CASTILLA-LA MANCHA'S UNIVERSITY

ESI – Escuela Superior de Informática

Flutter Report

Elena Ballesteros Morallón | Elena.Ballesteros2@alu.uclm.es

Antonio Campallo Gómez | Antonio.Campallo@alu.uclm.es

Course: Web Systems Development

Titulation: Computer Science Degree

Date: 13/02/25

Index

1. INTRODUCTION

1.1. What is Flutter?

Flutter is an open-source framework that enables the development of responsive and interactive web applications using a single Dart codebase. It allows for the “write once, run anywhere” methodology, which makes it an excellent option for cross-platform initiatives. This means that, the same code can be compiled to different platforms such as Android, iOS, web and desktop applications for Windows, Linux and macOS. In this report we will focus only on web applications so Flutter will be a synonym for Flutter Web.

Flutter Web is innovative, in contrast to conventional frameworks such as React, Angular, or Vue, it does not use HTML and CSS directly. Rather, it renders the complete UI with Canvas and WebAssembly, delivering a native app-like experience online.

1.2. Origins and Evolution

Flutter was launched in 2017 as a software development kit for creating mobile apps for Android and iOS, with the goal of streamlining UI development while delivering excellent performance. In 2019, Google introduced Flutter for web, broadening its functionalities to web browsers.

With the release of Flutter 2 in 2021, web support has reached stability and is now officially integrated into the Flutter ecosystem. Current version is 3.27 released on the 11th of December.

1.3. Introduction to Dart

Dart is a C-like, object-oriented programming language used to code Flutter apps. Dart is compiled to JavaScript, so it runs on all web browsers. Some key points are:

- The main() method is the entry point of a class.
- It supports inheritance, abstract classes and interfaces.
- It has type inference.
- All data types are objects and therefore their default value is null.

The objective of this report is not learning about Dart, so we will learn more about it in following code examples.

2. FLUTTER ARCHITECTURE

During development, Flutter apps run in a Virtual Machine that allows hot reloading and visualization of changes without needing to recompile.

In this section, we will explain the layer and components following a top-down approach:

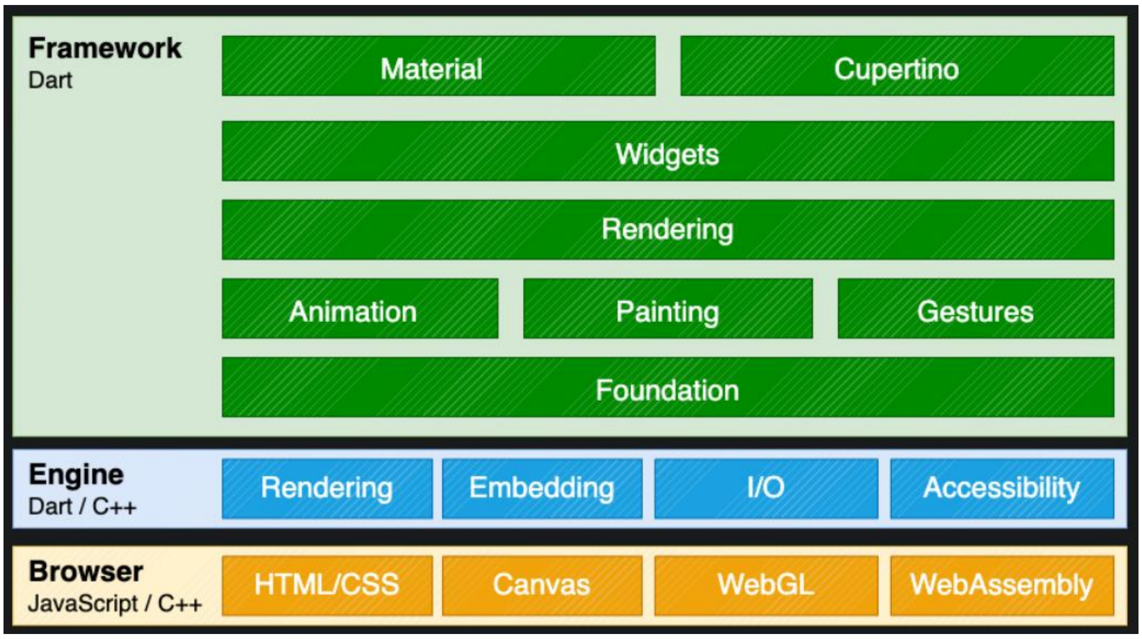


Fig. 1 Flutter layers and components

Framework layer oversees setting the layout, rendering, user interaction and visual appearance.

The engine is responsible for processing and rendering graphics whenever the screen needs to be updated. Additionally, it manages key functions such as text rendering, file and network operations, accessibility support, plugin integration, and executing Dart code. This engine creates a widget tree which will be passed into the render.

To execute in a browser, offers two build modes, and two renderers that convert UI primitives (stored as Scene objects) into pixels, using Skia. Flutter chooses the build mode when building the app, and determines which renderers are available at runtime:

- Default build and canvaskit.
- WebAssembly build and skwasm renderer or canvaskit if the browser does not support skwasm.

- Skwasm Rendering: A more compact version of Skia compiled to WebAssembly, designed to run on a separate thread for better performance. It requires the WebAssembly build mode and a server that supports **SharedArrayBuffer** to enable multi-threading. If these requirements are not met, it runs in single-threaded mode.
- CanvasKit Rendering: Is compatible with all modern browsers and is the renderer that is used in the *default* build mode. It has smoother animations and better perfomance in Complex UIs and is ideal for apps with advanced graphics, animations or visual effects. It also includes a copy of Skia compiled to WebAssembly.

3. KEY COMPONENTS

3.1. Widgets: The Core of UI

Flutter widgets work in an equivalent way to React components, acting as the building blocks of the UI. They represent an immutable declaration of part of the interface structure into a hierarchy. All widgets must implement a build method that returns another widget. This method is called when created and when its dependencies change, for example, its state.

We have distinct categories of widgets:

- Static: Text, Image, Container, Icon, etc
- Interactive: ElevatedButton, TextField, Checkbox, etc.
- Layout: LayoutBuilder, Column, Row, Stack, Card, ListTile, etc.

There is another category: stateful and stateless widgets. Choosing one or the other depends if it should change its appearance based on user interactions.

Here we have an example of a custom stateless widget:

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

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Hello, Flutter Web!', style: TextStyle(fontSize: 24)),
        ElevatedButton(onPressed: () {}, child: Text('Click Me'))
      ],
    )}
}
```

3.2. State Management

Managing app state is especially important for interactivity. Flutter provides multiple options like the next ones:

- SetState(): For local state updates in a widget.
- Provider: A structured approach for global state management.
- Riverpod: A safer and more flexible alternative to Provider.
- Bloc: An event-driven state management, ideal for large apps.
- ValueNotifier and InheritedNotifier: An approach using only Flutter provided tooling to update state and notify the UI of changes.
- InheritedWidget and InheritedModel: The low-level approach used to communicate between ancestors and children in the widget tree. This is what provider, and many other approaches use under the hood.

3.3. Design Systems

Flutter includes two design systems: Material Design and Cupertino. Material Design follows Google's UI guidelines that are great for web and Android, and Cupertino is designed to mimic IOS's style UI, this is going to be useful if you want an Apple-like design on web.

The following examples show that widgets are different but share the same syntax.

Material Design

```
ElevatedButton(
  onPressed: () {},
  child: Text('Click Me'),
)
```

Cupertino

```
CupertinoButton(
  onPressed: () {},
  child: Text('Click Me'),
)
```

4. NAVIGATION AND ROUTING

Flutter Web provides a comprehensive navigation system that supports deep linking and integrates with the browser's history. Navigation can be managed using Navigator for simple use cases or Router for more complex applications.

4.1. Navigation with Navigator

The Navigator widget manages a stack of routes and manages transitions between screens. It supports both imperative navigation (using push() and pop()) and named routes.

- **Imperative Navigation** (Recommended for Simple Apps): With imperative navigation, you manually push and pop screens in response to user actions:

  ```
  Navigator.of(context).push(
    MaterialPageRoute(builder: (context) => const SecondScreen()),
  );
  ```

  The Navigator maintains a stack of Route objects, allowing users to navigate forward and back.

  MaterialPageRoute is a subclass of Route that includes Material Design transitions.

- **Named Routes** (Limited Deep Link Support): Named routes allow to define a list of paths in MaterialApp.routes and navigate using pushNamed():

  ```
  Navigator.pushNamed(context, '/second');
  ```

  Named routes work with deep links but lack flexibility. They do not support browser forward/back buttons properly. Therefore, they are not recommended for complex apps.

4.2. Navigation with Router (Recommended for Web)

For applications with deep linking, multiple Navigators, or browser history integration, Flutter recommends using Router or a package like go_router.

- **Declarative** Routing with go_router: Deep links always navigate to the correct screen. Supports the browser's **back and forward buttons**. Uses **page-backed routes**, making them more predictable than Navigator.push().

```
context.go('/second');
```

- **Full Customization** with RouterDelegate and RouteInformationParser: Advanced developers can override RouterDelegate and RouteInformationParser for full control over navigation, including custom URL parsing, dynamic route management and state-driven navigation.

4.3. Using Router and Navigator Together

Both Navigator and Router can be used in the same app:

- Router for deep linking and web history integration.
- Navigator for in-app modal dialogs or transient screens.
- Router-backed pages are deep-linkable; Navigator's pageless routes are not.

5. MIGRATING FROM HTML

To interact directly with JavaScript and HTML elements using "dart:js" and "PlatformView". An example of calling JavaScript from Dart can be:

```
import 'dart:js' as js;

void runJS() {
  js.context.callMethod('alert', ['Hello from Flutter Web!']);
}
```

6. PROS AND CONS

To summarize, we have seen a brief introduction to the architecture and main components of Flutter. However, while it offers a smooth UI and powerful performance, it also has some limitations, especially for web development.

Pros:

- Unified codebase: Flutter enables developers to write a single codebase that can be deployed on mobile, web, and desktop.
- Smooth animations & native-like UI with Skia rendering.
- Support for Progressive Web Apps (PWA): web applications that offer a user experience like native apps while being access through a browser.
- Community support: with support to plugins, templates and other configured widgets.

Cons:

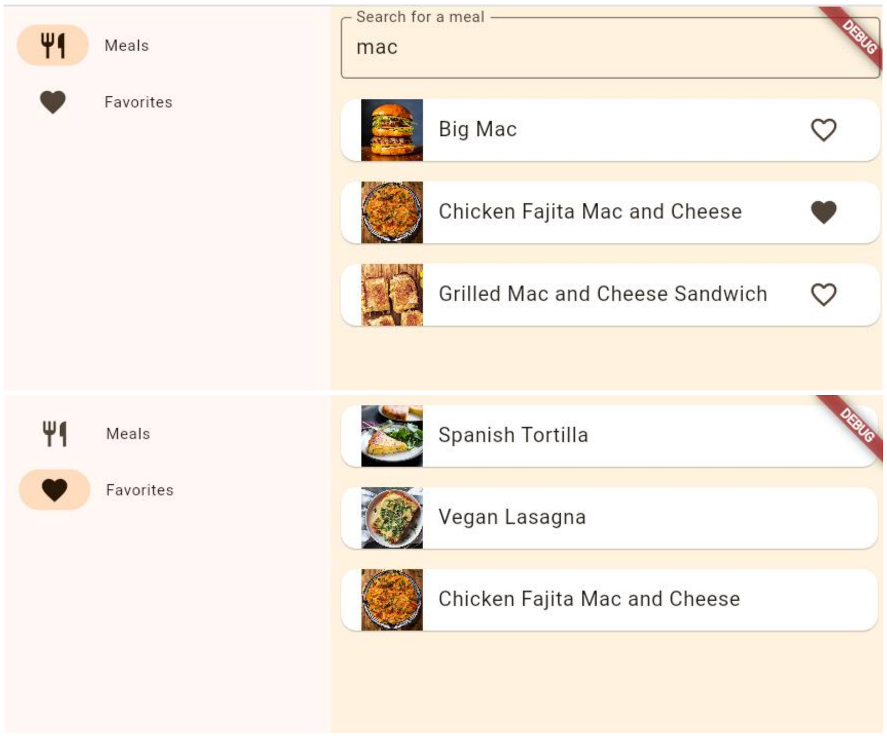
- Larger app size compared to native web frameworks (React, Angular).
- Not ideal for SEO-heavy websites.
- Performance limitations: sometimes the web page might take a few seconds to load.

Examples of use cases:

- Dashboards and enterprise tools.
- Data-driven apps and admin panels.
- Games and highly animated experiences.
- Progressive Web Apps (PWA).
- Apps with uniform UI across mobile and web.

7. TUTORIAL

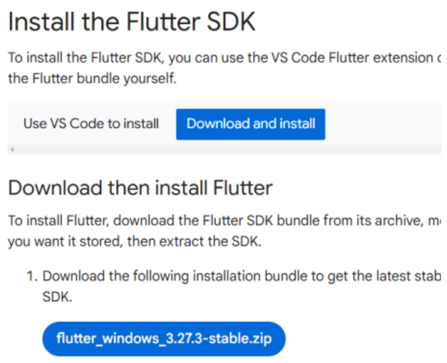
In this section, we will implement a web app to search and add to favorites different types of meals. Here is the final result of the app[1]:



7.1. Previous installations and configurations

First, we need to have an IDE installed. For example, we will consider using Visual Studio Code. We can install it here: Visual Studio Code Installation.

We will also need to have Flutter SDK installed, which can be done here: Flutter Installation. On the page, if you are on Windows, choose the "Web" option and then follow the installation steps, including unzipping the SDK.



Install the Flutter SDK

To install the Flutter SDK, you can use the VS Code Flutter extension o
the Flutter bundle yourself.

Download then install Flutter

To install Flutter, download the Flutter SDK bundle from its archive, m
you want it stored, then extract the SDK.

1. Download the following installation bundle to get the latest stab
SDK.

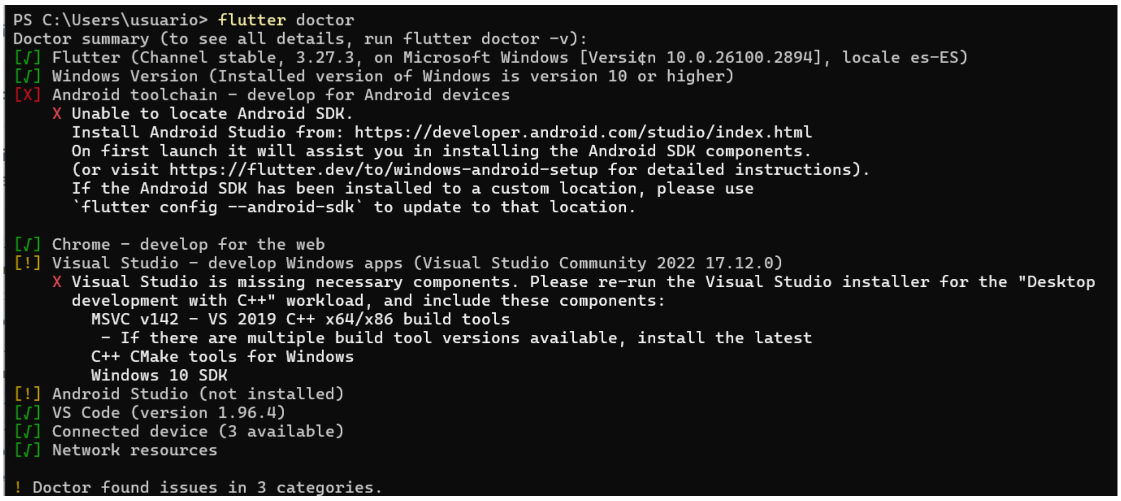
Now we need to update the PATH variable in Windows. To do this, follow these steps:

- Search for *"environment variables"* in the Windows search bar and click on *"Edit the system environment variables"*.
- In the window that opens, click on *"Environment Variables"*.

[1] Web demo and source code: https://github.com/elena-17/Flutter-welcoming

- In the *"User variables for [your user]"* section, select *"Path"* and click *"Edit"*.
- Check if there is an existing path for Flutter. In this example, we added *"C:\Users\[usuario]\Downloads\flutter_windows_3.27.3-stable\flutter\bin"*. If the path is not there, add it and move it to the top of the list.
- Once this is done, open PowerShell and run the following command to verify the installation:

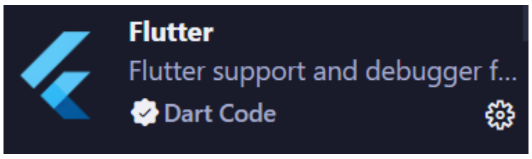
```
> flutter doctor
```



```
PS C:\Users\usuario> flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 3.27.3, on Microsoft Windows [Versi¢n 10.0.26100.2894], locale es-ES)
[√] Windows Version (Installed version of Windows is version 10 or higher)
[X] Android toolchain - develop for Android devices
    X Unable to locate Android SDK.
      Install Android Studio from: https://developer.android.com/studio/index.html
      On first launch it will assist you in installing the Android SDK components.
      (or visit https://flutter.dev/to/windows-android-setup for detailed instructions).
      If the Android SDK has been installed to a custom location, please use
      `flutter config --android-sdk` to update to that location.

[√] Chrome - develop for the web
[!] Visual Studio - develop Windows apps (Visual Studio Community 2022 17.12.0)
    X Visual Studio is missing necessary components. Please re-run the Visual Studio installer for the "Desktop
      development with C++" workload, and include these components:
        MSVC v142 - VS 2019 C++ x64/x86 build tools
         - If there are multiple build tool versions available, install the latest
        C++ CMake tools for Windows
        Windows 10 SDK
[!] Android Studio (not installed)
[√] VS Code (version 1.96.4)
[√] Connected device (3 available)
[√] Network resources

! Doctor found issues in 3 categories.
```

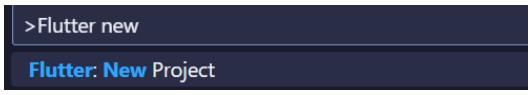
As we can see, since we are going to develop for the web, there is no need to fulfil the requirements related to Windows or Visual Studio Community 2022.

Finally, install Flutter extension in Visual Studio Code.

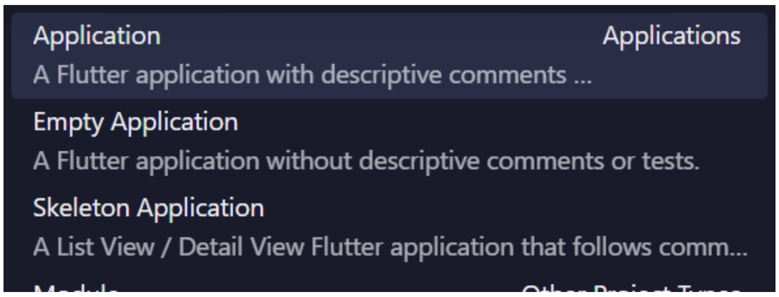


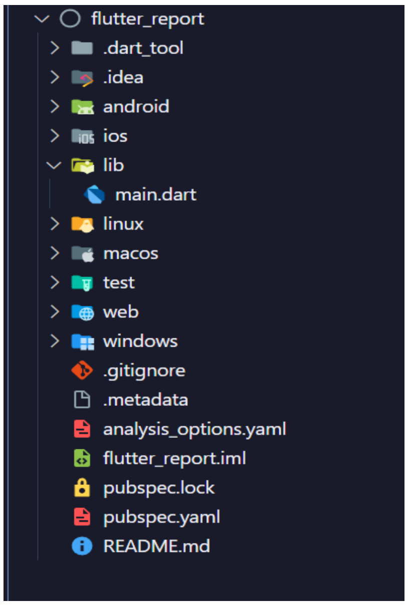
7.2. Creation of new project

We are going to open the command palette pressing F1 and we will type "flutter new", then we select the command **"Flutter: New Project".**



After clicking on it, we will see this:


Application
Applications
A Flutter application with descriptive comments ...
Empty Application
A Flutter application without descriptive comments or tests.
Skeleton Application
A List View / Detail View Flutter application that follows comm...


```
flutter_report
  .dart_tool
  .idea
  android
  ios
  lib
    main.dart
  linux
  macos
  test
  web
  windows
  .gitignore
  .metadata
  analysis_options.yaml
  flutter_report.iml
  pubspec.lock
  pubspec.yaml
  README.md
```

So, we need to click on *“Application”* and then, we choose the folder where we want to create our project. At this point, we should verify if our project structure matches this one or is similar.

Now, we need to open the *"pubspec.yaml"* file, which specifies the basic information about the app, like http version, and replace its content with this:

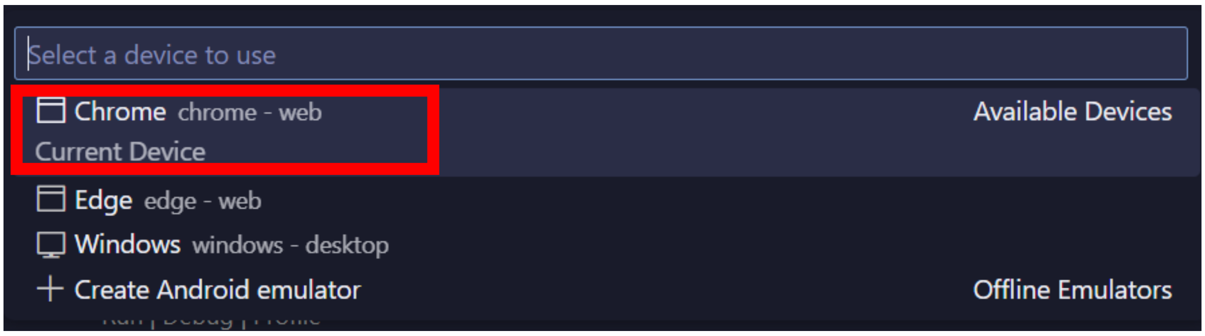
```
name: flutter_food_app
description: A new Flutter project.

publish_to: 'none' # Remove this line if you wish to publish to pub.dev

version: 0.0.1+1

environment:
  sdk: '>=2.19.4 <5.0.0'
  flutter: 3.27.3

dependencies:
  flutter:
    sdk: flutter

  provider: ^6.1.2
  http: ^0.13.6
  cached_network_image: ^3.3.0

dev_dependencies:
  flutter_test:
    sdk: flutter

  flutter_lints: ^5.0.0

flutter:
  uses-material-design: true
```

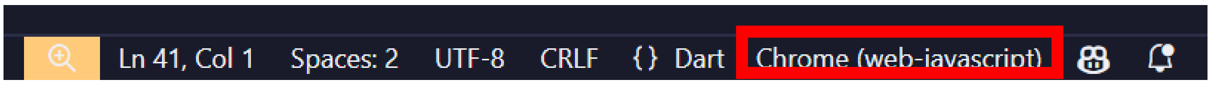
Now, open another configuration file called “analysis_options.yaml” and we replace its content with this one:

```
include: package:flutter_lints/flutter.yaml

linter:
  rules:
    prefer_const_constructors: false
    prefer_final_fields: false
    use_key_in_widget_constructors: false
    prefer_const_literals_to_create_immutables: false
    prefer_const_constructors_in_immutables: false
    avoid_print: false
```

This file helps configure various aspects of your Flutter project, including the level of code analysis. For beginners, it is better to keep the analysis relaxed, but as app gets ready for production, it is recommended to enforce stricter checks to ensure better quality and performance.

Finally, we are going to start our app, but, before adding buttons, we need to configurate how we run our app, our target device. We need to choose Chrome because we are doing a web with Flutter.





To run the app, we can click on the Run button or execute this command on the terminal:

```
> flutter run -d chrome
```

7.3. Structure and imports

Imports allow us to use external packages and functionalities within our application. In this case, place these imports at the top of the file:

- “dart:convert”: To convert JSON data.
- “package:flutter/material.dart”: To build the user interface.
- “package:http/http.dart as http”: To make HTTP requests to the API.
- “package:provider/provider.dart”: To manage the global state using the Provider pattern.

These imports are essential for the correct execution of the code and interaction with the meals API.

This is the structure that we will follow that is based on widgets, which allows us to separate responsibilities. Also, it makes the app more scalable:

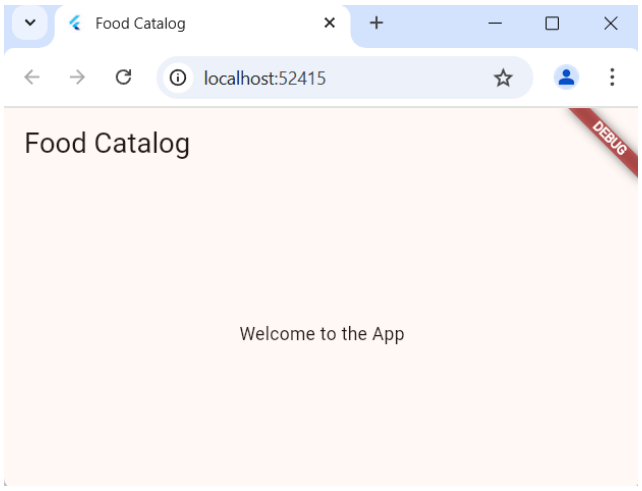
- **MyApp:** Defines the main application configuration and theme.
- **MealProvider:** Manages the application state, such as the list of meals and favorites.
- **MyHomePage:** Contains the navigation rail and manages the screen content.
- **MealsPage:** Displays the list of meals and the search bar.

- **MealItem:** Meal item representation.
- **FavoritesPage:** Displays the list of favorite meals.

7.4. MyApp, the root widget

To improve the application's appearance, we have applied Flutter's ColorScheme and adjusted the widget styles. We have updated the main application theme using "ThemeData", defined a color scheme based on a seed color. The home page will be define by the widget MyHomePage, for now it's just a template.

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

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Food Catalog',
      theme: ThemeData(
        useMaterial3: true,
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.orange),
      ),
      home: MyHomePage(),
    );
  }
}

class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Food Catalog')),
      body: Center(child: Text('Welcome to the App')),
    );
  }
}
```



Food Catalog

Welcome to the App

7.5. Stateless versus stateful widgets

We have divided the classes (widgets) into stateless and stateful as necessary.

“MyApp”, “MealItem” and “FavoritesPage” remain “StatelessWidget” because they do not need to change their state.

“MyHomePage” and “MealsPage” extends “StatefulWidget” class because they manage UI changes.

7.6. Adding the navigation rail.

To allow navigation between the meals screen and the favorites screen, we have added a NavigationRail, that provides a sidebar navigation system. The “selectedIndex” determines which screen is displayed: '0' for meals and '1' for favorites. In the left side of the screen, there’s a menu to change between pages and the main area will be related to each page, for now it only displays a text.

7.6.1. Responsiveness

The widget to use, in this case, is “LayoutBuilder”. It lets you change your widget tree depending on how much available space you have. We used Flutter's refactor menu to make the required changes.

1. We are going to change our “MyHomePage” like a “StatefulWidget”.

```
class MyHomePage extends StatefulWidget {
@override
State<MyHomePage> createState() => _MyHomePageState();
}
class _MyHomePageState extends State<MyHomePage> {
    Widget build(BuildContext context) {
            return Scaffold(
                    appBar: AppBar(title: Text('Food Catalog')),
                    body: Center(child: Text('Welcome to the App')),
            );
    }
}
```

2. Inside “_MyHomePageState”s build method, put your cursor on “Scaffold”.

3. Call up the refactor menu.

4. Select “Wrap with Builder”.

5. Modify the name of the newly added “Builder” to “LayoutBuilder”.

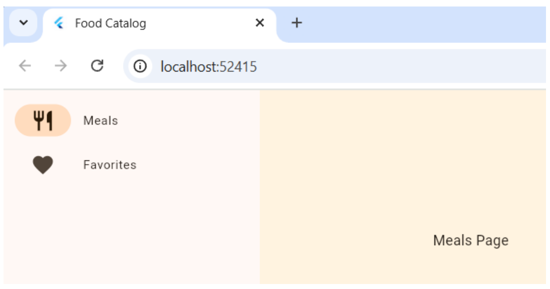
6. Modify the callback parameter list from (context) to (context, constraints).

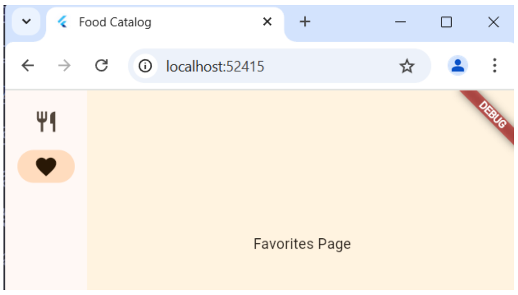
“LayoutBuilder”'s builder runs whenever constraints change, such as when resizing the window, rotating the device, or when nearby widgets grow. Now, your code can check constraints to decide whether to show the label.

At this point of the tutorial, the app adapts to its environment when: screen size, orientation and platform changes, thus making it fully responsive. The only work that remains is to replace each page with their full functionality.

```
class MyHomePage extends StatefulWidget {
```

```dart
  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  var selectedIndex = 0;

  @override
  Widget build(BuildContext context) {
    Widget page = selectedIndex == 0 ? Center(child: Text('Meals Page')) : Center(child: Text('Favorites Page'));

    return LayoutBuilder(
      builder: (context, constraints) {
        return Scaffold(
          body: Row(
            children: [
              SafeArea(
                child: NavigationRail(
                  extended: constraints.maxWidth >= 600, // Expande el menú si la pantalla es grande
                  selectedIndex: selectedIndex,
                  onDestinationSelected: (index) {
                    setState(() {
                      selectedIndex = index;
                    });
                  },
                  destinations: [
                    NavigationRailDestination(icon: Icon(Icons.restaurant), label: Text('Meals')),
                    NavigationRailDestination(icon: Icon(Icons.favorite), label: Text('Favorites')),
                  ],
                ),
              ),
              Expanded(
                child: Container(
                  color: Colors.orange[50],
                  child: page,
                ),
              ),
            ],
          ),
        );
      },
    );
  }
}

```





7.7. Adding MealProvider to manage data and search bar

First of all, we need to add more imports.

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

Then, we will create a Provider that manages meals. This will make global state management easier and allow us to update data efficiently. We also connect it with “ChangeNotifier”, because it integrates well with Flutter’s state management and allows widgets to listen for updates efficiently. When “notifyListeners” is called (line 6 and 21), it tells all listening widgets to rebuild the updated state. This is how we dynamically update the UI when the favorite meals list changes.

```dart
 1. class MealProvider extends ChangeNotifier {
 2.   List<Map<String, dynamic>> meals = [];
 3.   Future<void> fetchMeals(String query) async {
 4.     if (query.isEmpty) {
 5.       meals = [];
 6.       notifyListeners();
 7.       return;
 8.     }
 9.
10.     final response = await
http.get(Uri.parse('https://www.themealdb.com/api/json/v1/1/search.php?s=$query'))
;
11.     if (response.statusCode == 200) {
12.       final data = jsonDecode(response.body);
13.       meals = data['meals'] != null
14.           ? (data['meals'] as List)
15.               .map((meal) => {
16.                     'name': meal['strMeal'],
17.                     'image': meal['strMealThumb'],
18.                   })
19.               .toList()
20.           : [];
21.       notifyListeners();
22.     }
23.   }
24. }
25.
```

- “fetchMeals(query”): Fetches meals from the API, retrieves them, and stores them in meals.
- “notifyListeners()”: Notifies the app when there is new data, ensuring the UI updates accordingly.

Now, we need to wrap “MyApp” with “ChangeNotifier” so that the entire app can access “MealProvider”. We will modify our “MyApp” class.

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (context) => MealProvider(),
      child: MaterialApp(
        title: 'Food Catalog',
        theme: ThemeData(
          useMaterial3: true,
          colorScheme: ColorScheme.fromSeed(seedColor: Colors.orange),
        ),
        home: MyHomePage(),
      ),
    );
  }
}
```

7.8. Meal Item and Meals Page

To improve code reusability and organization, we have extracted the meal item representation into a separate widget called MealItem. We have created a new class “MealItem” that receives meal data and displays it using a Card.

```dart
class MealItem extends StatelessWidget {
  final Map<String, dynamic> meal;

  MealItem({required this.meal});

  @override
  Widget build(BuildContext context) {
   // var mealProvider = context.watch<MealProvider>();

    return Card(
      color: Colors.white,
      margin: EdgeInsets.all(8.0),
      child: ListTile(
        leading: Image.network(meal['image']!),
        title: Text(meal['name']!),
        /*trailing: IconButton(
          icon: Icon(
            mealProvider.favorites.contains(meal) ? Icons.favorite :
Icons.favorite_border),
          onPressed: () => mealProvider.toggleFavorite(meal),
        ),*/
      ),
    );
  }
}

```

Now, we are going to add our search bar for meals using and add our “MealsPage”:

- “TextField”: Allows users to type their search query.
- “onChanged”: Calls fetchMeals(query) every time the user types.
- “ListView.builder”: Displays the search results dynamically.

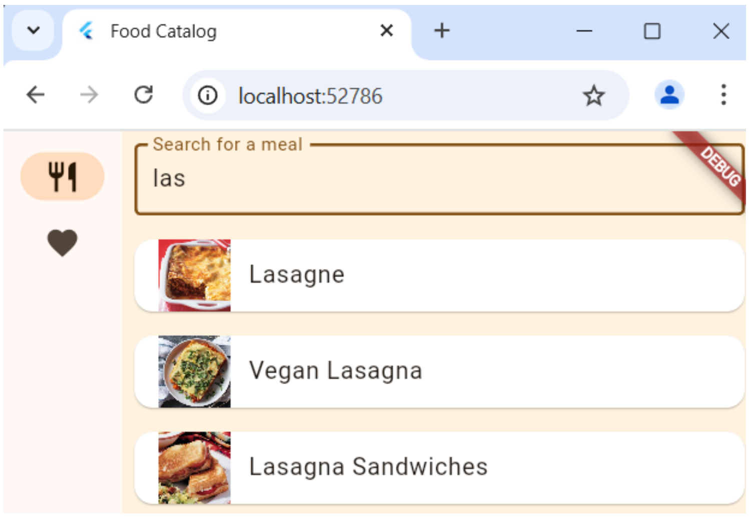
```dart
class MealsPage extends StatefulWidget {
  @override
  _MealsPageState createState() => _MealsPageState();
}

class _MealsPageState extends State<MealsPage> {
  TextEditingController searchController = TextEditingController();

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

```dart
  var mealProvider = context.watch<MealProvider>();

  return Column(
   children: [
    Padding(
     padding: const EdgeInsets.all(8.0),
     child: TextField(
      controller: searchController,
      decoration: InputDecoration(
       labelText: 'Search for a meal',
       border: OutlineInputBorder(),
      ),
      onChanged: (query) {
       mealProvider.fetchMeals(query);
      },
     ),
    ),
    Expanded(
     child: mealProvider.meals.isEmpty
       ? Center(child: Text('Type a meal name to search.'))
       : ListView.builder(
         itemCount: mealProvider.meals.length,
         itemBuilder: (context, index) {
          var meal = mealProvider.meals[index];
          return MealItem(meal: meal);
         },
        ),
    ),
   ],
  );
 }
}

```

Lastly, we need to add our new page inside “MyHomePage”, we will change the next line.

```dart
Widget page = selectedIndex == 0 ? Center(child: Text('Meals Page')) : Center(child: Text('Favorites Page'));
```

```dart
Widget page = selectedIndex == 0 ? MealsPage() : Center(child: Text('Favorites Page'));
```



Search for a meal
las

Lasagne

Vegan Lasagna

Lasagna Sandwiches

7.9. Adding favorites

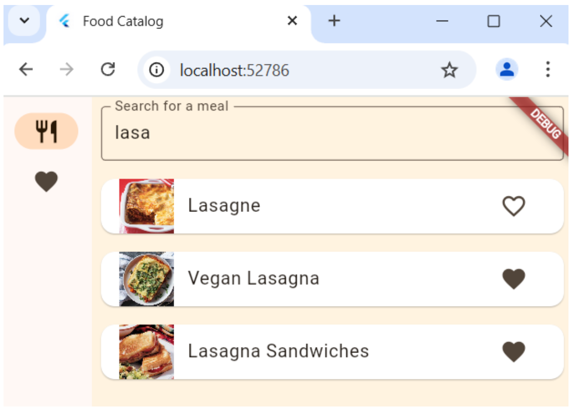
Now, we will modify our “MealProvider” to add a method called “toggleFavorite()” to manage adding/removing meals from favorites. We also need to include a new list for our favorites.

```
class MealProvider extends ChangeNotifier {
  List<Map<String, dynamic>> meals = [];
  List<Map<String, dynamic>> favorites = []; // Lista de favoritos
  Future<void> fetchMeals(String query) async {
    if (query.isEmpty) {
      meals = [];
      notifyListeners();
      return;
    }
    final response = await http.get(Uri.parse('https://www.themealdb.com/api/json/v1/1/search.php?s=$query'));
    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);
      meals = data['meals'] != null
          ? (data['meals'] as List)
              .map((meal) => {
                    'name': meal['strMeal'],
                    'image': meal['strMealThumb'],
                  })
              .toList()
          : [];
      notifyListeners();
    }
  }
  void toggleFavorite(Map<String, dynamic> meal) {
    if (favorites.contains(meal)) {
      favorites.remove(meal);
    } else {
      favorites.add(meal);
    }
    notifyListeners();
  }
}

```

Now, our “MealItem” with comments will be modify and we are gonna eliminate them.

```
class MealItem extends StatelessWidget {
  final Map<String, dynamic> meal;

  MealItem({required this.meal});

  @override
  Widget build(BuildContext context) {
   var mealProvider = context.watch<MealProvider>();

    return Card(
      color: Colors.white,
      margin: EdgeInsets.all(8.0),
      child: ListTile(
        leading: Image.network(meal['image']!),
        title: Text(meal['name']!),
        trailing: IconButton(
          icon: Icon(
            mealProvider.favorites.contains(meal) ? Icons.favorite : Icons.favorite_border),
          onPressed: () => mealProvider.toggleFavorite(meal),
        ),
      ),
    );
  }
}

```

When the button is pressed, it calls “toggleFavorite()”, which either adds or removes the meal from the favorite list. Also the button’s icon changes dynamically to show whether the meal is favorited or not.



7.10. Favorites page

Now that we know how to create a page, we will add a "Favorites" page. This page will display the meals that the user has marked as favorites.

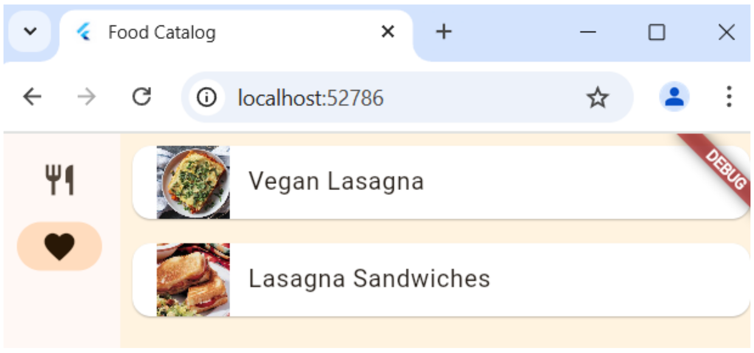
- When we need a scrollable column, we use “ListView”.
- To access the shared state (favorites list), we use “`context.watch<MealProvider>()`”.

```
class FavoritesPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    var mealProvider = context.watch<MealProvider>();

    if (mealProvider.favorites.isEmpty) {
      return Center(child: Text('No favorites yet.'));
    }

    return ListView.builder(
      itemCount: mealProvider.favorites.length,
      itemBuilder: (context, index) {
        var meal = mealProvider.favorites[index];
        return Card(
          color: Colors.white,
          margin: EdgeInsets.all(8.0),
          child: ListTile(
            leading: Image.network(meal['image']!),
            title: Text(meal['name']!),
          ),
        );
      },
    );
  }
}

```

We also need to change the page in “MyHomePage”:

```
Widget page = selectedIndex == 0 ? MealsPage() : Center(child: Text('Favorites Page'));
```

```
Widget page = selectedIndex == 0 ? MealsPage() : FavoritesPage();
```



7.11. Deploy the app

Once we have finished our web page, it’s time to deploy it. To build the app, execute the following commands:

```
> flutter config --enable-web

> flutter build web --release
```

The new folder *build/web* contains all generated files to be deployed in a service like Github Pages (see Annex: Deployment to Github Pages)

8. REFERENCES

Flutter. (s.f.). Flutter for web developers. Recuperado de https://docs.flutter.dev/get-started/flutter-for/web-devs

Flutter. (s.f.). Multi-platform web. Recuperado de https://flutter.dev/multi-platform/web

Flutter. (s.f.). Flutter widgets. Recuperado de https://docs.flutter.dev/ui/widgets

Gautam007. (2021). Flutter for web: Best practices and performance tips. Medium. Recuperado de https://gautam007.medium.com/flutter-for-web-best-practices-and-performance-tips-3ec603d791f8

Stack Overflow. (2021). Handling overflow in Flutter web. Recuperado de https://stackoverflow.com/questions/66989012/handling-overflow-in-flutter-web

Flutter. (s.f.). State management options. Recuperado de https://docs.flutter.dev/data-and-backend/state-mgmt/options)

Flutter. (s.f.). Navigation and routing. Recuperado de https://docs.flutter.dev/ui/navigation

Flutter. (s.f.). Web renderers. Recuperado de https://docs.flutter.dev/platform-integration/web/renderers

Flutter. (s.f.). Flutter GitHub repository. Recuperado de https://github.com/flutter

ANNEX

A. Annex: Deployment to Github Pages

For simplicity, we will use a Github Action to automatically deploy our web every time we push on the main branch.

First, we need to enable Github pages. Go to the repository settings, Pages and select source Github Actions. The next step is to create a file named *.github/workflows/deploy_gh-page.yml* with this content:

```yaml
name: Deploy Flutter Web to GitHub Pages

on:
  push:
    branches:
      - main

permissions:
  contents: read
  pages: write
  id-token: write

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
      - name: Setup Flutter
        uses: subosito/flutter-action@v2
        with:
          channel: stable
          flutter-version: 3.27.3

      - name: Enable Flutter Web
        run: flutter config --enable-web

      - name: Install dependencies
        run: flutter pub get

      - name: Build Flutter Web
        run: flutter build web –release --base-href="/REPO_NAME/"

      - name: Setup GitHub Pages
        uses: actions/configure-pages@v3

      - name: Upload GitHub Pages artifact
        uses: actions/upload-pages-artifact@v3
        with:
          path: build/web

  deploy:
    runs-on: ubuntu-latest
    needs: build
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}

    steps:
      - name: Deploy to GitHub Pages
        id: deployment
        uses: actions/deploy-pages@v4
```

Commit to main branch and the web page will be automatically deployed every time you implement a new functionality.