Flutter Course

This is a just a study guide or a guide when programming in Flutter. It has only the links to specific material. It does not intend to be a manual on Flutter.

Some Flutter resources:

a. Overview of Flutter - The official homepage for the Flutter framework, providing a high-level introduction, showcases, and news.

b. the main documentation - The landing page for Flutter's official documentation, the most critical resource for any developer. In particular, there is a page that is a cook book - A collection of practical, recipe-style guides for solving common problems and implementing specific features.

c. Comunidade Flutter do Brasil - The main hub for the Brazilian Flutter community, offering articles, tutorials, and events in Portuguese.

d. Heyflutter - A popular YouTube channel with concise, high-quality tutorials on a wide range of Flutter widgets and concepts.

e. another good one is Flutter Mapp. There is a YT video with hundreds of widgets. - A massive video that serves as a visual encyclopedia of Flutter widgets, with timestamps for easy navigation. The code is here - The corresponding GitHub repository containing the source code for all the widgets demonstrated in the video.

f. My own YT video list on Flutter - A curated YouTube playlist of useful Flutter-related videos.

g. Flutter gems is a repository of curated Dart and Flutter packages - An essential website for discovering the best and most popular Flutter packages, categorized for easy searching.

h. Flutter awesome webpage - A curated list of articles, libraries, projects, and other resources related to Flutter.

i. Flutter tips and tricks - A very valuable GitHub repository containing a large collection of practical code snippets and advanced techniques.

As any other SDK for Graphical User Interface, Flutter has its cons and pros:

a. Why Use Flutter: Pros and Cons of Flutter App Development - An article from a software development company analyzing the advantages and disadvantages of choosing Flutter for a project.

b. Flutter pros and cons summary - A blog post offering a concise summary of the key strengths and weaknesses of the Flutter framework.

c. Benefits of Flutter - An article focusing on the positive aspects and business advantages of developing applications with Flutter.

Installation

First of all, install Flutter. The Dart SDK will be installed too. Follow the steps below.

a. Install VSCode. - The official download page for the Visual Studio Code editor, a popular choice for Flutter development.

b. Setting up the Flutter environment. - The official, step-by-step guide from the Flutter team on how to install the SDK on your operating system.

c. Install the following extensions for VSCode. They are optional, but very useful. *GitHub Copilot, GitHub Copilot Chat, Bookmarks, Bracket Pair Color DLW, GitHub Pull Request, GitHub Repositories, Image Preview, Pubspec Assist, Remote Repositories.*

Introduction to Dart

See this page for introduction to Dart. - The main entry point to the Dart language tour, covering all fundamental concepts. Topics:

Beginner Topics

1. Basic Syntax

```dart
// The main() function is the entry point for all Dart applications.
// The code inside its curly braces {} is executed when the program starts.
void main() {
  /*
    This is a multi-line comment. It can span several lines
    and is useful for longer explanations.
  */

  // A statement is a single instruction that performs an action.
  // The following statement calls the print function to display text in the console.
  // Every statement in Dart must end with a semicolon (;).
  print('Hello, Dart! This is a statement.');
}
```

---

2. Variables & Data Types

```dart
void main() {
  // `var` infers the type from the initial value. Here, `age` is inferred as an `int`.
  var age = 30; // int: for whole numbers.

  // `String`: for text. Must be enclosed in single or double quotes.
```

```dart
  String name = 'Alice';

  // `double`: for floating-point or decimal numbers.
  double height = 1.75;

  // `bool`: for true/false values.
  bool isStudent = true;

  // `final`: for variables that are assigned once and never change (runtime constant).
  final String university = 'Dart University';

  // `const`: for variables whose value is known at compile-time (compile-time constant).
  const double PI = 3.14159;

  // `dynamic`: for variables that can hold values of any type. The type can change at runt
  dynamic flexibleVar = 'I am a string';
  flexibleVar = 100; // Now it's an integer.

  print(
      'Name: $name, Age: $age, Height: $height, Is Student: $isStudent, University: $univers
}
```

---

5. Operators

```dart
void main() {
  int a = 10;
  int b = 3;

  // Arithmetic Operators: perform mathematical calculations.
  print('Arithmetic: ${a + b}, ${a - b}, ${a * b}, ${a / b}, ${a % b}');

  // Relational Operators: compare values and return a boolean.
  bool areEqual = (a == b); // false
  print('Relational (a == b): $areEqual');

  // Logical Operators: combine boolean expressions.
  bool condition1 = true;
  bool condition2 = false;
  print('Logical (AND, OR, NOT): ${condition1 && condition2}, ${condition1 || condition2}, $

  // Assignment Operators: assign values to variables.
  double c = 5;
  c += 2.5; // Equivalent to c = c + 2.5;
  print('Assignment (c += 2.5): $c');
```

```dart
  // Null-aware Operators: for working with nullable variables.
  String? nullableText; // This variable can be null.

  // `?.` (Null-aware access): Access a property only if the object is not null. Otherwise,
  int? length = nullableText?.length;
  print('Null-aware access (?.): $length'); // Prints: null

  // `??` (Null-coalescing operator): Provide a default value if the expression on the left
  String nonNullText = nullableText ?? 'Default Text';
  print('Null-coalescing operator (??): $nonNullText');
}
```

---

6. Control Flow

```dart
void main() {
  int score = 85;

  // `if-else` statement: executes code based on a boolean condition.
  if (score > 90) {
    print('Grade: A');
  } else if (score > 80) {
    print('Grade: B');
  } else {
    print('Grade: C or lower');
  }

  // `switch-case` statement: compares a value against several possible cases.
  String fruit = 'apple';
  switch (fruit) {
    case 'apple':
      print('It is a red fruit.');
      break;
    case 'banana':
      print('It is a yellow fruit.');
      break;
    default:
      print('Unknown fruit.');
  }

  // `for` loop: executes code a specific number of times.
  print('For loop:');
  for (int i = 1; i <= 3; i++) {
    print('Count: $i');
```

```
  }

  // `while` loop: executes code as long as a condition is true.
  print('While loop:');
  int countdown = 3;
  while (countdown > 0) {
    print(countdown);
    countdown--;
  }

  // `do-while` loop: similar to while, but the body is executed at least once.
  print('Do-while loop:');
  int number = 5;
  do {
    print('This will print once, number is $number');
  } while (number < 5);
}
```

---

7. Collections

```
void main() {
  // `List`: An ordered collection of items. Can contain duplicates.
  List<int> numbers = [1, 2, 3, 3, 4];
  print('List: $numbers');

  // `Set`: An unordered collection of unique items. Duplicates are automatically removed.
  Set<String> fruits = {'apple', 'banana', 'orange', 'apple'};
  print('Set: $fruits');

  // `Map`: A collection of key-value pairs. Keys must be unique.
  Map<String, dynamic> person = {
    'name': 'John Doe',
    'age': 28,
    'isEmployed': true
  };
  print('Map: $person');
  print('Accessing map value: ${person['name']}');

  // Iteration with `forEach`: Performs an action for each element in the collection.
  print('Iterating over list with forEach:');
  numbers.forEach((number) {
    print('Number: $number');
  });
```

```
  // Iteration with `map`: Transforms each element into a new value, creating a new collect
  var squaredNumbers = numbers.map((number) => number * number).toList();
  print('List transformed with map (squared): $squaredNumbers');
}
```

---

8. Functions

```
// A standard function with a return type (`int`) and two required positional parameters.
int add(int a, int b) {
  return a + b;
}

// A function using arrow syntax (`=>`) for a concise, single-expression body.
// It has one required parameter and one optional positional parameter `[int factor = 2]`.
int multiply(int a, [int factor = 2]) => a * factor;

// A function with no return type (`void`) that uses named parameters.
// `name` is required, while `greeting` has a default value.
void greet({required String name, String greeting = 'Hello'}) {
  print('$greeting, $name!');
}

void main() {
  // Calling the functions defined above.
  print('add(5, 3) = ${add(5, 3)}');
  print('multiply(5) = ${multiply(5)}'); // Uses default factor
  print('multiply(5, 3) = ${multiply(5, 3)}'); // Provides optional factor
  greet(name: 'World'); // Provides only the required named parameter
}
```

---

9. Null Safety

```
// This function might return a String or null. The `?` indicates it's nullable.
String? findUser(int id) {
  if (id == 1) {
    return 'Alice';
  }
  return null; // Return null if user not found
}

// A class that demonstrates late initialization.
class Profile {
  // `late` promises that this non-nullable variable will be initialized before it's ever r
```

```
  late String username;

  void loadUsername() {
    // Simulates fetching a username from a database.
    username = 'SuperUser123';
  }
}

void main() {
  // `?` declares a nullable variable. `user` can hold a String or null.
  String? user = findUser(2);

  // We must check for null before using a nullable variable's properties.
  if (user != null) {
    print('User found: ${user.toUpperCase()}');
  } else {
    print('User not found.');
  }

  // `!` (bang operator): Asserts that a nullable variable is not null at this point.
  // Use with caution! This will crash if the variable is null.
  String realUser = findUser(1)!;
  print('Found user with ! operator: $realUser');

  // Example of using a `late` variable.
  var profile = Profile();
  profile.loadUsername(); // `username` is initialized here.
  print('Late initialized username: ${profile.username}'); // It's now safe to read.
}
```

---

10. Object-Oriented Programming (OOP)

```
class User {
  // Public property.
  String name;
  // Private property (indicated by the leading underscore).
  int _age;

  // This is a constructor, a special method for creating an object (instance of the class).
  // `this.name` and `this._age` assign the constructor arguments to the object's properties
  User(this.name, this._age);

  // A getter provides read-only access to a property.
  // It allows us to expose the private _age value without letting external code change it d
```

```dart
  int get age {
    return _age;
  }

  // A setter provides controlled write access to a property.
  // It allows us to add logic, like validation, before changing the value.
  set age(int value) {
    if (value >= 0) {
      _age = value;
    } else {
      print('Age cannot be negative.');
    }
  }

  void describe() {
    print('User: $name, Age: $_age');
  }
}

void main() {
  // Creating an object (an instance of the User class).
  var user1 = User('Alice', 30);

  // Using the getter to read the age.
  print('${user1.name} is ${user1.age} years old.');

  // Using the setter to change the age.
  user1.age = 31;
  user1.describe(); // Prints: User: Alice, Age: 31

  user1.age = -5; // Tries to set an invalid age.
  user1.describe(); // Prints: User: Alice, Age: 31 (age did not change)
}
```

Dart constructors - A Medium article offering a deep dive into how constructors work in Dart, including named, factory, and const constructors.

---

11. Inheritance & Polymorphism

```dart
// A base class.
class Animal {
  String name;
  Animal(this.name);

  void makeSound() {
```

```
    print('Some generic animal sound');
  }
}

// An interface (defined using an abstract class).
abstract class CanFly {
  void fly();
}

// A subclass that `extends` Animal. It inherits properties and methods.
class Dog extends Animal {
  // The constructor calls the parent's constructor using `super`.
  Dog(String name) : super(name);

  // Method overriding: providing a specific implementation for a method from the parent cl
  @override
  void makeSound() {
    print('Woof!');
  }
}

// A class that both extends a base class and `implements` an interface.
class Bird extends Animal implements CanFly {
  Bird(String name) : super(name);

  @override
  void makeSound() {
    print('Chirp!');
  }

  // Must provide an implementation for the `fly` method from the `CanFly` interface.
  @override
  void fly() {
    print('$name is flying.');
  }
}

void main() {
  // Polymorphism: Objects of different classes (Dog, Bird) can be treated as objects of a
  List<Animal> animals = [Dog('Buddy'), Bird('Sparrow')];

  for (var animal in animals) {
    print('Animal: ${animal.name}');
    animal.makeSound(); // Calls the overridden method specific to each object's actual typ

    // Check if the animal can fly before calling the fly method.
```

```
    if (animal is CanFly) {
      (animal as CanFly).fly();
    }
  }
}
```

---

12. Mixins & Abstract Classes

```
// An `abstract class` defines a contract but cannot be instantiated itself.
// It can have abstract methods (without a body) and concrete methods/properties.
abstract class Vehicle {
  int wheels;
  Vehicle(this.wheels);

  // An abstract method must be implemented by any concrete subclass.
  void move();

  void describe() {
    print('This vehicle has $wheels wheels.');
  }
}

// A `mixin` is a way to reuse a class's code in multiple class hierarchies.
// It adds functionality to a class.
mixin HasEngine {
  bool _isEngineOn = false;

  void startEngine() {
    _isEngineOn = true;
    print('Engine started.');
  }
}

// `extends` creates an "is-a" relationship (a Car is a Vehicle). It inherits implementatio
// `implements` creates a "can-do" relationship, forcing the class to re-implement the inte
// `with` applies a mixin to a class, "mixing in" its methods and properties.
class Car extends Vehicle with HasEngine {
  Car() : super(4);

  // Must implement the abstract `move` method from Vehicle.
  @override
  void move() {
    if (_isEngineOn) {
      print('The car is moving on its wheels.');
```

```
    } else {
      print('Start the engine first!');
    }
  }
}

void main() {
  var myCar = Car();
  myCar.describe();
  myCar.move(); // Engine is off
  myCar.startEngine(); // Using method from the `HasEngine` mixin
  myCar.move(); // Engine is on
}
```

---

13. Error Handling

```
// A custom exception class to represent a specific type of error.
class NegativeValueException implements Exception {
  final String message;
  NegativeValueException(this.message);

  @override
  String toString() => 'NegativeValueException: $message';
}

// A function that might fail and `throw` an exception.
double divide(int a, int b) {
  if (b == 0) {
    // `throw` is used to signal an error condition.
    throw IntegerDivisionByZeroException();
  }
  if (a < 0 || b < 0) {
    throw NegativeValueException('Cannot divide negative numbers.');
  }
  return a / b;
}

void main() {
  // The `try-catch` block is used to handle potential exceptions.
  try {
    // The code that might throw an exception is placed in the `try` block.
    print('Trying to divide 10 by -2...');
    var result = divide(10, -2);
    print('Result: $result'); // This line won't be reached.
```

```
  } on NegativeValueException catch (e) {
    // An `on...catch` block handles a specific type of exception.
    print('Caught a specific error: $e');
  } catch (e) {
    // A generic `catch` block handles any other type of exception.
    print('Caught an unknown error: $e');
  } finally {
    // The `finally` block contains code that is always executed, whether an exception occu
    print('This block always runs.');
  }
}
```

---

16. Libraries & Imports

```
// The `import` keyword is used to make a library's features available in the current file.
// `dart:math` is a standard library that comes with the Dart SDK, providing math constants
import 'dart:math';

void main() {
  // Using the `sqrt` function from the math library to calculate a square root.
  double number = 16;
  double squareRoot = sqrt(number);
  print('The square root of $number is $squareRoot.');

  // Using the `pi` constant from the math library.
  double radius = 5;
  double area = pi * pow(radius, 2);
  print('The area of a circle with radius $radius is $area.');

  // Using the `Random` class from the math library to generate a random number.
  var random = Random();
  int randomNumber = random.nextInt(100); // A random integer between 0 and 99.
  print('A random number: $randomNumber');
}
```

---

17. Generics

```
// A generic function `getFirst<T>` can work with any type `T`.
// It takes a list of type `T` and returns an element of type `T`.
T getFirst<T>(List<T> items) {
  if (items.isEmpty) {
    throw Exception('List cannot be empty.');
  }
```

```dart
  return items.first;
}

// A generic class `Cache<T>` can store a value of any type `T`.
// `Future<T>` is another common generic class for asynchronous operations.
class Cache<T> {
  T? _value;

  void setValue(T value) {
    _value = value;
  }

  T? getValue() {
    return _value;
  }
}

void main() {
  // Using the generic function with a List<int>.
  var numbers = [10, 20, 30];
  print('First number: ${getFirst<int>(numbers)}');

  // Using the same generic function with a List<String>.
  var names = ['Alice', 'Bob', 'Charlie'];
  print('First name: ${getFirst<String>(names)}');

  // Using the generic class to create a cache for a String.
  var nameCache = Cache<String>();
  nameCache.setValue('John Doe');
  print('Cached name: ${nameCache.getValue()}');

  // Using the generic class to create a cache for a double.
  var piCache = Cache<double>();
  piCache.setValue(3.14159);
  print('Cached pi value: ${piCache.getValue()}');
}
```

18. Isolates, Concurrency
19. Metaprogramming

A Deep Dive - An article explaining the concepts and capabilities of reflection in the Dart language.

Dart Object Description Using Reflection - A specific code example of using reflection to dynamically describe an object's properties.

It is possible to evaluate a `main` function in Dart at runtime. See the article

How to Eval in Dart - A blog post demonstrating a technique for dynamically executing Dart code at runtime.

Flutter Basics

Any doubts, go to the Flutter documentation.

A Basic Flutter App

In a directory, type

```
Flutter create myfirstapp
```

It will create a directory `myfirstapp` with a complete example of a Flutter app.

pubspec.yaml

The command "flutter create" creates a file named `pubspec.yaml` that specifies the dependencies and settings of the app project.

"Dependencies specify other packages that your package needs in order to work".

There are differences between dependencies and dev_dependencies. - An article clarifying the distinction between runtime dependencies (needed for the app to run) and development dependencies (only needed for testing and development).

"To automatically upgrade your package dependencies to the latest versions consider running 'flutter pub upgrade –major-versions'".

Basic Widgets

MaterialApp - The root widget that sets up a Material Design application, providing theming, navigation, and other core functionalities.

```
// The starting point for most Flutter apps.
MaterialApp(
  title: 'My Flutter App',
  theme: ThemeData(primarySwatch: Colors.blue),
  home: MyHomePage(),
);
```

Scaffold - A fundamental layout widget that implements the basic Material Design visual layout structure, including app bars, drawers, and floating action buttons.

```
// Provides the basic structure for a visual screen.
Scaffold(
  appBar: AppBar(title: Text('My App')),
  body: Center(child: Text('Hello!')),
```

```
  floatingActionButton: FloatingActionButton(onPressed: () {}),
);
```

Text - The basic widget for displaying a string of text with a single, uniform style.

```
// Displays text with custom styling.
Text(
  'Hello, styled text!',
  style: TextStyle(
    fontSize: 24,
    fontWeight: FontWeight.bold,
    color: Colors.red,
  ),
);
```

Center - A layout widget that centers its child widget within itself.

```
// Centers its child widget.
Center(
  child: Text('I am in the center.'),
);
```

Container - A powerful and versatile widget for combining painting, positioning, and sizing of its child.

```
// A decorated box with padding and a child.
Container(
  padding: const EdgeInsets.all(16.0),
  margin: const EdgeInsets.all(10.0),
  decoration: BoxDecoration(
    color: Colors.blue[100],
    borderRadius: BorderRadius.circular(8),
  ),
  child: Text('I am in a container'),
);
```

Column - A layout widget that arranges its children in a vertical array.

```
// Arranges widgets vertically.
Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
    Icon(Icons.star),
    Text('Item 1'),
    Text('Item 2'),
  ],
);
```

Row - A layout widget that arranges its children in a horizontal array.

```
// Arranges widgets horizontally.
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: <Widget>[
    Icon(Icons.thumb_up),
    Text('Like'),
    Icon(Icons.comment),
    Text('Comment'),
  ],
);
```

AppBar - The Material Design app bar, typically displayed at the top of a `Scaffold`.

```
// The bar at the top of the screen.
AppBar(
  title: Text('My Page Title'),
  actions: [
    IconButton(
      icon: Icon(Icons.search),
      onPressed: () { /* Perform search */ },
    ),
  ],
);
```

ElevatedButton - A Material Design button with a raised appearance, indicating a primary action.

```
// A standard raised button.
ElevatedButton(
  onPressed: () { /* Handle button press */ },
  child: Text('Click Me'),
);
```

Icon - A widget that displays a graphical icon. Some websites with list of icons are: icons from Google fonts, icon finder, icons from fluttergems.

```
// A graphical icon.
Icon(
  Icons.favorite,
  color: Colors.pink,
  size: 30.0,
);
```

Image - A widget for displaying images, either from assets, memory, or the network. Free images sites here. - A list of websites that offer royalty-free stock images.

```
// Display an image from project assets.
Image.asset('assets/images/my_image.png');
```

```dart
// Display an image from the internet.
Image.network('https://picsum.photos/250?image=9');
```

NetworkImage - An object used with the `Image` widget to display an image fetched from a URL.

```dart
// Use NetworkImage for more control, e.g., in a decoration.
Container(
  width: 100,
  height: 100,
  decoration: BoxDecoration(
    image: DecorationImage(
      image: NetworkImage('https://picsum.photos/250?image=9'),
      fit: BoxFit.cover,
    ),
  ),
);
```

Stateless vs. Stateful Widgets

Stateless and stateful widgets - The official documentation explaining the fundamental difference between widgets that can change over time (Stateful) and those that cannot (Stateless). For short, stateless widgets are used when there is no interactivity and stateful widgets when the user actions (like the press of a button) can change something.

Layouts in Flutter

The following text is from the page Layout fundamentals - The core documentation page explaining Flutter's unique layout model.

In the simplest example, the layout conversation looks like this:

1. A widget receives its constraints from its parent.
2. A constraint is just a set of 4 doubles: a minimum and maximum width, and a minimum and maximum height.
3. The widget determines what size it should be within those constraints, and passes its width and height back to the parent.
4. The parent looks at the size it wants to be and how it should be aligned, and sets the widget's position accordingly. Alignment can be set explicitly, using a variety of widgets like Center, and the alignment properties on Row and Column.

In Flutter, this layout conversation is often expressed with the simplified phrase, "Constraints go down. Sizes go up. Parent sets the position."

Widget lifecycle

Widgets are created, pass through a series of steps, and finish their short lifes in a garbage can. Their life journey is described by the following articles:

The Journey of a Widget: Understanding the Lifecycle in Flutter. - A blog post that provides a clear overview of the lifecycle methods of a Stateful widget, such as `initState`, `build`, and `dispose`.

A pragmatic guide to BuildContext in Flutter - A long, detailed article explaining the crucial role of `BuildContext` and its relationship to the widget tree and lifecycle.

Flutter App Lifecycle Explained in 4 Minutes - A short video that quickly explains the lifecycle of the entire application itself (e.g., resumed, inactive, paused), which is different from a widget's lifecycle.

Navigation and Routing

A route represents a single screen or page within your application.

Main link for navigation and routing. - The central documentation hub for everything related to navigating between screens in a Flutter app.

Navigation cookbook - A collection of practical recipes for common navigation tasks, like passing data to a new screen or returning data from it.

Another explanation. - A Medium article that offers a comprehensive guide to different navigation patterns and techniques in Flutter.

Navigator - The API documentation for the `Navigator` widget, which manages the stack of routes (screens).

```
// Push a new screen onto the navigation stack.
ElevatedButton(
  child: Text('Go to Second Screen'),
  onPressed: () {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => SecondScreen()),
    );
  },
);

// Pop the current screen to go back.
ElevatedButton(
  child: Text('Go Back'),
  onPressed: () {
    Navigator.pop(context);
```

```
  },
);
```

Basics of Themes

Main information. - The official cookbook recipe on how to create and use themes to share colors and font styles throughout an app.

Theme and ThemeData. - A video tutorial that explains the `Theme` and `ThemeData` classes and how to use them for app-wide styling.

Material Design. - The official website for Google's Material Design system, which is the foundation for Flutter's UI components.

Material Theme Builder. - An interactive online tool that helps you visualize and create a custom color theme for your Material 3 app.

Appainter - A third-party visual theme editor for Flutter that lets you customize your Material Design theme and export the Dart code.

Mastering Material Design 3: The Complete Guide to Theming in Flutter. - A detailed blog post covering the intricacies of theming a Flutter app using the latest Material Design 3 standards.

Themes Playground. - A powerful web app that allows you to experiment with the `FlexColorScheme` package to create beautiful and complex themes.

Figma. - A collaborative interface design tool. I have never used it. How to convert Figma Design into Flutter Code. - A video tutorial demonstrating workflows and tools for translating a design created in Figma into a functional Flutter UI.

Basics of State Management

State management deals with the listening of state changes in one part of the app to keep some other part updated. For example, whenever a data is changed, a widget is updated because it is listening for changes in that specific data. In this course, we will use the Riverpod package. - The official website and documentation for Riverpod, a popular and powerful state management library for Flutter. There are several other options for state management in Flutter that will not be studied. A simplified explanation of riverpod is here.

State management is done using *providers*, which are classes directed to certain kinds of data updating. There are several different kinds of *providers* in Riverpod. We will study *now* just `StateProvider` and `NotifierProvider`.

The references used are:

a. Flutter Riverpod 2.0: The Ultimate Guide - A comprehensive and highly-regarded tutorial from Code with Andrea, covering all major aspects of modern Riverpod.

b. How to use Notifier and AsyncNotifier ... - A specific guide from the same author on the modern `Notifier` and `AsyncNotifier` APIs in Riverpod.

c. State management like a Pro - Flutter Riverpod - A video tutorial from developer Robert Brunhage explaining how to use Riverpod for professional state management.

First of all, use

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

Use StateProvider for managing state of simple variables like `int` or `String`. From the above link, an example of using a provider:

```
final countProvider = StateProvider<int>(
  (ref){ return 0; }
);
```

For watching (listening) a provider:

```
class MyHomePage extends ConsumerWidget {
  const MyHomePage({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // 'watch' rebuilds the widget when the provider's state changes.
    final count = ref.watch(countProvider);

    return Scaffold(
      body: Center(
        child: Text('Count: $count'),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          // 'read' is used to change the state, usually in a callback.
          ref.read(countProvider.notifier).state++;
        },
        child: Icon(Icons.add),
      ),
    );
  }
}
```

You can also use the `Consumer` widget:

```
class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key});
  @override
```

```
  Widget build(BuildContext context) {
    return Center(
      // The Consumer widget provides a `ref` to its builder.
      child: Consumer(builder: (context, ref, child) {
        final name = ref.watch(nameProvider); // Assuming nameProvider exists
        return Text(name);
      }),
    );
  }
}
```

Using the `NotifierProvider`:

```
// 1. Define the Notifier class for more complex logic.
class Counter extends Notifier<int> {
  @override
  int build() => 0; // The initial state

  void increment() {
    state++;
  }
}

// 2. Create the provider.
final counterProvider = NotifierProvider<Counter, int>(Counter.new);

// 3. Use it in a widget.
class CounterWidget extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final counter = ref.watch(counterProvider);
    return ElevatedButton(
      child: Text('Value: $counter'),
      onPressed: () => ref.read(counterProvider.notifier).increment(),
    );
  }
}
```

Widget book

Essential tool for testing your widgets in a variaty of devices and situations.

- Widgetbook Docs. Don't forget to read about Knobs and Addons.
- Widgetbook Demo - a video

Intermediary Widgets

Generic Widgets and Classes

Builder - The API documentation for the `Builder` widget, which is used to obtain a `BuildContext` from a new scope, often to use with widgets like `Scaffold.of()`.

```
// Useful for getting a context deep within the widget tree.
Scaffold(
  appBar: AppBar(title: Text('Builder Example')),
  body: Builder(
    builder: (BuildContext innerContext) {
      // This context is a descendant of the Scaffold.
      return Center(
        child: ElevatedButton(
          child: Text('Show SnackBar'),
          onPressed: () {
            ScaffoldMessenger.of(innerContext).showSnackBar(
              SnackBar(content: Text('Hello from the Builder!')),
            );
          },
        ),
      );
    },
  ),
);
```

Builder (Flutter Widget of the Week) - A short, official video from the Flutter team explaining the purpose and use case for the `Builder` widget.

StatefulBuilder - A video explaining the `StatefulBuilder` widget, a useful tool for encapsulating and rebuilding a small piece of stateful UI without needing a full `StatefulWidget`.

```
// Creates a small piece of stateful UI without a full StatefulWidget.
showDialog(
  context: context,
  builder: (context) {
    int counter = 0;
    return StatefulBuilder(
      builder: (context, setState) {
        return AlertDialog(
          title: Text("Counter Dialog"),
          content: Text("Count: $counter"),
          actions: [
            TextButton(
              child: Text("Increment"),
```

```
              onPressed: () {
                setState(() {
                  counter++;
                });
              },
            ),
          ],
        );
      },
    );
  },
);
```

The dart.io package for file input and output. - Official documentation for Dart's core library for handling files, sockets, processes, and other system-level I/O.

Decode and encode a JSON string:

```
/// put in the beginning of file:
///  import 'dart:convert';

  var jsonObj = {
    "name": "John Smith",
    "email": "john.smith@example.com",
    "address": "123 Main Street",
  };
  /// encoding json object to a string
  var jsonStr = jsonEncode(jsonObj);
  print(jsonStr);
  /// decoding json string to an object
  var decodedJson = jsonDecode(jsonStr);

  /// convert json to a string with indentation of 2 spaces
  var encoder = JsonEncoder.withIndent('  ');
  var strJsonFormatted = encoder.convert(decodedJson);
  print(strJsonFormatted);
```

Classes for handling the assets folder. - The official guide on how to include and use assets like images and fonts in a Flutter application.

SharedPreferences - The pub.dev page for the `shared_preferences` package, which is used for storing simple key-value data persistently on the device.

Text

Text (advanced). - The main API documentation for the `Text` widget, detailing its numerous styling and configuration properties.

RichText - API documentation for a widget that allows displaying text with multiple different styles within the same block.

```
// For text with multiple styles in one block.
RichText(
  text: TextSpan(
    text: 'Hello ',
    style: DefaultTextStyle.of(context).style,
    children: <TextSpan>[
      TextSpan(text: 'bold', style: TextStyle(fontWeight: FontWeight.bold)),
      TextSpan(text: ' world!'),
    ],
  ),
);
```

DefaultTextStyle - A widget that applies a default text style to all of its descendant `Text` widgets.

```
// Applies a text style to all descendant text widgets.
DefaultTextStyle(
  style: TextStyle(color: Colors.purple, fontSize: 20),
  child: Column(
    children: [
      Text('This text is purple.'),
      Text('This text is also purple.'),
      Text('This one is red.', style: TextStyle(color: Colors.red)),
    ],
  ),
);
```

SelectableText - A widget that displays text that the user can select and copy.

```
// Text that can be selected by the user.
SelectableText(
  'You can copy this text. Long-press or double-tap.',
  style: TextStyle(fontSize: 18),
  textAlign: TextAlign.center,
);
```

Need a full editor? Use this. - A link to the `re_editor` package, a powerful and extensible code editor widget for Flutter.

Asynchronous Programming

Explanation from dart.dev. - The official Dart documentation explaining how to use `Future`, `async`, and `await` for asynchronous operations.

Text with examples. - A Medium article that provides practical examples of asynchronous programming in a Flutter context.

Futures - The detailed API documentation for the `Future` class, which represents a potential value or error that will be available at some time in the future.

Isolates. - A video explaining Dart's concurrency model using Isolates to perform heavy computations without blocking the UI thread.

Isolate technique. The example is:

```dart
Future<dynamic> loadJson(String filename) async {
  File f = File(filename);

  final String jsonStr = await f.readAsString();
  // Use Isolate.run to perform heavy parsing in the background.
  final json = await Isolate.run(() {
    final jsonObj = jsonDecode(jsonStr);
    return jsonObj;
  });
  return json;
}
```

Important: async gaps

Yandex dictionary lookup. - An example link to a dictionary API endpoint, likely used to demonstrate making network requests.

FutureBuilder - API documentation for a widget that builds itself based on the latest snapshot of interaction with a `Future`, perfect for displaying loading states and results from async operations.

```dart
// Builds UI based on the result of a Future.
FutureBuilder<String>(
  future: Future.delayed(Duration(seconds: 2), () => 'Data Loaded!'),
  builder: (BuildContext context, AsyncSnapshot<String> snapshot) {
    if (snapshot.connectionState == ConnectionState.waiting) {
      return CircularProgressIndicator();
    } else if (snapshot.hasError) {
      return Text('Error: ${snapshot.error}');
    } else {
      return Text('Result: ${snapshot.data}');
    }
  },
);
```

More widgets for layout

ListView. - A core widget for displaying a scrollable list of widgets linearly.

```dart
// A scrollable list of items, built on demand.
ListView.builder(
  itemCount: 50,
```

```
  itemBuilder: (context, index) {
    return ListTile(
      leading: Icon(Icons.person),
      title: Text('Item $index'),
    );
  },
);
```

Further info: ListView inside a Column - An article addressing the common layout issue of placing an unbounded `ListView` inside a `Column` and how to solve it.

video on nesting ListView and Column. - A video tutorial that visually explains solutions for nested scrolling widgets.

Nested ListViews and Columns - A video providing more advanced techniques for handling complex nested layouts.

Top 12 ListView Widgets. - A video showcasing various specialized list widgets and packages for different use cases.

SingleChildScrollView - A widget that makes its single child scrollable, often used to prevent content from overflowing on smaller screens.

```
// Makes a tall column of widgets scrollable.
SingleChildScrollView(
  child: Column(
    children: List.generate(20, (i) => ListTile(title: Text("Item ${i+1}"))),
  ),
);
```

Align - A widget that aligns its child within itself and can optionally size itself based on the child's size.

```
// Aligns a child to a specific position within a parent.
Container(
  height: 120.0,
  width: 120.0,
  color: Colors.blue[50],
  child: Align(
    alignment: Alignment.topRight,
    child: FlutterLogo(size: 60),
  ),
);
```

Padding - A widget that insets its child by a given padding.

```
// Adds space around a widget.
Padding(
  padding: const EdgeInsets.all(20.0),
```

```
  child: Text('This text has padding around it.'),
);
```

ConstrainedBox - A widget that imposes additional constraints (like a max width or height) on its child.

```
// Enforces size constraints on its child.
ConstrainedBox(
  constraints: BoxConstraints(
    minWidth: 70,
    maxWidth: 150,
    minHeight: 70,
    maxHeight: 150,
  ),
  child: Container(color: Colors.red, width: 200, height: 50), // Will be constrained
);
```

SizedBox - A box with a specified size, often used as a spacer between other widgets.

```
// Used as a spacer between two widgets in a Column.
Column(
  children: [
    Text('First Item'),
    SizedBox(height: 20), // 20 pixels of vertical space
    Text('Second Item'),
  ],
);
```

There are even more widgets for layout. - A link to the official documentation page that catalogs all of Flutter's layout widgets.

Widgets for Responsiveness

Some websites on Devices for Android, in Portuguese, are here. - The main landing page for Android developer guides in Portuguese. Specific details are cited below.

Visão geral de compatibilidade de dispositivos - The Portuguese version of the Android guide on device compatibility.

Visão geral de compatibilidade de tela - The Portuguese version of the Android guide on supporting different screen sizes.

Modo de compatibilidade do dispositivo - The Portuguese version of the Android guide on device compatibility mode.

Compatibilidade com diferentes densidades de pixels - The Portuguese version of the Android guide on supporting different pixel densities (DP).

A list of important widgets for responsiveness follows.

MediaQuery: this is the main widget to make responsive Apps. Among other things, you can get the width and height of the screen.

```
// Accesses screen size and other properties.
Widget build(BuildContext context) {
  final screenWidth = MediaQuery.of(context).size.width;
  return Container(
    width: screenWidth / 2, // Half the screen width
    color: Colors.green,
    child: Text('This is half the screen width.'),
  );
}
```

When the code uses `MediaQuery.of(context). ...` any changes in any property causes the widget related to `context` to be rebuild. There are new methods in MediaQuery that only rebuild the widget if that specific feature changes. See the example:

```
    // Old way (less performant if only size is needed)
    final size = MediaQuery.of(context).size;

    // Recommended way for size changes
    final size = MediaQuery.sizeOf(context);
    final width = MediaQuery.widthOf(context);
    final height = MediaQuery.heightOf(context);
```

If `sizeOf` is used, the widget is only rebuild if the screen size changes.

Wrap - A layout widget that arranges its children in multiple horizontal or vertical runs, wrapping to the next line when space runs out.

```
// Arranges items in a row and wraps to the next line if needed.
Wrap(
  spacing: 8.0, // Gap between adjacent chips.
  runSpacing: 4.0, // Gap between lines.
  children: <Widget>[
    Chip(label: Text('Tag 1')),
    Chip(label: Text('Tag 2')),
    Chip(label: Text('A Longer Tag 3')),
    Chip(label: Text('Tag 4')),
  ],
);
```

OrientationBuilder - A widget that builds a different UI depending on the device's current orientation (portrait or landscape).

```
// Rebuilds its child when the device orientation changes.
OrientationBuilder(
  builder: (context, orientation) {
    return Center(
```

```
      child: Text('Current orientation: $orientation'),
    );
  },
);
```

LayoutBuilder - A widget that builds a UI that can depend on the parent widget's size, providing constraints at layout time.

```
// Builds a different layout based on the parent's constraints.
LayoutBuilder(
  builder: (context, constraints) {
    if (constraints.maxWidth > 600) {
      return Text('This is a wide layout.');
    } else {
      return Text('This is a narrow layout.');
    }
  },
);
```

Expanded - A widget used inside a `Row` or `Column` that expands its child to fill the available space.

```
// Fills the available space in a Row or Column.
Row(
  children: <Widget>[
    Container(color: Colors.red, width: 100),
    Expanded(
      child: Container(
        color: Colors.blue,
        child: Text('I fill the rest of the space'),
      ),
    ),
    Container(color: Colors.green, width: 100),
  ],
);
```

Flexible - A widget used inside a `Row` or `Column` that controls how its child flexes to fill space, using a `flex` factor.

```
// Shares space in a Row or Column based on a flex factor.
Row(
  children: <Widget>[
    Flexible(
      flex: 2, // Takes twice the space of the other flexible widget
      child: Container(color: Colors.cyan),
    ),
    Flexible(
      flex: 1,
      child: Container(color: Colors.teal),
```

```
    ),
  ],
);
```

FittedBox - A widget that scales and positions its child within itself according to a specified `fit`.

```
// Scales its child to fit within the available space.
Container(
  height: 100,
  width: 200,
  color: Colors.orangeAccent,
  child: FittedBox(
    fit: BoxFit.contain,
    child: Text('This Text Is Huge But It Fits'),
  ),
);
```

FractionallySizedBox - A widget that sizes its child to a fraction of the total available space.

```
// Sizes its child as a fraction of the parent's size.
FractionallySizedBox(
  widthFactor: 0.5, // 50% of the parent's width
  heightFactor: 0.2, // 20% of the parent's height
  child: Container(
    color: Colors.amber,
  ),
);
```

IntrinsicHeight - A widget that sizes its children to the same, "intrinsic" height.

```
// Forces children of a Row to be the same height.
IntrinsicHeight(
  child: Row(
    crossAxisAlignment: CrossAxisAlignment.stretch,
    children: [
      Container(width: 100, color: Colors.red),
      Container(width: 50, color: Colors.blue),
    ],
  ),
);
```

IntrinsicWidth - A widget that sizes its children to the same, "intrinsic" width.
Intrinsic Height and Widgth in HeyFlutter - A video tutorial from HeyFlutter explaining how to use these two related intrinsic sizing widgets.

Handling User Input

This is a general documentation. - The official guide to handling user input in Flutter.

TextField - A Material Design text field for user input. A better explanation is here.

TextField in HeyFlutter - A video tutorial covering the features and usage of the `TextField` widget.

```
// A text input field.
TextField(
  decoration: InputDecoration(
    border: OutlineInputBorder(),
    labelText: 'Enter your name',
    hintText: 'John Doe',
  ),
);
```

Form - A widget that acts as a container for multiple `TextField` widgets, allowing for grouping and validation. See also this for form validation. - The official cookbook recipe for building a form with input validation.

```
// Groups and validates multiple text fields.
final _formKey = GlobalKey<FormState>();
Form(
  key: _formKey,
  child: Column(
    children: <Widget>[
      TextFormField(
        validator: (value) => value!.isEmpty ? 'Please enter some text' : null,
      ),
      ElevatedButton(
        onPressed: () {
          if (_formKey.currentState!.validate()) {
            // Process data
          }
        },
        child: Text('Submit'),
      ),
    ],
  ),
);
```

Flutter Form Widget - A video tutorial demonstrating how to use the `Form` widget for input management.

TextButton - A simple text label button without a border.

```
// A flat button with text.
TextButton(
  onPressed: () {},
  child: Text('Learn More'),
);
```

OutlinedButton - A button with a thin border.

```
// A button with a visible border.
OutlinedButton(
  onPressed: () {},
  child: Text('Cancel'),
);
```

IconButton - A picture-only button, typically an icon.

```
// A button that is just an icon.
IconButton(
  icon: Icon(Icons.settings),
  onPressed: () {},
  tooltip: 'Settings',
);
```

FloatingActionButton - A circular button that floats above the UI, typically used for a primary action.

```
// The main action button on a screen.
FloatingActionButton(
  onPressed: () {},
  child: Icon(Icons.add),
  tooltip: 'Add Item',
);
```

SegmentedButton - A set of buttons that allows for mutually exclusive selection, like a row of tabs.

```
// A row of buttons for selecting one option.
// Requires a stateful variable `_selection` to manage the state.
SegmentedButton<String>(
  segments: const [
    ButtonSegment(value: 'day', label: Text('Day')),
    ButtonSegment(value: 'week', label: Text('Week')),
    ButtonSegment(value: 'month', label: Text('Month')),
  ],
  selected: <String>{_selection},
  onSelectionChanged: (newSelection) {
    setState(() {
      _selection = newSelection.first;
    });
```

```
  },
);
```

Chip - A compact element representing an input, attribute, or action.

```
// A small, pill-shaped label.
Chip(
  avatar: CircleAvatar(child: Text('A')),
  label: Text('Action Chip'),
  onDeleted: () {},
);
```

DropdownMenu - A menu that shows a list of choices when tapped.

```
// A dropdown for selecting one item from a list.
DropdownMenu<String>(
  initialSelection: 'One',
  onSelected: (String? value) { /* Handle selection */ },
  dropdownMenuEntries: <String>['One', 'Two', 'Three'].map((String value) {
    return DropdownMenuEntry<String>(value: value, label: value);
  }).toList(),
);
```

Drawer - The Material Design navigation panel that slides in from the side.

```
// A side navigation menu, placed in Scaffold's `drawer` property.
Drawer(
  child: ListView(
    children: <Widget>[
      DrawerHeader(child: Text('Menu Header')),
      ListTile(title: Text('Item 1'), onTap: () {}),
      ListTile(title: Text('Item 2'), onTap: () {}),
    ],
  ),
);
```

Slider - A widget that allows users to select from a range of values by moving a thumb.

```
// A slider for selecting a value in a range.
// Requires a stateful variable `_sliderValue` to manage state.
Slider(
  value: _sliderValue,
  min: 0,
  max: 100,
  divisions: 10,
  label: _sliderValue.round().toString(),
  onChanged: (double value) {
    setState(() {
      _sliderValue = value;
```

```
    });
  },
);
```

Checkbox - A widget for selecting a boolean (true/false) state.

```
// A checkbox.
// Requires a stateful variable `_isChecked` to manage state.
Checkbox(
  value: _isChecked,
  onChanged: (bool? value) {
    setState(() {
      _isChecked = value!;
    });
  },
);
```

Switch - An on/off toggle switch.

```
// An on/off switch.
// Requires a stateful variable `_isSwitchedOn` to manage state.
Switch(
  value: _isSwitchedOn,
  onChanged: (bool value) {
    setState(() {
      _isSwitchedOn = value;
    });
  },
);
```

Radio - A widget for selecting one option from a set.

```
// A radio button for selecting one option from a group.
// Requires a stateful variable `_groupValue` to manage state.
Radio<String>(
  value: 'Option A',
  groupValue: _groupValue,
  onChanged: (String? value) {
    setState(() { _groupValue = value; });
  },
);
```

DatePickerDialog - A dialog for picking a date.

```
// Shows a calendar dialog to pick a date.
ElevatedButton(
  child: Text('Show Date Picker'),
  onPressed: () {
    showDatePicker(
      context: context,
```

```
      initialDate: DateTime.now(),
      firstDate: DateTime(2000),
      lastDate: DateTime(2100),
    );
  },
);
```

TimePickerDialog - A dialog for picking a time.

```
// Shows a clock dialog to pick a time.
ElevatedButton(
  child: Text('Show Time Picker'),
  onPressed: () {
    showTimePicker(
      context: context,
      initialTime: TimeOfDay.now(),
    );
  },
);
```

Dismissible - A widget that can be dismissed by swiping.

```
// A list item that can be swiped away.
Dismissible(
  key: Key('item_key'), // Must have a unique key.
  onDismissed: (direction) { /* Handle dismissal */ },
  background: Container(color: Colors.red),
  child: ListTile(title: Text('Swipe me away')),
);
```

Dismissible (Flutter Widget of the Week) - The official short video from the Flutter team explaining how to use the `Dismissible` widget.

InkWell - A rectangular area that responds to touch and shows a "ripple" splash effect.

```
// A generic widget that shows a ripple effect on tap.
InkWell(
  onTap: () {},
  child: Container(
    padding: EdgeInsets.all(12.0),
    child: Text('Tap Me'),
  ),
);
```

InkResponse - A more configurable version of `InkWell` that can have different shapes.

GestureDetector - A powerful widget for detecting a variety of gestures like taps, drags, and scales, without any visual feedback.

```dart
// Detects various gestures without any visual feedback.
GestureDetector(
  onDoubleTap: () {
    print('Double Tapped!');
  },
  onLongPress: () {
    print('Long Pressed!');
  },
  child: Container(
    width: 100,
    height: 100,
    color: Colors.grey,
    child: Center(child: Text('Gesture Area')),
  ),
);
```

Selecting the Position of Widgets

The Stack and Positioned classes:.

Stack in Flutter docs - API docs for a widget that lets you place widgets on top of each other.

Positioned in Flutter docs - API docs for a widget that must be used inside a `Stack` to control a child's position.

```dart
// Overlaps widgets on top of each other.
Stack(
  children: <Widget>[
    Container(width: 200, height: 200, color: Colors.green),
    Positioned(
      top: 20,
      right: 20,
      child: Container(width: 50, height: 50, color: Colors.yellow),
    ),
    Positioned(
      bottom: 20,
      left: 20,
      child: Text('On top'),
    ),
  ],
);
```

Stack (Flutter Widget of the Week) - The official short video explaining the `Stack` widget.

Flutter Tutorial - Stack — Deep Dive - A more in-depth video tutorial on how to use `Stack` for complex layouts.

Overlays, The Floating Widgets

Overlay in Flutter docs - API docs for the `Overlay` widget, a stack-like mechanism for showing widgets on top of everything else, used for things like tooltips and dropdowns.

HeyFlutter - A video from HeyFlutter explaining the basics of using `Overlay` and `OverlayEntry`.

Another from HeyFlutter - A second HeyFlutter video, likely covering more advanced `Overlay` use cases.

The OverlayPortal class:

OverlayPortal in Flutter docs - API docs for a newer, easier-to-use widget for managing overlays declaratively.

OverlayPortal (Widget of the Week) - The official short video explaining the benefits of using `OverlayPortal` over the traditional `Overlay` API.

```
// A modern way to show floating widgets.
// Requires a stateful controller `_overlayController`.
final _overlayController = OverlayPortalController();

// The widget that will be shown in the overlay.
@override
Widget build(BuildContext context) {
  return OverlayPortal(
    controller: _overlayController,
    overlayChildBuilder: (context) {
      return Positioned(top: 100, left: 50, child: Chip(label: Text('I am an overlay!')));
    },
    child: ElevatedButton(
      child: Text('Toggle Overlay'),
      onPressed: () {
        _overlayController.toggle();
      },
    ),
  );
}
```

Animations and Transitions

The Hero widget - A video explaining the `Hero` widget, used for creating beautiful "hero" transitions where a widget appears to fly between two screens.

Animations playlist by the Flutter team. - The official YouTube playlist from the Flutter team covering a wide range of animation topics.

How to choose which animation widget is right for you. - A guide from the Flutter team to help you decide between implicit and explicit animation widgets.

Making animations in Flutter - The official Flutter YouTube playlist dedicated to animations.

Animation Deep Dive — in text - A detailed article from the Flutter team explaining the core concepts behind the animation system.

Animation Tutorial - The official step-by-step tutorial for getting started with animations in Flutter.

AnimatedContainer - An implicitly animated version of `Container` that automatically animates changes to its properties.

```
// Automatically animates changes to its properties.
// Requires stateful variables `_width`, `_height`, `_color`.
AnimatedContainer(
  width: _width,
  height: _height,
  color: _color,
  duration: Duration(seconds: 1),
  curve: Curves.fastOutSlowIn,
);

// In a button's onPressed:
setState(() {
  _width = _width == 100 ? 200 : 100;
  _color = _color == Colors.blue ? Colors.red : Colors.blue;
});
```

Tween - API docs for the `Tween` class, which defines the interpolation between a beginning and ending value for an animation.

AnimatedWidget - Documentation for a base class that simplifies the creation of custom animation widgets.

Animations cookbook. - The official collection of practical recipes specifically for implementing animations.

Advanced Topics

General Widgets

Drag a UI element. - An official cookbook recipe on how to make a widget draggable using the `Draggable` and `DragTarget` widgets.

drag_and_drop_lists - A popular package for creating reorderable lists with drag-and-drop functionality.

Advanced Navigation and Routing

From the article Mastering navigation in Flutter, data is passed between routes using

```
Navigator.pushNamed(context, '/details', arguments: {'productId': 123});
```

The arguments are accessed in the previous route using

```
ModalRoute.of(context).settings.arguments
```

A good package for managing routes is go_router. - The official routing package recommended by the Flutter team for handling complex navigation scenarios, including deep linking. Use ShellRouter to keep the Scaffold fixed. - An article explaining how to use `ShellRoute` within `go_router` to create nested navigation with a persistent UI shell (like a bottom navigation bar).

State Management

FutureProvider. - Riverpod documentation for a provider that is perfect for handling asynchronous operations that return a single value, like a network request.

StreamProvider. - Riverpod documentation for a provider designed to work with Dart Streams, ideal for real-time data like a WebSocket connection or Firestore snapshots.

Videos:

Flutter Riverpod EASY Tutorial - A beginner-friendly video tutorial on the basics of Riverpod.

Flutter Tutorial - Riverpod - 1/3 The Complete Guide For Providers. - Part one of a comprehensive video series from HeyFlutter that covers Riverpod's providers in detail.

Flutter Riverpode State Management — Simplest Explanation. - A video that aims to simplify the core concepts of Riverpod for easy understanding.

Animations and Transitions

Simultaneous animations. - An official tutorial section on how to run multiple animations at the same time on a single widget.

List of animation widgets - The official catalog page for all of Flutter's built-in animation and motion widgets.

Package flutter_animate. - A video tutorial for the popular `flutter_animate` package, which provides a simple, fluent API for creating complex animations.

Animations Package Tutorial. - A video tutorial for the `animations` package, which provides pre-built, high-quality transitions like `FadeThroughTransition`

and `SharedAxisTransition`.

Lottie animations - The pub.dev page for the Lottie package, which allows you to play high-quality, vector-based animations created in Adobe After Effects.

Lottie em Heyflutter. - A video from HeyFlutter demonstrating how to implement Lottie animations in a Flutter app.

Rive documentation and in Heyflutter.

Design

Design Cookbook. - The official collection of recipes related to UI design, including themes, fonts, and custom painters.

O que é DP? - The Portuguese Android guide explaining what a Density-independent Pixel (DP) is, a crucial concept for responsive design.

How to choose fonts. - A video providing guidance on selecting appropriate fonts for UI design.

The ONLY 8 Fonts UI Designers Need. Forget The Rest. - A video offering an opinionated list of versatile fonts for designers.

Designers Only Need These 6 Fonts. Trash the Rest. - Another video with a curated list of essential fonts for UI design.

What Is Neumorphism and Why Should Designers Care? - An article explaining the neumorphic design trend, which uses soft shadows to create a "soft UI" effect.

More on neumorphism - A link to the `flutter_neumorphic` package for easily creating this style.

Neumorphism: The art of shadow and light. - Another article detailing the principles behind the neumorphic design style.

A complete Neumorphic ui kit for Flutter - A tutorial or resource for a full set of neumorphic UI components.

Neumorphism Button - A video tutorial on how to create a neumorphic-style button in Flutter.

Flutter neumorphic login page. - A tutorial showing how to build a login screen using the neumorphic design style.

Sign in and sign up neumorphic. - A resource for a sign-in/sign-up UI built with neumorphism from scratch, without external packages.

Networking and API Integration

Asynchronous programming. - A Medium article that provides practical examples of asynchronous programming in a Flutter context.

Fetching data from the internet. - The official cookbook recipe on how to make HTTP requests to fetch data from a server.

CircularProgressIndicator - API docs for the spinning indicator widget used to show that something is loading.

Obter dados da internet. - An article in Portuguese from Alura on how to get data from the internet using the `http` package.

A guide to HTTP requests, JSON parsing, and File Transfers - A detailed guide covering common networking tasks in Flutter.

Use of Dart patterns to validade JSON - Official Dart documentation on using modern pattern matching to validate the structure of JSON data.

StreamBuilder - API docs for a widget that builds itself based on interaction with a `Stream`, perfect for real-time data.

Other text. - An alternative tutorial explaining the use of `StreamBuilder`.

Jump to the Clouds: Firebase and Firestore

Firebase is an app development platform that includes backend services like authentication, storage, hosting, and real-time databases.

Firestore is a NoSQL document database that's part of Firebase, and is used to store, retrieve, and manage large amounts of data.

Using Firestore as a backend to your Flutter app. - A video tutorial explaining how to connect and use Firestore in a Flutter application.

Add Firebase to your Flutter app - The official documentation for setting up Firebase in a Flutter project.

Flutter Tutorial - Firebase Setup For Android 1/3 - A video guide specifically for configuring Firebase on the Android platform.

Use Firebase to host your Flutter app on the web. - An article from the Flutter team on how to deploy a Flutter web app using Firebase Hosting.

Building chat app with Flutter and Firebase. - A step-by-step tutorial on creating a real-time chat application.

Firebase - Back to the Basics - A video from the official Firebase channel reviewing fundamental concepts.

100 Firebase Tips, Tricks, and Screw-ups. - A presentation from a Google I/O event sharing valuable real-world experience with Firebase.

Authentication using Firebase

Get started. - The official documentation for getting started with Firebase Authentication in Flutter.

Video from Heyflutter. - A tutorial from HeyFlutter on implementing Firebase Authentication.

Flutter Firebase Auth — The Cleanest & Fastest Way - IOS & Android - A video tutorial that promises a clean and efficient method for setting up authentication.

package firebase_ui_auth - The pub.dev page for a package that provides pre-built UI screens for Firebase Authentication, saving a lot of development time.

Databases

Cloudflare, cloud-based database.

Supabase, cloud-based database.

Objectbox, local database, NoSQL. See also the package objectbox., the pub.dev page for ObjectBox, a high-performance, easy-to-use NoSQL database that runs directly on the device. Como usar Objectbox em Flutter., an article in Portuguese on how to use ObjectBox in a Flutter project.

Internationalizing Flutter apps

Introduction - The official documentation on internationalization (i18n) and localization (l10n) in Flutter.

Package flutter_localization - A package designed to simplify the process of adding multiple language support to a Flutter app.

Localization of Flutterando - A localization package created by the Flutterando community.

Automatically Translate Your Flutter App - An article describing a workflow for using tools to automate the translation of localization files.

Slang, a simpler package for internationalization

Testing and Debugging

An introduction to unit testing. - The official guide on how to write unit tests for your Dart and Flutter code to verify the logic of individual functions and classes.

Deploying Flutter Apps

Deployment. - The main documentation hub for all topics related to building and deploying Flutter apps to different platforms.

Build and release an Android app - The official guide for preparing and publishing a Flutter app to the Google Play Store.

Build and release an iOS app. - The official guide for preparing and publishing a Flutter app to the Apple App Store.

Flutter In-App Subscription Tutorial (with RevenueCat). - A video tutorial on how to implement in-app purchases and subscriptions using the RevenueCat service.

Continuous delivery with Flutter - The official guide on setting up Continuous Integration and Continuous Delivery (CI/CD) pipelines for your Flutter projects.

Advanced Widgets, Packages, and Techniques

Package Mason for pre-build apps. A video tutorial on Mason, a tool for creating and using reusable templates (called "bricks") to generate code files quickly. A good article is here. A comprehensive written guide to using Mason. Use the link for searching for bricks. The official repository for discovering and sharing Mason bricks. You will need to install the command-line Git first.

How To Host Flutter Website On Custom Domain - A video tutorial on deploying a Flutter web app and connecting it to a custom domain name.

How to Change App Icon and App Name - A tutorial showing the process of changing the launcher icon and display name for a Flutter app on iOS and Android.

Flutter Tutorial - Create App Shortcuts For Home Screen - A tutorial on how to implement home screen quick actions (app shortcuts) for your application.

Flutter Tutorial - Detect App Background & App Closed - A tutorial explaining how to detect when the app's lifecycle state changes (e.g., goes into the background).

Flutter Update: App Monetization - An official update from the Flutter team regarding new packages and strategies for monetizing apps.

Monetizing apps with Flutter - A presentation on different methods for monetizing Flutter apps, including ads and in-app purchases.

Flutter Push Notifications using Firebase - A tutorial on how to implement push notifications using Firebase Cloud Messaging (FCM).

OneSignal Notifications in Flutter Made Easy - A tutorial demonstrating how to use the OneSignal service as an alternative to Firebase for push notifications.