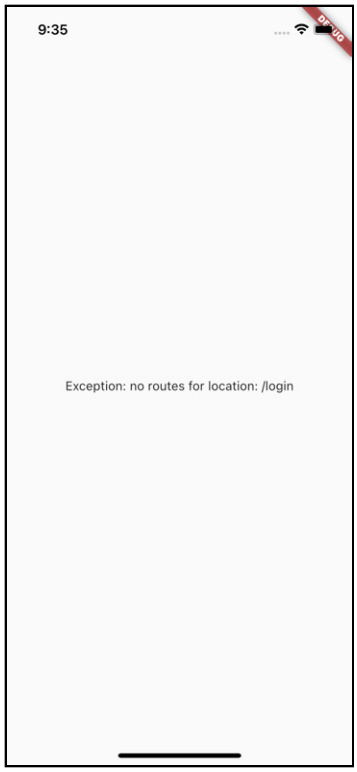
You can tweak `GoRouter` to show a custom error page. It's common for users to enter the wrong URL path, especially with web apps. Web apps usually show a **404** error screen.

Next locate `// TODO: Add Error Handler` and replace it with:

```
errorPageBuilder: (context, state) {
  return MaterialPage(
    key: state.pageKey,
    child: Scaffold(
      body: Center(
        child: Text(
          state.error.toString(),
        ),
      ),
    ),
  );
},
```

Here you simply show your error page and the error exception.

Trigger a hot restart. Your custom error page now displays.



Next, you'll start working on your login page.

Adding the Login Route

You'll start by displaying the **Login** screen.

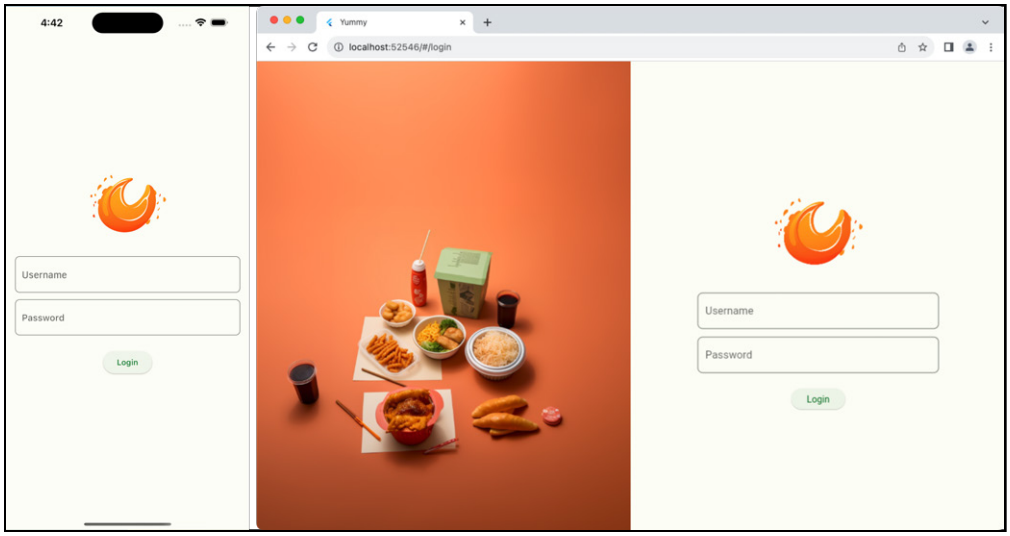
Locate `// TODO: Add Login Route` and replace it with:

```
GoRoute(
  // 1
  path: '/login',
  // 2
  builder: (context, state) =>
    // 3
    LoginPage(
      // 4
      onLogIn: (Credentials credentials) async {
        // 5
        _auth
          .signIn(credentials.username, credentials.password)
          // 6
          .then((_) => context.go('/${YummyTab.home.value}'));
    })),
```

Here's how you define a route:

1. The route is set to `/login`. When the URL or path matches `/login` go to the login route.
2. The `builder()` function creates the widget to display when the user hits a route.
3. The function returns a `Login` widget.
4. The `Login` widget takes a callback named `onLogIn` which returns the user credentials.
5. Use the credentials to log in.
6. If the login is successful, navigate to the path `/0`, which is the first tab.

Trigger a hot restart. You'll see the Login Page:



You just added your first route!

Adding the Home Route

Once you log in, you need to navigate to the home route. Locate the comment `// TODO: Add Home Route` and replace it with the following:

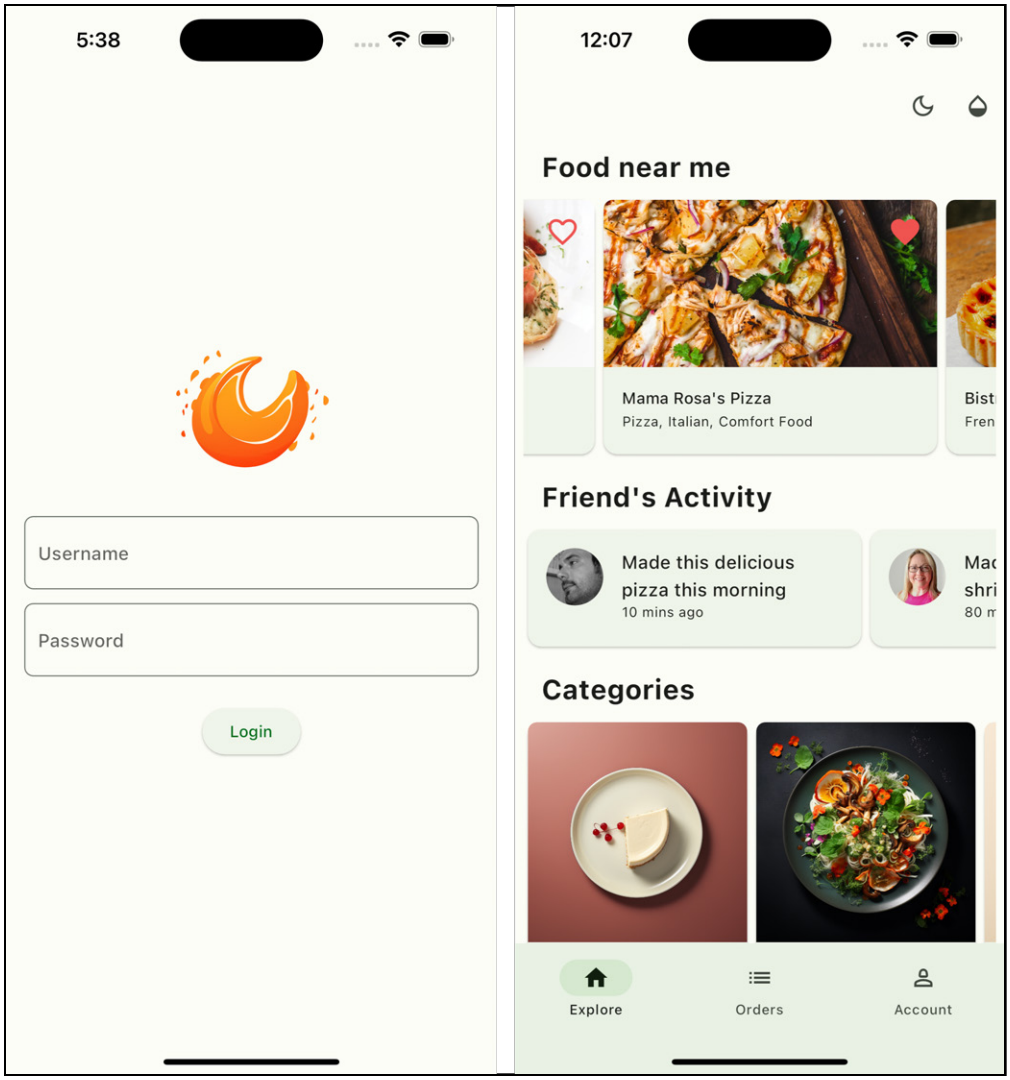
```
// 1
GoRoute(
  path: '/:tab',
  builder: (context, state) {
    // 2
  return Home(
    //3
    auth: _auth,
    //4
    cartManager: _cartManager,
    //5
    ordersManager: _orderManager,
    //6
    changeTheme: changeThemeMode,
    //7
    changeColor: changeColor,
    //8
    colorSelected: colorSelected,
    //9
    tab: int.tryParse(state.pathParameters['tab'] ?? '') ?? 0);
```

```
      },
      // 10
      routes: [
      // TODO: Add Restaurant Route
    ]),
```

Here's how it works:

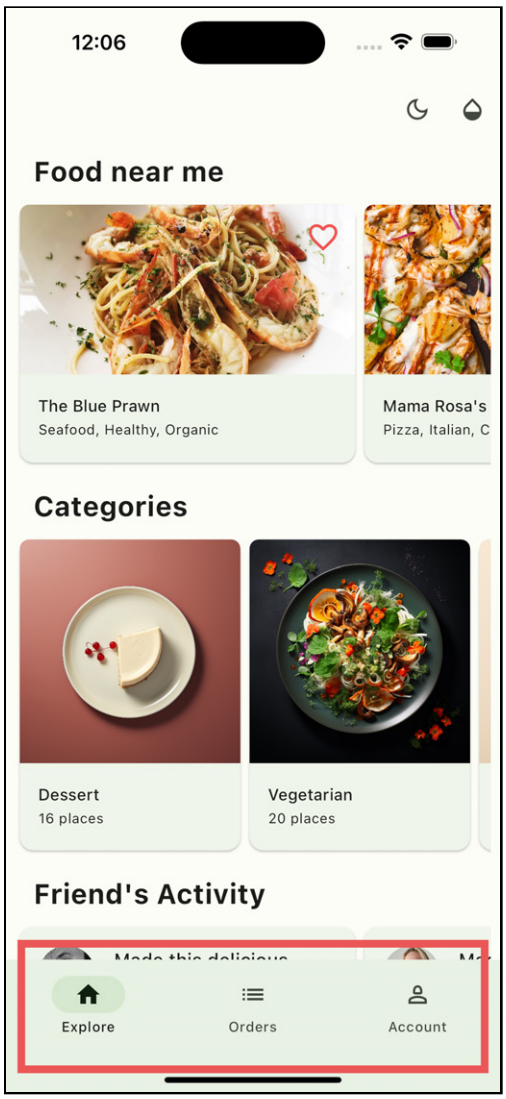
1. The route is set to `/`. When the URL or path matches `/` go to the home route. `:tab` is a path parameter used to switch between different tabs.
2. The builder function returns a `Home` widget.
3. Pass `auth` for handling authentication
4. Use `cartManager` to manage the items that the user added to the cart.
5. Use `ordersManager` to manage all the orders submitted.
6. Set a callback to handle user changes from light to dark mode.
7. Set a callback to handle user app color theme changes.
8. Pass the currently selected color theme.
9. Set the current tab, default to **0** if the path parameter is absent or not an integer.

Perform a hot reload if needed, click the **Login** button and now you'll land on **Home**.



Navigate to the Current Tab

Try clicking on the tab bar items and notice that nothing works. You'll now add a way to navigate between tabs.



In **lib/home.dart** locate `// TODO: Navigate to specific tab` and replace it with the following:

```
context.go('/$index');
```

Don't forget to import `go_router`:

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

Now you can navigate to different tabs.

Hot reload again and notice that the app goes back to the login screen. Wouldn't it be great when the user opens **Yummy** app again to go straight to the home page if the user is already logged in?

Handling Redirects

You redirect when you want your app to go to a different location. `GoRouter` lets you do this with its `redirect` handler.

Most apps require some type of login authentication flow, and redirect is perfect for this situation. For example, some of these scenarios may happen to your app:

- The user logs out of the app.
- The user tries to go to a restricted page that requires them to log in.
- The user's session token expires. In this case, they're automatically logged out.

It would be nice to redirect the user back to the login screen in all these cases. Open **lib/main.dart** and locate the comment `// TODO: Add Redirect Handler` and replace it with:

```
// 1
Future<String?> _appRedirect(
  BuildContext context, GoRouterState state) async {
  // 2
  final loggedIn = await _auth.loggedIn;
  // 3
  final isOnLoginPage = state.matchedLocation == '/login';

  // 4
  // Go to /login if the user is not signed in
  if (!loggedIn) {
    return '/login';
  }
  // 5
  // Go to root if the user is already signed in
  else if (loggedIn && isOnLoginPage) {
    return '/${YummyTab.home.value}';
  }
```

```
    // 6
    // no redirect
    return null;
  }
```

Here's how it works:

1. `_appRedirect()` is an asynchronous function that returns a future, optional string. It takes in a build context and the go router state.
2. Get the login status.
3. Check if the user is currently on the login page.
4. If the user is not logged in yet, redirect to the login page.
5. If the user is logged in and is on the login page, redirect to the home page.
6. Don't redirect if no condition is met.

Next to apply the handler, locate `// TODO: Add App Redirect` and replace it with:

```
  redirect: _appRedirect,
```

Hot reload and you will notice that the app now goes to the home page directly.

Adding the Restaurant Route

When the user taps on a restaurant on the **Explore** page, the app navigates to a subroute. Locate `// TODO: Add Restaurant Route` and replace it with:

```
  GoRoute(
    // 1
    path: 'restaurant/:id',
    builder: (context, state) {
      // 2
      final id =
          int.tryParse(state.pathParameters['id'] ?? '') ?? 0;
      // 3
      final restaurant = restaurants[id];
      // 4
      return RestaurantPage(
        restaurant: restaurant,
        cartManager: _cartManager,
        ordersManager: _orderManager,
      );
    }),
```

Here's how it works:

1. The route is defined with the path `restaurant/:id`. The `:id` part is a path parameter, which allows for dynamic routing based on the restaurant's ID.
2. Within the `builder()` function, you extract the `id` from `pathParameters`.
3. Get the restaurant based on the `id``.
4. Return the `RestaurantPage` widget with the specific restaurant, cart and order manager.

Now that you have set up the restaurant route, you need to navigate to it.

Navigate to the Restaurant Page

Open **lib/components/restaurant_section.dart**, locate the comment `// TODO: Navigate to Restaurant` and replace it with the following:

```
context.go('/${YummyTab.home.value}/restaurant/$
{restaurants[index].id}');
```

Don't forget to add the necessary imports:

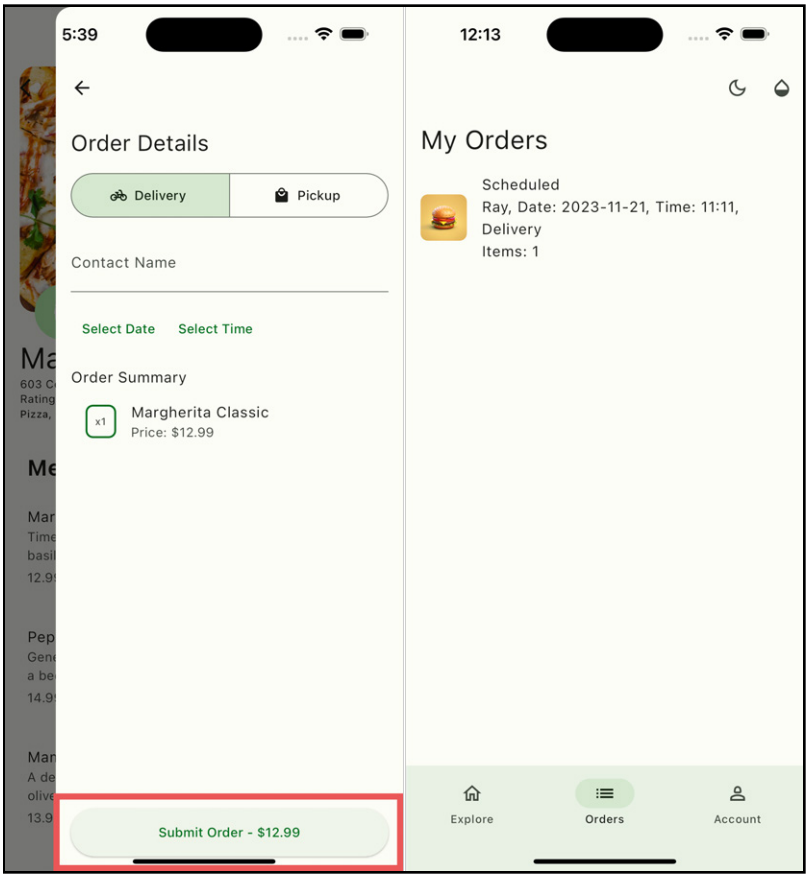
```
import 'package:go_router/go_router.dart';
import '../constants.dart';
```

From the home page, based on the selected restaurant, navigate to the specific restaurant with the specific restaurant id.

> **Note**: There are two ways to navigate to different routes:
>
> 1.) `context.go(path)`
>
> 2.) `context.goNamed(name)`
>
> You should use `goNamed()` instead of `go()` as it's **error-prone**, and the actual URI format can change over time.
>
> `goNamed()` performs a **case-insensitive** lookup by using the `name` parameter you set with each route. It also helps you pass query parameters to your route.

Navigate to the Order Page

Once a user adds items to the cart and submits an order, it would be nice to navigate to the orders tab, so that customers can review the order.



Open **restaurant_page.dart** and add the following imports:

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

Next locate `// TODO: Navigate to Orders Page` and replace it with:

```
context.pop();
context.go('/${YummyTab.orders.value}');
```

Now, when the user taps on the **Submit order** button, the app navigates to the **Orders** tab.

Handle Log Out

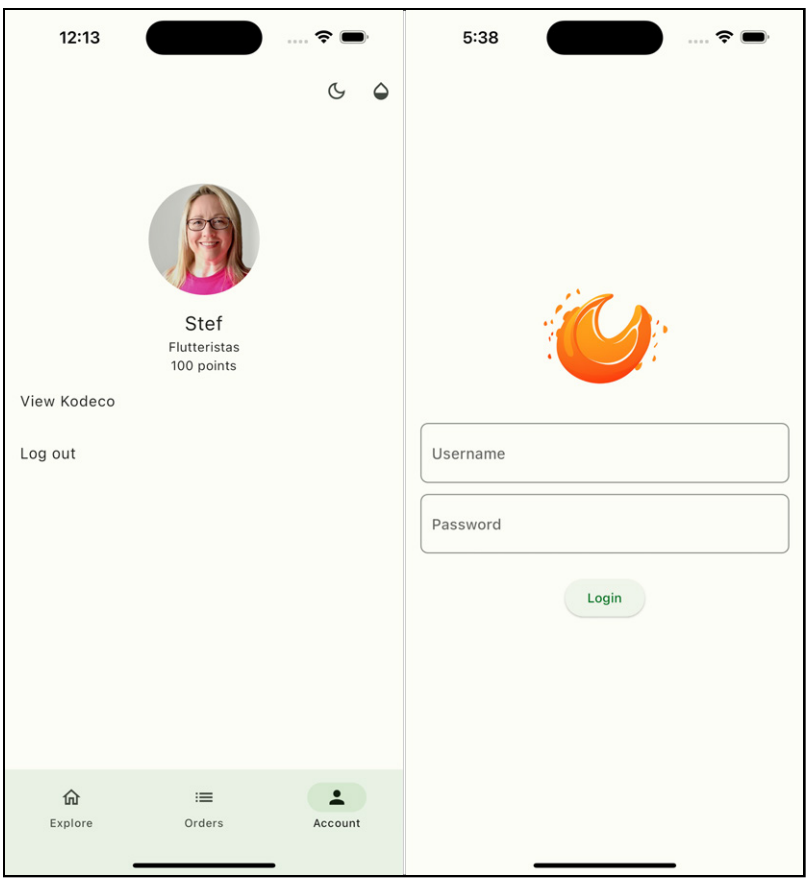
Lastly you'll work on the logout functionality.

Open **home.dart**, locate the `// TODO: Logout and go to login` and replace it with the following:

```
widget.auth.signOut().then((value) => context.go('/login'));
```

Here you call `signOut()`, which resets the entire app state and redirects you back to the **Login** screen.

Save your changes. Now, tap the **Log out** button on the **Account** screen. You'll notice it goes back to the **Login** screen, as shown below:



Congratulations, you've now completed the entire **UI navigation flow**.

Key Points

- **Navigator 1.0** is useful for quick and simple prototypes, presenting alerts and dialogs.
- **Router API** is useful when you need more control when managing the **navigation stack**.
- `GoRouter` is a wrapper around the **Router API** that makes it easier for developers to build navigation logic.
- With `GoRouter`, you navigate to other routes using `goNamed()` instead of `go()`.
- Use a **router widget** to listen to navigation state changes and configure your navigator's list of pages.
- If you need to navigate to another page after some state change, handle that with the `redirect()` handler.
- You can customize the error page by implementing the `errorPageBuilder`.

Where to Go From Here?

You've now learned how to navigate between screens the **declarative** way. Instead of calling `push()` and `pop()` in different widgets, you use multiple managers to manage your state.

You also learned how to create a `GoRouter` widget, which encapsulates and configures all the routes for a navigator. Now, you can easily manage your navigation flow in a single router object!

To learn about navigation in Flutter, here are some recommendations:

- To understand the motivation behind Navigator 2.0, check out the design document (https://docs.google.com/document/d/1Q0jx0l4-xymph9O6zLaOY4d_f7YFpNWX_eGbzYxr9wY/edit).
- Watch this presentation by Chun-Heng Tai (https://youtu.be/xFFQKvcad3s?t=3158), who contributed to the new declarative API.
- In this video, Simon Lightfoot walks you through a Navigator 2.0 example (https://www.youtube.com/watch?v=Y6kh5UonEZ0).

- Flutter Navigation 2.0 by Dominik Roszkowski goes through the differences between Navigator 1.0 and 2.0, including a video example (https://youtu.be/JmfYeF4gUu0?t=9728).
- For in-depth knowledge about `Navigator`, check out Flutter's documentation (https://api.flutter.dev/flutter/widgets/Navigator-class.html).
- Finally, here's the GoRouter documentation (https://pub.dev/documentation/go_router/latest/).

Other Libraries to Check Out

`GoRouter` is just one of the many libraries trying to make the Router API easier to use. Check them out here:

- Beamer (https://pub.dev/packages/beamer)
- Flow Builder (https://pub.dev/packages/flow_builder)
- Fluro (https://pub.dev/packages/fluro)
- Vrouter (https://pub.dev/packages/vrouter)
- Auto Route (https://pub.dev/packages/auto_route)

There are so many more things you can do with Router API. In the next chapter, you'll look at supporting web URLs and deep linking!

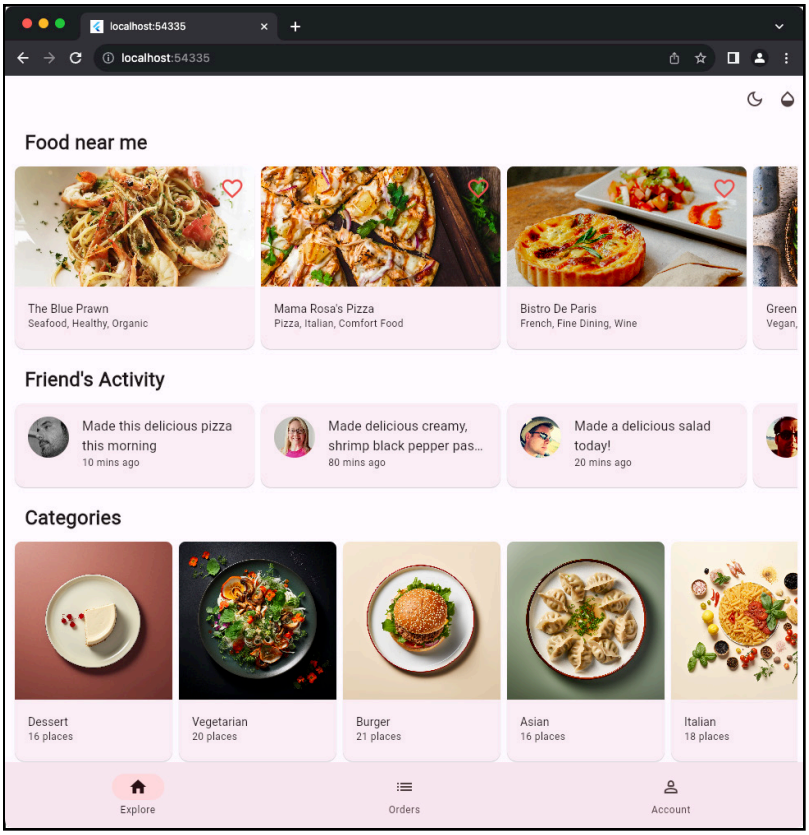
Chapter 9: Deep Links & Web URLs

By Vincent Ngo

Sometimes, opening your app and working through the navigation to get to a screen is just too much trouble for the user. Redirecting to a specific part of your app is a powerful marketing tool for user engagement. For example, generating a special QR code for a promotion that users can scan to visit that specific product in your app is a cool and effective way to build interest in the product.

In the last chapter, you learned how to use `GoRouter` to move between screens, navigating your app in a declarative way. Now you'll learn how to **deep link** to screens in your app and explore web URLs on the web.

Take a look at how **Yummy** looks in the Chrome web browser:



By the end of this chapter, you'll:

- Have a better understanding of the **router API**.
- Know how to support **deep linking** on iOS and Android.
- Explore the Yummy app on the **web**.

You'll learn how to **direct** users to any screen of your choice.

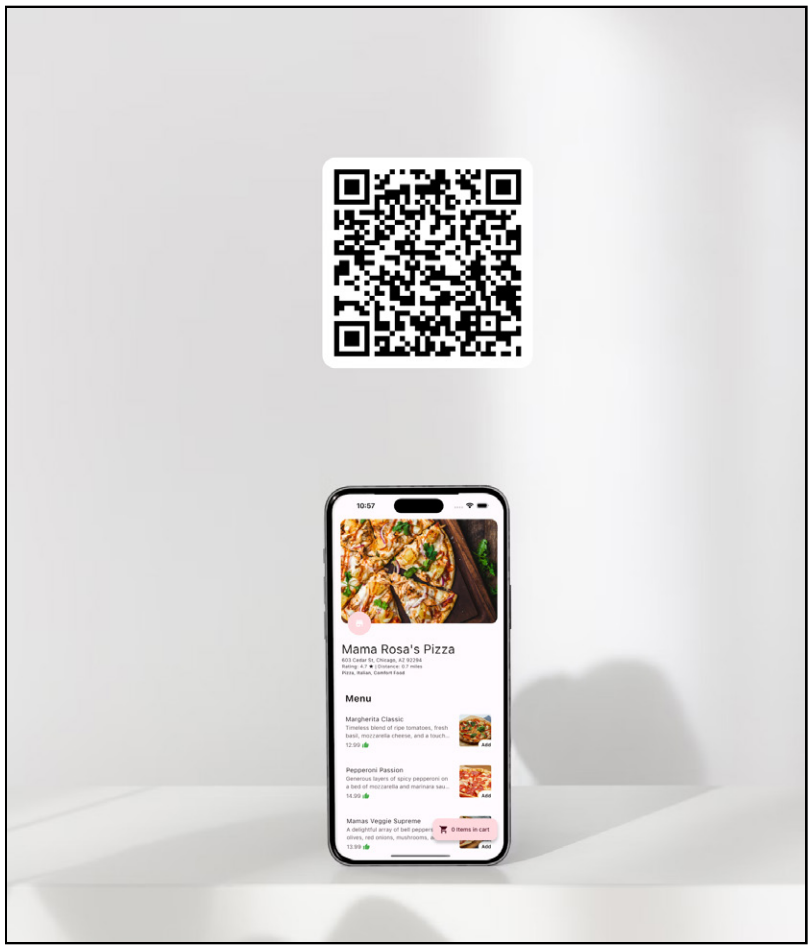
> **Note**: You'll need to install the Chrome web browser to view Yummy on the web. If you don't have Chrome, you can get it here (https://www.google.com/chrome/). The Flutter web project can run on other browsers, but this chapter only covers testing and development with Chrome.

Understanding Deep Links

A **deep link** is a URL that navigates to a **specific destination** in your mobile app. Think of deep links like a URL address you enter into a web browser to go to a specific page of a website rather than the home page.

Deep links help with **user engagement** and **business marketing**. For example, if you're running a sale, you can direct the user to a specific product page in your app instead of making them search for it.

Just imagine, your app **Yummy** is a user-friendly food app that allows customers to quickly scan a QR code at restaurants, instantly access menus and seamlessly deep-link to detailed restaurant pages in for an enhanced dining experience.



With deep linking, Yummy is more automated. It brings the user directly to the restaurant page making it easier to view the menu.Without deep linking, the process is more manual. The user has to launch the app, navigate to the **Explore** tab find the correct restaurant, or search the restaurant name, and finally get to the restaurant page to view the menu. That takes three steps instead of one and likely some head-scratching, too!

Types of Deep Links

There are three types of deep links:

- **URI schemes**: An app's own URI scheme. **yummy://kodeco.com/home** is an example of Yummy's URI scheme. This form of deep link only works if the user has **installed** your app.
- **iOS Universal Links**: In the root of your web domain, you place a file that points to a specific **app ID** to say whether to **open** your app or to direct the user to the **App Store**. You must register that specific app ID with Apple to handle links from that domain.
- **Android App Links**: Like iOS Universal Links, Android App Links take users to a link's specific content directly in your app. They leverage **HTTP URLs** and are associated with a website. For users that don't have your app installed, these links go directly to the content of your **website**.

In this chapter, you'll only look at **URI Schemes**. For more information on how to set up iOS Universal Links and Android App Links, check out these tutorials:

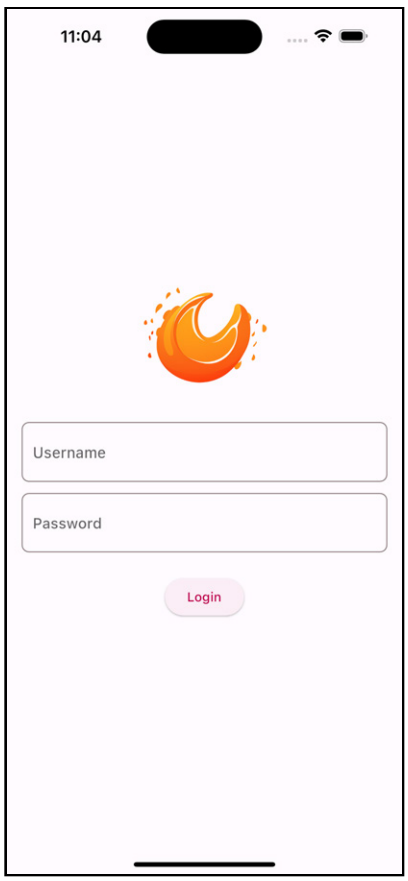
- Universal Links: Make the Connection (https://www.kodeco.com/6080-universal-links-make-the-connection)
- Deep Links in Android: Getting Started (https://www.kodeco.com/18330247-deep-links-in-android-getting-started)

Getting Started

> **Note**: We recommend you use the starter project for this chapter rather than continuing with the project from the last chapter.

Open the starter project in Android Studio and run `flutter pub get`. Then, run the app on iOS or Android.

You'll see that **Yummy** shows the **Login** screen.



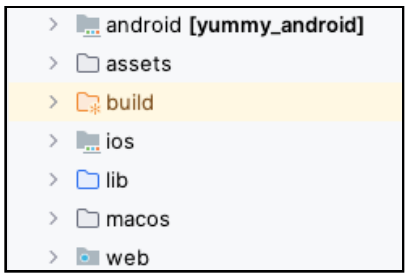
Soon, you'll be able to redirect users to different parts of the app. But first, take a moment to review what's changed in the starter project since the last chapter.

Project Files

Before diving in, you need to be aware of some new files.

New Flutter Web Project

The starter project includes a pre-built Flutter web project.



> **Note**: To speed things up, the web project is pre-built in your starter project. To learn how to create a Flutter web app, check out the Flutter documentation (https://flutter.dev/docs/get-started/web#add-web-support-to-an-existing-app).

Setting Up Deep Links

To enable deep linking on iOS and Android, you must add metadata tags on the respective platforms. These tags have already been added to the starter project.

Setting Up Deep Links on iOS

Open **ios/Runner/Info.plist**. You'll see some new key-value pairs, which enable deep linking for iOS:

```
...
<key>FlutterDeepLinkingEnabled</key>
<true/>
<key>CFBundleURLTypes</key>
<array>
  <dict>
  <key>CFBundleTypeRole</key>
  <string>Editor</string>
  <key>CFBundleURLName</key>
  <string>kodeco.com</string>
  <key>CFBundleURLSchemes</key>
```

```
    <array>
    <string>yummy</string>
    </array>
    </dict>
  </array>
  ...
```

`CFBundleURLName` is a **unique URL** that distinguishes your app from others that use the same scheme. `yummy` is the URL scheme you'll use later.

Setting Up Deep Links on Android

Open **android/app/src/main/AndroidManifest.xml**. Here you'll also find two new definitions in the `<data>` tag:

```
  ...
  <!-- Deep linking -->
  <meta-data android:name="flutter_deeplinking_enabled"
  android:value="true" />
  <intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  <data
    android:scheme="yummy"
    android:host="kodeco.com" />
  </intent-filter>
  ...
```

Like in iOS, you set the same values for `scheme` and `host`.

When you create a deep link for Yummy, the custom URL scheme looks like this:

```
yummy://kodeco.com/<path>
```

Now, take a quick look at the URL paths you'll create.

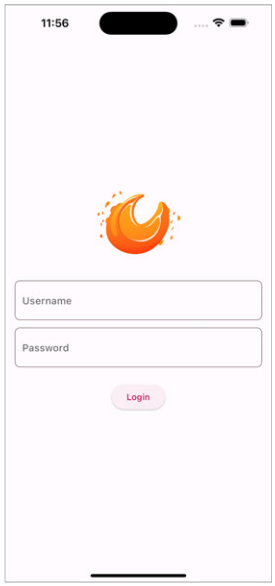
Overview of Yummy Paths

Yummy has many screens you can deep link to. Here are all the possible paths you can direct your users to the following.

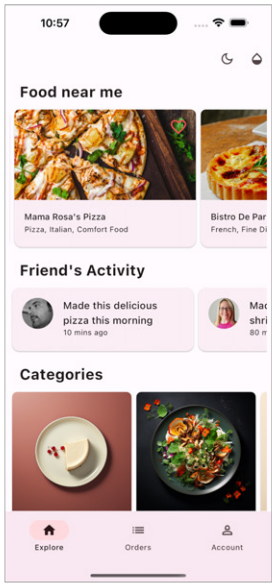
Path: /

The app initializes and checks the app cache to see if the user is logged in.

- **/login**: Redirects to the **Login** screen if the user isn't logged in yet.



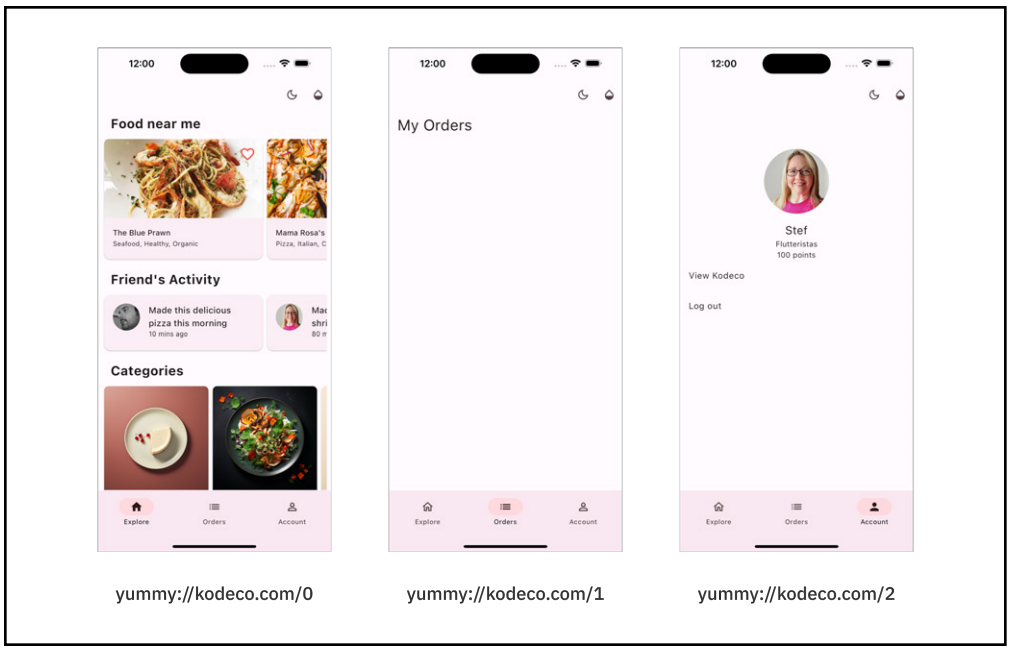
yummy://kodeco.com/login



yummy://kodeco.com/

Path: /:tab

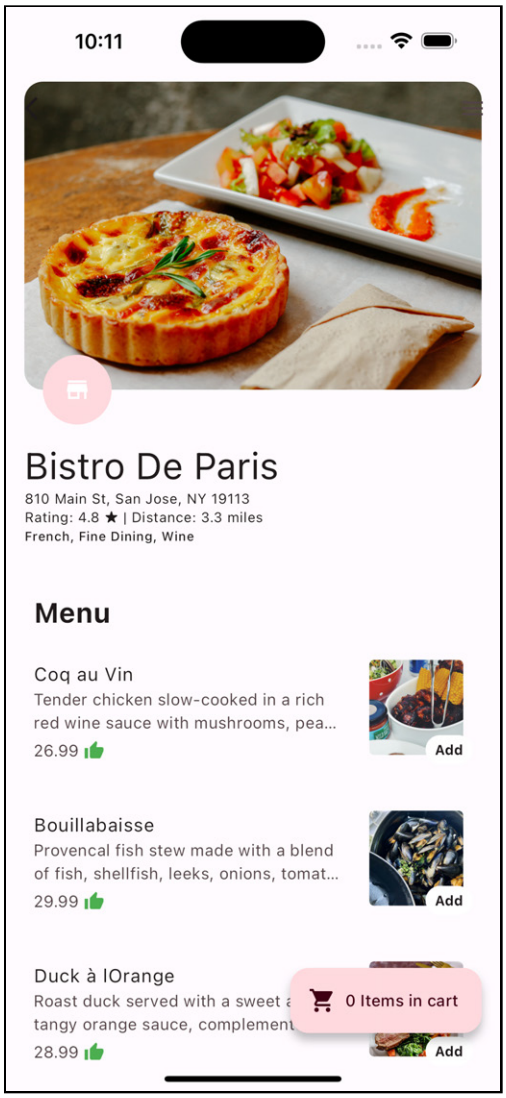
Once the user logs in, they're redirected to `/:tab`. It contains one parameter, `tab`, which directs to a **tab index**. The screenshots below show that the tab index is `0`, `1` or 2, respectively.



Path: /restaurant/:id

The restaurant page is a **sub route** of the **Explore** page. You can present a restaurant from any tab.

`/restaurant/:id` contains one parameter, `id`.



Note: Keep in mind that these URL paths work similarly for mobile and web apps.

When you deep link on **mobile**, you'll use the following URI scheme:

```
yummy://kodeco.com/<path>
```

On the **web**, the URI scheme is like any web browser URL:

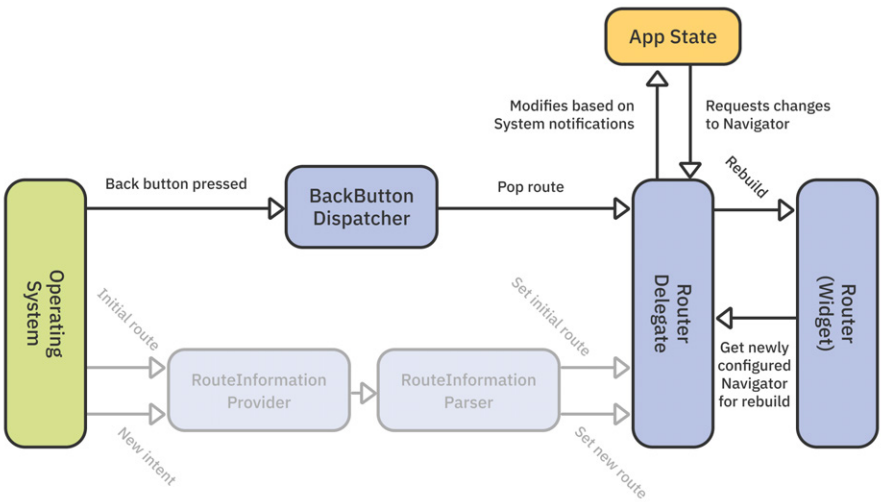
```
http://localhost:60738/#/<path>
```

Before exploring deep links, take a moment for a quick Router API recap.

Router API Recap

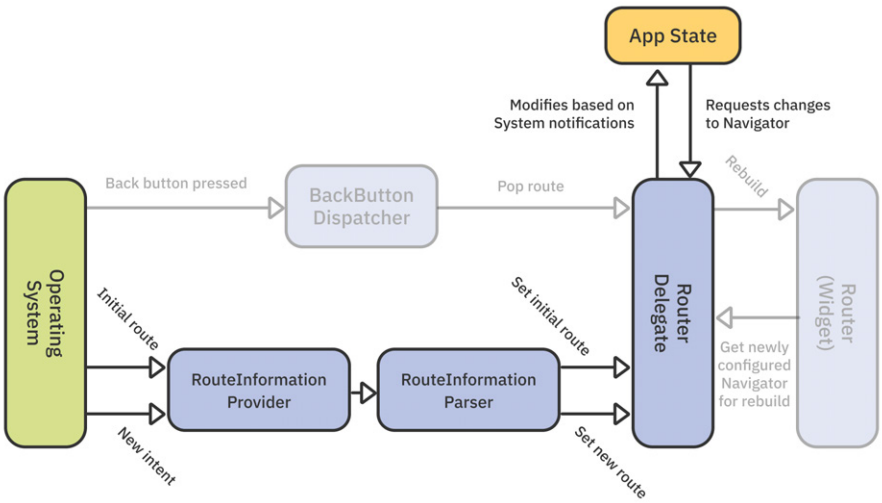
In the last chapter, you learned how to use `GoRouter` to set up routes and navigate to screens. `GoRouter` conveniently manages the Router API for you. How amazing is that? :]

However, it's still good to understand how routing works behind the scenes. Here's a diagram of what makes up the Router API:



- **Router** is a widget that extends `RouterDelegate`. The router ensures that the messages get to the `RouterDelegate`.
- **Navigator** defines a stack of `MaterialPages` in a declarative way. It also handles any `onPopPage()` event.
- **BackButtonDispatcher** handles platform-specific system **back button presses**. It listens to requests by the OS and tells the router delegate to pop a route.

Next, you'll look at **RouteInformationProvider** and **RouteInformationParser**.



- **RouteInformationProvider**: Provides the route information to the router. It informs the router about the **initial route** and notifies **new intents**.
- **RouteInformationParser**: Gets the route string from `RouteInformationProvider`, then parses the URL string to a generic **user-defined** data type. This data type is a navigation configuration.

> **Note**: `GoRouter` implements its own `RouteInformationParser` called **GoRouteInformationParser**. Based on the `routeInformation`, it tries to search for a route match based on the route's location. Check out the code in this GitHub repository (https://github.com/flutter/packages/blob/main/packages/go_router/lib/src/parser.dart).

Since `GoRouter` provides and manages all of these components, it's a good idea to jump straight into `GoRouter`'s implementation to learn more and see how they configure things.

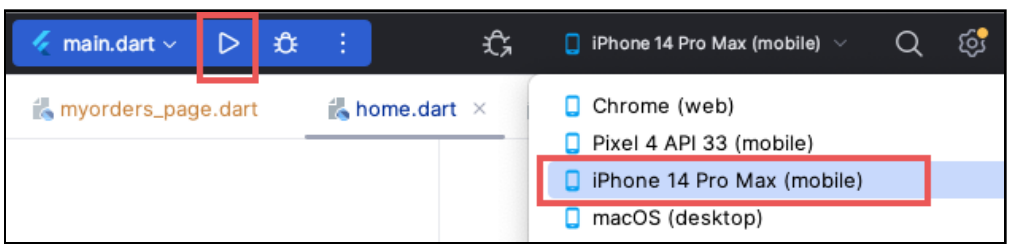
Enough theory. It's time to get started!

Testing Deep Links

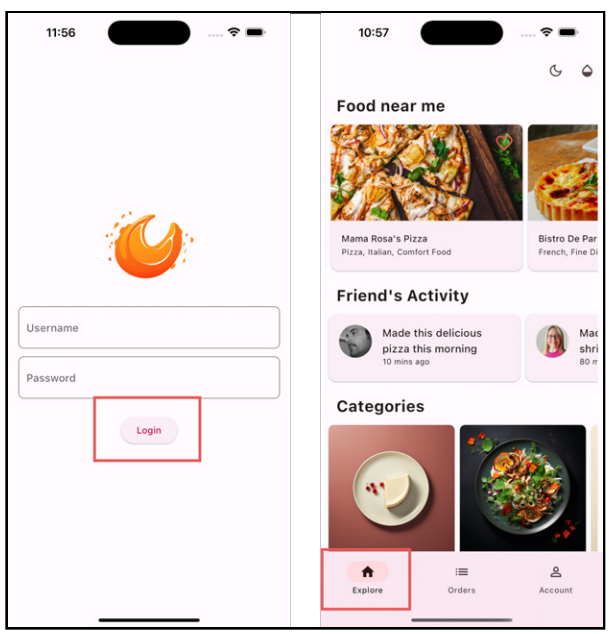
Next, you'll test how deep linking works on iOS, Android and the web.

Testing Deep Links on iOS

In Android Studio, select an iOS device and press **Run**:



Once the simulator is running, log in as shown below:



Deep Linking to the Orders Page

Enter the following in your terminal:

```
xcrun simctl openurl booted 'yummy://kodeco.com/1'
```

> **Note**: You have to be logged into the app. Otherwise, it will just show the **Login** page. Note that the first time you run this command the simulator might show a popup. If so, allow it to proceed.

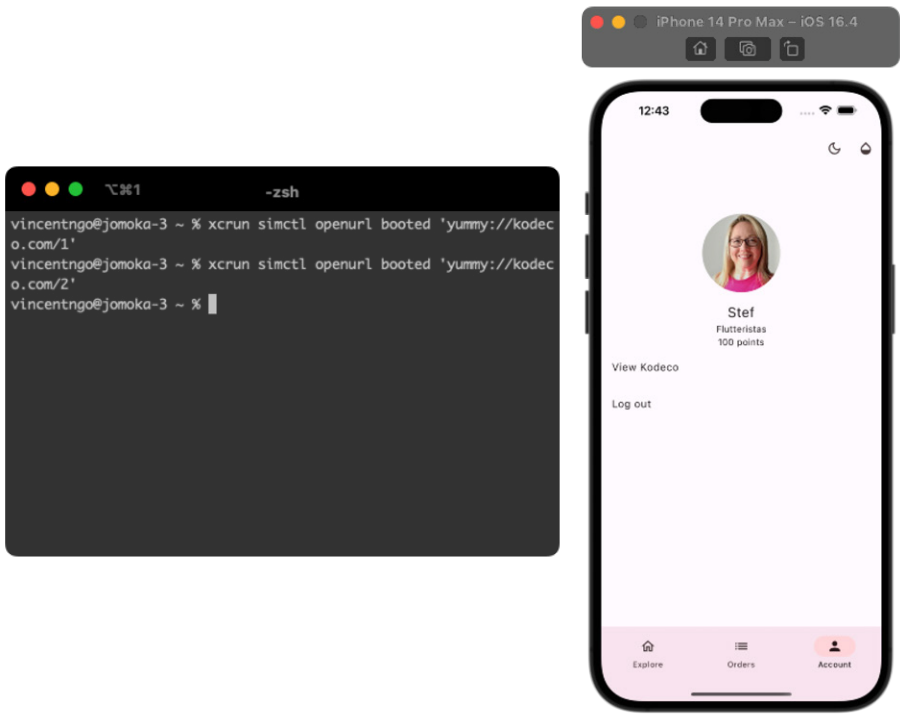
In the simulator, this automatically switches to the second tab, as shown below:

Deep Linking to the Account Page

Next, run the following command:

```
xcrun simctl openurl booted 'yummy://kodeco.com/2'
```

This command directs to the **Account** page:

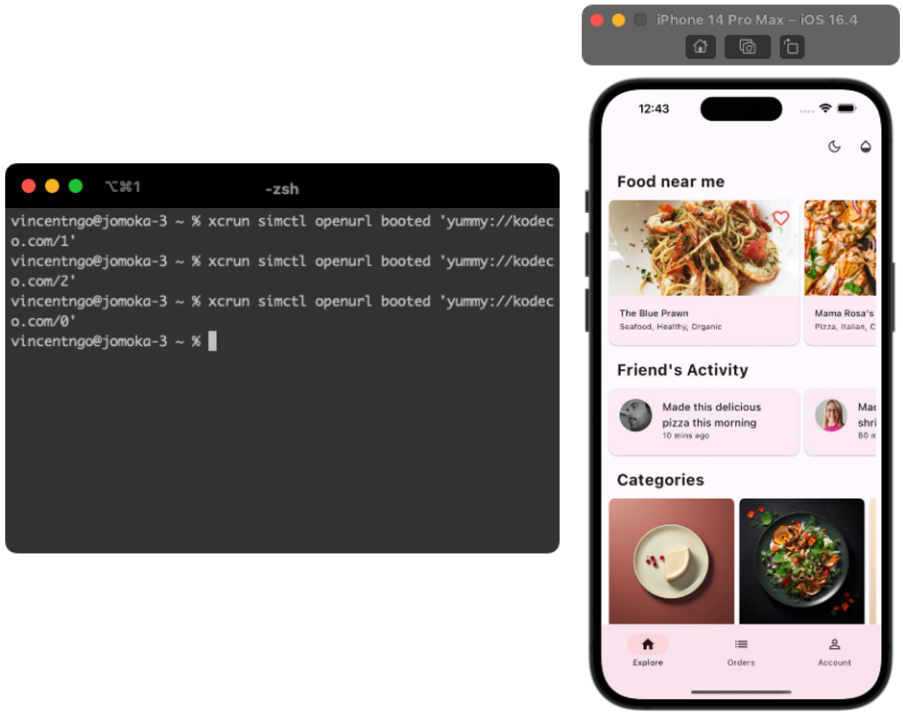


Deep Linking to the Home Page

Next, run the following command:

```
xcrun simctl openurl booted 'yummy://kodeco.com/0'
```

This command directs to the home page, named **Explore**:



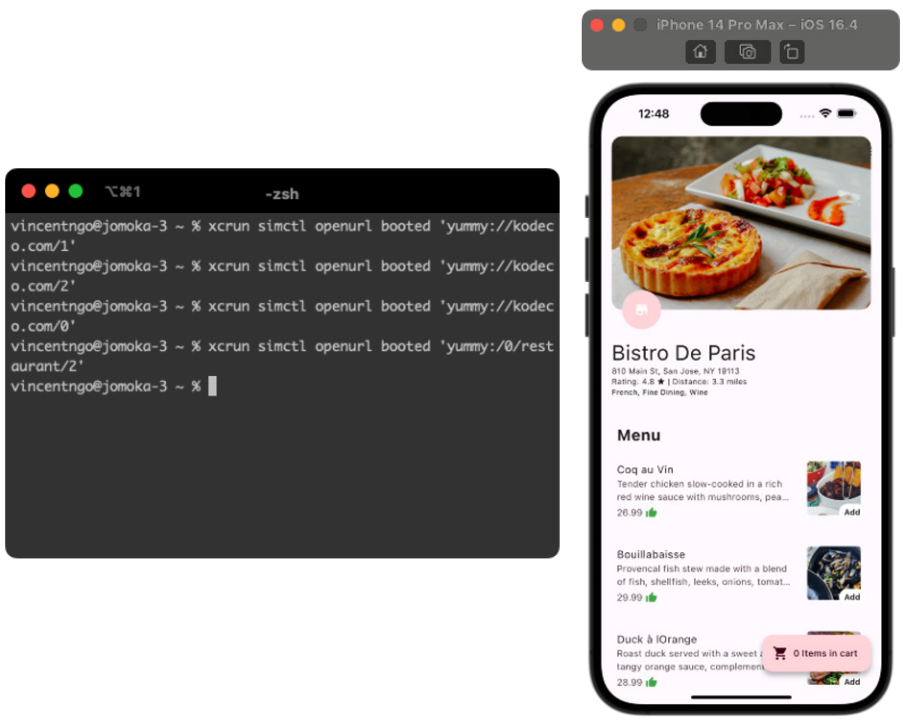
Deep Linking to a Specific Restaurant

Next, run the following command:

```
xcrun simctl openurl booted 'yummy:/0/restaurant/2'
```

Observe that the route is structured as `/:tab/restaurant/:id`. Here, the restaurant page acts as a **subroute** under the home route. To navigate to a specific restaurant, you need to identify the **active tab**, followed by the restaurant's **unique ID**. This hierarchical routing ensures precise and context-aware navigation within the app.

The restaurant page will now show:

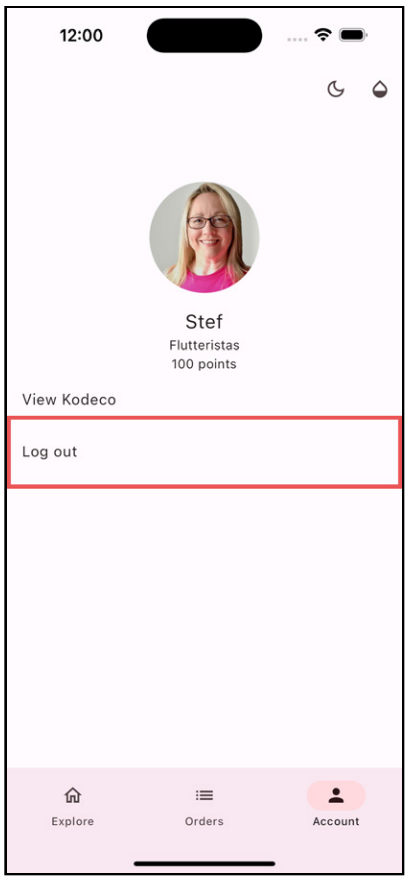


Following this pattern, you can build paths to any location in your app!

Resetting the Cache in the iOS Simulator

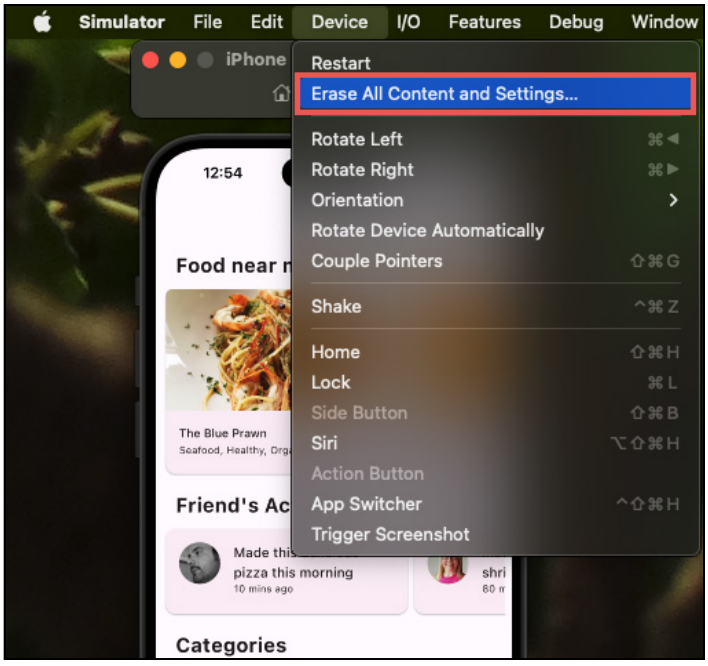
Recall that `AppStateManager` checks with `AppCache` to see whether the user is logged in. If you want to reset the cache to see the **Login** screen again, you have two options:

1. Go to the **Account** view and tap **Log out** to invalidate the app cache.



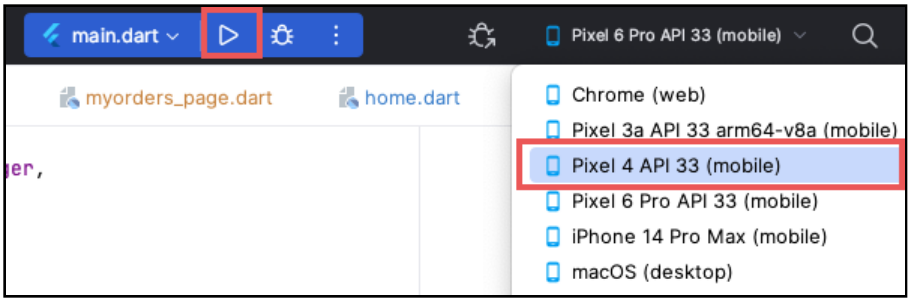
2. In the iOS simulator menu, you can select **Erase All Content and Settings...** to clear the cache.

> **Note**: This will delete any other apps you have on the simulator.

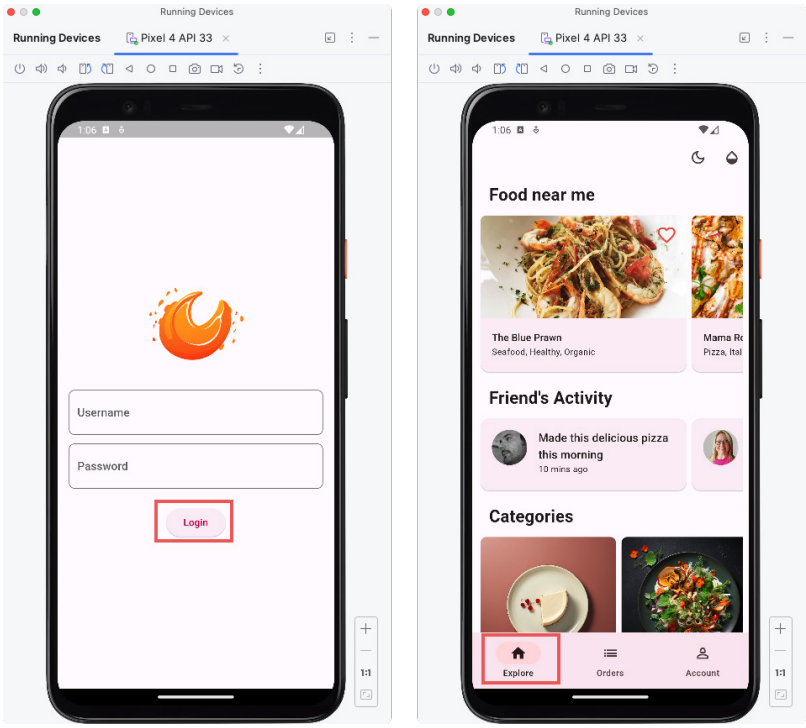


Testing Deep Links on Android

Stop running on iOS. In **Android Studio**, select an Android emulator or device and click the **Run** button:



Once the emulator or device is running, log in:



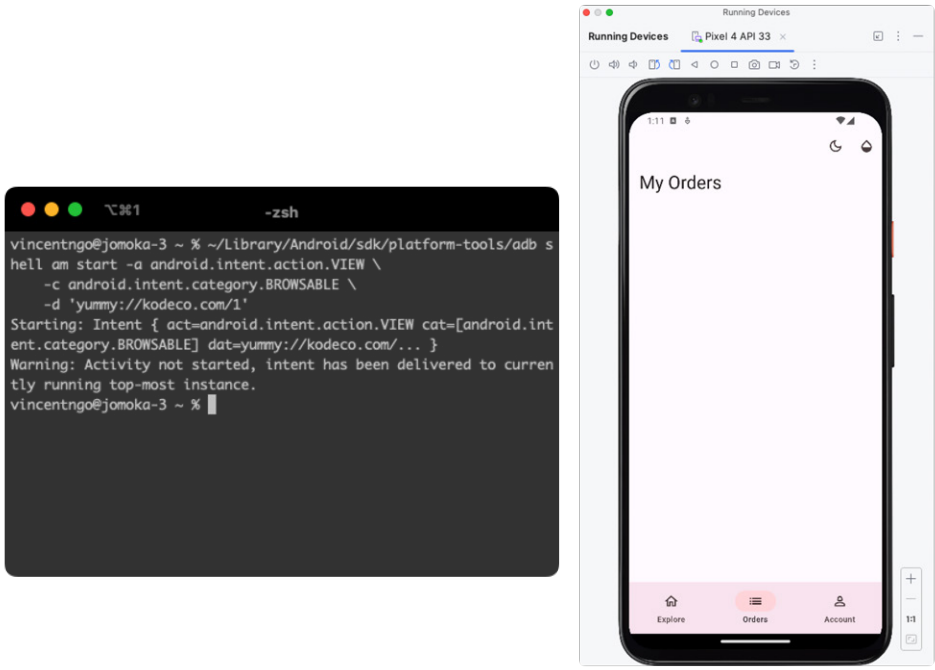
Deep Linking to the Orders Page

Enter the following in your terminal:

```
~/Library/Android/sdk/platform-tools/adb shell am start -a
android.intent.action.VIEW \
    -c android.intent.category.BROWSABLE \
    -d 'yummy://kodeco.com/1'
```

> **Note**: If you receive a message in Terminal like: `Warning: Activity not started, intent has been delivered to currently running top-most instance`, ignore it. It just means that the app is already running.
>
> The entire path is listed to ensure that you can still execute this command if you don't have `adb` in your $PATH. The `\` at each line's end formats the script nicely across multiple lines.

This command directs to the second tab of Yummy, that is **Orders**:

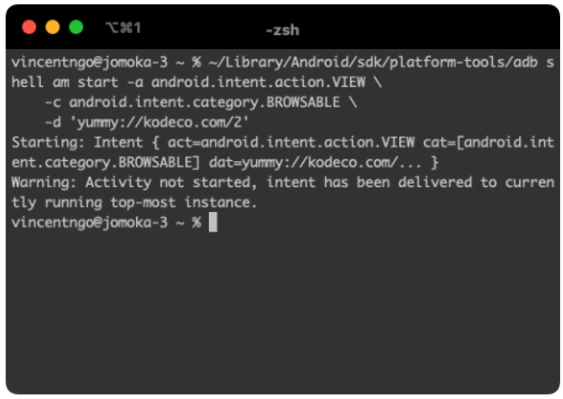


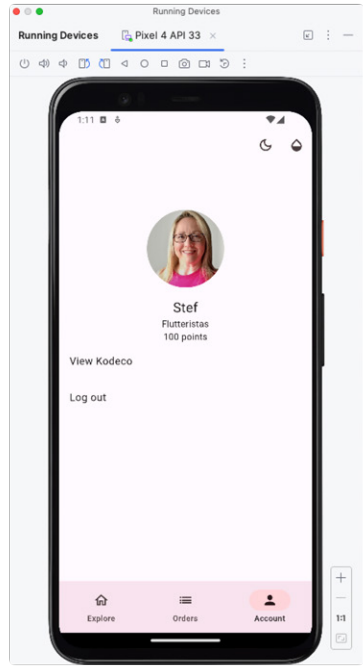
Deep Linking to the Account Page

Next, run the following command:

```
~/Library/Android/sdk/platform-tools/adb shell am start -a
android.intent.action.VIEW \
    -c android.intent.category.BROWSABLE \
    -d 'yummy://kodeco.com/2'
```

This command navigates to the **Account** screen:



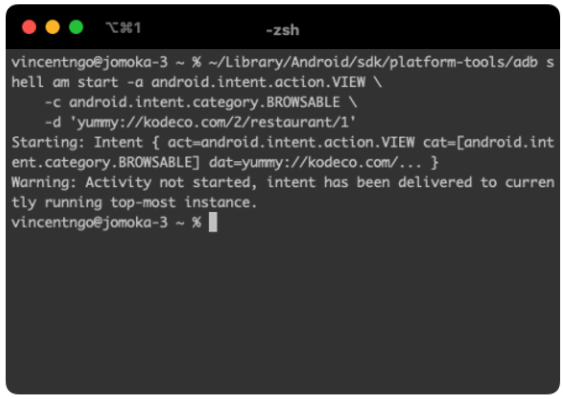


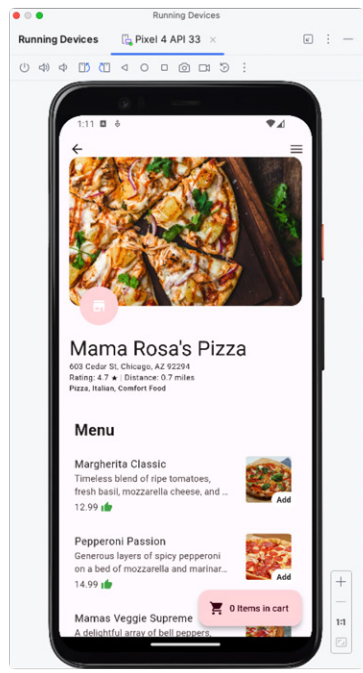
Deep Linking to Restaurant Page

Next, run the following:

```
~/Library/Android/sdk/platform-tools/adb shell am start -a
android.intent.action.VIEW \
    -c android.intent.category.BROWSABLE \
    -d 'yummy://kodeco.com/2/restaurant/1'
```

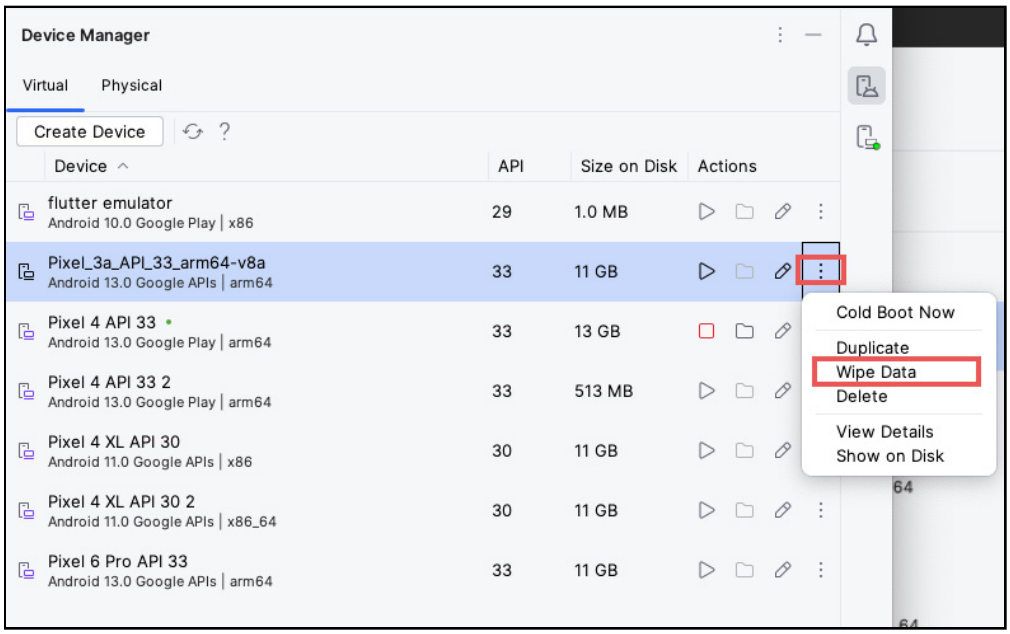
The selected restaurant page appears, as shown below:





Resetting the Cache in Android

If you need to reset your emulator cache:

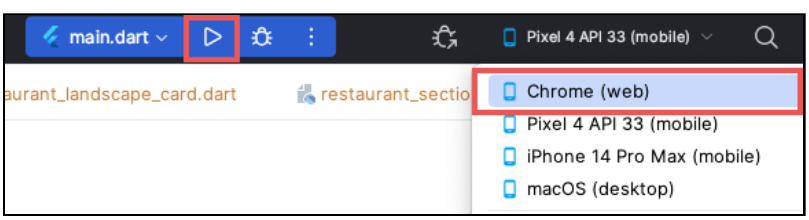


In **Android Studio** go to **Tools/Device Manager** and you'll see your list of virtual devices. Click the 3 dotted action bar and select **Wipe Data**

Now, it's time to test how Yummy handles URLs on the web.

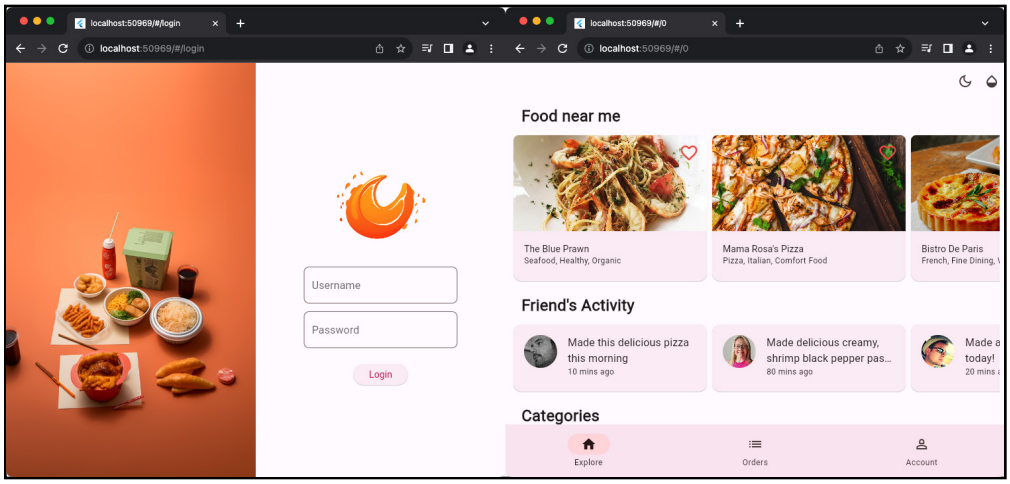
Running the Web App

Stop running on Android. In Android Studio, select **Chrome (web)** and click **Run**:

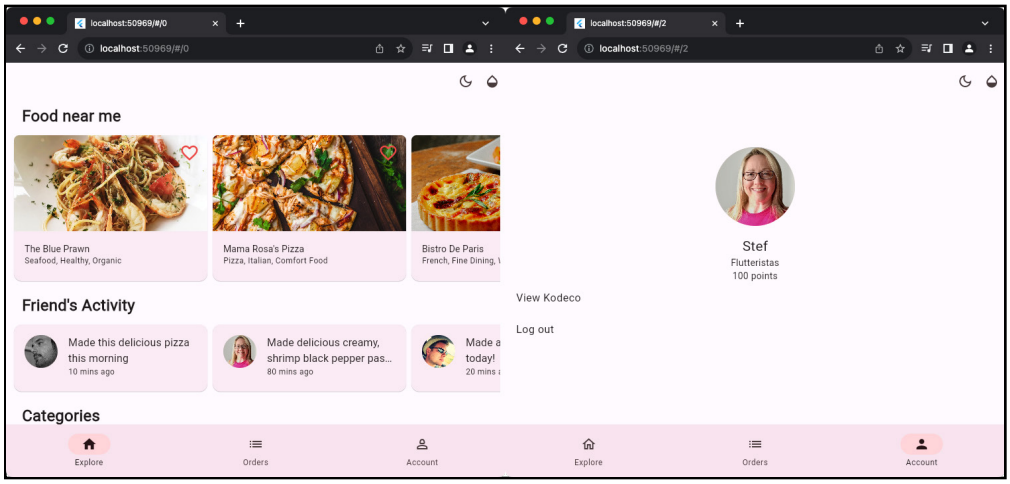


> **Note**: Your data won't persist between app launches because Flutter web runs the equivalent of incognito mode (https://support.google.com/chrome/answer/95464) during development.
>
> If you build and release your Flutter web app, it'll work as expected. For more information on how to build for release, check the Flutter documentation (https://flutter.dev/docs/deployment/web#building-the-app-for-release).

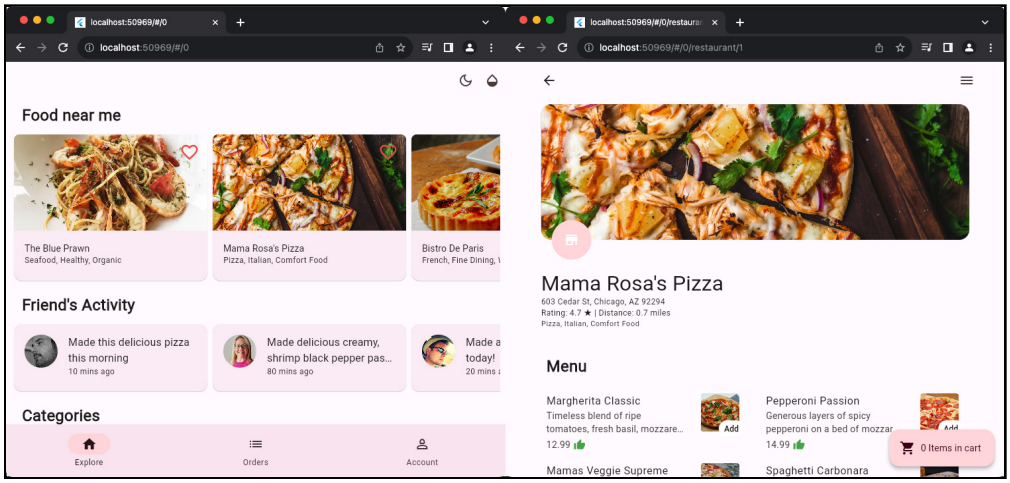
Go through the Yummy UI flow, and you'll see that the web browser's address bar changes:



If you change the `tab` query parameter's value to **0**, **1** or **2**, the app automatically switches to that tab.

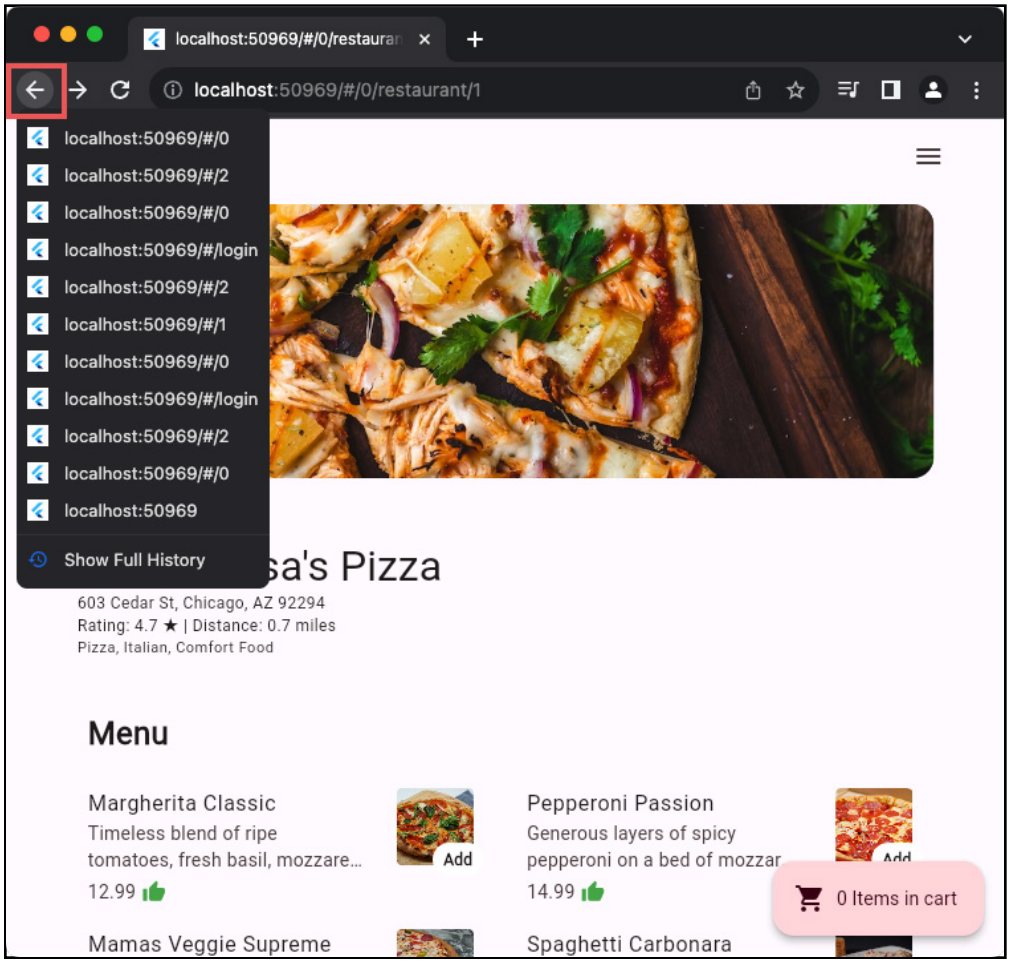


Next on the **Explore** tab, tap on a restaurant



Notice that the app stores the entire browser history. Pretty cool!

Tap the **Back** and **Forward** buttons and the app restores that state! How cool is that? You can also **long-press** the **Back** button to jump to a specific state in the browser history.



Congratulations on learning how to work with deep links in your Flutter app!

Key Points

- The app notifies `RouteInformationProvider` when there's a new route to navigate to.
- The provider passes the route information to `RouteInformationParser` to parse the URL string.
- The parser converts the route information state to and from a URL string.
- `GoRouter` converts route information state to and from a `RouteMatchList`.
- `GoRouter` supports deep linking and web browser address bar paths out of the box.
- In development mode, the Flutter web app doesn't persist data between app launches. The web app generated in release mode will work on other browsers.

Where to Go From Here?

Flutter renders web apps in two different ways. Explore how that works in the Flutter documentation on web renderers (https://flutter.dev/docs/development/tools/web-renderers).

For more examples of various navigation use-cases with `GoRouter`, check out these examples (https://github.com/flutter/packages/tree/main/packages/go_router/example).

In this chapter, you continued to learn how the Router API works behind the scenes and explore how to test and perform deep links on iOS, Android and the Web.

Deep linking helps bring users to specific destinations within your app, building better user engagement!

Flutter's ability to support routes and navigation for multiple platforms isn't just powerful; it's magical.

Section IV: Networking, Persistence & State

Most apps interact with the network to retrieve data and then persist that data locally in some form of cache, such as a database. In this section, you'll build a new app that lets you search the Internet for recipes, bookmark recipes, and save their ingredients into a shopping list.

You'll learn about making network requests, parsing the network JSON response, and saving data in a SQLite database. You'll also get an introduction to using Dart streams.

Finally, this section will also dive deeper into the important topic of app state, which determines where and how your user interface stores and refreshes data in the user interface as a user interacts with your app.

Chapter 10: Handling Shared Preferences

By Kevin David Moore

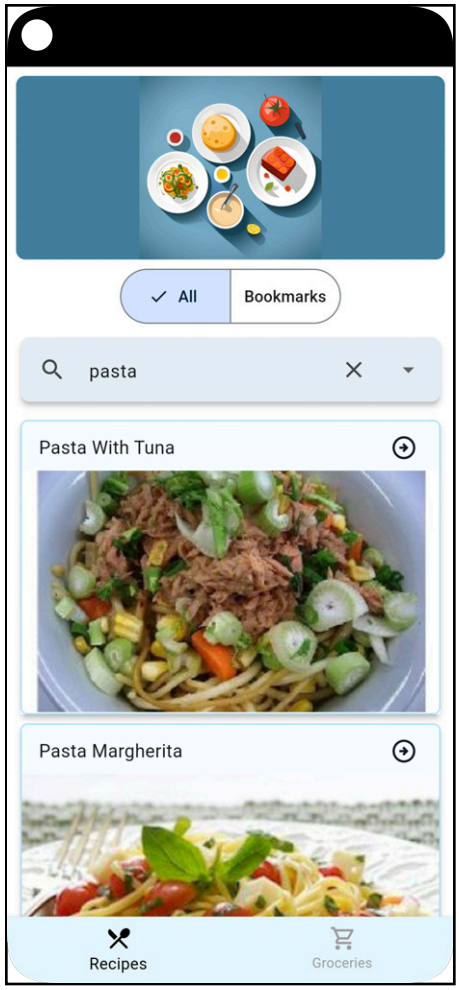
Picture this: You're browsing recipes and find one you like. You're in a hurry and want to bookmark it to check it later. Can you build a Flutter app that does that? You sure can! Read on to find out how.

In this chapter, your goal is to learn how to use the **shared_preferences** plugin to save important pieces of information to your device.

You'll start with a new project showing two tabs at the bottom of the screen for two views: Recipes and Groceries.

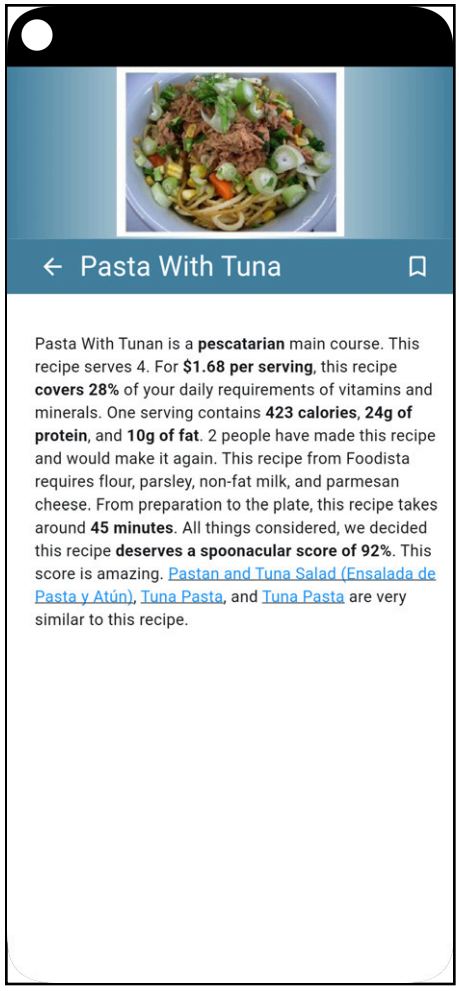
The first screen is where you'll search for recipes you want to prepare. Once you find a recipe you like, just bookmark it, and the app will add the recipe to your **Bookmarks** page. It will also add all the ingredients you need to your shopping list. You'll use a web API to search for recipes and store the ones you bookmark in a local database.

The completed app will look something like this:

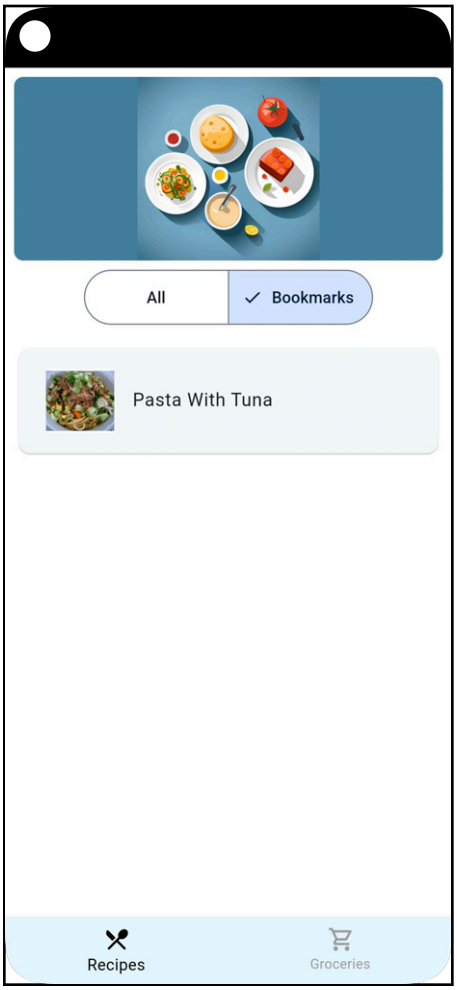


This shows the **Recipes** tab with the results you get when searching for **Pasta**. It's as easy as typing in the search text field and tapping the **Search** icon. The app stores your search term history in the combo box to the right of the text field.

When you tap a card, you'll see something like this:

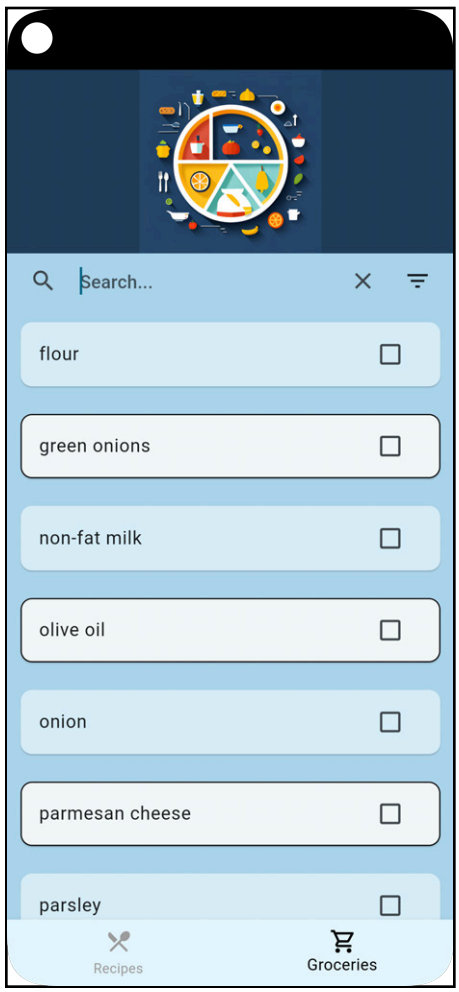


To save a recipe, just tap the **Bookmark** button. When you tap on the **Bookmarks** selector, you'll see that the recipe has been saved:



If you don't want the recipe anymore, swipe left or right, and you'll see a delete button that allows you to remove it from the list of bookmarked recipes.

The **Groceries** tab shows the ingredients you need to make the recipes you've bookmarked.



You'll build this app over the next few chapters. In this chapter, you'll use shared_preferences to save simple data like the selected tab and also to cache the searched items in the Recipes tab.

By the end of the chapter, you'll know:

- What shared preferences are.
- How to use the shared_preferences plugin to save and retrieve objects.

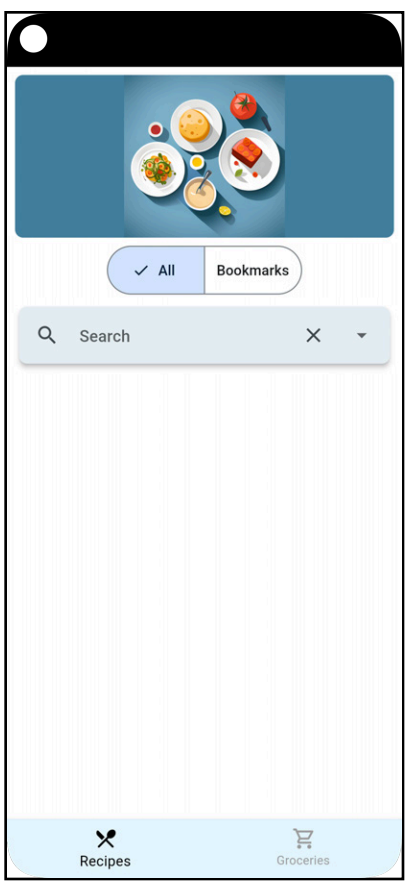
> **Note**: Feel free to explore the entire app. There is a lot there to explore and learn that isn't covered in the book. Copy any code you like for your projects.

Now that you know what your goal is, it's time to jump in!

Getting Started

Open the **starter** project for this chapter in Android Studio. Open the **pubspec.yaml** file and click `pub get`, then run the app.

Notice the two tabs at the bottom — each will show a different screen when you tap it. Only the **Recipes** screen currently shows any UI. It looks like this:



App Libraries

The starter project includes quite a few libraries in **pubspec.yaml**:

```
dependencies:
  ...
  auto_size_text:
  flutter_adaptive_scaffold:
  desktop_window:
  path:
  cached_network_image:
  flutter_slidable:
  platform:
  freezed_annotation:
  flutter_svg:
  ....
  flutter_riverpod:
```

Here's what they help you do:

- **auto_size_text**: Useful library for ensuring text fits in the given space.
- **flutter_adaptive_scaffold**: Library changing your UI based on changing sizes. Useful for the desktop, web and folding phones.
- **desktop_window**: Library for the desktop app for setting the window size.
- **path**: Library for handling files.
- **cached_network_image**: Download and cache the images you'll use in the app.
- **flutter_slidable**: Build a widget that lets the user slide a card left and right to perform different actions, like deleting a saved recipe.
- **platform**: For accessing platform-specific information.
- **freezed_annotation**: Part of the freezed library. Generates useful JSON and related utility functions.
- **flutter_svg**: Load SVG images without the need to use a program to convert them to vector files.
- **flutter_riverpod**: State management library. You'll learn more about this library in Chapter 13, "Managing State".

Now that you've looked at the libraries take a moment to think about how you save data before you begin coding your app.

Saving Data

There are three primary ways to save data to your device:

1. Write formatted data, like JSON, to a file.
2. Use a library or plugin to write simple data to a shared location.
3. Use a SQLite database.

Writing data to a file is simple, but it requires you to handle reading and writing data in the correct format and order.

You can also use a library or plugin to write simple data to a shared location managed by the platform, like iOS and Android. This is what you'll do in this chapter.

You can save the information to a local database for more complex data. You'll learn more about that in **Chapter 15, "Saving Data Locally"**.

Saving Small Bits of Data

Why would you save small bits of data? Well, there are many reasons to do this. For example, you could save the user ID when the user has logged in — or if the user has logged in at all. You could also save the onboarding state or data the user has bookmarked to consult later.

Note that this simple data saved to a shared location is lost when the user uninstalls the app.

The shared_preferences Plugin

shared_preferences is a Flutter plugin that allows you to save data in a key-value format so you can easily retrieve it later. Behind the scenes, it uses the aptly named **SharedPreferences** on Android and the similar **UserDefaults** on iOS.

For this app, you'll learn to use the plugin by saving the search terms the user entered and the tab currently selected.

One of the great things about this plugin is that it doesn't require any setup or configuration. Just create an instance of the plugin, and you're ready to fetch and save data.

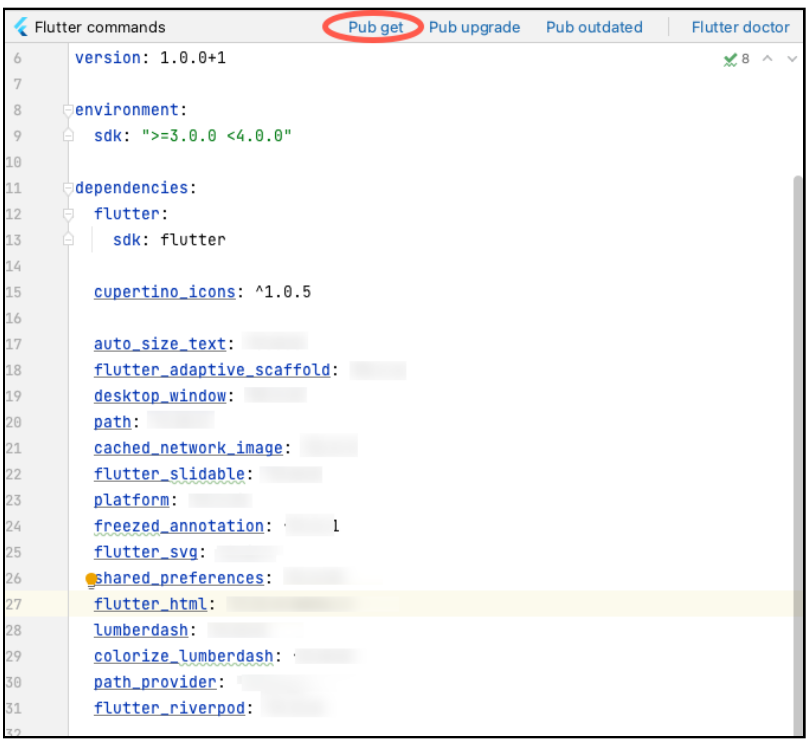
> **Note**: The shared_preferences plugin gives you a quick way to persist and retrieve data, but it only supports saving simple properties like strings, numbers, and Boolean values.
>
> In later chapters, you'll learn about alternatives you can use when you want to save complex data.
>
> Be aware that shared_preferences is not a good fit to store sensitive data. To store passwords or access tokens, check out the Android Keystore for Android and Keychain Services for iOS, or consider using the **flutter_secure_storage** plugin.

To use shared_preferences, you first need to add it as a dependency. Open **pubspec.yaml** and underneath the `flutter_svg` library, add the following:

```
shared_preferences: ^2.2.0
```

Make sure you indent it the same as the other libraries.

Now, click the **Pub Get** button to get the shared_preferences library.



You can also run pub get from the command line:

```
flutter pub get
```

How Does it Work?

The shared_preferences library uses the system's API's to store data into a file. These are small bits of information like integers, strings or Booleans. It has three main sets of function calls:

1. `setXXX`: Set methods save the data of that specific data type.
2. `getXXX`: Get methods retrieve the data of that specific data type.
3. `clear()`: This method deletes all saved data.
4. `remove()`: This method removes a specific value.

All of these methods except the `clear()` method use a **key** to access an item. By giving the library a unique key, you can store, retrieve and delete specific items. Here's an example:

```
final CURRENT_USER_KEY = 'CURRENT_USER_KEY';
final sharedPrefs = await SharedPreferences.getInstance();
sharedPrefs.setString(CURRENT_USER_KEY, '1011442433');
...
sharedPrefs.remove(CURRENT_USER_KEY);
```

In this example, you get an instance of the shared preference library and then set a string using that key. Later on, you remove that item if the user logged out, for example.

There are several other interesting methods:

1. `containsKey()`: Returns true if the key exists.
2. `getBool()`, `getDouble()`, `getInt()`: Methods to retrieve specific types.
3. `setBool()`, `setDouble()`, `setInt()`: Methods to store specific types.

There aren't a lot of methods, but it's a very useful library.

You're now ready to store data. You'll start by saving the **searches** the user makes so they can easily select them again in the future.

Running Code in the Background

To understand the code you'll be adding next, you need to know a bit about running code in the **background**.

Most modern UI toolkits have a **main thread** that runs the UI code. Any code that takes a long time needs to run on a different thread or process so it doesn't block the UI. Dart uses a technique similar to *JavaScript* to achieve this. The language includes these two keywords:

- `async`
- `await`

`async` marks a method or code section as **asynchronous**. You then use the `await` keyword inside that method to wait until an asynchronous process finishes in the background.

Saving UI States

You'll use **shared_preferences** to save a list of saved searches in this section. Later, you'll also save the tab that the user has selected so the app always opens to that tab.

You'll start by preparing your search to store that information.

Adding Shared Preferences as a Provider

This app uses the **Riverpod** library to provide resources to other parts of the app. Chapter 13, "Managing State" covers **Riverpod** in more detail. For now, you want to create an instance of the `SharedPreferences` library on startup and provide it to other parts of the app. To do so, open up **lib/providers.dart** and import shared preferences library:

```
import 'package:shared_preferences/shared_preferences.dart';
```

Then replace `// TODO Add Shared Pref Provider` with the following:

```
final sharedPrefProvider = Provider<SharedPreferences>((ref) {
  throw UnimplementedError();
});
```

This creates a **Riverpod Provider** for our shared preference. Notice how we throw a `UnimplementedError`. This is because you'll provide it in the **main.dart** file.

Open up **main.dart**. Add the shared preferences library and providers import:

```
import 'package:shared_preferences/shared_preferences.dart';
import 'providers.dart';
```

Find `// TODO Add Shared Preferences`. Replace this and the line below it with the following:

```
// 1
final sharedPrefs = await SharedPreferences.getInstance();
// 2
runApp(ProviderScope(overrides: [
  // 3
  sharedPrefProvider.overrideWithValue(sharedPrefs),
], child: const MyApp()));
```

Here's what that code's doing:

1. Create an instance of the **SharedPreferences** library. Notice the `await` keyword. This will wait until the instance is created.
2. **Riverpod** requires a `ProviderScope` above the app where you'll provide providers. These allow you to make functionalities like shared_preferences available to other parts of the app.
3. Override the `sharedPrefProvider` value with the shared pref you just created.

Because the main function has the `async` keyword, you can `await` getting an instance of `SharedPreferences`. By using `overrideWithValue()`, you replace the `unimplemented exception` with a real value. `ProviderScope` will be discussed more in Chapter 13, "Managing State" but is required for **Riverpod** to run.

Next, you'll add an entry to the search list.

Adding an Entry

First, you'll change the UI so that when the user presses the search icon, the app will add the search entry to the search list.

Open **lib/ui/recipes/recipe_list.dart**, locate `// TODO: Add imports` and replace it with:

```
import '../../providers.dart';
```

That imports the provider's file.

Next, you'll give each search term a unique key. Find `// TODO Add Search Index Key` and replace it with the following:

```
static const String prefSearchKey = 'previousSearches';
```

All preferences need to use a unique key. Here, you're simply defining a constant for the **preference search key**.

Saving Previous Searches

Still in **recipe_list.dart**, replace `// TODO Save Current Index` with:

```
// 1
final prefs = ref.read(sharedPrefProvider);
// 2
prefs.setStringList(prefSearchKey, previousSearches);
```

1. **ref.read** extracts the preferences from the provider you set up previously.
2. Saves the list of previous searches using the `prefSearchKey` key.

The `setStringList` method is a nice way to save a list of strings. Next, replace `// TODO: TODO Get Current Index` with the following:

```
// 1
final prefs = ref.read(sharedPrefProvider);
// 2
if (prefs.containsKey(prefSearchKey)) {
  // 3
  final searches = prefs.getStringList(prefSearchKey);
  // 4
  if (searches != null) {
    previousSearches = searches;
  } else {
    previousSearches = <String>[];
  }
}
```

Here, you:

1. Extract the preferences object.
2. Check if the **preference** for your saved list already exists.
3. Get the list of **previous searches**.
4. If the list is not `null`, set the previous searches, otherwise initialize an empty list.

This method is called when the **recipe list** starts loading any previous searches.

Showing the Previous Searches

To perform a search, you first need to clear any of your variables and save the new search value. Take a look at `startSearch()`:

```
void startSearch(String value) {
  //1
  if (value.isEmpty) {
    return;
  }
  // 2
  setState(() {
    // 3
    currentSearchList.clear();
    currentCount = 0;
    currentEndPosition = pageCount;
    currentStartPosition = 0;
    hasMore = true;
    value = value.trim();

    // 4
    if (!previousSearches.contains(value)) {
      // 5
      previousSearches.add(value);
      // 6
      savePreviousSearches();
    }
  });
}
```

In this method, you:

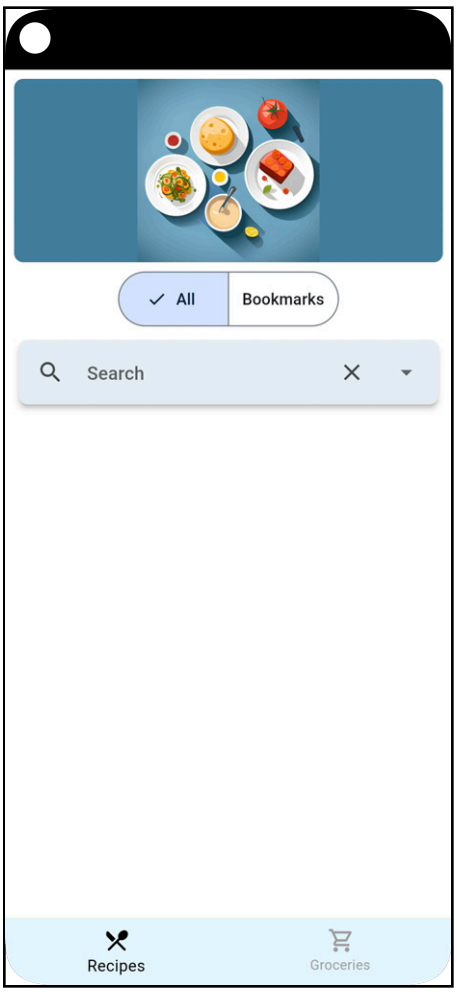
1. Checks the input value to make sure it's not empty.
2. Tell the system to update the widgets by calling `setState()`.
3. Clear the current search list and reset the `currentCount`, `currentStartPosition` and `currentEndPosition`.
4. Check to ensure the **search text** hasn't already been added to the previous search list.
5. Add the **search item** to the previous search list.
6. Save the new list of previous searches.

You used a text field with a drop-down menu to show the list of previous text searches. That's a row with a `TextField` and a `CustomDropDownMenuItem`. The menu item shows the search term and an icon on the right. It will look something like this:

Tapping the **X** will delete the corresponding entry from the list.

Testing the App

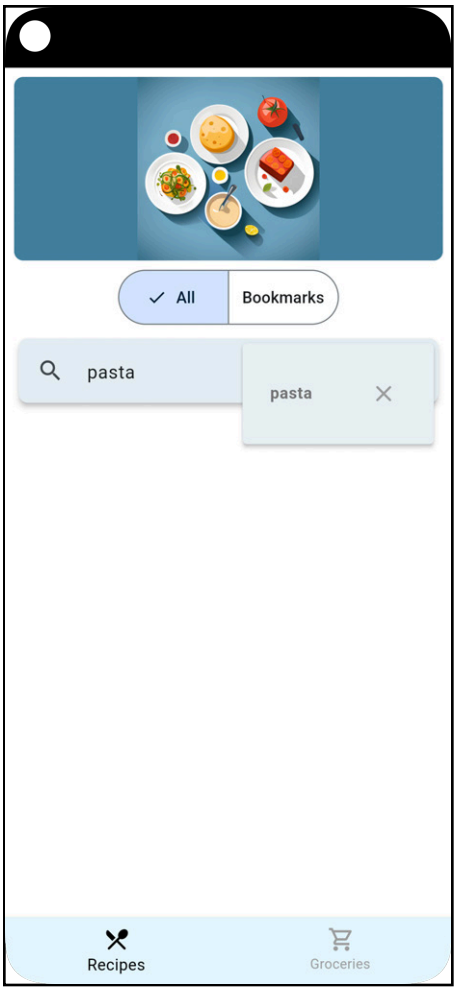
It's time to test the app. You'll see something like this:



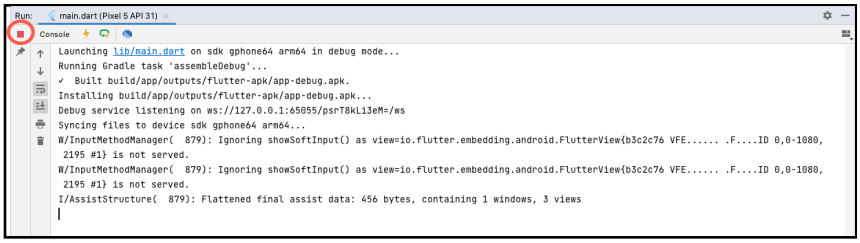
The arrow button displays a menu when tapped and calls the method `onSelected()` when the user selects a menu item.

Enter a food item like **pasta** and you hit the search button. Then make sure that the app adds your search entry to the drop-down list.

Don't worry about errors — that happens when no data exists. Your app should look like this when you tap the drop-down arrow:



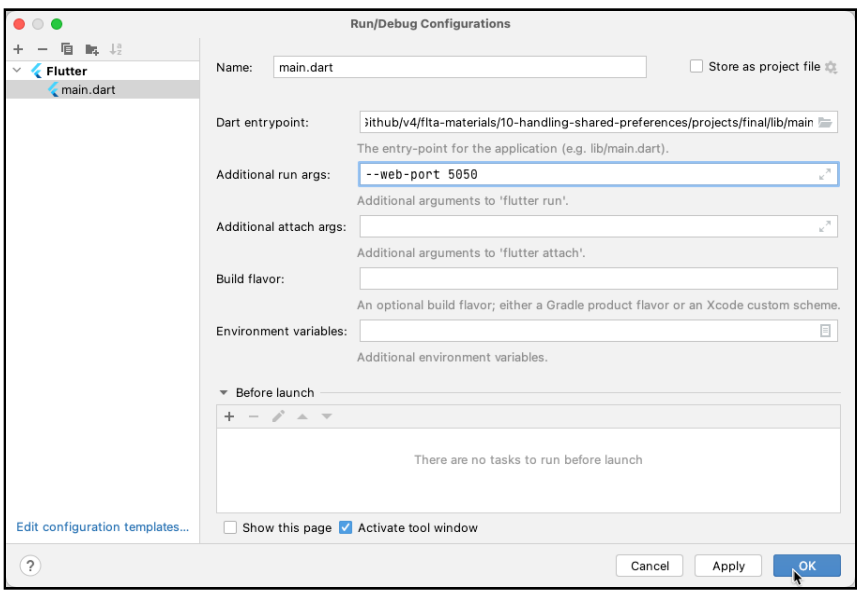
Now, stop the app by clicking the red stop button.



Run the app again and tap the drop-down button. The pasta entry is there. It's time to celebrate. :]

The next step is to use the same approach to save the selected tab.

> **Note**: If you're testing this on the web, you may notice that the drop-down menus are empty. This is because Android Studio will use random port numbers for the web, and this will cause different values to be shown. To fix this, you need to start the web with the same port number each time. You can do that by adding the `--web-port` launch parameter.



> This will ensure you'll see the same list each time.

Saving the Selected Tab

In this section, you'll use shared_preferences to save the current UI tab that the user has navigated to.

Open **lib/ui/main_screen.dart** and add the following import:

```
import '../providers.dart';
```

Next, replace `// TODO Add Index Key` with:

```
static const String prefSelectedIndexKey = 'selectedIndex';
```

You'll use this constant for the selected index **preference key**.

Next, add this in the `saveCurrentIndex()` method by replacing `// TODO Save Current Index` with this:

```
final prefs = ref.read(sharedPrefProvider);
prefs.setInt(prefSelectedIndexKey, _selectedIndex);
```

Here, you:

1. **ref.read** extracts the shared preferences as usual.
2. Save the **selected index** as an integer.

Now, find and replace `// TODO Get Current Index` with this:

```
  // 1
final prefs = ref.read(sharedPrefProvider);
  // 2
if (prefs.containsKey(prefSelectedIndexKey)) {
    // 3
  setState(() {
    final index = prefs.getInt(prefSelectedIndexKey);
    if (index != null) {
      _selectedIndex = index;
    }
  });
}
```

With this code, you:

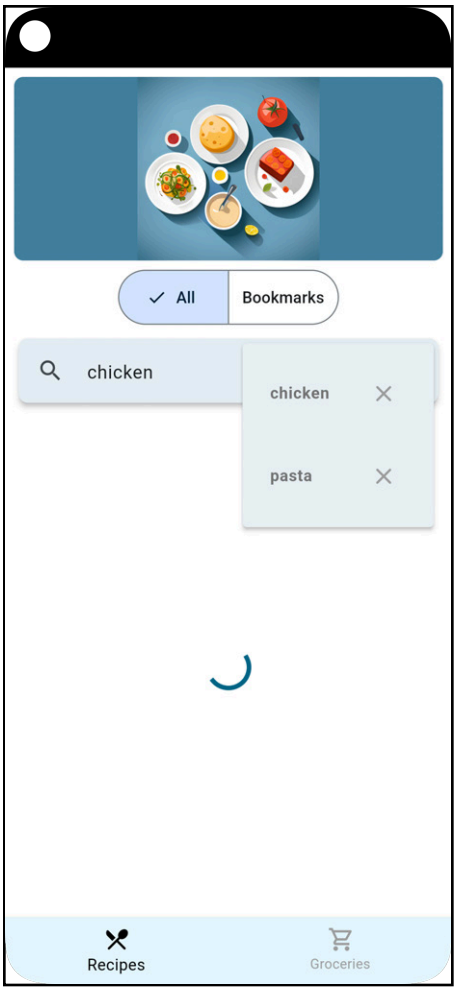
1. Get the shared preferences reference.
2. Check if a **preference** for your current index already exists.

3. Get the **current index** and update the state accordingly.

Now, *hot reload* the app and select either the **first** or the **second** tab.

Go to the **Groceries** tab and quit the app. Run it again to make sure the app uses the **saved index** to go to the Groceries tab when it starts.

At this point, your app should show a list of **previously searched** items and also take you to the last selected tab when you start the app again. Here's what it will look like:



Congratulations! You've saved the state for both the **current tab** and any **previous searches** the user made.

Key Points

- There are multiple ways to save data in an app: to **files**, in **shared preferences** and to a **SQLite** database.
- Shared preferences are best used to store simple, **key-value pairs** of primitive types like `strings`, `numbers` and `Booleans`.
- An example of when to use **shared preferences** is to save the tab a user is viewing, so the next time the user starts the app, they're brought to the same tab.
- The `async`/`await` keyword pair lets you run **asynchronous** code off the main UI thread and then wait for the response. An example is getting an instance of `SharedPreferences`.
- The shared_preferences plugin shouldn't be used to hold sensitive data. Instead, consider using the **flutter_secure_storage** plugin.

Where to Go From Here?

In this chapter, you learned how to persist simple data types in your app using the shared_preferences plugin.

If you want to learn more about Android **SharedPreferences**, go to https://developer.android.com/reference/kotlin/android/content/SharedPreferences?hl=en.

For iOS, check **UserDefaults** https://developer.apple.com/documentation/foundation/userdefaults.

In the next chapter, you'll continue building the same app and learn how to serialize JSON in preparation for getting data from the internet. See you there!

Chapter 11: Serialization With JSON

By Kevin David Moore

In this chapter, you'll learn how to **serialize** JSON data into model classes. A **model class** represents **data structure** and defines **attributes** and **operations** for a particular object. An example is a recipe model class, which usually has a title, an ingredient list and steps to cook it.

You'll continue with the previous project, which is the starter project for this chapter. You'll add a class that models a recipe, and its properties. Then, you'll integrate that class into the existing project.

By the end of this chapter, you'll know:

- How to **serialize JSON** into model classes.
- How to use Dart tools to automate the generation of model classes from JSON.

What is JSON?

JSON, which stands for **JavaScript Object Notation**, is an **open-standard** format used on the web and in mobile clients. It's the most widely used format for **Representational State Transfer** (REST)-based APIs that servers provide (https://en.wikipedia.org/wiki/Representational_state_transfer). If you talk to a server that has a **REST API**, it will most likely return data in a **JSON format**. An example of a JSON response looks something like this:

```
{
  "results": [
    {
      "id": 296687,
      "title": "Chicken",
      "image": "https://spoonacular.com/recipeImages/
296687-312x231.jpeg",
      "imageType": "jpeg"
    },
    ...
    ]
  }
```

That's an example recipe response containing a list of results with four fields inside an object.

While it's possible to treat the JSON as just a long string and try to **parse** out the data, it's much easier to use a package that already knows how to do that. Flutter has a built-in package for **decoding** JSON, but in this chapter, you'll use the **json_serializable** and **json_annotation** packages to help make the process easier.

> **Note**: **JSON parsing** is the process of converting a JSON object in **String format** to a **Dart object** that can be used inside a program.

Flutter's built-in **dart:convert** package contains methods like `json.decode()` and `json.encode()`, which converts a JSON string to a `Map<String, dynamic>` and back. While this is a step ahead of manually parsing JSON, you'd still have to write extra code that takes that map and puts the values into a new class.

The **json_serializable** package is useful because it can generate model classes for you according to the annotations you provide via **json_annotation**. Before taking a look at **automated serialization**, you need to see how to manually serialize JSON.

Writing the Code Yourself

So, how do you go about writing code to serialize JSON yourself? Typical model classes have `toJson()` and `fromJson()` methods. The `toJson()` method helps to convert objects into JSON strings, and the `fromJson()` method helps to parse a JSON string into an object so you can use it inside the program.

In the next section, you learn how to use **automated serialization**. For now, you don't need to type this into your project, but you need to understand the methods to convert the JSON above to a model class.

First, you'd create a `Recipe` model class:

```
class Recipe {
  final String uri;
  final String label;

  Recipe({this.uri, this.label});
}
```

Then you'd add a `toJson()` factory method and a `fromJson()` method:

```
factory Recipe.fromJson(Map<String, dynamic> json) {
  return Recipe(json['uri'] as String, json['label'] as String);
}

Map<String, dynamic> toJson() {
  return <String, dynamic>{ 'uri': uri, 'label': label}
}
```

In `fromJson()`, you grab data from the JSON map variable named `json` and convert it to arguments you pass to the `Recipe` constructor. In `toJson()`, you construct a map using the JSON field names.

While it doesn't take much effort to do that by hand for two fields, what if you had multiple model classes, each with, say, five fields, or more? What if you renamed one of the fields? Would you remember to rename all of the occurrences of that field?

The more model classes you have, the more complicated it becomes to maintain the code behind them. Fear not, that's where automated code generation comes to the rescue.

Automating JSON Serialization

In this chapter, you'll use two packages: **json_annotation** and **json_serializable** from Google.

You use the first to add annotations to model classes so that **json_serializable** can generate **helper classes** to convert a JSON `string` to a model and back.

To do that, you mark a class with the `@JsonSerializable()` annotation so the **builder package** can generate code for you. Each field in the class should either have the same name as the field in the JSON string, or use the `@JsonKey()` annotation to give it a different name.

Most builder packages work by generating a **.part** file. That will be a file that's automatically created for you. All you need to do is add a few factory methods, which will call the generated code.

> **Note**: The **freezed** package is also used in the project and uses the **json_serializable** package to generate serialization code. You could just use the freezed package by itself if you wanted to, as it has additional functionality.

Adding Dependencies for JSON Serialization and Deserialization

Continue with your current project, or open the **starter** project in the **projects** folder. Add the following package to **pubspec.yaml** in the Flutter `dependencies` section underneath and make sure it's aligned with `flutter_riverpod`:

```
json_annotation: ^4.8.1
```

In the `dev_dependencies` section replace `# TODO: Add new dev_dependencies` with the following:

```
json_serializable: ^6.7.1
```

Make sure these are all indented correctly. `build_runner`, which is already included, is the package that helps generate the code.

Finally, click the **Pub get** button you should see at the top of the file, or run `flutter pub get` in the terminal. You're now ready to generate **model classes**.

Generating Model Classes From JSON

The JSON that you're trying to serialize looks something like this:

```
{
  "results": [
    {
      "id": 296687,
      "title": "Chicken",
      "image": "https://spoonacular.com/recipeImages/
296687-312x231.jpeg",
      "imageType": "jpeg"
    },
    {
      "id": 379523,
      "title": "Chicken",
      "image": "https://spoonacular.com/recipeImages/
379523-312x231.jpeg",
      "imageType": "jpeg"
    },
    ...
  ],
  "offset": 0,
  "number": 10,
  "totalResults": 51412
}
```

- The `results` list is a list of recipe objects.
- Each recipe has an `id`, `title`, `image` and `imageType`.
- `offset` is the **starting position** for the search. 0 means start at the beginning, while a value of 10 would start at the 10th element. This is useful for paging long lists.
- `number` is the total number of results returned in this list.
- `totalResults` is the total results available for this search query.

Your next step is to generate the **classes** that model that data.

Creating Model Classes

Start by opening **lib/network/spoonacular_model.dart** and add the following import at the top:

```
import 'package:json_annotation/json_annotation.dart';
```

```
import '../data/models/models.dart';

part 'spoonacular_model.g.dart';
```

The `json_annotation` library lets you mark a class as **serializable**. The file **spoonacular_model.g.dart** doesn't exist yet, you'll generate it in a later step.

Next, replace `// TODO: Add SpoonacularResults class` with a class named `SpoonacularResults` with a `@JsonSerializable()` annotation:

```
@JsonSerializable()
class SpoonacularResults {
  // TODO: Add Fields
  // TODO: Add Constructor
  // TODO: Add fromJson
  // TODO: Add toJson
}

// TODO: Add SpoonacularResult
```

That marks the `SpoonacularResults` class as serializable so the **json_serializable** package can generate the corresponding **.g.dart** file.

Command-Click on `JsonSerializable`, scroll down, and you'll see its definition:

```
...

/// Creates a new [JsonSerializable] instance.
const JsonSerializable({
  @Deprecated('Has no effect') bool? nullable,
  this.anyMap,
  this.checked,
  this.constructor,
  this.createFieldMap,
  this.createFactory,
  this.createToJson,
  this.disallowUnrecognizedKeys,
  this.explicitToJson,
  this.fieldRename,
  this.ignoreUnannotated,
  this.includeIfNull,
  this.converters,
  this.genericArgumentFactories,
  this.createPerFieldToJson,
});

...
```

For example, you can make the class `nullable` and add extra checks for **validating** JSON properly. Close the **json_serialization.dart** source file after reviewing it.

Converting to and From JSON

Now, you need to add JSON conversion methods within the `SpoonacularResults` class. Return to **spoonacular_model.dart** and replace `// TODO: Add Fields` with:

```
List<SpoonacularResult> results;
int offset;
int number;
int totalResults;
```

This is the list of results, offset, number and total results. The `SpoonacularResult` class doesn't exist yet. Next, replace `// TODO: Add Constructor` with:

```
SpoonacularResults({
  required this.results,
  required this.offset,
  required this.number,
  required this.totalResults,
});
```

The `required` annotation says that these fields are mandatory when creating a new instance.

Next, replace `// TODO: Add fromJson` with:

```
factory SpoonacularResults.fromJson(Map<String, dynamic> json)
=>
    _$SpoonacularResultsFromJson(json);
```

The above method converts the JSON string to a `SpoonacularResults` object.

Note that the method to the right of the arrow operator doesn't exist yet and will be present in `spoonacular_model.g.dart` after generating the code, so ignore any red squiggles. They'll be created later by running the **build_runner** command.

Also note that this is a `factory` method. That's because you need a class-level method when creating the instance.

> **Note**: To know more about factory methods check Chapter 9 in the Dart Apprentice: Fundamentals Book (https://www.kodeco.com/books/dart-apprentice-fundamentals/v1.0/chapters/9-constructors#b8d9168fc0febd62f39468aa2163ed3dc1155760373beaad55df1175605bda58).

Now, replace `// TODO: Add toJson` with the following:

```
Map<String, dynamic> toJson() =>
_$SpoonacularResultsToJson(this);
```

The above method connects `SpoonacularResultsToJson` to the `toJson()` method.The `_$SpoonacularResultsToJson` method will be created for you. This method will return a map and is useful for saving its data.

Then, find `// TODO: Add SpoonacularResult`, and replace it with the following new class, continuing to ignore the red squiggles:

```
// 1
@JsonSerializable()
class SpoonacularResult {
  // 2
  int id;
  String title;
  String image;
  String imageType;

  // 3
  SpoonacularResult({
    required this.id,
    required this.title,
    required this.image,
    required this.imageType,
  });

  // 4
  factory SpoonacularResult.fromJson(Map<String, dynamic> json)
=>
      _$SpoonacularResultFromJson(json);


  Map<String, dynamic> toJson() =>
_$SpoonacularResultToJson(this);
}
```

Here's what this code does:

1. Marks the class `JsonSerializable`.
2. Defines several fields: `id`, `title`, `image` and `imageType`.
3. Defines a `constructor` that accepts these fields.
4. Adds the methods for JSON serialization.

Now, uncomment the rest of the code in **lib/network/spoonacular_model.dart**. This will add two new classes, `SpoonacularRecipe` and `ExtendedIngredient` plus some conversion methods. This is to save time, as the detailed recipe information from **Spoonacular** is quite extensive.

For your next step, you'll generate the code to automatically parse the recipes' JSON.

Generating the code for JSON Serialization and Deserialization

Open the terminal in Android Studio by clicking the `Terminal` panel in the lower left, or by selecting **View ▸ Tool Windows ▸ Terminal**, and type:

```
dart run build_runner build
```

The expected output will look something like this:

```
[INFO] Generating build script completed, took 155ms
[INFO] Precompiling build script... completed, took 3.3s
[INFO] Building new asset graph completed, took 372ms
[INFO] Checking for unexpected pre-existing outputs. completed,
took 15.0s
[INFO] Generating SDK summary completed, took 2.3s
[INFO] Running build completed, took 11.8s
[INFO] Caching finalized dependency graph completed, took 44ms
[INFO] Succeeded after 11.8s with 11 outputs (82 actions)
→
```

> **Note**: If you have problems running the command, ensure you've installed Flutter on your computer and you have a path set up to point to it. See Flutter installation documentation for more details, https://docs.flutter.dev/get-started/install.

You may encounter a problem that looks like this:

```
[INFO] Found 6 declared outputs which already exist on disk.
This is likely because the`.dart_tool/build` folder was deleted,
or you are submitting generated files to your source repository.
Delete these files?
1 - Delete
2 - Cancel build
3 - List conflicts
1
```

Choose 1 to delete the files.

This command creates the **spoonacular_model.g.dart** file, which has all the generated code in the **network** folder. If you don't see the file, right-click on the network folder and choose **Reload from disk**.

If you still don't see it, restart Android Studio, so it recognizes the presence of the newly generated file when it starts up.

If you want the program to run every time you make a change to your file, you can use the `watch` command like this:

```
dart run build_runner watch
```

The command will continue to run and watch for changes to files. To stop the process, you can press Ctrl-C. Now, open **spoonacular_model.g.dart**. Here's the first generated method:

```
// GENERATED CODE - DO NOT MODIFY BY HAND

part of 'spoonacular_model.dart';
// 1
SpoonacularResults _$SpoonacularResultsFromJson(Map<String,
dynamic> json) =>
    SpoonacularResults(
      // 2
      results: (json['results'] as List<dynamic>)
          .map((e) => SpoonacularResult.fromJson(e as
Map<String, dynamic>))
          .toList(),
      // 3
      offset: json['offset'] as int,
      // 4
      number: json['number'] as int,
      // 5
      totalResults: json['totalResults'] as int,
    );
```

Notice that it takes a map of `<String, dynamic>`, which is typical of JSON data in Flutter. The key is the string, and the value will either be a primitive, a list or another map. The method:

1. Returns a new `SpoonacularResults` class.
2. Maps each element of the `results` list to an instance of `SpoonacularResult`.
3. Maps the `offset` key to a `offset` field.

4. Maps the `number` integer to the `number` field.
5. Maps the `totalResults` integer to the `totalResults` field.

You could've written this code yourself, but it can get a bit tedious and is error-prone. Having a tool generate the code for you saves a lot of time and effort. Look through the rest of the file to see how the generated code converts the JSON data to all the other model classes.

Hot restart the app to make sure it still compiles and works as before. You won't see any changes in the UI, but the code is now set up to parse recipe data.

Testing the Generated JSON Code

Now that you can parse model objects from JSON, you'll read one of the JSON files included in the starter project and show one card to make sure you can use the generated code.

Open **ui/recipes/recipe_list.dart** and add the following imports at the top:

```
import 'dart:convert';
import '../../network/spoonacular_model.dart';
import 'package:flutter/services.dart';
```

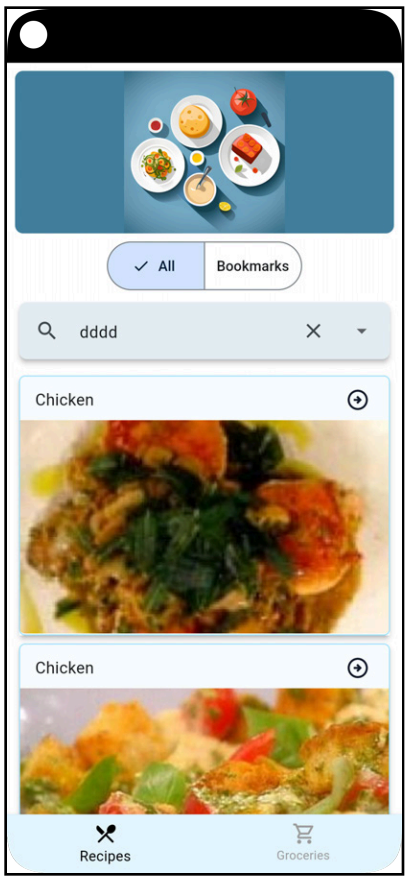
In `fetchData()`, replace `// TODO: Load Recipes` with:

```
// 1
final jsonString = await rootBundle.loadString('assets/
recipes1.json');
// 2
final spoonacularResults =
    SpoonacularResults.fromJson(jsonDecode(jsonString));
// 3
final recipes = spoonacularResultsToRecipe(spoonacularResults);
// 4
final apiQueryResults = QueryResult(
    offset: spoonacularResults.offset,
    number: spoonacularResults.number,
    totalResults: spoonacularResults.totalResults,
    recipes: recipes);
// 5
currentResponse = Future.value(Success(apiQueryResults));
```

This is what that code does:

1. `rootBundle` is from the services page and allows you to load data from the **assets** directory.
2. **Decode** the JSON string and convert it to a `SpoonacularResults` class.
3. Convert that result into a list of recipes.
4. Create a new query result that contains the results. This is the class that will be used in Chapter 12, "Networking in Flutter".
5. Return a `Success` response.

Perform a hot reload, run a search and the app will show some chicken recipe cards:



Mock Service

Now that you've manually loaded a sample JSON file, it's time to implement the **Mock Service**. This service class will randomly load one of two recipe files: One for **chicken** and one for **pasta**.

While in **recipe_list.dart**, comment out the code you just entered and uncomment this code:

```
final recipeService = ref.watch(serviceProvider);
currentResponse = recipeService.queryRecipes(
    searchTextController.text.trim(), currentStartPosition,
pageCount);
```

Open **mock_service/mock_service.dart** and add the following imports:

```
import 'dart:convert';
import 'package:flutter/services.dart';
import '../network/spoonacular_model.dart';
```

Uncomment the code in `loadRecipes()` This will randomly load recipes either from **recipes1.json** or **recipes2.json** in the **assets** folder. Next, open **main.dart** and add the following import:

```
import 'mock_service/mock_service.dart';
```

Then replace `// TODO: Create Mock service` with:

```
final service = await MockService.create();
```

Finally replace `// TODO: Inject mock service` with:

```
serviceProvider.overrideWithValue(service),
```

This will inject this service via the **Riverpod** library. You'll read more about that library in Chapter 13, "Managing State". Do a hot restart *not* reload. Type anything in the search field and press enter, or click the search icon. You should see a list of chicken or pasta recipes.

Now that the data model classes work as expected, you're ready to load recipes from the web. Fasten your seat belt. :]

Key Points

- JSON is an open-standard format used on the web and in mobile clients, especially with REST APIs.
- In mobile apps, JSON code is usually parsed into the model objects that your app will work with.
- You can write JSON parsing code yourself, but it's usually easier to let a JSON package generate the parsing code for you.
- **json_annotation** and **json_serializable** are packages that will let you generate the parsing code.

Where to Go From Here?

In this chapter, you've learned how to create models that you can parse from JSON and then use when you fetch JSON data from the network. If you want to learn more about json_serializable, go to https://pub.dev/packages/json_serializable.

In the next chapter, you'll build on what you've done so far and learn about loading recipes from the internet.

Chapter 12: Networking in Flutter

By Kevin David Moore

Loading data from the network to show it in a UI is a very common task for apps. In the previous chapter, you learned how to serialize JSON data. Now, you'll continue the project to learn about retrieving JSON data from the network.

> **Note**: You can also start fresh by opening this chapter's **starter** project. If you choose to do this, remember to click the **pub get** button or execute `flutter pub get` from Terminal.

By the end of this chapter, you'll know how to:

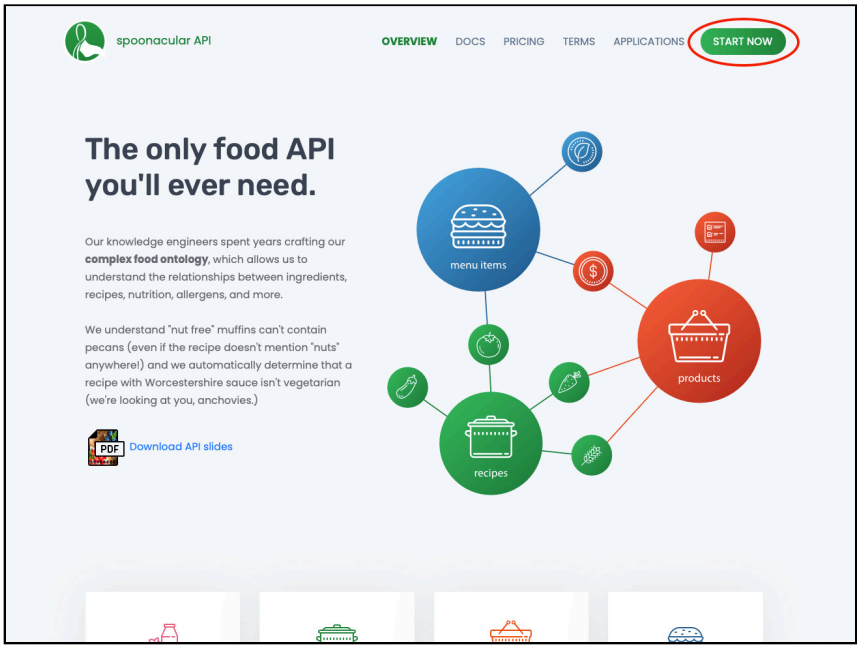
- Sign up for a recipe **API service**.
- Trigger a search for **recipes by name**.
- Convert data returned by the API to **model classes**.
- **Display recipes** in the current UI.

Without further ado, it's time to get started!

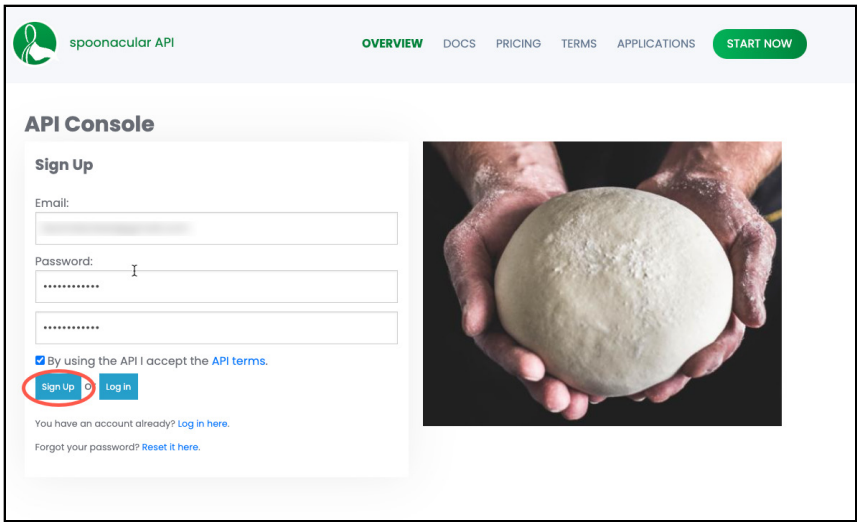
Signing Up With the Recipe API

For your remote content, you'll use the **Spoonacular Food API**. Open this link in your browser: https://spoonacular.com/food-api.

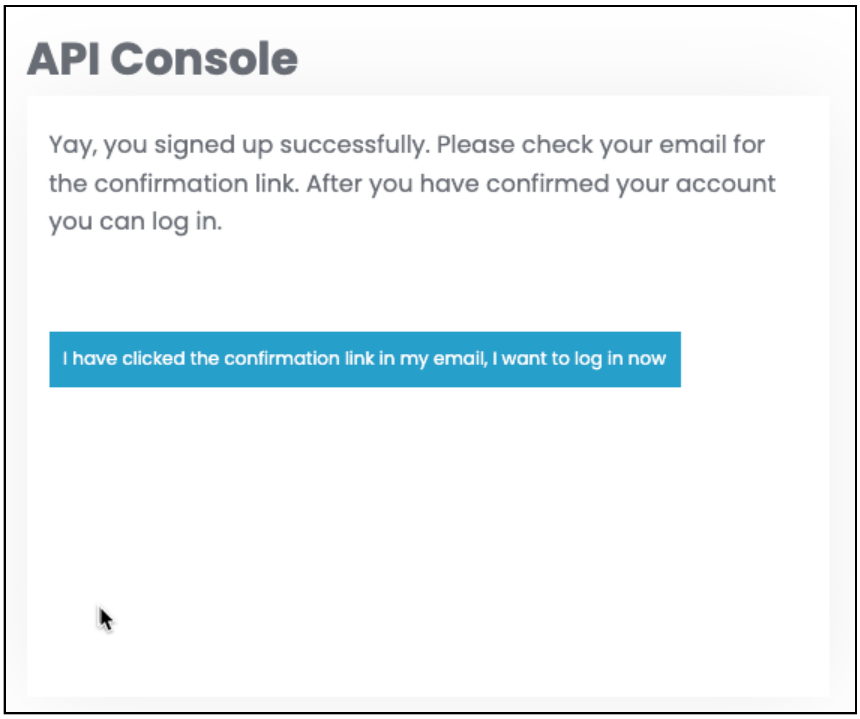
Click the **Start Now** button in the top right to create an account.



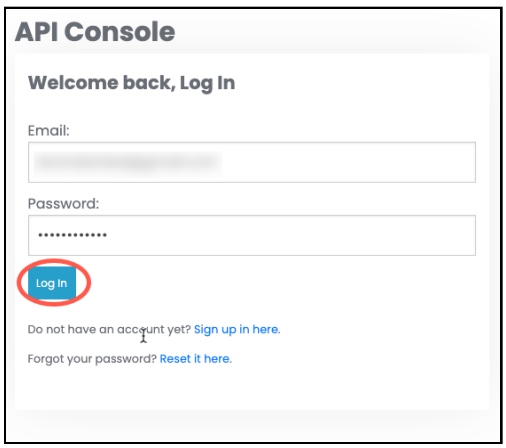
Fill in an email and password, then click the checkbox and sign up. Go through the steps to finish the process. You can choose the free tier.



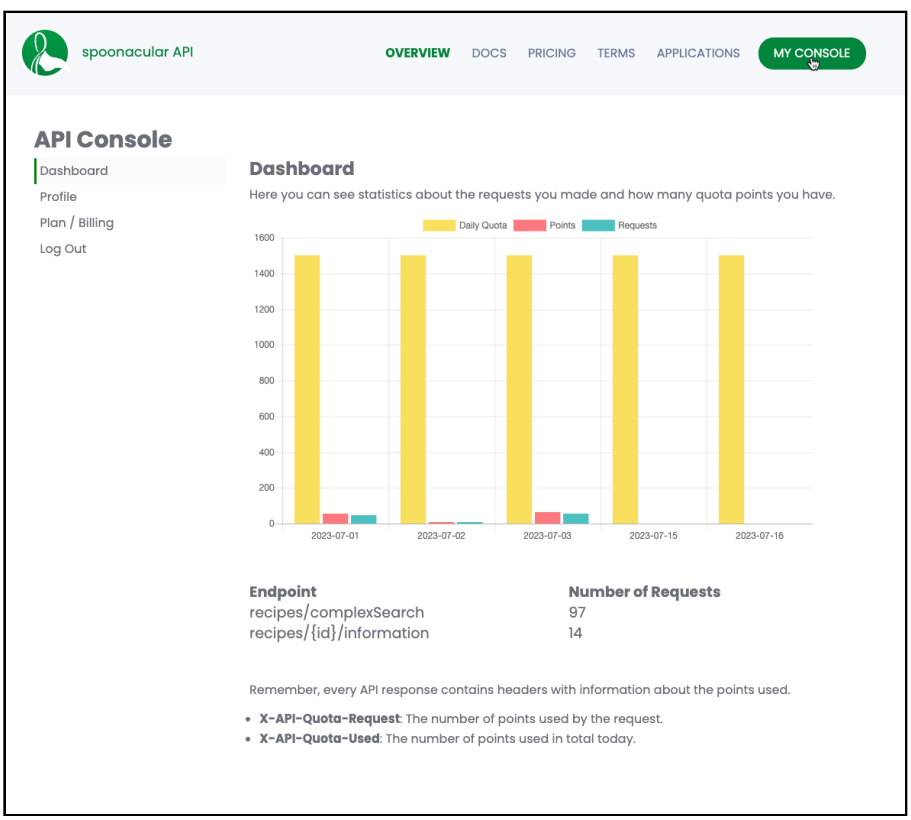
Click **Sign Up**, and you should see this:



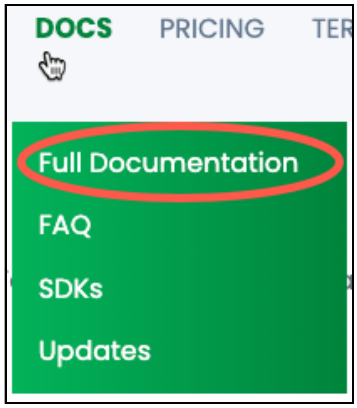
Once you've confirmed your email, visit https://spoonacular.com/food-api/console and log in.



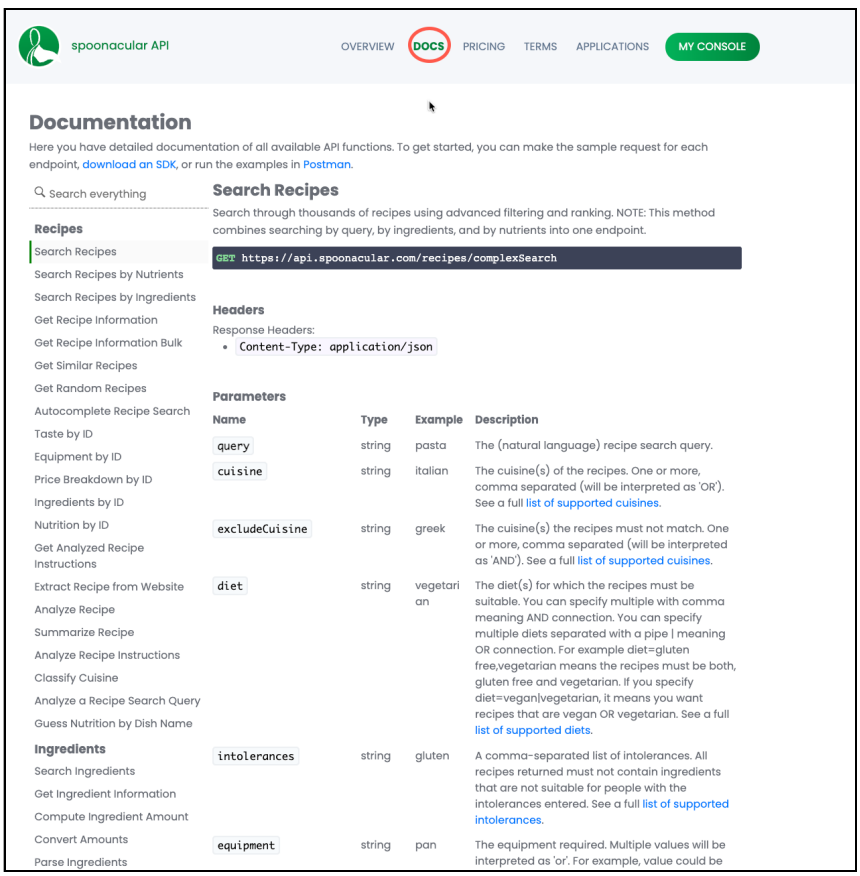
You should see your Console. Once you start making requests, you'll see the graph fill up.



Now go to **docs** and click **Full Documentation**:

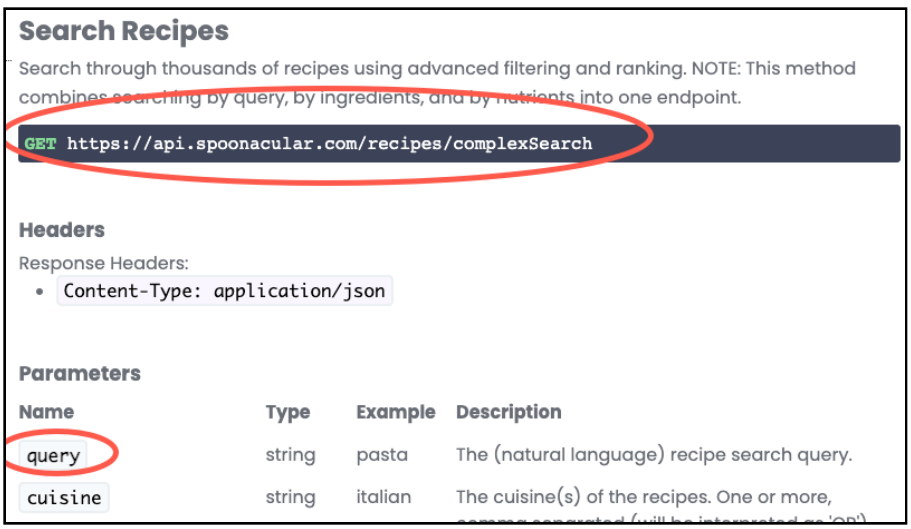


Here, you can see the docs for searching for recipes:



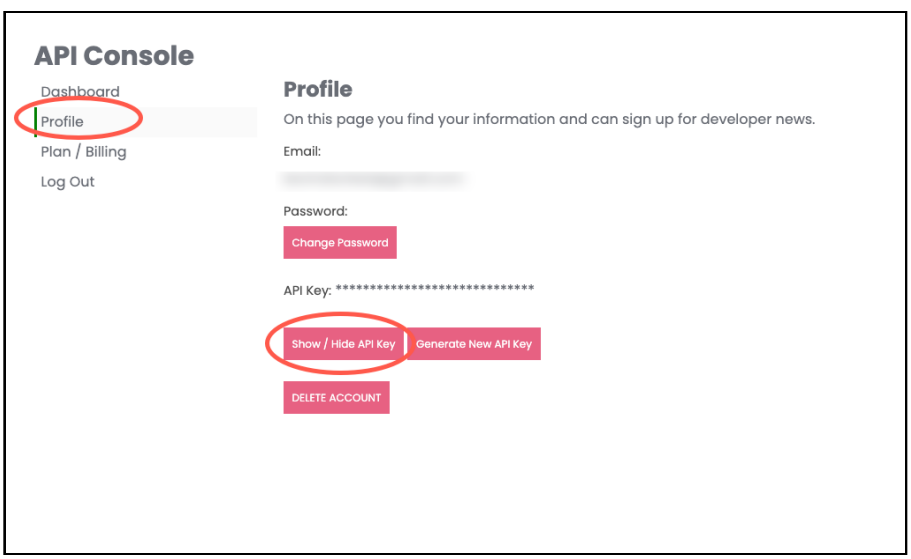
If you scroll down, you can see a lot of fields returned. We're not interested in most of these fields.

You'll see a complete API URL and a list of the parameters available for the `GET` request you'll make.



There's much more API information on this page than you'll need for your app, so you might want to bookmark it for the future.

Click **My Console**, then the **Profile** section and you'll end up on this link https://spoonacular.com/food-api/console#Profile:



Click **Show/Hide API key**. Copy the API Key and save it in a secure place.

For your next step, you'll use your newly created API key to fetch recipes via HTTP requests.

> **Note**: The free developer version of the API is rate-limited. If you use the API a lot, you'll probably receive some JSON responses with errors and emails warning you about the limit.

Preparing the Pubspec File

Open either your project or the chapter's **starter** project. To use the **http** package for this app, you need to add it to **pubspec.yaml**, so open that file and add the following after the **json_annotation** package:

```
http: ^1.1.0
```

Click the **Pub get** button to install the package, or run `flutter pub get` from the **Terminal**.

Using the HTTP package

The package contains only a few files and methods that you'll use in this chapter. The REST protocol has methods such as:

- **GET**: Gets the data.
- **POST**: Posts/sends new data.
- **PUT**: Updates data.
- **DELETE**: Deletes data.

You'll use `GET`, specifically the function `get()` in the http package, to retrieve recipe data from the API. This function uses the API's URL and a list of optional headers to retrieve data from the API service. In this case, you'll send all the information via **query parameters**.

Connecting to the Recipe Service

To fetch data from the recipe API, you'll create a **Dart class** to manage the connection. Such a class file will contain your API Key and URL.

In the Project sidebar, right-click **lib/network**, create a new Dart file and name it **spoonacular_service.dart**. After the file opens, import the HTTP package along with the required files:

```
import 'dart:convert';
import 'dart:developer';
import 'package:http/http.dart' as http;

import '../data/models/recipe.dart';
import '../mock_service/mock_service.dart';
import 'model_response.dart';
import 'query_result.dart';
import 'service_interface.dart';
import 'spoonacular_model.dart';
```

> **Note**: Here, we import the http package as `http` so that we can append `http` to the `get()` method and prevent any naming conflicts or confusion.

Now, add the constants that you'll use when calling the APIs:

```
const String apiKey = '<Add Your Key Here>';
const String apiUrl = 'https://api.spoonacular.com/';
```

Copy the **API key** from your Spoonacular account and replace the existing `apiKey` string with your value.

The `apiUrl` constant holds the base URL for the Spoonacular search API from the recipe API documentation. You'll append the path to this URL to get the data you want.

Still in **spoonacular_service.dart**, add the following class and method to get the data from the API:

```
class SpoonacularService implements ServiceInterface {
   // 1
  Future getData(String url) async {
    // 2
    final response = await http.get(Uri.parse(url));
    // 3
```

```
      if (response.statusCode == 200) {
        // 4
        return response.body;
      } else {
        // 5
        log(response.statusCode.toString());
      }
    }
    // TODO: Add getRecipes
  }
```

Here's a breakdown of what's going on:

1. `getData()` returns a value in `Future`, with an upper case "F", because it takes some time to get the data from the Server. An API's returned data type is determined in the future, lower case "f". `async` signifies this method performs an asynchronous operation.
2. `response` has to wait until the HTTP gets the data from the server. The `await` keyword tells the function to wait. `Response` and `get()` are from the HTTP package. `get()` fetches data from the provided `url`.
3. A `statusCode` of **200** means the request was successful.
4. You `return` the results embedded in `response.body`.
5. Otherwise, print the `statusCode` to the console if you have an error.

> **Note**: To learn more about `Future` and `async` operations, check out chapters 11 and 12 of **Dart Apprentice: Beyond the Basics** book https://www.kodeco.com/books/dart-apprentice-beyond-the-basics.

Now, replace `// TODO: Add getRecipes` with:

```
  // 1
  @override
  Future<Result<Recipe>> queryRecipe(String recipeId) {
    // TODO: implement queryRecipe
    throw UnimplementedError();
  }

  // 2
  @override
  Future<RecipeResponse> queryRecipes(
      String query, int offset, int number) async {
    // 3
```

```
    final recipeData = await getData(
        '${apiUrl}recipes/complexSearch?
apiKey=$apiKey&query=$query&offset=$offset&number=$number');
    // 4
    final spoonacularResults =
        SpoonacularResults.fromJson(jsonDecode(recipeData));
    // 5
    final recipes =
spoonacularResultsToRecipe(spoonacularResults);
    // 6
    final apiQueryResults = QueryResult(
        offset: spoonacularResults.offset,
        number: spoonacularResults.number,
        totalResults: spoonacularResults.totalResults,
        recipes: recipes);
    // 7
    return Success(apiQueryResults);
  }
```

In this code, you:

1. Override an unimplemented method for querying a specific recipe. That will be implemented later.
2. Create a new method, `queryRecipes()`, with the parameters `query`, `offset` and `number`. These help you get specific pages from the complete query. `offset` starts at 0, and `number` is calculated by adding the `offset` index to your page size. You use a return type of `Future<RecipeResponse>` for this method because the response will be a `RecipeResponse` in the future when it finishes. `async` signals that this method runs asynchronously.
3. `final` creates a non-changing variable. You use `await` to tell the app to wait until `getData()` returns its result. Look closely at `getData()` and note that you're creating the API URL with the variables passed in.
4. Convert the **JSON string** to a `SpoonacularResults` class with the help of `fromJson` method.
5. Convert the `SpoonacularResults` class object into a list of recipes.
6. Create a `QueryResult` object with those results.
7. Return a Success with the query results.

> **Note**: This method doesn't handle errors.

Now that you've written the service, it's time to update the UI code to use it.

Updating the User Interface

Open **main.dart** and add the following:

```
import 'network/spoonacular_service.dart';
```

Then, after the `sharedPrefProvider` override, replace:

```
final service = await MockService.create();
```

with:

```
final service = SpoonacularService();
```

This creates a new instance of `SpoonacularService`. This will be the start of using real data taken from the internet instead of mock data.

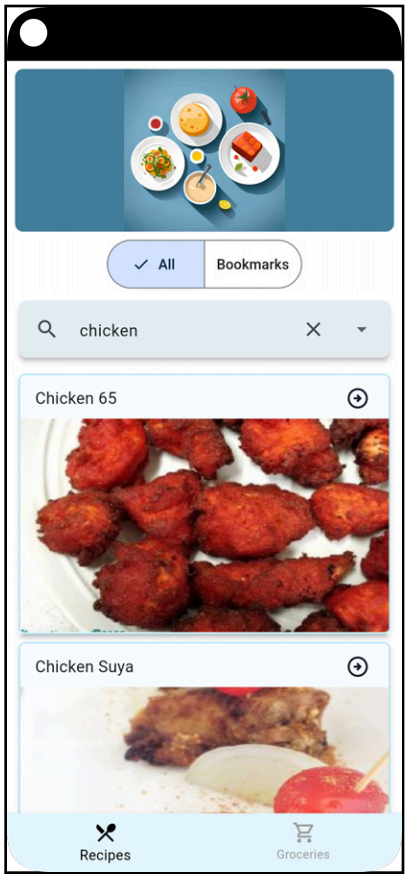
Now remove the import of `mock_service.dart`, as it's not needed anymore.

Retrieving Recipe Data

Great, it's time to try out the app!

Run the app, type **Chicken** in the text field, and tap the **Search** icon. While the app gets data from the API, you'll see the circular progress bar.

After the app receives the data, you'll see a list of images with different types of chicken recipes.



Well done! You've updated your app to receive real data from the internet. Try different search queries and go and show your friends what you've created. :]

> **Note**: If you make too many queries, you could get an error from the Spoonacular site. That's because the free account limits your number of calls.

The http package is easy to use to handle network calls, but it's also pretty basic. Let's explore Chopper, a library that simplifies the creation of code that manages HTTP calls.

Why Chopper?

Chopper is a library that streamlines the process of writing code that performs HTTP requests. For example:

- It generates code to simplify the development of networking code.
- It allows you to organize that code modularly, making it easier to change.

> **Note**: If you come from the Android side of mobile development, you're probably familiar with the **Retrofit** library, which is similar. If you have an iOS background, **AlamoFire** is a very similar library.

Preparing to use Chopper

To use Chopper, you need to add the package to **pubspec.yaml**. To log network calls, you also need the **logging** package, which is already included in the project.

Open **pubspec.yaml** and add the following after the HTTP package:

```
chopper: ^6.1.4
```

You also need **chopper_generator**, which is a package that generates the boilerplate code for you in the form of a part file. In the **dev_dependencies** section, after **json_serializable**, add the following:

```
chopper_generator: ^6.0.3
```

Next, either click **Pub get** or run `flutter pub get` in Terminal to get the new packages.

Now that the new packages are ready to be used… fasten your seat belt! :]

Handling Recipe Results

In this scenario, creating a generic response class that holds either a successful response or an error is good practice. While these classes aren't required, they make it easier to deal with the responses that the server returns.

Take a look inside the **lib/network** folder and open **model_response.dart**.

```
// 1
sealed class Result<T> {
}

// 2
class Success<T> extends Result<T> {
  final T value;

  Success(this.value);
}

// 3
class Error<T> extends Result<T> {
  final Exception exception;

  Error(this.exception);
}
```

Here's what that does:

1. Defines a `sealed class`. It's a simple blueprint for a result with a generic type `T`.
2. The `Success` class extends `Result` and holds a value when the response is successful. This could hold JSON data or a de-serialized class.
3. The `Error` class extends `Result` and holds an exception. This will model errors that occur during an HTTP call, like using the wrong credentials or trying to fetch data without authorization.

> **Note**: The sealed modifier prevents a class from being extended or implemented outside its own library. Sealed classes are implicitly abstract. To refresh your knowledge of classes in Dart, check out our **Dart Apprentice: Fundamentals** book https://www.kodeco.com/books/dart-apprentice-fundamentals/.

You'll use these classes to model the data fetched via HTTP using Chopper. Now that Chopper has been added, you need to update the definition of the result types defined in **lib/network/service_interface.dart**. In that file, add the Chooper import:

```
import 'package:chopper/chopper.dart';
```

Then replace:

```
typedef RecipeResponse = Result<QueryResult>;
typedef RecipeDetailsResponse = Result<Recipe>;
```

with:

```
typedef RecipeResponse = Response<Result<QueryResult>>;
typedef RecipeDetailsResponse = Response<Result<Recipe>>;
```

Instead of returning a `Result` directly, you'll return a `Response` that contains a `Result`. This is because Chopper will handle the conversion of the response to a `Result` for you.

This will mess up the existing `MockService` class as now you have passed `Response` instead of `QueryResult`. Open up **mock_service.dart** and add the following imports:

```
import 'package:http/http.dart' as http;
import 'package:chopper/chopper.dart';
```

In the `queryRecipes` methods, wrap each `Success` call with a `Response`. Like this:

```
return Future.value(
          Response(
            http.Response(
              'Dummy',
              200,
              request: null,
            ),
            Success<QueryResult>(_currentRecipes1),
          ),
        );
```

Do this three times. Make sure you keep the correct values. Also, modify `queryRecipe()` like this:

```
    return Future.value(
        Response(
          http.Response(
            'Dummy',
            200,
            request: null,
          ),
          Success<Recipe>(recipeDetails),
        ),
      );
```

It's time to integrate the code that Chopper will generate into the existing service.

Preparing the Recipe Service

Open **spoonacular_service.dart**.

Replace the existing imports with the following:

```
import 'package:chopper/chopper.dart';

import 'model_response.dart';
import 'query_result.dart';
import 'service_interface.dart';
import '../data/models/models.dart';

part 'spoonacular_service.chopper.dart';
```

The **.chopper** file doesn't exist yet, but you'll generate it soon. Change the definition of the class to look like this:

```
// 1
@ChopperApi()
// 2
abstract class SpoonacularService extends ChopperService
    implements ServiceInterface {
```

1. `@ChopperApi()` tells the Chopper generator to build a file. This generated file will have the same name as this file but with **.chopper** added to it. In this case, it will be **spoonacular_service.chopper.dart**. Such a file will hold the boilerplate code.

2. Define an abstract class. Chopper will create the real class that extends the `ChopperService` and implements the `ServiceInterface`.

Now remove the `getData()` method. It's now time to set up Chopper!

Setting Up the Chopper Client

Your next step is to update the queries needed to implement the service. Replace the definitions of `queryRecipes()` and `queryRecipe()` with:

```
/// Get the details of a specific recipe
@override
@Get(path: 'recipes/{id}/information?includeNutrition=false')
Future<RecipeDetailsResponse> queryRecipe(
  @Path('id') String id,
);

/// Get a list of recipes that match the query string
```

```
  @override
  @Get(path: 'recipes/complexSearch')
  Future<RecipeResponse> queryRecipes(
    @Query('query') String query,
    @Query('offset') int offset,
    @Query('number') int number,
  );

  // TODO: Add create Service
```

The first method returns the details of a specific recipe. The second method returns a list of recipes:

- `@Get` is an annotation that tells the generator this is a GET request.
- `path` is the path to the API call. Chopper will append this path to the base URL, which you've defined as the `apiUrl` constant in `SpoonacularService` class.
- In the first method, you're using a path parameter to get the details of the specific recipe by passing a recipe ID as a dynamic parameter. In the second method, you're using a path to get a list of recipes.
- There are other HTTP methods you can use, such as `@Post`, `@Put` and `@Delete`, but you won't use them in this chapter.
- `@Query` is a query parameter used to define the query name in the URL that's created for this API call. In the second method, you're using `@Query` to get the `query`, `offset` and `number` of recipes.
- These methods return a `Future` response.

Note that you have defined a generic interface to make network calls so far. No actual code performs tasks like adding the API key to the `request` or transforming the `response` into data objects. This is a job for converters and interceptors.

Converting Request and Response

To use the returned API data, you need a converter to transform `requests` and `responses`. To attach a converter to a Chopper client, you need an **interceptor**. You can think of an interceptor as a function that runs every time you send a `request` or receive a `response`. It's a sort of hook to which you can attach functionalities, like converting or decorating data, before passing such data along.

Right-click **lib/network**, create a new file named **spoonacular_converter.dart** and add the following imports:

```
import 'dart:convert';
import 'package:chopper/chopper.dart';
import 'model_response.dart';
import 'query_result.dart';
import 'spoonacular_model.dart';
```

This adds the built-in Dart convert package, which transforms data **to** and **from** JSON, plus the Chopper package and your model files.

Next, create `SpoonacularConverter` by adding the following:

```
// 1
class SpoonacularConverter implements Converter {
  // 2
  @override
  Request convertRequest(Request request) {
    // 3
    final req = applyHeader(
      request,
      contentTypeKey,
      jsonHeaders,
      override: false,
    );

    // 4
    return encodeJson(req);
  }

  // TODO encode JSON

  // TODO Decode Json

  // TODO Convert Response to Model
}
```

Here's what you're doing with this code:

1. Create `SpoonacularConverter` class to implement the Chopper `Converter` abstract class.
2. Override `convertRequest()`, which takes in a request and returns a new request.
3. Add a header to the request that says you have a request type of `application/json` using `jsonHeaders`. These constants are part of Chopper.
4. Call `encodeJson()` to convert the request to a JSON-encoded one, as required by the server API.

The remaining code consists of placeholders, which you'll include in the next section.

Encoding and Decoding JSON

To make it easy to expand your app in the future, you'll separate encoding and decoding. This gives you flexibility if you need to use them separately later.

Whenever you make network calls, you want to ensure that you encode the request before you send it and decode the response string into your **model classes**, which you'll use to display data in the UI.

Encoding JSON

To encode the request in JSON format, replace `// TODO encode JSON` with the following:

```
Request encodeJson(Request request) {
  // 1
  final contentType = request.headers[contentTypeKey];
  // 2
  if (contentType != null && contentType.contains(jsonHeaders))
{
    // 3
    return request.copyWith(body: json.encode(request.body));
  }
  return request;
}
```

In this code, you:

1. Get the content type from the request headers.
2. Check if `contentType` is not `null` and `contentType` is of type `application/json`.
3. Return a copy of the `request` with a JSON-encoded body.

Essentially, this method takes a `Request` instance and returns an encoded copy ready to be sent to the server. What about decoding? Well, I'm glad you asked. :]

Decoding JSON

Now, it's time to add the functionality to decode JSON. A server response is usually a `String`, so you'll have to parse the JSON string and transform it into the corresponding model class.

Replace `// TODO Decode Json` with:

```
Response<BodyType> decodeJson<BodyType, InnerType>(Response
response) {
    final contentType = response.headers[contentTypeKey];
    var body = response.body;
    // 1
    if (contentType != null &&
contentType.contains(jsonHeaders)) {
      body = utf8.decode(response.bodyBytes);
    }
    try {
      // 2
      final mapData = json.decode(body) as Map<String, dynamic>;

      // 3
      // This is the list of recipes
      if (mapData.keys.contains('totalResults')) {
        // 4
        final spoonacularResults =
SpoonacularResults.fromJson(mapData);
        // 5
        final recipes =
spoonacularResultsToRecipe(spoonacularResults);
        // 6
        final apiQueryResults = QueryResult(
            offset: spoonacularResults.offset,
            number: spoonacularResults.number,
            totalResults: spoonacularResults.totalResults,
            recipes: recipes);
        // 7
        return response.copyWith<BodyType>(
          body: Success(apiQueryResults) as BodyType,
        );
      } else {
        // This is the recipe details
        // 8
        final spoonacularRecipe =
SpoonacularRecipe.fromJson(mapData);
        // 9
        final recipe =
spoonacularRecipeToRecipe(spoonacularRecipe);
        // 10
        return response.copyWith<BodyType>(
          body: Success(recipe) as BodyType,
```

```
          );
        }
      } catch (e) {
        // 11
        chopperLogger.warning(e);
        final error = Error<InnerType>(Exception(e.toString()));
        return Response(response.base, null,
            error: error);
      }
  }
```

There's a lot to think about here. To break it down, you:

1. Check if the `contentType` is not null and check if `contentType` contains the `jsonHeaders`. Later you decode the `response` and save to body.
2. Use JSON decoding to convert that string into a map representation.
3. Check if the call has the "totalResults" text. This means it's from the `queryRecipes` call.
4. Convert the JSON to a `SpoonacularResults` instance using `fromJson()`.
5. Convert `SpoonacularResults` to a list of recipes.
6. Create a `QueryResult` with the recipes.
7. Return a copy of `Response` with `Success` result.
8. Convert the map to a detailed `SpoonacularRecipe`.
9. Convert the `spoonacularRecipe` to the recipe.
10. Return a copy of `Response` with `Success` that wraps the result.
11. If you get any kind of error, wrap the `response` with a generic instance of `Error`.

You still have to override one more method: `convertResponse()`. This method changes the given response to the one you want.

Replace the existing `// TODO Convert Response to Model` with the following:

```
  @override
  Response<BodyType> convertResponse<BodyType, InnerType>(Response
  response) {
    // 1
    return decodeJson<BodyType, InnerType>(response);
  }
```

1. This returns the decoded JSON response by calling `decodeJson()`, which you defined earlier.

Now, it's time to use the converter in the appropriate spots and to add some interceptors.

Using Interceptors

As mentioned earlier, interceptors can intercept either the request, the response or both. In a request interceptor, you can add headers or handle authentication. In a response interceptor, you can manipulate a response and transform it into another type, as you'll see shortly. You'll start with decorating the request.

Automatically Including Your API Key

To request any recipes, the API needs your `api_key`. Instead of adding this field manually to each query, you can use an interceptor to add this to each call.

Open **spoonacular_service.dart** and add the following method *outside* of the `SpoonacularService` class definition:

```
Request _addQuery(Request req) {
  // 1
  final params = Map<String, dynamic>.from(req.parameters);
  // 2
  params['apiKey'] = apiKey;
  // 3
  return req.copyWith(parameters: params);
}
```

This is a request interceptor that adds the API key to the query parameters. Here's what the code does:

1. Creates a `Map`, which contains key-value pairs from the existing `Request` parameters.
2. Adds the `apiKey` parameter to the map.
3. Returns a new copy of the `Request` with the parameters contained in the map.

The benefit of this method is that once you hook it up, all your calls will use it. While you only have one call for now, if you add more, they'll include those parameters automatically. Also, if you want to add a new parameter to every call, you'll change only this method.

I hope you're starting to see the advantages of Chopper. :]

You have interceptors to decorate `requests`, and you have a converter to transform `responses` into model classes. Next, you'll put them to use!

Wiring Up Interceptors and Converters

It's time to create an instance of the service that will fetch recipes.

Still in **spoonacular_service.dart**, add the following import, and make sure it's placed *before* the `part` statement:

```
import 'spoonacular_converter.dart';
```

Then locate `// TODO: Add create Service` and replace it with the following code. Don't worry about the red squiggles; they're warning you that the boilerplate code is missing because you haven't generated it yet.

```
static SpoonacularService create() {
  // 1
 final client = ChopperClient(
    // 2
    baseUrl: Uri.parse(apiUrl),
    // 3
    interceptors: [_addQuery, HttpLoggingInterceptor()],
    // 4
    converter: SpoonacularConverter(),
    // 5
    errorConverter: const JsonConverter(),
    // 6
    services: [
      _$SpoonacularService(),
    ],
  );
  // 7
  return _$SpoonacularService(client);
}
```

In this code, you:

1. Create a `ChopperClient` instance.
2. Pass in a base URL using the `apiUrl` constant.
3. Pass in two interceptors. `_addQuery()` adds your API key to the query. `HttpLoggingInterceptor` is part of Chopper and logs all calls. While you're developing, it's handy to see traffic between the app and the server.

4. Set the `converter` as an instance of `SpoonacularConverter`.
5. Use the built-in `JsonConverter` to decode any errors.
6. Define the services created when you run the generator script.
7. Return an instance of the generated service.

It's all set, you're ready to generate the boilerplate code!

Generating the Chopper File

Your next step is to generate **spoonacular_service.chopper.dart**, which works with the `part` keyword. Remember from Chapter 11, "Serialization With JSON", `part` will include the specified file and make it part of one big file.

> **Note**: It might seem weird to import a file before it's been created, but the generator script will fail if it doesn't know what file to create.

Now, open **Terminal** in Android Studio. By default, it'll be in your project folder.

Execute the following:

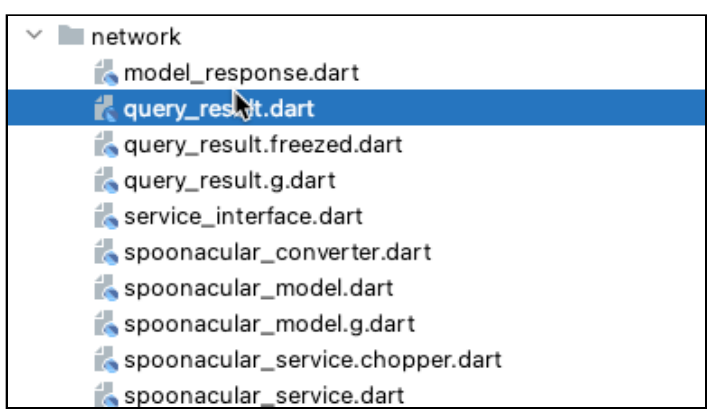
```
dart run build_runner build --delete-conflicting-outputs
```

> **Note**: Using `--delete-conflicting-outputs` will delete all generated files before generating new ones.

While it's executing, you'll see something like this:

```
Terminal:  Local  +
→  starter git:(master) ✗ flutter pub run build_runner build --delete-conflicting-outputs
[INFO] Generating build script...
[INFO] Generating build script completed, took 354ms

[WARNING] Deleted previous snapshot due to missing asset graph.
[INFO] Creating build script snapshot......
[INFO] Creating build script snapshot... completed, took 11.8s

[INFO] Initializing inputs
[INFO] Building new asset graph...
[INFO] Building new asset graph completed, took 622ms

[INFO] Checking for unexpected pre-existing outputs....
[INFO] Deleting 1 declared outputs which already existed on disk.
[INFO] Checking for unexpected pre-existing outputs. completed, took 3ms

[INFO] Running build...
[INFO] Generating SDK summary...
[INFO] 3.8s elapsed, 0/12 actions completed.
[INFO] Generating SDK summary completed, took 3.8s

[INFO] 5.0s elapsed, 0/12 actions completed.
[INFO] 6.0s elapsed, 2/12 actions completed.
[INFO] 7.1s elapsed, 2/12 actions completed.
[INFO] 8.1s elapsed, 2/12 actions completed.
[INFO] 10.5s elapsed, 2/12 actions completed.
[INFO] 11.5s elapsed, 2/12 actions completed.
[INFO] 16.3s elapsed, 4/12 actions completed.
[INFO] 17.3s elapsed, 7/12 actions completed.
[INFO] Running build completed, took 17.8s

[INFO] Caching finalized dependency graph...
[INFO] Caching finalized dependency graph completed, took 41ms

[INFO] Succeeded after 17.9s with 3 outputs (39 actions)

→  starter git:(master) ✗
```

Once it finishes, you'll see the new **spoonacular_service.chopper.dart** in **lib/network**. You may need to refresh the **network** folder before it appears.



> **Note**: In case you don't see the file or Android Studio doesn't detect its presence, right-click on the network folder and select "Reload from disk".

Open it and check it out. The first thing you'll see is a comment stating not to modify the file by hand. Looking farther down, you'll see a class called `_$SpoonacularService`. Below that, you'll notice that `queryRecipes()` has been overridden to build the parameters and the request. It uses the client to send the request.

It may not seem like much, but as you add different calls with different paths and parameters, you'll start to appreciate the help of a code generator like the one included in Chopper.

Now that you've changed `SpoonacularService` to use Chopper, it's time to put on the finishing touches.

Using the Chopper Client

Open **main.dart** and after `sharedPrefs`, replace:

```
    final service = SpoonacularService();
```

with:

```
  final service = SpoonacularService.create();
```

This method will create a new instance of the service with the Chopper client.

Updating the UI

Now open **lib/ui/recipe_details.dart**. In `loadRecipe()`, replace:

```
  final result = response;
  if (result is Success<Recipe>) {
    final body = result.value;
    recipeDetail = body;
    if (mounted) {
      setState(() {});
    }
  } else  {
    logMessage('Problems getting Recipe $result');
  }
```

with:

```
  final result = response.body;
  if (result is Success<Recipe>) {
```

```
    final body = result.value;
    recipeDetail = body;
    if (mounted) {
      setState(() {});
    }
  } else  {
    logMessage('Problems getting Recipe $result');
  }
```

This will retrieve the recipe detail.

In `readRecipe()`, replace:

```
  final result = snapshot.data;
  if (result is Success<Recipe>) {
    final body = result.value;
    recipeDetail = body;
  }
```

with:

```
  final result = snapshot.data?.body;
  if (result is Success<Recipe>) {
    final body = result.value;
    recipeDetail = body;
  }
```

Open **recipe_list.dart** and add:

```
  import 'dart:collection';
```

In `_buildRecipeLoader()`, replace:

```
  final result = snapshot.data;
  // Hit an error
  if (result is Error) {
    const errorMessage = 'Problems getting data';
    return const SliverFillRemaining(
      child: Center(
        child: Text(
          errorMessage,
          textAlign: TextAlign.center,
          style: TextStyle(fontSize: 18.0),
        ),
      ),
    );
  }
```

with:

```
if (false == snapshot.data?.isSuccessful) {
  var errorMessage = 'Problems getting data';
  if (snapshot.data?.error != null &&
      snapshot.data?.error is LinkedHashMap) {
    final map = snapshot.data?.error as LinkedHashMap;
    errorMessage = map['message'];
  }
  return SliverFillRemaining(
    child: Center(
      child: Text(
        errorMessage,
        textAlign: TextAlign.center,
        style: const TextStyle(fontSize: 18.0),
      ),
    ),
  );
}
final result = snapshot.data?.body;
if (result == null || result is Error) {
  inErrorState = true;
  return _buildRecipeList(context, currentSearchList);
}
```

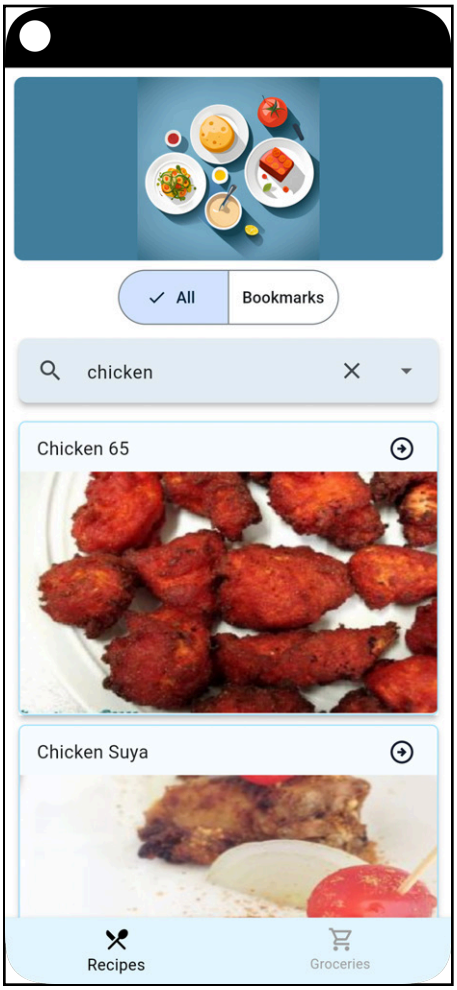
This uses the new response type that wraps the result of an API call.

Now, replace the existing `fetchData()` with:

```
Future<RecipeResponse> fetchData() async {
  if (!newDataRequired && currentResponse != null) {
    return currentResponse!;
  }
  newDataRequired = false;
  final recipeService = ref.watch(serviceProvider);
  currentResponse = recipeService.queryRecipes(
      searchTextController.text.trim(), currentStartPosition,
pageCount);
  return currentResponse!;
}
```

This will use the services `queryRecipes()` instead of the older HTTP call.

Stop the app, run it again and choose the search value **chicken** from the drop-down button. Verify that you see the recipes displayed in the UI.



Now, look in the **Run** window of **Android Studio**, where you'll see lots of `[log] INFO` messages related to your network calls. This is a great way to see how your requests and responses look and figure out what's causing problems.

You made it! You can now use Chopper to make calls to the server API and retrieve recipes.

Key Points

- The **http** package is a simple-to-use set of methods for retrieving data from the internet.
- The built-in `json.decode()` transforms JSON strings into a map of objects that you can use in your code.
- The Chopper package provides easy ways to retrieve data from the internet.
- You can add headers to each network request.
- Interceptors can intercept both requests and responses and change those values.
- Converters can modify requests and responses.

Where to Go From Here?

You've learned how to retrieve data from the internet and parse it into data models. If you want to learn more about the HTTP package and get the latest version, go to https://pub.dev/packages/http.

If you want to learn more about the Chopper package, go to https://pub.dev/packages/chopper. For more info on the Logging library, visit https://pub.dev/packages/logging.

In the next chapter, you'll learn about the important topic of state management.

Chapter 13: Managing State

By Kevin David Moore

The main job of a UI is to represent **state**. Imagine, for example, you're loading a list of recipes from the network. While the recipes are loading, you show a spinning widget. When the data loads, you swap the spinner with the list of loaded recipes. In this case, you move from a **loading** to a **loaded** state. Handling such state changes manually, without following a specific pattern, quickly leads to code that's difficult to understand, update and maintain. One solution is to adopt a pattern that programmatically establishes how to track changes and broadcast details about states to the rest of your app. This is called **state management**.

To learn about state management and see how it works for yourself, you'll continue working with the previous project.

> **Note**: You can also start fresh by opening this chapter's **starter** project. If you choose to do this, remember to click the **Get dependencies** button or execute `flutter pub get` from Terminal. You'll also need to add your API Key to **lib/network/spoonacular_service.dart**.

By the end of the chapter, you'll know:

- Why you need state management.
- How to implement state management using **Riverpod**.

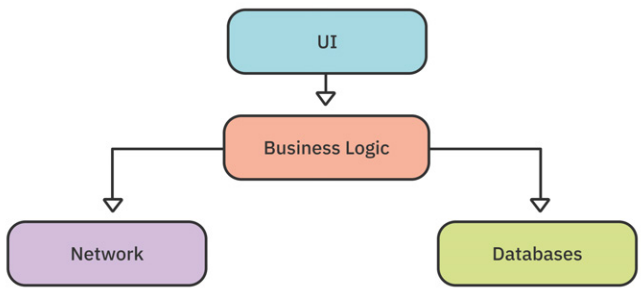
- How to save the current list of bookmarks and ingredients.
- What a repository is.
- Different ways to manage state.

Architecture

When you write apps and the amount of code gets larger and larger over time, you learn to appreciate the importance of separating code into manageable pieces. When files contain more than one **class** or when **classes** combine multiple functionalities, it's harder to fix bugs and add new features.

One way to handle this is to follow **Clean Architecture** principles by organizing your project so it's easy to change and understand. You do this by separating your code into directories and classes, each handling just *one* task. You also use interfaces to define contracts that different classes can implement, allowing you to easily swap in different classes or reuse classes in other apps.

You should design your app with some or all of the components below:



Notice that the **UI** is separate from the **business logic**. It's easy to start an app and put your database and business logic into your UI code — but what happens when you need to change your app's behavior and that behavior is spread throughout your UI code? That makes it difficult to change and causes duplicate code you might forget to update.

Communicating between these layers is important as well. How does one layer talk to the other? The easy way is just to create those classes when you need them. However, this results in multiple instances of the same class, which causes problems coordinating calls.

For example, what if two classes each have their own **database handler class** and make conflicting calls to the database? Both Android and iOS use **Dependency Injection** or **DI** to create instances in one place and inject them into other classes that need them. This chapter will cover the **Riverpod** package for DI and state management.

> **Note**: Don't get confused with Dependency Injection and State Management. They are two different things. Dependency Injection is a way to inject or provide the dependencies needed inside the app, and State Management is a way to manage the app's state.

Ultimately, the business logic layer should decide how to react to the user's actions and delegate tasks like **retrieving** and **saving data** to other classes.

Why You Need State Management

First, what do the terms **state** and **state management** mean? State is when a widget is active and stores its data in memory. The Flutter framework handles some state, but as mentioned earlier, Flutter is declarative. That means it rebuilds the UI from memory when the state or data changes or when another part of your app uses it.

State management is, as the name implies, how you manage the state of your widgets and app.

There are two types of state to consider - **ephemeral state**, also known as **local state**, which is limited to the widget, and **app state**, also known as **global state**.

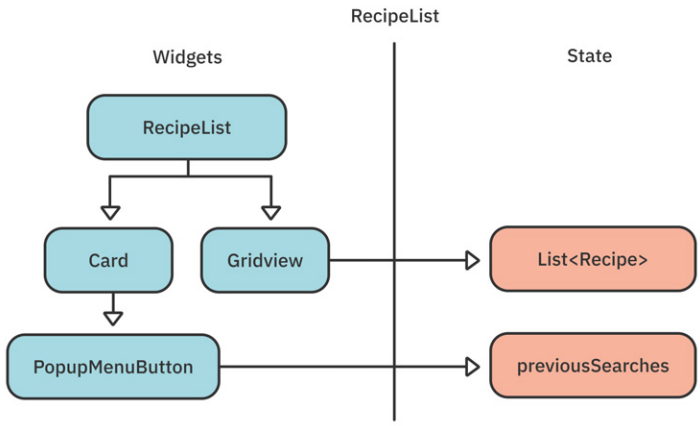
- Use ephemeral state when no other component in the widget tree needs to access a widget's data. Examples include whether a `TabBarView` tab is selected or `FloatingActionButton` is pressed.
- Use app state to manage the entire state of the app and when other parts of your app need to access some state data. One example is an image that changes over time, like an icon for the current weather. Another example is information that the user selects on one screen, which should then display on another screen, like when the user adds an item to a shopping cart.

Next, you'll learn more about the different types of state and how they apply to your recipe app.

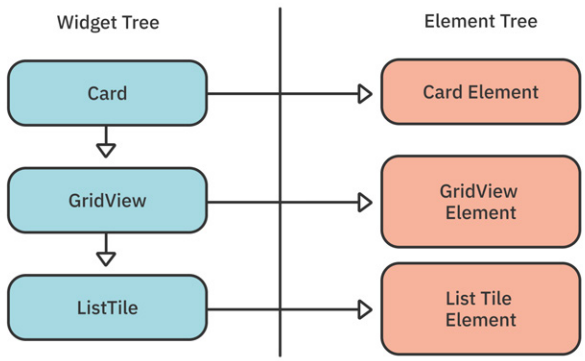
Widget State

In Chapter 4, "Understanding Widgets", you saw the difference between **stateless** and **stateful widgets**. A stateless widget is drawn with the same state it had when it was created. A stateful widget preserves its state and uses it to (re)draw itself when there's any change in the widget's state.

Your current **Recipes** screen has a card with the list of previous searches and a `GridView` with a list of recipes:



The left side shows some of the `RecipeList` widgets, while the right side shows the state objects that store the information each widget uses. An element tree stores both the widgets themselves and the states of all the stateful widgets in `RecipeList`:



If the state of a widget updates, the state object also updates, and the widget is redrawn with that updated state.

Application State

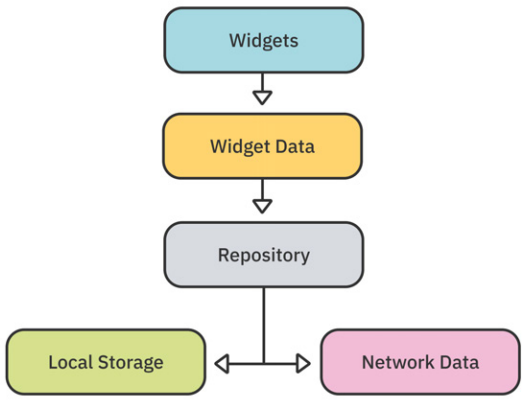
In Flutter, a `StatefulWidget` can hold state. Its children can access it, and even pass (pieces of) it to other screens. However, that complicates your code, and you have to remember to pass data objects down the tree. Wouldn't it be great if child widgets could easily access their parent data without having to pass in that data?

There are several different ways to achieve that, both with **built-in widgets** and with **third-party packages**. You'll look at built-in widgets first.

Managing State in Your App

The Recipes Finder app needs to save four things: the **currently selected screen**, the list to show in the **Recipes** screen, the user's **bookmarks** and the **ingredients**. In this chapter, you'll use state management to save this information so other screens can use it.

These methods are still relevant for sharing data between screens. Here's a general idea of how your classes will look:



Stateful Widgets

`StatefulWidget` is one of the most basic ways of saving state. The `RecipeList` widget, for example, saves several fields for later usage, including the current search list and the start and end positions of search results for pagination.

When you create a `StatefulWidget`, the `createState()` method gets called, which creates and stores the state internally in Flutter. The parent needs to rebuild the widget when there's a change in the state of the widget.

You use `initstate()` to initialize the widget in its starting state. You use it for one-time work, like initializing text controllers. Then, you use `setState()` to set the new changed state, triggering a rebuild of the widget.

For example, in Chapter 10, "Handling Shared Preferences", you used `setState()` to set the selected tab. This tells the system to rebuild the UI to select a page. `StatefulWidget` is great for maintaining an internal state, but not for a state outside the widget.

One way to achieve an architecture that allows sharing state between widgets is to adopt **InheritedWidget**.

InheritedWidget

`InheritedWidget` is a built-in class allowing child widgets to access its data. It's the basis for a lot of other state management widgets. If you create a class that extends `InheritedWidget` and gives it some data, any child widget can access it by calling `context.dependOnInheritedWidgetOfExactType<class>()`.

Wow, that's quite a mouthful! As shown below, `<class>` represents the name of the class extending `InheritedWidget`.

```
class RecipeWidget extends InheritedWidget {
  final Recipe recipe;
  RecipeWidget(Key? key, required this.recipe, required Widget
child}) :
      super(key: key, child: child);

  @override
  bool updateShouldNotify(RecipeWidget oldWidget) => recipe !=
oldWidget.recipe;

  static RecipeWidget of(BuildContext context) =>
context.dependOnInheritedWidgetOfExactType<RecipeWidget>()!;

}
```

You can then extract data from that widget. Since that's such a long method name to call, the convention is to create an `of()` method.

Then a child widget, like the text field that displays the recipe title, can just use:

```
RecipeWidget recipeWidget = RecipeWidget.of(context);
print(recipeWidget.recipe.label);
```

> **Note**: `updateShouldNotify()` compares two recipes, which requires `Recipe` to implement `equals`. Otherwise, you need to compare each field.

An advantage of using `InheritedWidget` is it's a built-in widget so you don't need to worry about using external packages.

A disadvantage of using `InheritedWidget` is that the value of a recipe can't change unless you rebuild the whole widget tree because `InheritedWidget` is immutable. So, if you want to change the displayed recipe title, you'll have to rebuild the whole `RecipeWidget`.

Provider

Remi Rousselet designed **Provider** to build state management functionalities on top of `InheritedWidget`.

Google even includes details about it in their state management docs https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple#providerof.

RiverPod

Provider's author, Remi Rousselet, wrote Riverpod to address some of Provider's weaknesses. In fact, Riverpod is an anagram of Provider! Rousselet wanted to solve the following problems:

1. Easily **access state** from anywhere.
2. Allow the **combination of states**.
3. Enable **override providers** for testing.

You'll use RiverPod to implement state management in your app.

Keypoints of Riverpod

Before you start using Riverpod, you need to understand some of its key points.

- **ProviderScope**: A provider scope is a widget that provides a **scope** for providers. The `AppWidget` must be wrapped in ProviderScope to use Riverpod.
- **Provider**: A provider is a class that provides a value to other classes. It's the most basic class in Riverpod. There are many types of providers. You'll see them later.
- **Consumer**: A consumer is a widget that listens to changes in a provider and rebuilds itself when the value changes. There are two types of consumers: **Consumer** and **ConsumerWidget**. You'll see examples later.
- **Ref**: A ref is a reference to a provider. You use it to access other providers. You can obtain a **ref** from providers and ConsumerWidgets.

Types of Providers

There are several different types of providers:

- **Provider**: Returns any **value**. Useful as DI.
- **StateProvider**: Returns any **type** and provides a way to modify it's **state**.
- **FutureProvider**: Returns a **Future**.
- **StreamProvider**: Returns a **Stream**.
- **StateNotifierProvider**: Returns a subclass of **StateProvider** and provides a way to modify its state through an interface.
- **NotifierProvider**: Listen to and expose a **Notifier**.
- **AsyncNotifierProvider**: Listen to and expose an **Asyncotifier**, AsyncNotifier is a Notifier that can be asynchronously initialized.
- **ChangeNotifierProvider**: Returns a **ChangeNotifier**. This is for migrating from the old ChangeNotifier.

> **Note**: **ChangeNotifierProvider** is a mutable provider, and its use is discouraged. It's only for transitioning from provider to Riverpod. It's advisable to use **NotifierProvider** instead.

Provider

`Provider` is the most basic class that provides a value to other classes. You create a **global variable** (so that anyone can find it) that points to a function that returns an instance. You create a provider like this:

```
final myProvider = Provider((ref) {
  return MyValue();
});
```

The variable `myProvider` is final and doesn't change. It provides a function that will create the state. You can also use the `ref` variable to access other providers. You can also provide multiple providers that return the same type.

StateProvider

`StateProvider` is a simplified version of `StateNotifierProvider`. It allows you to modify simple variables. This includes strings, Booleans, numbers or lists of items. You can also use classes. A simple example looks like this:

```
class Item {
  Item({required this.name, required this.title});

  final String name;
  final String title;
}

final itemProvider = StateProvider<Item>((ref) => Item(name:
'Item1', title: 'Title1'));
```

The variable `itemProvider` is final and doesn't change. You use this variable to access the state of the value provided by the provider and can change the value as follows:

```
ref.read(itemProvider.notifier).state = Item(name: 'Item2',
title: 'Title2');
```

There is also the `update()` method:

```
ref.read(itemProvider.notifier).update((state) => Item(name:
'Item2', title: 'Title2'));
```

FutureProvider

`FutureProvider` works like other providers but for asynchronous code and returns a `Future`. They are generally used in place of `FutureBuilder`.

```
final itemProvider = FutureProvider<Item>((ref) async {
  return someLongRunningFunction();
});
```

A `Future` is handy when a value is not readily available but will be in the future. Examples include calls that request data from the internet or asynchronously read data from a database. You can use `FutureProvider` like this:

```
AsyncValue<Item> futureItem = ref.watch(itemProvider);
  return futureItem.when(
    loading: () => const CircularProgressIndicator(),
    error: (err, stack) => Text('Error: $err'),
    data: (item) {
      return Text(item.name);
    },
  );
```

StreamProvider

You'll learn about **streams** in detail in the next chapter. For now, you just need to know that Riverpod also has a provider specifically for streams and works the same way as `FutureProvider`. `StreamProviders` are handy when data comes in via streams and values change over time, like, for example, when you're monitoring the connectivity of a device.

StateNotifierProvider

`StateNotifierProvider` is used to listen to changes in `StateNotifier`. A simple example looks like this:

```
class ItemNotifier extends StateNotifier<Item> {
  ItemNotifier() : super(Item(name: 'Item1', title: 'Title1'));

  void updateItem(Item item) {
    state = item;
  }
}

final itemProvider = StateNotifierProvider<ItemNotifier,
Item>((ref) => ItemNotifier());
```

Here the constructor of `ItemNotifier` sets the initial state for an `Item`.To change the value of the provider, you use its `updateItem()` method as follows:

```
ref.read(itemProvider.notifier).updateItem(Item(name: 'Item2',
title: 'Title2'));
```

NotifierProvider and AsyncNotifierProvider

`NotifierProvider` is used to listen to and expose a Notifier.
`AsyncNotifierProvider` is a Notifier that you can asynchronously initialize. You generally use it to expose the state, which can change over time after reacting to custom events, like button taps and data changes.

```
class ItemNotifier extends Notifier<Item> {
  @override
  Item build(){
    return Item(name: 'Item1', title: 'Title1');
  }

  void updateItem(Item item) {
    state = item;
  }
}

final itemNotifierProvider = NotifierProvider<ItemNotifier,
Item>(() => ItemNotifier());
```

The `build()` function returns the initial state of the `Item` and is called when the provider is first accessed.

To change the value of the provider, you use again its `updateItem()` method:

```
ref.read(itemNotifierProvider.notifier).updateItem(Item(name:
'Item2', title: 'Title2'));
```

Adopting Riverpod in the Recipe Finder App

You're now ready to start working on your recipe project. If you're following along with your app from the previous chapters, open it and keep using it with this chapter. If not, just locate this chapter's **projects** folder and open **starter** in Android Studio.

> **Note**: If you use the starter app, don't forget to add your `apiKey` in **network/spoonacular_service.dart**.

Overview of Existing Providers

Open up **providers.dart**. It should look like this:

```
// 1
final sharedPrefProvider = Provider<SharedPreferences>((ref) {
  throw UnimplementedError();
});

// 2
final repositoryProvider =
ChangeNotifierProvider<MemoryRepository>((ref) {
  return MemoryRepository();
});

// 3
final serviceProvider = Provider<ServiceInterface>((ref) {
  throw UnimplementedError();
});
```

This code:

1. Defines a provider for **Shared preferences**. Note that it throws an `UnimplementedError`. Explanation below.
2. Defines a `ChangeNotifierProvider` for the`MemoryRepository`.
3. Defines a provider for `ServiceInterface`. This will allow you to substitute any `ServiceInterface` class.

Now open **main.dart** and look at the following:

```
// 1
final sharedPrefs = await SharedPreferences.getInstance();
// 2
final service = SpoonacularService.create();
// 3
runApp(ProviderScope(overrides: [
  sharedPrefProvider.overrideWithValue(sharedPrefs),
  serviceProvider.overrideWithValue(service),
], child: const MyApp()));
```

1. Get an **instance** of the `SharedPreferences` library.
2. Create a `SpoonacularService`.
3. **Override** the definitions above with these newly created instances.

Since getting a shared preference instance is an `asynchronous` call, we do this in the main method that uses the `async` keyword.

Updating Repositories

Inside the **data/repositories** directory are two repository files: **repository.dart** contains the abstract definition of a repository, and **memory_repository.dart** defines a memory-based repository. This repository will hold your **recipes** and **ingredients** while running. Once the app closes, the data goes away. In Chapter 15, "Saving Data Locally", you'll learn how to store such data locally.

Updating the Memory Repository

Open up **data/repositories/memory_repository.dart**. Notice that it currently uses `ChangeNotifier, which isn't recommended when using **Riverpod**. You'll convert this class to the Riverpod **Notifier** class.

To be a **Notifier** - a class has to have an object that notifies others about the change. This class will be **CurrentRecipeData**. This will contain the current **recipes** and **ingredients** list. Create a new file in **data/models** called **current_recipe_data.dart**.

Add the following:

```
import 'package:freezed_annotation/freezed_annotation.dart';
import 'models.dart';
part 'current_recipe_data.freezed.dart';

@freezed
class CurrentRecipeData with _$CurrentRecipeData {
  const factory CurrentRecipeData({
    @Default(<Recipe>[]) List<Recipe> currentRecipes,
    @Default(<Ingredient>[]) List<Ingredient>
currentIngredients,
  }) = _CurrentRecipeData;
}
```

This uses the Freezed package to create a few helper methods like `copyWith()`. The `@Default` annotation helps assign the default value to the variables. From a terminal run:

```
dart run build_runner build
```

This will create the **current_recipe_data.freezed.dart** file.

Back in **memory_repository.dart**, replace the import of **foundation.dart** with:

```
import 'package:flutter_riverpod/flutter_riverpod.dart';
import '../models/current_recipe_data.dart';
```

Then change the class definition to:

```
class MemoryRepository extends Notifier<CurrentRecipeData>
    implements Repository {
```

On the next line, add the following method. This will initialize the notifier and set the initial state of `CurrentRecipeData`.

```
@override
CurrentRecipeData build() {
  const currentRecipeData = CurrentRecipeData();
  return currentRecipeData;
}
```

Now, remove all of the calls to `notifyListeners()` methods, as they'll be throwing compilation errors. Next, remove the declaration of `_currentRecipes` and `_currentIngredients` fields. Since those two fields are in `CurrentRecipeData` and we have a state of `currentRecipeData`, you'll just use that.

Substitute all the occurrences of `_currentRecipes` with `state.currentRecipes`.

For example, `findAllRecipes()` should turn into this:

```
@override
List<Recipe> findAllRecipes() {
  return state.currentRecipes;
}
```

Similarly, change all occurrences of `_currentIngredients` with `state.currentIngredients`.

> **Note**: **State** is a getter that returns the current state of the **notifier** and you can access the current state. You can also update the state of the notifier by assigning **new state**. You don't need to call `notifyListeners()` as it's done automatically.

If you need help, look at the file in the final project. Hint - find and replace works great!

Since `CurrentRecipeData` is immutable, meaning you can't modify it, you'll have to create new instances instead of modifying the lists. Where are the lists modified, you may wonder? In the methods that insert and delete recipes and ingredients. Find `// TODO: Update insertRecipe()` and replace the line below it with:

```
if(state.currentRecipes.contains(recipe)) {
  return 0;
}
state = state.copyWith(currentRecipes: [...state.currentRecipes,
recipe]);
```

First, you check if the recipe is already on the list. If it is, you return 0. If not, you assign the current state with a new instance of state of `CurrentRecipeData` by copying the existing one (`copyWith()` comes from Freezed) with the current list of recipes and a new one. Notice that using `[]` makes a new list.

Now replace the line below `// TODO: Update insertIngredients()` with:

```
state = state.copyWith(currentIngredients:
[...state.currentIngredients,
  ...ingredients]);
```

This does something similar but adds two lists together. Next, replace `// TODO: Update deleteRecipe()` and the subsequent line with the following:

```
final updatedList = [...state.currentRecipes];
updatedList.remove(recipe);
state = state.copyWith(currentRecipes: updatedList);
```

This creates a new list using the **spread operator**: `...`, which unfolds the list of items.Now replace the whole body of `deleteIngredient()` with:

```
final updatedList = [...state.currentIngredients];
updatedList.remove(ingredient);
state = state.copyWith(currentIngredients: updatedList);
```

Then substitute the whole body of `deleteIngredients()` with:

```
final updatedList = [...state.currentIngredients];
updatedList.removeWhere((ingredient) =>
ingredients.contains(ingredient));
state = state.copyWith(currentIngredients: updatedList);
```

And finally, change the body of `deleteRecipeIngredients()` as follows:

```
final updatedList = [...state.currentIngredients];
updatedList.removeWhere((ingredient) => ingredient.recipeId ==
recipeId);
state = state.copyWith(currentIngredients: updatedList);
```

Now that `MemoryRepository` has changed, open **lib/providers.dart** to adopt NotifierProvider.Add the following import:

```
import 'data/models/current_recipe_data.dart';
```

and then change `repositoryProvider` to:

```
final repositoryProvider =
    NotifierProvider<MemoryRepository, CurrentRecipeData>(() {
  return MemoryRepository();
});
```

Rerun your app to make sure it compiles successfully.

It's now time to use the new **repositoryProvider** in the UI.

Using the Repository for Recipes

You'll implement code to add a recipe to the **Bookmarks** screen and ingredients to the **Groceries** screen. First, open **ui/recipes/recipe_details.dart**.

Displaying the Recipes' Details

You need to show the recipe's image, label and calories on the **Details** page. The repository already stores all of your currently bookmarked recipes.

> **Note**: If your **recipe_details.dart** file does not have the // TODO comments, take a look at the starter project.

Find `// TODO: Add Repository` and replace it with:

```
final repository = ref.read(repositoryProvider.notifier);
```

This reads the `repositoryProvider` as a **class instance** so that you can use it to access the **functions** you defined in the repository. You'll use it to add the bookmark.

Next, replace `// TODO: Insert Recipe` with:

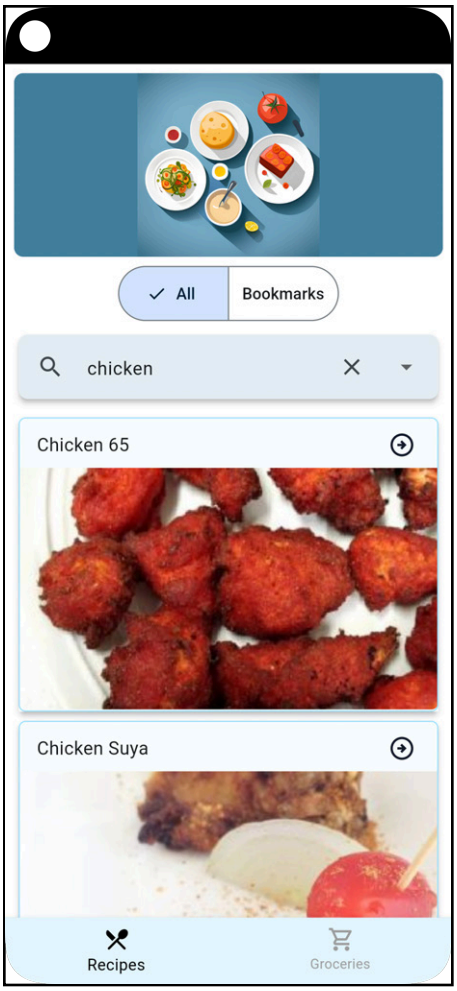
```
repository.insertRecipe(recipeDetail!);
```

This adds the recipe to your repository's list of recipes. To delete the recipe, replace: `// TODO: Delete Recipe` with:

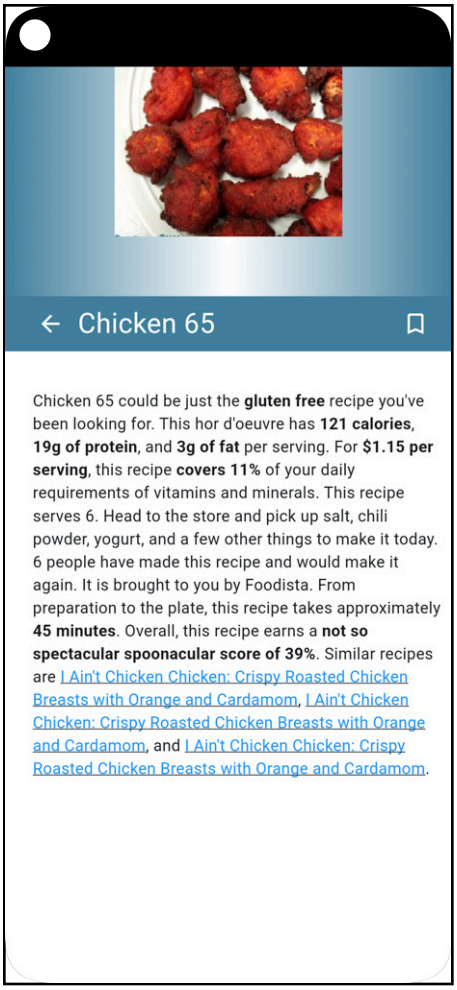
```
repository.deleteRecipe(recipeDetail!);
```

This just removes it from the memory repository's list of recipes.

Now, hot reload the app. Enter **chicken** in the search box and tap the magnifying glass to perform the search. You'll see something like this:



Select a recipe to go to the details page:



Tap the **Bookmark** button and the details page will disappear.

Now, select the **Bookmarks** tab. At this point, you'll see a blank screen — you haven't implemented it yet.

Showing bookmarked recipes in the **Bookmarks** tab is your next step.

Implementing the Bookmarks Screen

Open **ui/bookmarks/bookmarks.dart** and add the following imports:

```
import '../../providers.dart';
import '../recipes/recipe_details.dart';
```

This includes the Riverpod providers to retrieve the repository as well as the `RecipeDetails` class.

Find `// TODO: Add Repository` and add:

```
final repository = ref.watch(repositoryProvider);
recipes = repository.currentRecipes;
```

This watches the repository for changes and updates the widget. It also gets the current list of recipes from the repository.

On the **Bookmarks** page, the user can delete a bookmarked recipe by swiping left or right and selecting the delete icon. To implement this, find and replace `// TODO: Add Delete Recipe` at the bottom of the class with:

```
  void deleteRecipe(Recipe recipe) {
    ref.read(repositoryProvider.notifier).deleteRecipe(recipe);
  }
```

In this code, you use: `ref.read` to get the repository and then call `deleteRecipe()` on it.

Go back up in the file and replace the two instances of `// TODO Add Delete` with:

```
deleteRecipe(recipe);
```

This will call the method you just created and pass the recipe to delete. Replace `// TODO: Add Push to Recipe Details Page` with:

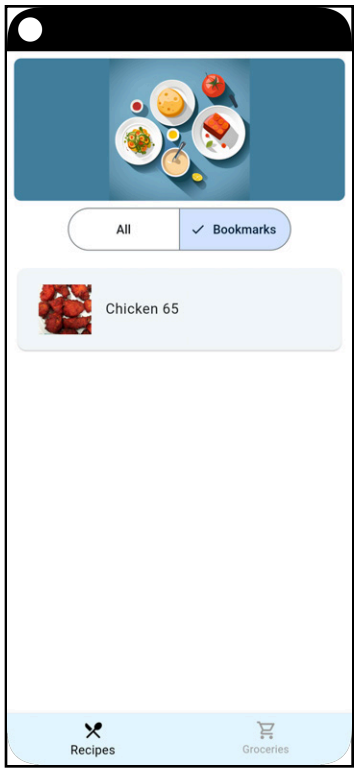
```
Navigator.push(context, MaterialPageRoute(
  builder: (context) {
    return RecipeDetails(
        recipe: recipe.copyWith(bookmarked: true));
  },
));
```

This will take the user to the **recipe details** page with a copy of the recipe and `bookmarked` set to true.

If you left your app running while making all of the above changes, hot reload the app.

If you stopped your app or did a hot restart instead of a hot reload, then return to the **Recipes** tab and bookmark a recipe.

Select the **Bookmarks** tab, and you should see the recipe you bookmarked. Something like this:



You're almost done, but if you go to the **Groceries** tab, you'll see that the view is currently blank. Your next step is to add the functionality to show the ingredients of bookmarked recipes.

Implementing the Groceries Screen

Open **ui/groceries/groceries.dart** and add the following:

```
import '../../providers.dart';
```

Here, you import your providers.

Find `// TODO: Add Repository 1` and replace with:

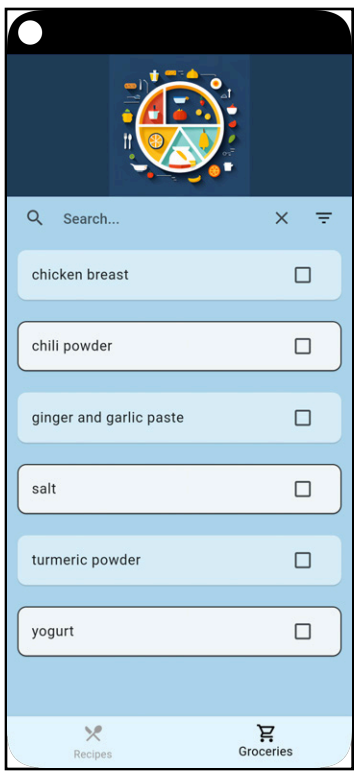
```
    final repository = ref.watch(repositoryProvider);
    currentIngredients = repository.currentIngredients;
```

Find `// TODO: Add Repository 2` and replace with:

```
    final repository = ref.watch(repositoryProvider);
    currentIngredients = repository.currentIngredients;
```

Hot reload and make sure you still have one bookmark saved.

Now, go to the **Groceries** tab to see the ingredients of the recipe you bookmarked. You'll see something like this:



Congratulations, you made it! You now have an app where you can monitor state changes and get notifications across different screens, thanks to the infrastructure of Riverpod.

Implementing the Main Screen State

The main screen also has a state, and that is the currently selected bottom navigation item. This state will use the `StateProvider` class from Riverpod.

In the **ui** directory, create a new file named **main_screen_state.dart**. Add the following:

```
import 'package:flutter_riverpod/flutter_riverpod.dart';
import 'package:freezed_annotation/freezed_annotation.dart';

part 'main_screen_state.freezed.dart';

// 1
@freezed
class MainScreenState with _$MainScreenState {
  const factory MainScreenState({
    @Default(0) int selectedIndex,
  }) = _MainScreenState;
}

// 2
class MainScreenStateProvider extends
StateNotifier<MainScreenState> {
  MainScreenStateProvider() : super(const MainScreenState());

  // 3
  void updateSelectedIndex(int index) {
    state = MainScreenState(selectedIndex: index);
  }
}
```

1. `MainScreenState` just holds the currently selected index.
2. `MainScreenStateProvider` is a provider for that state.
3. One method (`updateSelectedIndex`) is provided to update the index. It creates a new state.

This uses the Freezed package to create a few helper methods. From a terminal run:

```
dart run build_runner build
```

Now add this provider to **providers.dart**:

```
import 'ui/main_screen_state.dart';

final bottomNavigationProvider =
    StateNotifierProvider<MainScreenStateProvider,
MainScreenState>((ref) {
  return MainScreenStateProvider();
});
```

Open up **main_screen.dart** and remove `int _selectedIndex = 0;`. Inside of `saveCurrentIndex()` replace the last line with:

```
final bottomNavigation = ref.read(bottomNavigationProvider);
prefs.setInt(prefSelectedIndexKey,
bottomNavigation.selectedIndex);
```

Find `// TODO: Update getCurrentIndex()` and replace the line below it with:

```
ref
    .read(bottomNavigationProvider.notifier)
    .updateSelectedIndex(index);
```

Change the line below `// TODO: Update _onItemTapped()` with:

```
ref.read(bottomNavigationProvider.notifier).updateSelectedIndex(
index);
```

Then find `// TODO: Update largeLayout() 1` and replace the line below it with:

```
selectedIndex:
    ref.watch(bottomNavigationProvider).selectedIndex,
```

Finally find `// TODO: Update largeLayout() 2` and replace the subsequent line with:

```
index: ref.watch(bottomNavigationProvider).selectedIndex,
```

The next step is to update `getRailNavigations()`. First, replace the line below `// TODO: Update getRailNavigations() 1` with:

```
ref.watch(bottomNavigationProvider).selectedIndex == 0
    ? selectedColor
    : Colors.black,
```

And then change the line after `// TODO: Update getRailNavigations() 2` with:

```
ref.watch(bottomNavigationProvider).selectedIndex == 0
    ? selectedColor
    : Colors.black,
```

Now find `// TODO: Update mobileLayout()` and change the line below it to:

```
index: ref.watch(bottomNavigationProvider).selectedIndex,
```

In `createBottomNavigationBar()`, find `// TODO: Add index` and replace it with:

```
final bottomNavigationIndex =
    ref.read(bottomNavigationProvider).selectedIndex;
```

Finally, change all of the remaining instances of `_selectedIndex` to:

```
bottomNavigationIndex
```

Now it's time to get rid of the calls to `setState()`. Update `_onItemTapped()` so it looks like this:

```
void _onItemTapped(int index) {

ref.read(bottomNavigationProvider.notifier).updateSelectedIndex(
index);
  saveCurrentIndex();
}
```

and change `getCurrentIndex()` as follows:

```
void getCurrentIndex() async {
  final prefs = ref.read(sharedPrefProvider);
  if (prefs.containsKey(prefSelectedIndexKey)) {
    final index = prefs.getInt(prefSelectedIndexKey);
    if (index != null) {

ref.read(bottomNavigationProvider.notifier).updateSelectedIndex(
index);
    }
  }
}
```

Finally, make sure that `getCurrentIndex()` is called after the build method. To achieve that, change the last line of `initState()` like this.

```
Future.microtask(() async {
  getCurrentIndex();
});
```

Stop and restart the app and verify that you can add and delete bookmarks. Check also that the **Groceries** tab shows the ingredients of the bookmarked recipes.

Congrats! Now you know how to manage state across different screens of your app using Riverpod. And that's just the beginning!

Is Riverpod the only option for state management? No. Here's a quick tour of alternative libraries.

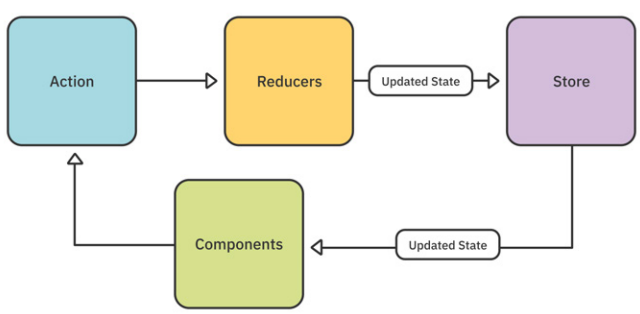
Other State Management Libraries

There are other packages that help with state management and provide even more flexibility when managing state in your app. While Riverpod features classes for widgets lower in the widget tree, other packages provide more generic state management solutions for the whole app, often enabling a unidirectional data flow architecture.

Such libraries include **Redux**, **BLoC** and **MobX**. Here's a quick overview of each.

Redux

If you come from web or React development, you might be familiar with Redux, which uses concepts such as actions, reducers, views and stores. The flow looks like this:



Actions, like clicks on the UI or events from network operations, are sent to reducers, which turn them into a state. That state is saved in a store, which notifies listeners, like views and components, about changes.

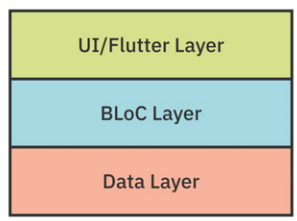
The nice thing about the Redux architecture is that a view can simply send actions and wait for updates from the store.

You need two packages to use Redux in Flutter: **redux** and **flutter_redux**.

For React developers migrating to Flutter, an advantage of Redux is that it's already familiar. It might take a bit to learn if you aren't familiar with it.

BLoC

BLoC stands for **B**usiness **Lo**gic **C**omponent. It's designed to separate UI code from the data layer and business logic, helping you create reusable code that's easy to test. Think of it as a stream of events; some widgets submit events, and others respond to them. BLoC sits in the middle and directs the conversation, leveraging the power of streams.



It's quite popular in the Flutter Community and very well documented.

MobX

MobX comes to Dart from the web world. It uses the following concepts:

- **Observables**: Hold the state.
- **Actions**: Mutate the state.
- **Reactions**: React to the change in observables.

MobX has annotations that help you write your code and simplify it.

One advantage is that MobX allows you to wrap any data in an observable. It's relatively easy to learn and requires smaller generated code files than BLoC.

Key Points

- State management is key to Flutter development.
- Riverpod is a great package that helps with state management.
- Other packages for handling application state include Redux, Bloc, and MobX.
- Repositories are a pattern for providing data.
- You can switch between repositories by providing an interface for the repository. For example, you can switch between real and mocked repositories.
- Mock services are a way to provide dummy data.

Where to Go From Here?

If you want to learn more about:

- State management, go to https://flutter.dev/docs/development/data-and-backend/state-mgmt/intro.
- Flutter Redux go to https://pub.dev/packages/flutter_redux.
- Bloc, go to https://bloclibrary.dev/#/.
- MobX, go to https://github.com/mobxjs/mobx.dart.
- Riverpod, go to https://riverpod.dev/.
- Clean Architecture, go to https://pusher.com/tutorials/clean-architecture-introduction.

In the next chapter, you'll learn all about **streams** that handle data that can be sent and received continuously. See you there!

Chapter 14: Working With Streams

By Kevin David Moore

Imagine yourself sitting by a creek, having a wonderful time. While watching the water flow, you see a piece of wood or a leaf floating down the stream and decide to take it out of the water. You could even have someone upstream purposely float things down the creek for you to grab.

You can imagine Dart **streams** in a similar way: as data flowing down a creek, waiting for someone to grab it. That's what a stream does in Dart — it sends data events for a listener to grab.

With Dart streams, you can send one data event at a time while other parts of your app listen for those events. Such events can be collections, maps or any other type of data you've created.

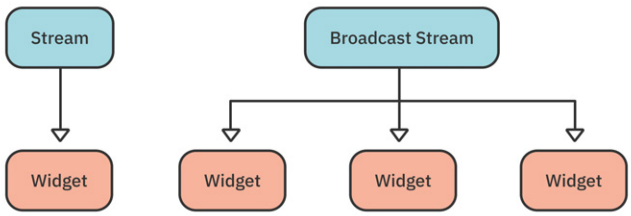
Streams can send errors in addition to data; you can also stop the stream if you need to.

In this chapter, you'll update Recipe Finder to use streams in two different locations. You'll use one for **bookmarks** to let the user mark favorite recipes and automatically update the UI to display them. You'll also use one to update your **ingredient** and **grocery** lists.

But before you jump into the code, you'll learn more about how streams work.

Types of Streams

Streams are part of Dart, and Flutter inherits them. There are two types of streams in Flutter: **single subscription streams** and **broadcast streams**.



Single subscription streams are the default. They work well when you're only using a particular stream on one screen.

A single subscription stream can only be listened to once. It doesn't start generating events until it has a **listener**, and it stops sending events when the listener stops listening, even if the source of events could still provide more data.

Single subscription streams are useful for downloading a file or for any single-use operation. For example, a widget can subscribe to a stream to receive updates about a value, like the progress of a download, and update its UI accordingly.

If you need multiple parts of your app to access the same stream, use a **broadcast stream**, instead.

A broadcast stream allows any number of listeners. It fires when its events are ready, whether there are listeners or not.

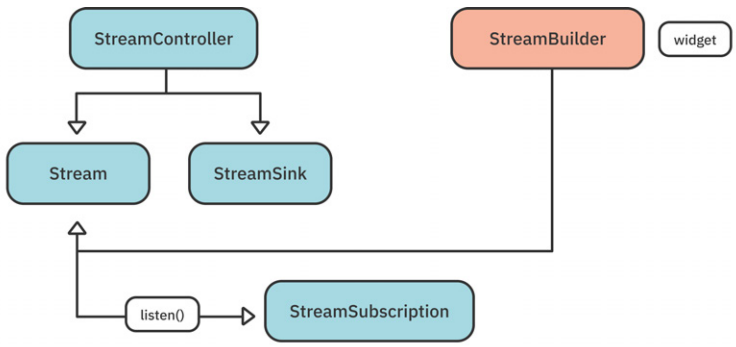
To create a broadcast stream, you simply call `asBroadcastStream()` on an existing single subscription stream.

```
final broadcastStream = singleStream.asBroadcastStream();
```

You can differentiate a broadcast stream from a single subscription stream by inspecting its Boolean property `isBroadcast`.

In Flutter, some key classes are built on top of `Stream` that simplify programming with streams.

The following diagram shows the main classes used with streams:



Next, you'll take a deeper look at each one.

StreamController and Sink

When you create a stream, you usually use `StreamController`, which holds both the stream and `StreamSink`.

A sink is a destination for data. When you want to add data to a stream, you'll add it to the **sink**. Since the `StreamController` owns the sink, it listens for data on the sink and sends the data to its stream listeners.

Here's an example that uses `StreamController`:

```
final _recipeStreamController =
StreamController<List<Recipe>>();
final _stream = _recipeStreamController.stream;
```

To add data to a stream, you add it to its sink:

```
_recipeStreamController.sink.add(_recipesList);
```

This uses the sink field of the controller to "place" a list of recipes on the stream. That data will be sent to any current listeners.

When you're done with the stream, make sure you close it, like this:

```
_recipeStreamController.close();
```

StreamSubscription

Using `listen()` on a stream returns a `StreamSubscription`. You can use this subscription class to cancel the stream when you're done, like this:

```
StreamSubscription subscription = stream.listen((value) {
    print('Value from controller: $value');
});
...
...
// You are done with the subscription
subscription.cancel();
```

Sometimes, it's helpful to have an automated mechanism to avoid managing subscriptions manually. That's where `StreamBuilder` comes in.

StreamBuilder

`StreamBuilder` is handy when you want to use a stream. It takes two parameters: a stream and a builder. As you receive data from the stream, the builder takes care of building or updating the UI.

Here's an example:

```
final repository = ref.watch(repositoryProvider);
return StreamBuilder<List<Recipe>>(
  stream: repository.recipesStream(),
  builder: (context, AsyncSnapshot<List<Recipe>> snapshot) {
    // extract recipes from snapshot and build the view
  }
)
...
```

`StreamBuilder` is handy because you don't need to use a subscription directly, and it unsubscribes from the stream automatically when the widget is destroyed.

> **Note**: Riverpod has a **StreamProvider**, which you can use to provide a stream to a widget. You can learn more about it at https://riverpod.dev/docs/providers/stream_provider.

Now that you understand how streams work, you'll convert your existing project to use them.

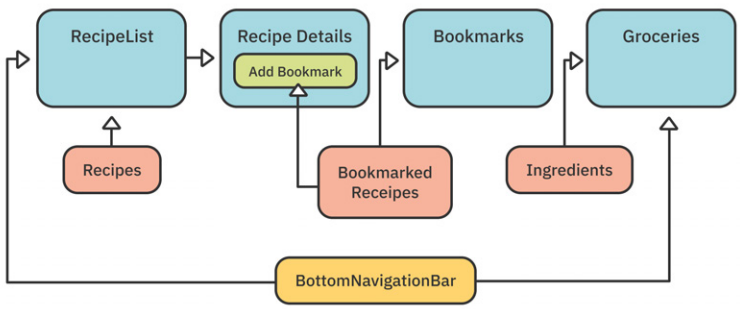
Adding Streams to Recipe Finder

You're now ready to start working on your recipe project. If you're following along with your app from the previous chapters, open it and keep using it with this chapter. If not, just locate the **projects** folder for this chapter and open **starter** in Android Studio.

> **Note**: If you use the starter app, don't forget to add your `apiKey` in **network/spoonacular_service.dart**.

To convert your project to use streams, you need to change `MemoryRepository` to add two new methods that return one stream for **recipes** and another for **ingredients**. Instead of just returning a list of static recipes, you'll use streams to modify that list and refresh the UI to display the change.

This is what the flow of the app looks like:



Here, you can see that the **Recipes** screen has a list of recipes. Bookmarking a recipe adds it to the bookmarked recipe list and updates both the bookmarks and the groceries screens.

You'll start by converting your repository code to return `Streams` and `Futures`.

Adding Futures and Streams to the Repository

Open **data/repositories/repository.dart** and change all of the return types to return a `Future`, except for the `init` and `close` methods. For example, change the existing `findAllRecipes()` to:

```
Future<List<Recipe>> findAllRecipes();
```

Do this for all the methods except `init()` and `close()`.

Your final class should look like this:

```
Future<List<Recipe>> findAllRecipes();

Future<Recipe> findRecipeById(int id);

Future<List<Ingredient>> findAllIngredients();

Future<List<Ingredient>> findRecipeIngredients(int recipeId);

Future<int> insertRecipe(Recipe recipe);

Future<List<int>> insertIngredients(List<Ingredient>
ingredients);

Future<void> deleteRecipe(Recipe recipe);

Future<void> deleteIngredient(Ingredient ingredient);

Future<void> deleteIngredients(List<Ingredient> ingredients);

Future<void> deleteRecipeIngredients(int recipeId);

Future init();

void close();
```

These updates allow you to have methods that work asynchronously to process data from a database or the network.

Next, add two new `Streams` after `findAllRecipes()`:

```
// 1
Stream<List<Recipe>> watchAllRecipes();
// 2
Stream<List<Ingredient>> watchAllIngredients();
```

Here's what this code does:

1. `watchAllRecipes()` listens for any changes to the list of recipes. For example, if the user does a new search, it updates the list of recipes and notifies listeners accordingly.
2. `watchAllIngredients()` listens for changes in the list of ingredients displayed on the **Groceries** screen.

You've now changed the interface, so you need to update the memory repository.

Cleaning Up the Repository Code

Before updating the code to use `streams` and `futures`, there are some minor housekeeping updates.

Open **data/repositories/memory_repository.dart** and notice there are some red squiggles. We'll address them step by step in a bit.

First, import the Dart **async** library:

```
import 'dart:async';
```

Next, add these new properties within the class:

```
//1
late Stream<List<Recipe>> _recipeStream;
late Stream<List<Ingredient>> _ingredientStream;
// 2
final StreamController _recipeStreamController =
    StreamController<List<Recipe>>();
final StreamController _ingredientStreamController =
    StreamController<List<Ingredient>>();
```

Here's what's going on:

1. `_recipeStream` and `_ingredientStream` are private fields for the streams. These will be captured the first time a stream is requested, which prevents new streams from being created for each call.
2. Creates `StreamControllers` for recipes and ingredients.

Next, add a constructor:

```
MemoryRepository() {
  // 1
  _recipeStream =
```

```
  _recipeStreamController.stream.asBroadcastStream(
      // 2
      onListen: (subscription) {
        // 3
        // This is to send the current recipes to new subscriber
        _recipeStreamController.sink.add(state.currentRecipes);
      },
    ) as Stream<List<Recipe>>;
    _ingredientStream =
  _ingredientStreamController.stream.asBroadcastStream(
      onListen: (subscription) {
        // This is to send the current ingredients to new
  subscriber

  _ingredientStreamController.sink.add(state.currentIngredients);
      },
    ) as Stream<List<Ingredient>>;
  }
```

This will initialize the streams.

1. Create a broadcast stream so that multiple listeners are available.
2. Add an `onListen` method to listen for new subscriptions.
3. Send the existing recipes to the new listener.

Here, you create a broadcast stream, which you need for multiple listeners, and then update the listener with the current list of recipes when they subscribe.

And now, add these new methods after `findAllRecipes()`:

```
  // 3
  @override
  Stream<List<Recipe>> watchAllRecipes() {
    return _recipeStream;
  }

  // 4
  @override
  Stream<List<Ingredient>> watchAllIngredients() {
     return _ingredientStream;
  }
```

The above functions are self-explanatory, `watchAllRecipes()` returns the stream of recipes as `_recipeStream` and `watchAllIngredients()` returns the stream of ingredients as `_ingredientStream`.

Updating the Existing Repository

`MemoryRepository` is full of red squiggles. That's because all the methods use the old signatures, and everything's now based on `Futures`.

Still in **data/repositories/memory_repository.dart**, replace the existing `findAllRecipes()` with this:

```
@override
// 1
Future<List<Recipe>> findAllRecipes() {
  // 2
  return Future.value(state.currentRecipes);
}
```

These updates:

1. Change the method to return a `Future`.
2. Wrap the return value with a `Future.value()`.

There are a few more updates you need to make before moving on to the next section.

First, in `init()` remove the `null` from the `return` statement so it looks like this:

```
@override
Future init() {
  return Future.value();
}
```

For this repository, there is no initialization needed, so just an empty future is returned. Then, update `close()` so it closes the streams.

```
@override
void close() {
  _recipeStreamController.close();
  _ingredientStreamController.close();
}
```

When dealing with `streams` and their controllers, you need to make sure you close them when you are finished. Closing them in the close method makes sure those streams are closed. In the next section, you'll update the remaining methods to return **futures** and add data to the **stream** using `StreamController`.

Sending Recipes Over the Stream

As you learned earlier, `StreamController`'s `sink` property adds data to streams. Since this happens in the future, you need to change the return type to `Future` and then update the methods to add data to the stream.

> **Note**: Make sure you add `@override` above each method.

To start, change `insertRecipe()` to:

```
@override
// 1
Future<int> insertRecipe(Recipe recipe) {
  if (state.currentRecipes.contains(recipe)) {
    return Future.value(0);
  }
  // 2
  state = state.copyWith(currentRecipes:
[...state.currentRecipes, recipe]);
  // 3
  _recipeStreamController.sink.add(state.currentRecipes);
  // 4
  final ingredients = <Ingredient>[];
  for (final ingredient in recipe.ingredients) {
    ingredients.add(ingredient.copyWith(recipeId: recipe.id));
  }
  insertIngredients(ingredients);
  // 5
  return Future.value(0);
}
```

Here's what you've updated:

1. Update the method's return type to `Future<int>`.
2. Update the state by adding the new recipe to the existing list.
3. Add the list to the recipe `sink`. You might wonder why you call `add()` with the same list instead of adding a single ingredient or recipe. The reason is that the stream expects a list, not a single value. Doing it this way replaces the previous list with the updated one.

4. Update all of the ingredients with the **recipe ID** and then insert the ingredients.
5. Return a `Future` value. You'll learn how to return the ID of the new item in a later chapter.

This replaces the previous list with the new list and notifies any stream listeners that the data has changed.

Now that you know how to convert the first method, it's time to convert the rest of the methods as an exercise. Don't worry, you can do it! :]

Exercise

Convert the remaining methods like you did with `insertRecipe()`. You'll need to do the following:

1. Update `MemoryRepository` methods to return a `Future` that matches the new `Repository` interface methods.
2. For all methods that change a watched item, add a call to add the item to the `sink`.
3. Remove all the calls to `notifyListeners()`. Hint - not all methods have this statement.
4. Wrap the return values in `Futures`.
5. Add `@override` before each method.

What do you think the return will look like for a method that returns a `Future<void>`? Got it? There might be a future for you yet.

```
  return Future.value();
```

If you get stuck, check out **memory_repository.dart** in this chapter's **challenge** folder — but first, give it your best shot!

After you complete the exercise, `MemoryRepository` shouldn't have any more red squiggles — but you still have a few more tweaks to make before you can run your new, stream-powered app.

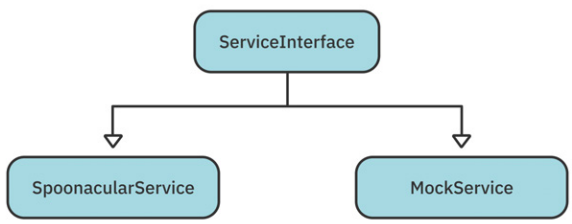
Note: It's very important that you add recipes to the `_recipeStreamController.sink` method for recipes and `_ingredientStreamController.sink` for ingredients. Check the **challenge** project to ensure you did this correctly. You'll need to do the same for the delete methods as well.

Switching Between Services

In an earlier chapter, you used a `MockService` to provide local data that never changes, but you also have access to `SpoonacularService`.

An easy way to do that is with an interface, or, as it's known in Dart, an **abstract class**. Remember that an interface or abstract class is just a contract that implementing classes will provide the given methods.

It'll look like this:



Go to the **network** folder and open **service_interface.dart**.

Here's what it looks like:

```
abstract class ServiceInterface {
  /// Query recipes with the given query string
  /// offset is the starting point
  /// number is the number of items
  Future<RecipeResponse> queryRecipes(
    String query,
    int offset,
    int number,
  );

  /// Get the details of a specific recipe
  Future<Response<Result<Recipe>>> queryRecipe(
    String id,
  );
}
```

This defines a class with two methods. One named `queryRecipes()`, for a list of recipes and `queryRecipe` for just a single recipe.

It has the same parameters and return values as `SpoonacularService` and `MockService`. Having each service implement this interface allows you to change the **providers** to provide this interface instead of a specific class.

You're now ready to integrate the new code based on streams. Fasten your seat belt! :]

Adding Streams to Bookmarks

The **Bookmarks** page uses `Consumer`, but you want to change it to a stream so it can react when a user bookmarks a recipe. To do this, you need to replace the reference to `MemoryRepository` with `Repository` and use a `StreamBuilder` widget.

Start by opening **ui/bookmarks/bookmarks.dart**. Replace `// TODO: Add Recipe Stream` with:

```
late Stream<List<Recipe>> recipeStream;
```

Next, replace `// TODO: Add initState` with:

```
@override
void initState() {
  super.initState();
  final repository = ref.read(repositoryProvider.notifier);
  recipeStream = repository.watchAllRecipes();
}
```

This will initialize the recipe stream.

Replace `// TODO: Replace with Stream` and the two subsequent lines with:

```
// 1
return StreamBuilder<List<Recipe>>(
  // 2
  stream: recipeStream,
  // 3
  builder: (context, AsyncSnapshot<List<Recipe>> snapshot) {
    // 4
    if (snapshot.connectionState == ConnectionState.active) {
      // 5
      recipes = snapshot.data ?? [];
    }
```

Don't worry about the red squiggles for now. This code:

1. Uses `StreamBuilder`, which uses a `List<Recipe>` stream type.
2. Uses the new `recipeStream` to return a stream of recipes for the builder to use.
3. Uses the builder callback to receive your **snapshot**.
4. Checks the state of the connection. When the state is active, you have data.

At the bottom of the method, find `// TODO: Add closing brackets` and replace it with:

```
    },
  );
```

At this point, you've achieved one of your two goals: you've changed the **Recipes** screen to use streams. Next, you'll do the same for the **Groceries** tab.

Adding Streams to Groceries

To add streams to the grocery list, you'll need to watch the ingredient stream.

Open **ui/groceries/groceries.dart**.

Find the `initState()` method and replace `// TODO: Add Ingredient Stream` with:

```
  final repository = ref.read(repositoryProvider.notifier);
  final ingredientStream = repository.watchAllIngredients();
  ingredientStream.listen(
    (ingredients) {
      setState(() {
        currentIngredients = ingredients;
      });
    },
  );
```

In the `buildIngredientList()` method, remove:

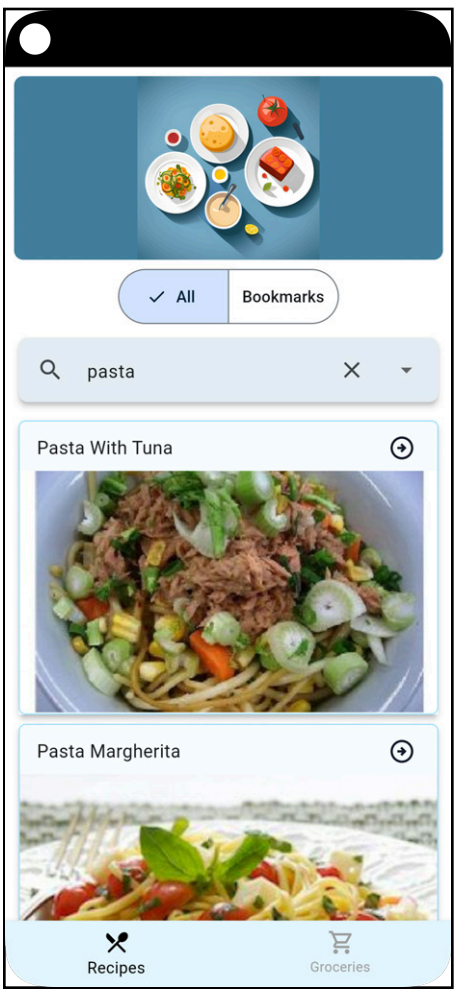
```
  final repository = ref.watch(repositoryProvider);
  currentIngredients = repository.currentIngredients;
```

This is no longer needed as the stream is listened to above.

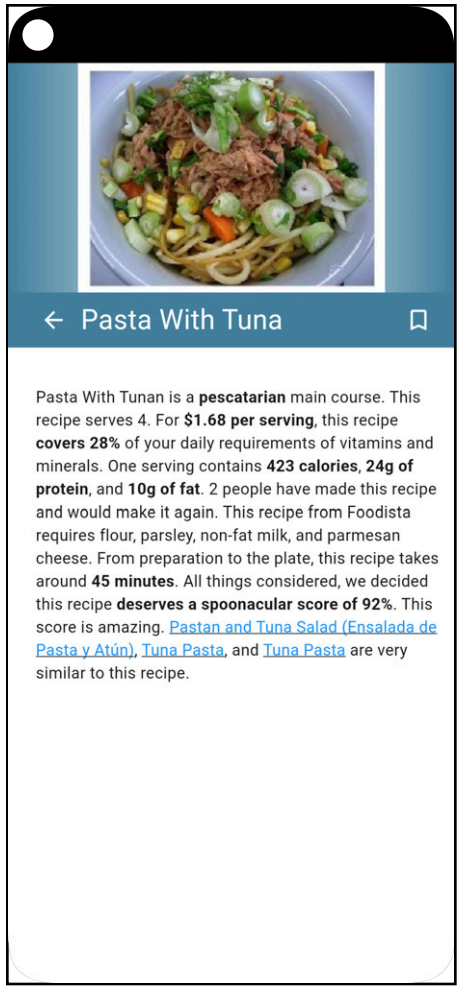
Finally, modify `startSearch()` as follows:

```
void startSearch(String searchString) {
  searching = searchString.isNotEmpty;
  searchIngredients = currentIngredients
      .where((element) => true ==
element.name?.contains(searchString))
      .toList();
  setState(() {});
}
```

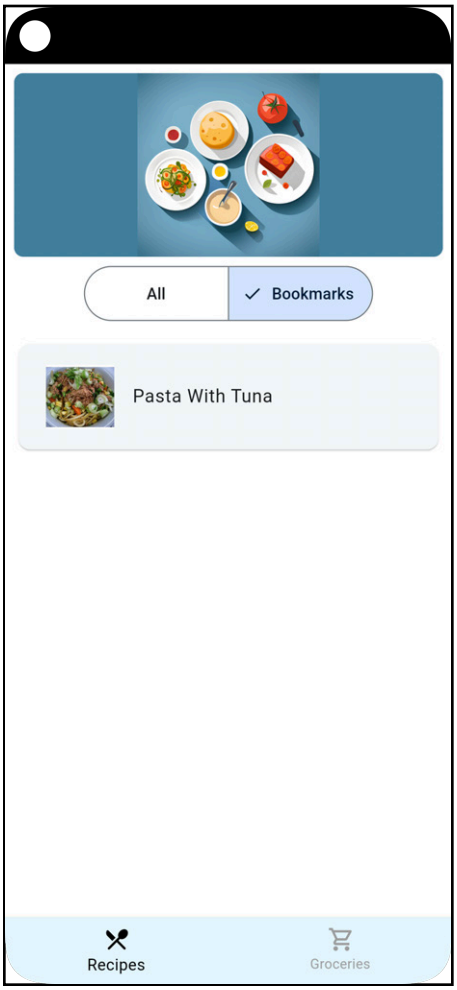
Stop and restart your app. Make sure it works as before. Your main screen will look something like this after a search:



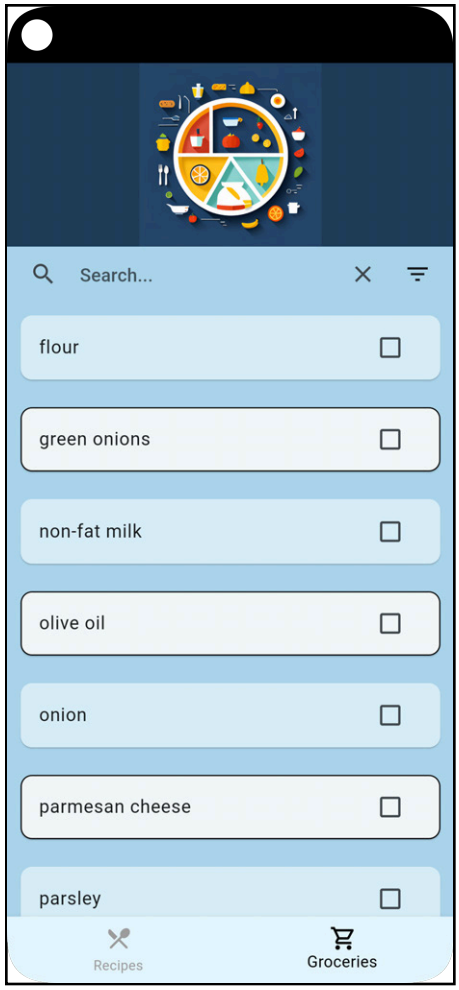
Tap a recipe. The **Details** page will look like this:



Next, tap the **Bookmark** button to return to the **Recipes** screen, then tap on the **Bookmarks** switch to see the recipe you just added:



Finally, go to the **Groceries** tab and make sure the recipe ingredients are all showing.



Congratulations! You're now using streams to control the flow of data. If any of the screens change, the other screens will know about that change and will update the screen.

You're also using the `Repository` interface, so you can go back and forth between a memory class and a different class in the future.

Key Points

- Streams are a way to asynchronously send data to other parts of your app.
- You usually create streams by using `StreamController`.
- Use `StreamBuilder` to add a stream to your UI.
- Abstract classes, or interfaces, are a great way to abstract functionality.

Where to Go From Here?

In this chapter, you learned how to use streams. If you want to learn more about the topic, visit the Dart documentation at https://dart.dev/tutorials/language/streams.

In the next chapter, you'll learn about databases and how to persist your data locally.

Chapter 15: Saving Data Locally

By Kevin David Moore

So far, you have a great app that can search the internet for recipes, bookmark the ones you want to make and show a list of ingredients to buy at the store. But what happens if you close the app, go to the store and try to look up your ingredients? They're gone! As you might have guessed, having an in-memory repository means that the data doesn't persist after your app closes.

One of the best ways to persist data is with a **database**. Android, iOS, macOS, Windows and the web provide the **SQLite** database system access. This allows you to **insert**, **read**, **update** and **remove** structured data that are persisted on disk.

In this chapter, you'll learn about using the **Drift** and **sqlbrite** packages.

By the end of the chapter, you'll know:

- How to **insert**, **fetch** and **remove recipes** or **ingredients**.
- How to use the **sqlbrite** library and receive updates via streams.
- How to leverage the features of the **Drift** library when working with databases.