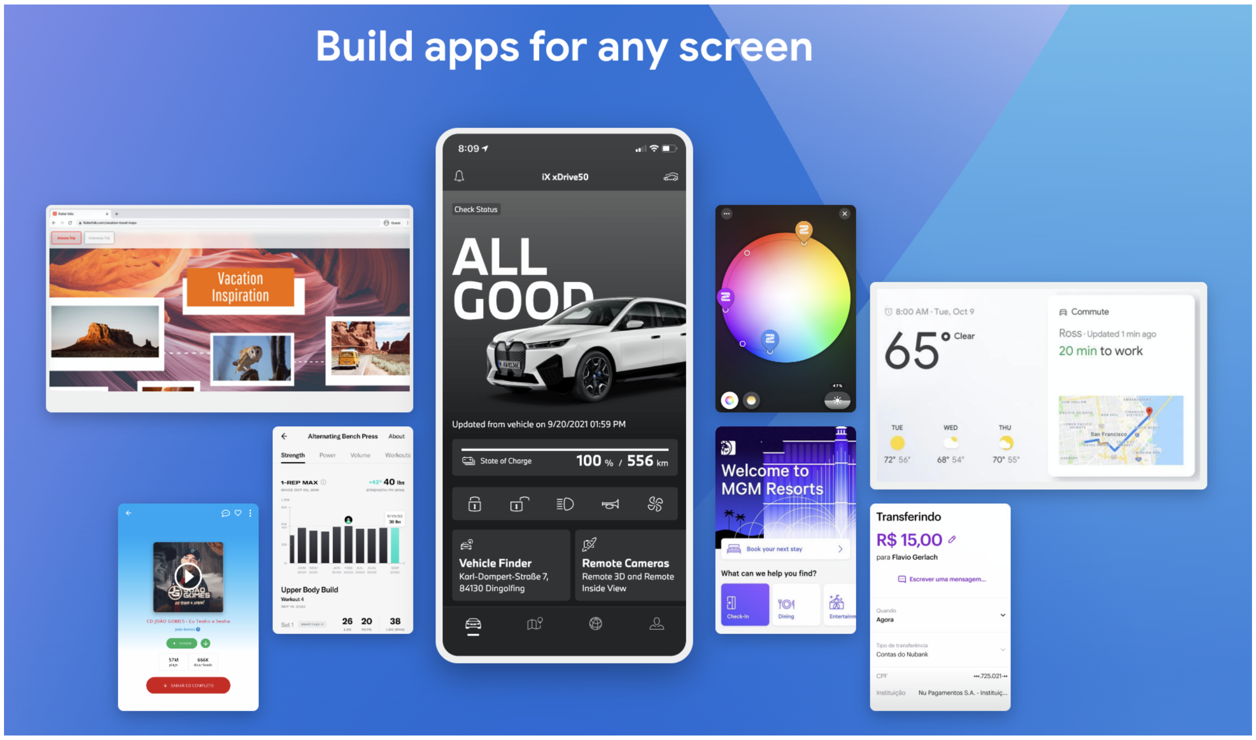
Flutter

Tags

- Introduction
- React native vs Flutter
- Learning steps
- Basic Flutter app
- Flutter components
- Build a Portfolio app
- Architecture patterns
- Bloc pattern
    - Bloc provider
- Web3 Flutter app
    - Add wallet button
    - Connect to blockchain
- Web3 libraries
- Flutter flow
- Flutter TODO list app
- Flutter + Backend
    - Connect to backend
- Flutter + Firebase auth
- Flutter + Firestore
- Complete Firestore functions
- Famous Flutter libraries
- Flutter UI libraries
- Official resources
- More resources
- Project ideas



Introduction

Flutter is an open-source mobile application development framework created by Google. It allows developers to create natively-compiled applications for mobile, web, and desktop from a single codebase. Some key points about Flutter include:

- Flutter uses the Dart programming language, which is also developed by Google and has a syntax similar to other popular languages like JavaScript and C#.
- Flutter includes a rich set of customizable widgets and tools for building beautiful and responsive user interfaces.
- Flutter has hot reload, which allows developers to see the effects of their changes in real-time, without having to manually stop and restart the app. This makes the development process faster and more efficient.
- Flutter has strong support for both iOS and Android, and can be used to build apps for both platforms with a single codebase.
- Flutter has a growing community of developers and contributors, with many plugins and packages available to extend the capabilities of the framework.

- Flutter is free and open-source, and is released under the BSD license. This means anyone can use it to build their own apps, and contribute to the framework if they wish.

React native vs Flutter

React Native and Flutter are both mobile application development frameworks, used for building natively-compiled apps for iOS and Android. However, there are some key differences between the two frameworks:

- Language: React Native uses JavaScript, a widely-used and popular programming language, while Flutter uses Dart, which is less commonly known but has a syntax similar to other C-style languages.
- User interface: React Native uses native components for building the user interface, which provides a more "native" look and feel for the app. Flutter, on the other hand, includes its own set of customizable widgets, which provides more flexibility and control over the app's design but may not always match the platform's native design.
- Performance: Both React Native and Flutter are known for their fast performance, but Flutter has an edge because it uses the Dart platform, which allows for faster compilation and better runtime performance than JavaScript.
- Community and ecosystem: React Native has a larger and more established community, with more third-party libraries and packages available. However, Flutter is growing quickly and has a strong and supportive community of its own.
- Development workflow: React Native has a more traditional development workflow, with a separation between the "native" code written in Java or Swift and the JavaScript code. Flutter, on the other hand, uses a single codebase for all platform-specific implementations.

Overall, both React Native and Flutter are powerful frameworks for building mobile apps, and the choice between the two will depend on the specific needs and preferences of the developer.

Learning steps

Here are some potential topics you could cover in a blog on learning Flutter:

1. Introduction to Flutter and its key features
2. Setting up your development environment for Flutter
3. Creating your first Flutter app and running it on an emulator or device
4. Understanding the Flutter widget system and how to use common widgets
5. Building beautiful and responsive user interfaces with Flutter
6. Adding interactivity to your Flutter app with gestures and animations
7. Working with data in Flutter, including using external APIs and persistence
8. Testing and debugging your Flutter app
9. Building and deploying your Flutter app for release
10. Advanced Flutter topics, such as using the Flutter framework to build for web, desktop, and other platforms.

Basic Flutter app

To create a simple Flutter app, you will need the following:

1. The Flutter SDK, which you can download from the official Flutter website (**https://flutter.dev/**).
2. An IDE or text editor, such as Android Studio or Visual Studio Code, to write and edit your Flutter code.
3. A device or emulator to run your app on, such as an Android emulator or an iOS simulator.

Once you have these tools installed, you can create a new Flutter project in your IDE and start writing code. Here is an example of a simple Flutter app that displays a button and a text message:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Text('Hello Flutter'),
              RaisedButton(
                child: Text('Click me'),
                onPressed: () {},
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

In this example, the `MyApp` class extends `StatelessWidget`, which means it does not have any internal state that can change. The `build` method returns a `MaterialApp` widget, which is the root of the app and provides a default app bar and a `Scaffold` widget, which provides a default white background and a body property. The `Scaffold`'s body is a `Center` widget, which centers its child widgets horizontally and vertically. The `Center` widget has a `Column` as its child, which arranges its children in a vertical list. The `Column` has two children: a `Text` widget that displays the text "Hello Flutter", and a `RaisedButton` widget that displays a button with the text "Click me".

When you run this app, you should see a screen with the text "Hello Flutter" and a button that you can click. This is just a simple example, but it shows the basics of how to create a Flutter app and use the widget system to build a user interface. For more information and examples, you can check out the Flutter documentation and tutorials on the official website.

Flutter components

Flutter includes a rich set of customizable and extensible widgets for building the user interface of a Flutter app. Some common UI components in Flutter include:

- `Text`: a widget for displaying text.
- `Image`: a widget for displaying images.

- `Icon`: a widget for displaying icons.
- `Button`: a widget for creating buttons.
- `TextField`: a widget for creating text input fields.
- `Checkbox`: a widget for creating checkboxes.
- `Radio`: a widget for creating radio buttons.
- `Slider`: a widget for creating sliders.
- `Switch`: a widget for creating switches.
- `ListView`: a widget for creating scrolling lists.
- `GridView`: a widget for creating grid-based layouts.
- `Card`: a widget for creating cards.
- `AppBar`: a widget for creating an app bar.
- `Drawer`: a widget for creating a navigation drawer.
- `BottomNavigationBar`: a widget for creating a bottom navigation bar.

These are just some examples of the many UI components available in Flutter. You can find a complete list of the built-in widgets in the Flutter documentation. In addition to the built-in widgets, Flutter also has a growing ecosystem of third-party libraries and packages that provide even more UI components and functionality.

Build a Portfolio app

create a new Flutter project in your IDE and start writing code. Here is an example of a simple portfolio app that has different sections for showcasing your skills, experience, and projects:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: SingleChildScrollView(
          child: Column(
            children: [
              ProfileSection(),
              SkillsSection(),
              ExperienceSection(),
              ProjectsSection(),
            ],
          ),
        ),
      ),
    );
  }
}

class ProfileSection extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(16),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          Text(
            'John Doe',
            style: Theme.of(context).textTheme.headline4,
          ),
          SizedBox(height: 8),
          Text(
            'Software Engineer',
            style: Theme.of(context).textTheme.subtitle1,
          ),
          SizedBox(height: 8),
          Text(
            'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas non arcu ac
enim varius finibus.',
            style: Theme.of(context).textTheme.bodyText1,
          ),
          SizedBox(height: 16),
        ....
        ....
}
```

Architecture patterns

Some of the most common architecture patterns in Flutter include:

1. The `BLoC` (Business Logic Component) pattern, which is a reactive and declarative way to manage the data flow in a Flutter app, using the `flutter_bloc` package.

2. The `Scoped Model` pattern, which is a simple and efficient way to manage shared state in a Flutter app, using the `scoped_model` package.
3. The `Provider` pattern, which is a reactive and dependency-injection-based way to manage the state and the dependencies of a Flutter app, using the `provider` package.
4. The `Redux` pattern, which is a predictable and unidirectional way to manage the state of a Flutter app, using the `flutter_redux` package.
5. The `MVVM` (Model-View-ViewModel) pattern, which is a separation-of-concerns and testability-focused way to structure a Flutter app, using the `flutter_mvvm` package.

Bloc pattern

The BLoC (Business Logic Component) pattern is a reactive and declarative way to manage the data flow in a Flutter app. It is based on the idea of separating the business logic of an app from its user interface, so that the same business logic can be used in different contexts and across multiple platforms.

In the BLoC pattern, the business logic of an app is implemented in a `Bloc` (Business Logic Component) class. This `Bloc` class acts as a mediator between the data source (such as a remote server or a local database) and the user interface of the app. It receives events from the user interface, such as user actions or data changes, and triggers the appropriate business logic. It also emits states, which represent the current state of the data and the UI, and can be used to update the user interface accordingly.

To implement the BLoC pattern in a Flutter app, you can use the `flutter_bloc` package, which provides a set of tools and abstractions for implementing a `Bloc` class and integrating it with your app. The `flutter_bloc` package also provides `BlocProvider` and `BlocBuilder` widgets, which can be used to provide and consume a `Bloc` instance in the widget tree of your app.

The BLoC pattern can help you write clean, maintainable, and testable code for your Flutter app, by providing a clear separation of concerns between the business logic and the user interface. It also enables you to reuse the same business logic in multiple contexts and platforms, such as web, desktop, and mobile.

Bloc provider

The `BlocProvider` is a widget from the `flutter_bloc` package that provides a `Bloc` instance to its descendants. A `Bloc` is a reactive and declarative way to manage the data flow in a Flutter app, using the `BLoC` (Business Logic Component) pattern.

The `BlocProvider` takes a `create` function as a parameter, which is used to create a new instance of the `Bloc` when it is accessed for the first time. This function is called only once and the returned `Bloc` instance is cached and reused for all subsequent accesses. The `BlocProvider` also has a `child` property, which is the widget tree that the `Bloc` instance will be provided to.

To use the `BlocProvider`, you first need to create a `Bloc` class that extends the `Bloc` base class from the `flutter_bloc` package. This `Bloc` class should define the state and the events that the `Bloc` can handle, as well as the `mapEventToState` method that specifies how the `Bloc` should respond to the events.

Once you have defined your `Bloc` class, you can use the `BlocProvider` to provide it to the widgets in your app. To access the `Bloc` instance from a descendant widget, you can use the `BlocProvider.of` method

Here is an example of how you can use the `flutter_bloc` package to implement the BLoC pattern in a Flutter app:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: BlocProvider(
          create: (context) => MyBloc(),
          child: MyWidget(),
        ),
      ),
    );
  }
}

class MyWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final myBloc = BlocProvider.of<MyBloc>(context);
    return BlocBuilder<MyBloc, MyState>(
      bloc: myBloc,
      builder: (context, state) {
        if (state is Loading) {
          return CircularProgressIndicator();
        }
        if (state is Loaded) {
          return ListView.builder(
            itemCount: state.items.length,
            itemBuilder: (context, index) {
              return Text(state.items[index].name);
            },
          );
        }
        return Container();
      },
    );
  }
}

class MyBloc extends Bloc<MyEvent, MyState> {
  @override
  MyState get initialState => Loading();

  @override
  Stream<MyState> mapEventToState(MyEvent event) async* {
    if (event is Load) {
      yield Loading();
      final items = await fetchItems();
      yield Loaded(items: items);
    }
    if (event is Add) {
      final items = List.from(state.items)..add(event.item);
      yield Loaded(items: items);
    }
  }
}

class MyState {
  final List<Item> items;
  MyState({this.items});
}

class Loading extends MyState {}

class Loaded extends MyState {
  Loaded({@required this.items}) : super(items: items);
}

class MyEvent {}

class Load extends MyEvent {}
class Add extends MyEvent {
  final Item item;
  Add({@required this.item}) : super();
}
```

Web3 Flutter app

Add wallet button

To add a "Connect Wallet" button to a simple Flutter app, you can use the `FlatButton` widget from the `material.dart` package. Here is an example of how you can use the `FlatButton` widget to create a "Connect Wallet" button and add it to your app:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Center(
          child: ConnectWalletButton(),
        ),
      ),
    );
  }
}

class ConnectWalletButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return FlatButton(
      child: Text('Connect Wallet'),
      onPressed: () {
        // TODO: Add code to connect the wallet
      },
    );
  }
}
```

In this example, the `ConnectWalletButton` widget extends the `FlatButton` widget and sets its `child` property to a `Text` widget with the label "Connect Wallet". It also sets its `onPressed` property to a callback function that will be called when the button is pressed.

Connect to blockchain

To connect a web3 wallet with the "Connect Wallet" button in a Flutter app, you will need to use a web3 library that provides the necessary APIs and functionalities for interacting with a web3 wallet provider. Some popular web3 libraries for Flutter include `web3dart`, `etherscan_api`, and `web3.dart`.

Once you have added the web3 library to your Flutter project, you can use it to connect to a web3 wallet provider and handle the authentication and the authorization of the user. Here is an example of how you can use the `web3dart` library to connect a web3 wallet with the "Connect Wallet" button in a Flutter app:

```
import 'package:flutter/material.dart';
import 'package:web3dart/web3dart.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Center(
          child: ConnectWalletButton(),
        ),
      ),
    );
  }
}

class ConnectWalletButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return FlatButton(
      child: Text('Connect Wallet'),
      onPressed: () async {
        // Connect to the wallet provider
        final httpClient = new Web3Client(
          'https://mainnet.infura.io/v3/your-api-key',
          http.Client(),
        );

        // Request the authorization of the user
        try {
          final credentials = await httpClient.credentialsFromPopup();

          // TODO: Add code to handle the credentials and access the wallet
        } catch (e) {
          // TODO: Add code to handle the error
        }
      },
    );
  }
}
```

In this example, the `ConnectWalletButton` widget uses the `web3dart` library to connect to a web3 wallet provider and request the authorization of the user. When the button is pressed, it calls the `credentialsFromPopup` method of the `Web3Client` class, which opens a popup window and prompts the user to authorize the app. If the authorization is successful, it returns the user's credentials, which can be used to access the wallet and perform web3 transactions. If the authorization fails, it throws an error, which can be caught and handled by the app.

Web3 libraries

There are several web3 libraries that can be used in Flutter apps to interact with a web3 wallet provider and perform web3 transactions. Some popular web3 libraries for Flutter include:

1. `web3dart`: This is a Dart library for interacting with Ethereum, which provides a high-level API for accessing Ethereum nodes, querying and mutating Ethereum state, and signing and sending Ethereum transactions.
2. `etherscan_api`: This is a Dart library for accessing the Etherscan API, which provides a simple and powerful way to access Ethereum blockchain data, such as blocks, transactions, addresses, and tokens.
3. `web3.dart`: This is a Dart library for accessing Ethereum JSON-RPC APIs, which provides a low-level API for calling Ethereum JSON-RPC methods and parsing the responses.

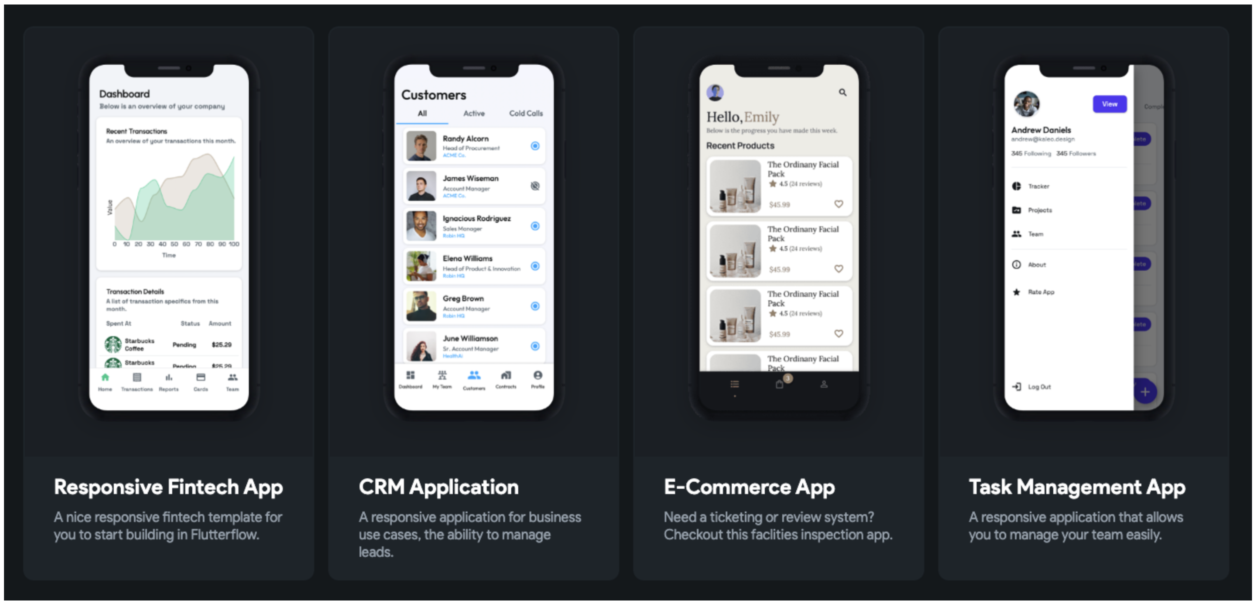
Flutter flow

Flutterflow is a visual development tool for creating Flutter apps, which allows you to design and build your app's user interface and business logic without writing any code.

To build a Flutter app with Flutterflow, you can use the Flutterflow designer to design your app's user interface by dragging and dropping pre-made widgets and customizing their properties. You can also use the Flutterflow editor to define the business logic of your app, using a simple and intuitive visual language.

Once you have designed and built your Flutter app with Flutterflow, you can export the generated Flutter code and run it on your device or emulator, just like any other Flutter app. Flutterflow also provides built-in support for hot reloading, testing, and debugging, so you can iterate on your app quickly and easily.

Overall, Flutterflow is a powerful and flexible tool for building Flutter apps, which can help you save time and effort, and focus on creating a great user experience.


Responsive Fintech App
A nice responsive fintech template for you to start building in Flutterflow.
CRM Application
A responsive application for business use cases, the ability to manage leads.
E-Commerce App
Need a ticketing or review system? Checkout this faclities inspection app.
Task Management App
A responsive application that allows you to manage your team easily.

Here is an example of how you can use Flutterflow to design and build a simple Flutter app:

1. Open Flutterflow and create a new project.
2. In the Flutterflow designer, drag and drop a `Text` widget and a `Button` widget onto the canvas.
3. Use the properties panel to customize the text and the style of the `Text` widget and the `Button` widget.
4. In the Flutterflow editor, create a new variable called `counter` and set its initial value to 0.

5. Create a new event called `incrementCounter` and set its action to increment the value of the `counter` variable by 1.
6. In the Flutterflow designer, select the `Button` widget and use the events panel to bind the `incrementCounter` event to the `onPressed` property of the `Button` widget.
7. In the Flutterflow designer, select the `Text` widget and use the bindings panel to bind the `counter` variable to the `text` property of the `Text` widget.
8. Export the generated Flutter code and run it on your device or emulator.

Flutter TODO list app

To make a simple to-do list Flutter app, you will need to use the `Flutter` and the `sqflite` packages, which provide the necessary APIs and functionalities for building a Flutter app and accessing a local SQLite database. The `Flutter` package provides the core Flutter functionality, such as building the user interface and handling the user input, and the `sqflite` package provides the specific SQLite functionality, such as creating and querying the database.

Here is an example of how you can make a simple to-do list Flutter app and access the local SQLite database:

- Install the `Flutter` and the `sqflite` packages by adding the following lines to your `pubspec.yaml` file:

```
dependencies:
  flutter:
    sdk: flutter
  sqflite: ^1.3.0
```

- Run `flutter packages get` in the terminal or command prompt to install the packages.
- Create a new Flutter app by running the following command in the terminal or command prompt:

```
flutter create todo_list
```

- In the `main.dart` file, import the `sqflite` and the `path_provider` packages and create a new class that extends the `Database` class and implements the `init` and the `todoDao` methods. The `init` method will initialize and open the database, and the `todoDao` method will return a new instance of the `TodoDao` class, which will provide the necessary APIs and functionalities for accessing the to-do list data in the database.

Here is an example of how you can complete the simple to-do list Flutter app and use it to add, mark as complete, and delete to-do items:

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

void main() => runApp(TodoListApp());

class TodoListApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Todo List',
      home: TodoListHome(),
    );
  }
}

class TodoListHome extends StatefulWidget {
  @override
  _TodoListHomeState createState() => _TodoListHomeState();
}

class _TodoListHomeState extends State<TodoListHome> {
  final TextEditingController _textController = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Todo List'),
      ),
      body: Column(
        children: [
          TodoList(),
          TextField(
            controller: _textController,
            decoration: InputDecoration(
              hintText: 'Add a todo item',
            ),
            onSubmitted: (text) async {
              final todoDao = TodoListDatabase.instance.todoDao;
              final todo = Todo(text: text);
              await todoDao.insertTodo(todo);
              _textController.clear();
            },
          ),
        ],
      ),
    );
  }
}

class TodoList extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final todoDao = TodoListDatabase.instance.todoDao;
    return StreamBuilder(
      stream: todoDao.watchAllTodos(),
      builder: (context, AsyncSnapshot<List<Todo>> snapshot) {
        final todos = snapshot.data ?? List();
        return ListView.builder(
          itemCount: todos.length,
          itemBuilder: (context, index) {
            final todo = todos[index];
            return TodoItem(
              todo: todo,
              onChanged: (completed) async {
                await todoDao.updateTodo(todo.copyWith(completed: completed));
              },
              onDismissed: (direction) async {
                await todoDao.deleteTodo(todo);
              },
            );
          },
        );
      },
    );
  }
}
```

In this example, the `TodoList` widget uses the `StreamBuilder` widget to listen to the changes in the to-do list data in the database and update the user interface in real-time. The `StreamBuilder` widget listens to the `watchAllTodos` stream of the `TodoDao` instance, which emits the updated list of `Todo` objects whenever the data changes in the database. The `ListView.builder` widget then uses the list of `Todo` objects to build the to-do items in the list.

Each `TodoItem` widget in the list represents a to-do item and displays its text and completion status. The `TodoItem` widget also allows the user to mark the to-do item as complete or incomplete, and to swipe the to-do item to dismiss it and delete it from the database. The `TodoItem` widget calls the `updateTodo` and the `deleteTodo` methods of the `TodoDao` instance to update and delete the to-do item data in the database.

You can customize the `TodoList` widget by modifying the parameters and the options of the `StreamBuilder`, the `ListView.builder`, and the `TodoItem` widgets, such as the data source, the item appearance, and the dismissal action. You can also add more advanced functionality to the `TodoList` widget, such as filtering and sorting the to-do items, and using transactions and batched writes.

Flutter + Backend

To create a simple Flask backend for a Flutter app, you will need to install Flask, a micro web framework for Python, and write some Flask code to handle the requests and the responses of your app. Flask allows you to create a web server and define the routes and the methods that handle the requests and the responses, using a simple and expressive syntax.

Here is an example of how you can create a simple Flask backend for a Flutter app:

1. Install Flask by running `pip install Flask` in a terminal or command prompt.
2. Create a new file called `app.py` and add the following code to it:

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

1. In the same directory, run the `app.py` file by running `python app.py` in the terminal or command prompt. This will start a local web server on your machine, listening on port 5000 by default.
2. Open a web browser and go to the URL `http://localhost:5000/`. You should see the message "Hello, World!" on the page.

3. In the `app.py` file, add more routes and methods to handle the requests and the responses of your app. For example, you can add a `POST` method to a `/login` route that receives a username and a password, and returns a token if the login is successful, or an error if the login fails.

Connect to backend

To connect the Flask backend to a Flutter app, you will need to use the `http` package, which provides a simple and efficient way to make HTTP requests and handle HTTP responses in Flutter. The `http` package allows you to send a request to the Flask backend, and receive and parse the response in your Flutter app's business logic or user interface.

Here is an example of how you can connect the Flask backend to a Flutter app using the `http` package:

```
=import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: Center(
          child: LoginButton(),
        ),
      ),
    );
  }
}

class LoginButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return FlatButton(
      child: Text('Login'),
      onPressed: () async {
        // Send a login request to the Flask backend
        final response = await http.post(
          'http://localhost:5000/login',
          body: {'username': 'john', 'password': 'doe'},
        );
```

```
          // Parse the response
          if (response.statusCode == 200) {
            final token = response.body;
            // TODO: Add code to handle the token
          } else {
            final error = response.body;
            // TODO: Add code to handle the error
          }
        },
      );
    }
  }
```

In this example, the `LoginButton` widget uses the `http` package to send a `POST` request to the `/login` route of the Flask backend.

Flutter + Firebase auth

To connect a Flutter app to Firebase, you will need to use the `firebase_core` and the `firebase_*` packages, which provide the necessary APIs and functionalities for integrating Firebase services into your Flutter app. The `firebase_core` package provides the core Firebase functionality, such as initializing and configuring Firebase, and the `firebase_*` packages provide the specific Firebase services, such as authentication, database, storage, and analytics.

Here is an example of how you can connect a Flutter app to Firebase and use the Firebase authentication service:

- Install the `firebase_core` and the `firebase_auth` packages by adding the following lines to your `pubspec.yaml` file:

```
dependencies:
  firebase_core: ^0.5.0
  firebase_auth: ^0.18.0
```

- Run `flutter packages get` in the terminal or command prompt to install the packages.
- Initialize and configure Firebase by calling the `initializeApp` method of the `Firebase` class and passing the necessary parameters, such as the project ID and the API key. You can find these parameters in the Firebase console.

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

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}
```

- In the widget tree of your app, wrap the root widget with the `FirebaseAuthProvider` widget, which will provide the `FirebaseAuth` instance to all the descendant widgets.

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: FirebaseAuthProvider(
          child: Center(
            child: LoginButton(),
          ),
        ),
      ),
    );
  }
}
```

- In the `LoginButton` widget, use the `FirebaseAuth` instance to sign in with an email and a password.

```
class LoginButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final auth = FirebaseAuth.of(context);
    return FlatButton(
      child: Text('Login'),
      onPressed: () async {
        try {
          final result = await auth.signInWithEmailAndPassword(
            email: 'john.doe@example.com',
            password: 'password',
          );
          final user = result.user;
          // TODO: Add code to handle the authenticated user
        } catch (e) {
          // TODO: Add code to handle the error
        }
      },
    );
  }
}
```

In this example, the `LoginButton` widget uses the `FirebaseAuth` instance to sign in with an email and a password by calling the `signInWithEmailAndPassword` method. If the sign in is successful, it returns a `UserCredential` object, which contains the authenticated `User` object. You can use the `User` object to access the user's data, such as the email, the display name, and the photo URL. If the sign in fails, it throws an error, which can be caught and handled by the app.

You can customize the sign in flow by modifying the parameters and the options of the `signInWithEmailAndPassword` method, such as the email verification, the password reset, and the authentication persistence. You can also add more advanced functionality to the `LoginButton` widget, such as displaying the user's data and enabling the user to sign out.

Flutter + Firestore

To connect Firestore to a simple Flutter app, you will need to use the `firebase_core` and the `cloud_firestore` packages, which provide the necessary APIs and functionalities for integrating Firestore into your Flutter app. The `firebase_core` package provides the core Firebase functionality, such as initializing and configuring Firebase, and the `cloud_firestore` package provides the specific Firestore functionality, such as accessing and querying the Firestore database.

Here is an example of how you can connect Firestore to a simple Flutter app and query the Firestore database:

1. Install the `firebase_core` and the `cloud_firestore` packages by adding the following lines to your `pubspec.yaml` file:

```
dependencies:
  firebase_core: ^0.5.0
  cloud_firestore: ^0.14.0
```

1. Run `flutter packages get` in the terminal or command prompt to install the packages.

2. Initialize and configure Firebase by calling the `initializeApp` method of the `Firebase` class and passing the necessary parameters, such as the project ID and the API key. You can find these parameters in the Firebase console.

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

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}
```

1. In the widget tree of your app, wrap the root widget with the `FirestoreProvider` widget, which will provide the `Firestore` instance to all the descendant widgets.

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: FirestoreProvider(
          child: Center(
            child: QueryButton(),
          ),
        ),
      ),
    );
  }
}
```

Here is an example of how you can complete the steps and use Firestore to query the database in a simple Flutter app:

```
class QueryButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final firestore = Firestore.of(context);
    return FlatButton(
      child: Text('Query'),
      onPressed: () async {
        final snapshot = await firestore
            .collection('users')
            .where('name', isEqualTo: 'John Doe')
            .getDocuments();
        final documents = snapshot.documents;
        // TODO: Add code to handle the documents
      },
    );
  }
}
```

In this example, the `QueryButton` widget uses the `Firestore` instance to query the `users` collection in the Firestore database by calling the `where` and the `getDocuments` methods. The query returns a `QuerySnapshot` object, which contains the `DocumentSnapshot` objects that match the query criteria. You can use the `DocumentSnapshot` objects to access the data of the documents, such as the name, the email, and the age. You can also use the `StreamBuilder` widget to listen to the changes in the query results and update the app's user interface in real-time.

You can customize the query by modifying the parameters and the options of the `where` and the `getDocuments` methods, such as the order, the limit, and the pagination. You can also add more advanced functionality to the `QueryButton` widget, such as updating and deleting the documents, and using transactions and batched writes.

Complete Firestore functions

Here is an example of how you can use the Firestore functions separately in a simple Flutter app:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: FirestoreProvider(
          child: Center(
            child: Column(
              children: [
                InsertButton(),
                UpdateButton(),
                DeleteButton(),
              ],
            ),
          ),
        ),
      ),
    );
  }
}

class InsertButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final firestore = Firestore.of(context);
    return FlatButton(
      child: Text('Insert'),
      onPressed: () async {
        final document = firestore.collection('users').document();
        await document.setData({
          'name': 'John Doe',
          'email': 'john.doe@example.com',
          'age': 30,
        });
        // TODO: Add code to handle the inserted document
      },
    );
  }
}

class UpdateButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final firestore = Firestore.of(context);
    return FlatButton(
      child: Text('Update'),
      onPressed: () async {
        final document = firestore.collection('users').document('user1');
        await document.updateData({'age': 31});
        // TODO: Add code to handle the updated document
      },
    );
  }
}

class DeleteButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final firestore = Firestore.of(context);
    return FlatButton(
      child: Text('Delete'),
      onPressed: () async {
        final document = firestore.collection('users').document('user1');
        await document.delete();
        // TODO: Add code to handle the deleted document
      },
    );
  }
}
```

Famous Flutter libraries

Flutter has a growing ecosystem of third-party libraries and packages that provide additional functionality and UI components. Some popular Flutter libraries include:

- `flutter_blue`: a Bluetooth plugin for Flutter, which allows apps to communicate with Bluetooth Low Energy (BLE) devices.
- `sqflite`: a SQLite plugin for Flutter, which allows apps to store and retrieve data locally using SQLite databases.
- `flutter_map`: a plugin for integrating map views into Flutter apps, using popular map providers such as Google Maps and OpenStreetMap.
- `url_launcher`: a plugin for launching URLs in the default browser or a webview from a Flutter app.
- `flutter_svg`: a plugin for rendering SVG images and displaying them in a Flutter app.
- `flutter_calendar`: a plugin for creating calendar views in a Flutter app.
- `flutter_local_notifications`: a plugin for displaying and scheduling local notifications in a Flutter app.
- `flutter_audio_recorder`: a plugin for recording audio in a Flutter app.
- `flutter_charts`: a plugin for creating charts and graphs in a Flutter app.
- `flutter_i18n`: a plugin for internationalizing and localizing a Flutter app.

Flutter UI libraries

Flutter has a growing ecosystem of third-party libraries and packages that provide additional UI components and functionality. Some popular Flutter UI libraries include:

- `flutter_material_color_picker`: a library for creating a color picker using Material Design components.
- `flutter_cupertino_date_picker`: a library for creating a date picker using Cupertino (iOS-style) components.
- `flutter_sticky_header`: a library for creating sticky headers that can be scrolled with a list of items.

- `flutter_slidable`: a library for creating slidable list items that can be dragged to reveal hidden actions.
- `flutter_swipe_action_cell`: a library for creating swipeable list items that can be swiped to reveal hidden actions.
- `flutter_staggered_grid_view`: a library for creating grid-based layouts with support for custom cross-axis ratios and animation effects.
- `flutter_rounded_progress_bar`: a library for creating rounded progress bars with customizable colors and labels.
- `flutter_circular_chart`: a library for creating circular charts and gauges, such as pie charts, doughnut charts, and speedometers.
- `flutter_speed_dial`: a library for creating a speed dial (floating action button) with multiple actions.
- `flutter_form_builder`: a library for creating and validating forms with customizable fields and validation rules.

Official resources

- The official Flutter website (**https://flutter.dev/**) is a great place to start, with documentation, tutorials, and examples for learning Flutter.
- The Flutter documentation (**https://flutter.dev/docs**) is a comprehensive resource for learning the fundamentals of Flutter and its APIs.
- The Flutter Codelabs (**https://codelabs.developers.google.com/flutter/**) are a series of step-by-step tutorials for learning Flutter through hands-on examples.
- The Flutter tutorials on the official Flutter YouTube channel (**https://www.youtube.com/flutterdev**) are a collection of videos that cover various topics in Flutter, from the basics to advanced topics.
- The Flutter community on Stack Overflow (**https://stackoverflow.com/questions/tagged/flutter**) is a great resource for getting answers to your questions and learning from other developers who are using Flutter.
- The FlutterDev subreddit (**https://www.reddit.com/r/FlutterDev/**) is a community forum for discussing Flutter and sharing tips, tricks, and resources.

- The Flutter Awesome list (**https://flutterawesome.com/**) is a curated list of Flutter libraries, packages, and resources, maintained by the Flutter community.

More resources

Here are five YouTube videos that provide comprehensive resources for learning Flutter:

1. "Flutter Crash Course" by Traversy Media (**https://www.youtube.com/watch?v=d_m5cSmrf7E**) is a fast-paced and comprehensive introduction to Flutter, covering the basics of the framework and its key features.
2. "Flutter in 7 Days" by Reso Coder (**https://www.youtube.com/watch?v=Hm3JfKG3Mh4**) is a seven-part series that covers the fundamentals of Flutter in a week, with one lesson per day.
3. "Building a Complete Flutter App from Scratch" by The Net Ninja (**https://www.youtube.com/watch?v=7Yc3PjJDFpE**) is a step-by-step tutorial for building a complete Flutter app from start to finish, covering topics such as navigation, database storage, and user authentication.
4. "Flutter App Development: A Complete Guide" by freeCodeCamp.org (**https://www.youtube.com/watch?v=zT62eVxShsY**) is a comprehensive course that covers the fundamentals of Flutter app development, including the Dart language, the Flutter widget system, and advanced topics such as testing and deployment.
5. "Flutter Weekly" by Flutter (**https://www.youtube.com/watch?v=2uOgDYKdGc4**) is a weekly video series that provides updates on the latest developments in the Flutter community, including new features, libraries, and resources.

Project ideas

1. A simple to-do list app that allows the user to add, mark as complete, and delete tasks. The app can also group the tasks by date and display them in a list or a calendar view.
2. A weather app that shows the current and the forecast weather of the user's location or any other location. The app can use the device's GPS or the user's input

to get the location, and the OpenWeatherMap API or any other weather API to get the weather data.

3. A calculator app that allows the user to perform basic and advanced calculations, such as addition, subtraction, multiplication, division, square root, and trigonometric functions. The app can also save the calculation history and display it in a list or a graph.
4. A chat app that allows the user to communicate with other users in real-time. The app can use the Firebase Realtime Database or any other real-time database to store and retrieve the messages, and the Firebase Authentication or any other authentication service to verify the user's identity.
5. A recipe app that displays a collection of recipes with their ingredients, instructions, and nutritional information. The app can use the Firebase Cloud Firestore or any other database to store and retrieve the recipes, and the Firebase Storage or any other storage service to store and retrieve the recipe images.
6. A fitness app that tracks the user's physical activity, such as steps, distance, calories, and heart rate. The app can use the device's sensors or the user's input to get the activity data, and the Google Fit API or any other fitness API to store and retrieve the activity history.
7. A social media app that allows the user to create and share posts, photos, videos, and links with their friends and followers. The app can use the Firebase Realtime Database or any other real-time database to store and retrieve the posts, and the Firebase Storage or any other storage service to store and retrieve the post media.