Understanding Flutter

Martin Estanislao Escalante Leiva

May 1st, 2024

Understanding Flutter:

A Comprehensive Exploration of Key Concepts and Lifecycles

Topics:

Junior

- What is Flutter? x
- Dart x
- OOP Concepts x
- Dart Collections x
- Basics of Asynchronous Programming x
- Widgets x
- Stateful vs Stateless widgets x
- Lifecycle methods x
- BuildContext x
- Packages vs plugins x
- Hot reload & Hot restart x
- Build Types x
- runApp() vs main() x
- UI (Scaffold, Row, Column…) x
- State management (Provider, BloC)
- GIT

Intermediate

- API integration (HTTP, Dio, JSON)
- Database integration (Cloud Firestore)
- Animations & Navigation
- State Management Library (Provider)
- Types

Senior

- Advanced Asynchronous programming (Streams, Isolates, Event Loops)
- Responsive UI
- Testing (unit, widget, integration)
- State Management
- Project Architecture
- Firebase services
- CI/CD
- Performance
- App release

Scenarios and solutions

- How to release apps

Clean Code and folder organization

video

What is Flutter?

Flutter is an open-source UI software development toolkit created by Google, open source. It enables developers to build natively compiled applications for mobile, web, and desktop from a single codebase.

1. **Hot Reload**: Developers to instantly see the changes they make to the code reflected in the app without having to restart it, making the development process faster and more efficient.
2. **Cross-platform Development**: Create applications for multiple platforms (iOS, Android, Web, and Desktop) using a single codebase, thereby reducing development time and effort.
3. **Developer Experience**: Provides a rich set of developer tools and features to enhance the development experience:
   - Hot reload, which speeds up iteration cycles
   - Pre-built widgets for building UIs quickly
   - Comprehensive documentation, and strong community support.
   - Flutter's reactive framework and declarative UI paradigm make it easier for developers to build complex UIs with less code.
4. **High Performance**: Flutter uses a compiled programming language (Dart) and a graphics engine (Skia) to render UI elements directly on the canvas, bypassing the *OEM widgets used by traditional mobile frameworks*. This approach results in high performance and smooth animations, even on less powerful devices. Additionally, Flutter's architecture enables it to achieve consistent 60 frames per second (fps) performance, providing users with a smooth and responsive experience across different platforms. > Many mobile development frameworks rely on the native widgets provided by the operating system (such as UIKit for iOS or Android SDK for Android) to render UI elements. These widgets are often designed specifically for their respective platforms and are tightly integrated with the operating system. In contrast, Flutter takes a different approach. It doesn't use these native widgets directly. Instead, it provides its own set of widgets that are rendered directly on the canvas using *Skia, a 2D graphics engine*. This allows Flutter to achieve high performance and consistent behavior across different platforms, as it doesn't rely on platform-specific widgets and rendering mechanisms.

Dart

Dart is a programming language developed by Google, designed for building web, server, and mobile applications. Dart is the programming language used to develop Flutter apps.

Flutter is a UI toolkit/framework also developed by Google and it uses Dart as its primary programming language.

Dart provides the foundational language features and libraries that Flutter relies on to create user interfaces, manage app state, handle asynchronous operations, and more.

> Dart is the language you use to write Flutter apps, while Flutter provides the UI framework and runtime environment for running Dart code and rendering user interfaces.

Key features and characteristics:

1. **General-Purpose Language**: Dart is a versatile language suitable for a wide range of application domains, including web development, mobile app development, server-side programming, and command-line tools.

2. **Object-Oriented**: Dart is an object-oriented language, which means it supports concepts such as classes, objects, inheritance, and encapsulation. This makes it easy to organize and structure code into reusable components.

3. **Optional Typing**: Dart supports both static and dynamic typing. You can choose to specify types for variables and function parameters, or you can rely on type inference to determine types automatically. This

flexibility allows developers to write code that is both statically and dynamically typed, depending on their preferences and requirements.

4. **Asynchronous Programming**: Dart provides built-in support for asynchronous programming using features such as async and await. This allows developers to write code that performs asynchronous operations, such as fetching data from a network or accessing files, without blocking the main thread.
5. **Single-Threaded with Event Loop**: Dart is primarily a single-threaded language with an event loop, similar to JavaScript. This means that Dart code runs on a single thread, but asynchronous operations are scheduled on the event loop, allowing for non-blocking I/O and concurrency.
6. **Garbage Collection**: Dart uses automatic memory management through garbage collection. Developers don't need to manually allocate or deallocate memory, as Dart's garbage collector automatically manages memory allocation and deallocation, freeing developers from memory management concerns.
7. **Ahead-of-Time (AOT) and Just-in-Time (JIT) Compilation**: Dart supports both AOT and JIT compilation. JIT compilation is used during development for hot reload and fast iteration cycles, while AOT compilation is used for production builds to generate optimized machine code for better performance.
8. **Strong Tooling and Ecosystem**: Dart comes with a rich set of development tools, including the Dart SDK, Dart DevTools, and the Dart Analyzer. Additionally, Dart has a growing ecosystem of libraries and packages that provide functionality for a wide range of tasks, such as web development, HTTP requests, database access, and more.

Overall, Dart is a modern, expressive, and productive language that offers a powerful set of features for building a variety of applications across different platforms. Its simplicity, flexibility, and strong tooling make it an attractive choice for developers looking to build high-quality software efficiently.

OOP Concepts

Dart is an object-oriented programming (OOP) language, meaning it supports the fundamental principles of OOP. Here are the main OOP concepts in Dart:

1. **Classes and Objects**:
   - Classes are blueprints for creating objects. They define the properties (attributes) and behaviors (methods) of objects.
   - Objects are instances of classes. They represent individual entities with their own state and behavior.
   - Dart uses the `class` keyword to define classes.

```
class Person {
  String name;
  int age;

  void sayHello() {
    print('Hello, my name is $name and I am $age years old.');
  }
}

void main() {
  var person = Person();
  person.name = 'John';
  person.age = 30;
  person.sayHello(); // Output: Hello, my name is John and I am 30 years old.
}
```

2. **Inheritance**:
   - Inheritance is a mechanism where a class (subclass) can inherit properties and behaviors from another class (superclass).
   - Dart supports single inheritance, meaning a subclass can only inherit from one superclass.
   - Use the `extends` keyword to establish inheritance.

```
class Animal {
  void eat() {
    print('Animal is eating.');
  }
}

class Dog extends Animal {
  void bark() {
    print('Woof! Woof!');
  }
}

void main() {
  var dog = Dog();
  dog.eat(); // Output: Animal is eating.
  dog.bark(); // Output: Woof! Woof!
}
```

3. **Encapsulation**:
   - Encapsulation is the bundling of data (properties) and methods that operate on that data within a single unit (class).
   - It helps hide the internal implementation details of a class and provides controlled access to its members.
   - Dart supports encapsulation through the use of access modifiers like `public`, `private`, and `protected`.

```
class Counter {
  int _count = 0; // Private variable

  void increment() {
    _count++;
  }

  int getCount() {
    return _count;
  }
}

void main() {
  var counter = Counter();
  counter.increment();
  print(counter.getCount()); // Output: 1
}
```

4. **Polymorphism**:
   - Polymorphism allows objects of different classes to be treated as objects of a common superclass.
   - Dart supports method overriding, where a subclass can provide a specific implementation of a method defined in its superclass.

```
class Animal {
  void makeSound() {
    print('Animal makes a sound.');
  }
}

class Dog extends Animal {
  @override
  void makeSound() {
    print('Dog barks.');
  }
}
```

```
void main() {
  Animal animal = Dog();
  animal.makeSound(); // Output: Dog barks.
}
```

Dart Collections

Dart provides a variety of collection types to store and manipulate data.

1. **List**:
   - A List is an ordered collection of elements.
   - Elements can be accessed by their index, starting from zero.
   - Lists can grow or shrink dynamically as elements are added or removed.
   - Dart provides two main types of lists: `List` and `List<T>` (a generic list).
   - Example:
   ```
   List<int> numbers = [1, 2, 3, 4, 5];
   numbers.add(6); // Add an element
   numbers.removeAt(0); // Remove an element by index
   ```
2. **Set**:
   - A Set is an unordered collection of unique elements.
   - Sets do not allow duplicate elements; if you add an element that already exists, it won't be added again.
   - Dart provides two main types of sets: `Set` and `Set<T>` (a generic set).
   - Example:
   ```
   Set<String> fruits = {'apple', 'banana', 'orange'};
   fruits.add('apple'); // Duplicate element won't be added
   ```
3. **Map**:
   - A Map is a collection of key-value pairs.
   - Keys are unique within the map, and each key maps to a single value.
   - Maps provide efficient lookup and retrieval of values based on their keys.
   - Dart provides two main types of maps: `Map` and `Map<K, V>` (a generic map).
   - Example:
   ```
   Map<String, int> ages = {
     'John': 30,
     'Alice': 25,
     'Bob': 35,
   };
   ages['John']; // Access value by key
   ```
4. **Queue**:
   - A Queue is a collection that operates on a first-in, first-out (FIFO) basis.
   - Elements are inserted at the end of the queue and removed from the front of the queue.
   - Dart provides the `Queue` class in the `dart:collection` library.
   - Example:
   ```
   import 'dart:collection';

   Queue<int> queue = Queue();
   queue.add(1); // Add element to the end of the queue
   queue.removeFirst(); // Remove element from the front of the queue
   ```

These collection types in Dart provide developers with flexible and efficient ways to store, manipulate, and access data in their applications, depending on the specific requirements and use cases.

Basics of Asynchronous Programming

Asynchronous programming in Flutter allows us to perform tasks concurrently without blocking the main thread, enabling the app to remain responsive while executing time-consuming operations such as network requests, file I/O, or database queries. Dart, the programming language used in Flutter, provides built-in support for asynchronous programming through features like async and await.

1. **Future**:
    - A Future represents a potential value or error that will be available at some point in the future.
    - It is used to perform asynchronous operations and obtain their results asynchronously.
    - You can use the `then()` method to register a callback that will be called when the future completes.

```
Future<String> fetchData() {
  return Future.delayed(Duration(seconds: 2), () {
    return 'Data fetched successfully';
  });
}

fetchData().then((value) {
  print(value); // Output: Data fetched successfully
});
```

2. **async and await**:
    - The async keyword is used to mark a function as asynchronous, allowing it to use the await keyword.
    - The await keyword is used to pause the execution of a function until a Future completes, and then resumes the function's execution with the result of the Future.

```
Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 2));
  return 'Data fetched successfully';
}

void fetchDataAndPrint() async {
  String data = await fetchData();
  print(data); // Output: Data fetched successfully
}

fetchDataAndPrint();
```

3. **Error handling**:
    - You can use try-catch blocks to handle errors that occur during asynchronous operations.
    - Futures can complete with either a value or an error, and you can use the catchError() method to handle errors.

```
Future<void> fetchData() async {
  try {
    // Simulate an error
    throw Exception('An error occurred');
  } catch (e) {
    print('Error: $e'); // Output: Error: Exception: An error occurred
  }
}

fetchData();
```

4. **Concurrency**:
    - Asynchronous programming allows you to execute multiple asynchronous tasks concurrently.
    - You can use features like Future.wait() to wait for multiple futures to complete simultaneously.

```
Future<void> fetchData() async {
  List<Future<String>> futures = [
    Future.delayed(Duration(seconds: 2), () => 'Data 1'),
    Future.delayed(Duration(seconds: 3), () => 'Data 2'),
  ];

  List<String> results = await Future.wait(futures);
  print(results); // Output: [Data 1, Data 2]
}
```

```
fetchData();
```

Widgets

In Flutter, it is not a component, rather it is a widget. A widget is responsible for a small unit in a Flutter app. As in SPAs, the widget tree has a root widget. This is the widget from which other widgets are stacked upon, meaning there will also be parent widgets and child widgets.

A widget that renders another widget is the parent widget, which renders the the child widget. The MyApp widget is usually the root widget.

```
        MyApp
          |
      Container
          |
         Row
          |
      ----------
      |        |
    Text   FlatButton
```

In the example: The MyApp is the root widget, and it is the parent of all the widgets in the tree. The MyApp widget renders the Container widget, and the Container widget renders the Row widget. The Row widget renders both Text and FlatButton widgets.

So, we can say that MyApp is the parent of Container, and that the Container is the parent of Row and the child of MyApp. The Row is the child of Container and parent of Text and FlatButton. Text and FlatButton are the children of Row.

Stateful vs Stateless widgets

Stateless Widgets: - Don't maintain any internal state. - Immutable, meaning their properties (parameters) cannot change once they are initialized. - Typically used for UI components that don't need to change dynamically based on user interactions or other factors. - Examples: static text, icons, buttons.

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

class MyButton extends StatelessWidget {
  final String text;
  final VoidCallback onPressed;

  const MyButton({required this.text, required this.onPressed});

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      child: Text(text),
    );
  }
}
```

Stateful Widgets: - Maintain internal state that can change over time. - Mutable, meaning they can be rebuilt with different property values or internal states in response to user interactions, changes in data, or other events. - Used for UI components that need to update their appearance or behavior dynamically based on user input, data changes, or other factors. - Examples: forms, interactive elements like sliders and switches, and UI components that display dynamic data fetched from APIs or stored locally.

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

class Counter extends StatefulWidget {
  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _count = 0;

  void _increment() {
    setState(() {
      _count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text('Count: $_count'),
        ElevatedButton(
          onPressed: _increment,
          child: Text('Increment'),
        ),
      ],
    );
  }
}
```

In addition to the states differences it also involves that stateful widgets have **lifecycle methods**.

Lifecycle methods

Methods for StatefulWidgets:

1. **initState()**:
    - Called when the stateful widget is inserted into the widget tree for the first time.
    - Typically used to initialize state variables or to perform any setup that needs to be done once when the widget is created.
2. **didChangeDependencies()**:
    - Called immediately after initState() and whenever the dependencies of the widget change.
    - It is often used to perform tasks that rely on the widget's context or to subscribe to data streams from inherited widgets.
3. **build()**:
    - This method is called whenever the widget needs to rebuild its UI, which can occur due to changes in state or when the parent widget rebuilds.
    - It is where you define the structure and layout of the widget's UI by returning a widget tree.
4. **didUpdateWidget()**:
    - Called whenever the widget is rebuilt with new properties.
    - It is useful for comparing the old and new widget properties and reacting to changes.
5. **setState()**:
    - This method is used to update the state of the widget and trigger a rebuild of the widget's UI.
    - It is typically called in response to user interactions or asynchronous events that change the widget's state.
6. **dispose()**:

  - Called when the stateful widget is removed from the widget tree and destroyed.
  - It is often used to release any resources held by the widget, such as closing streams or canceling timers.

For StatelessWidget, there's only one lifecycle method:

1. **build()**:
   - Since stateless widgets don't have mutable state, they only need to define their UI structure in the build method, which is called whenever the widget needs to rebuild its UI.

   These lifecycle methods provide hooks for performing various tasks at different stages of the widget's lifecycle, allowing you to manage state, perform setup and cleanup, and update the UI as needed.

BuildContext

It represents the location of a widget within the widget tree. It provides access to information about the widget's location and allows you to interact with other widgets in the tree. Everything in Flutter is a widget. Whether it is a container, text, button, providers, image, etc., everything is virtually a widget, no matter if they display a UI in the app or not.

> In Flutter, the visual presentation, known as the user interface (UI), is constructed from stacks of widgets commonly referred to as a "widget tree.

Key points:

1. **Location in the Widget Tree**:
   - Each widget in the Flutter widget tree has its own BuildContext, which identifies its position within the tree.
   - The BuildContext represents the widget's relationship with its parent, siblings, and children in the widget hierarchy.
2. **Accessing Theme and Localization**:
   - BuildContext provides access to the theme data and localization data defined higher up in the widget tree using the Theme and MaterialApp/WidgetsApp widgets.
   - This allows widgets to retrieve theme-related information such as colors, fonts, and text direction based on their location in the widget tree.
3. **Navigating the Widget Tree**:
   - BuildContext allows you to navigate up and down the widget tree to access parent, sibling, or child widgets.
   - You can use methods like `BuildContext.parent`, `BuildContext.ancestorWidgetOfExactType`, and `BuildContext.findAncestorStateOfType` to locate specific widgets or their associated state objects.
4. **Building Widgets**:
   - BuildContext is passed to the build() method of stateless and stateful widgets, allowing them to build their UI based on their current context.
   - Widgets use the context to obtain information about the widget tree and its configuration when constructing their UI.
5. **Creating New Widgets**:
   - BuildContext is also used to create new widgets within the build() method of existing widgets.
   - You can use methods like `BuildContext.build` or `BuildContext.inheritFromWidgetOfExactType` to create and return new widgets with specific configurations.

Packages vs plugins

What packages did you use?

- GraphQL Flutter
- Cupertino Icons
- Google Fonts
- SVG Flutter
- Flutter Native Splash ## Packages: A package is a collection of Dart files that provides a set of functionalities or features that can be easily integrated.

- Can be published to the Dart package repository (pub.dev) for others to use, or they can be private and used only within your own projects.
- Typically contain reusable code, such as utility functions, UI components, API wrappers, or other Dart libraries. > Examples of packages include **http** for making HTTP requests, **shared_preferences** for storing simple data persistently, and **flutter_bloc** for implementing the BLoC state management pattern.

Plugins:

Plugins in Flutter are similar to packages but are specifically designed to interact with native code on the platform level (Android or iOS). Plugins typically wrap native code libraries (written in Java/Kotlin for Android or Objective-C/Swift for iOS) and expose their functionalities to Dart code in Flutter. Plugins are used when you need to access platform-specific features that are not available through Flutter's existing APIs. > Examples of plugins include **camera** for accessing the device's camera, **firebase_core** for integrating Firebase services, and **location** for retrieving the device's location.

Hot restart vs Hot reload

Hot Reload:

Developers make changes to their Flutter code and see those changes reflected almost instantly in the running app. Tries to only changed the widgets and keep the state. With Hot Reload, the app state is preserved, meaning that any stateful changes (such as text entered into a text field) are maintained across reloads. It's a quick way to iterate on UI changes, fix bugs, and experiment with code without losing the current state of the app.

Hot Restart:

Completely restarts the Flutter app. Recompiled and restarted from scratch. The app state is reset to its initial state. This means that any stateful changes made during the previous session are lost. Re run the *main function*. Hot Restart is useful when you've made changes that require modifications to the app's structure, such as adding or removing dependencies, modifying the app's entry point, or changing platform-specific code. It involves recompiling and restarting the entire app. Both offer different advantages depending on the situation.

Build Types

It's refer to different configurations of app's build settings for different environments, such as development, testing, staging, and production. These build types are defined in your app's build.gradle file (for Android) and Info.plist file (for iOS), and they determine various aspects of the build process, including optimization levels, signing configurations, and resource handling.

Debug

Used for development purposes. It typically includes debugging symbols, minimal optimizations, and additional development features like *hot reload*. Debug builds are not optimized for performance but provide a fast development cycle. ## Release Optimized for performance and size. They exclude debugging symbols and enable various compiler optimizations to make the app run faster and consume less memory. *Release builds are suitable for production deployment.* ## Profile Optimized for profiling and performance analysis. They include some level of optimization but still retain debugging information to allow for profiling and performance monitoring tools to gather data. Profile builds are useful for identifying performance bottlenecks in your app. ## Custom Build Types Tailored for specific requirements. For example, you might create a "staging" build type with configurations specific to your staging environment, such as pointing to a different API endpoint or enabling additional logging.

> Each build type can have its own set of configurations, such as signing keys, package names, and resource directories, allowing you to customize your app's behavior and appearance for different deployment scenarios. When you run `flutter run` or `flutter build`, you can specify the desired build type using the `--release`, `--profile`, or `--debug` flags, or by specifying a custom build type if you've defined one.

runApp() vs main()

They serve different purposes: ## `main()` function - The entry point of a Dart application, including Flutter apps. - It's similar to the `main()` function in many other programming languages, except Javascript and Python - Starting point of the program's execution. - In Flutter, the `main()` function typically calls `runApp()` to start the app's execution by passing it the root widget of the application.

`runApp()` function

- `runApp()` is a function provided by the Flutter framework, and its purpose is to run the given widget as the root of the widget tree.
- It initializes the Flutter framework and starts the execution of the Flutter application.
- `runApp()` takes a widget (usually a `MaterialApp` or `CupertinoApp`) as its argument, which represents the root of the widget tree for the Flutter application.
- Widgets in Flutter are hierarchical, and the widget passed to `runApp()` serves as the top-level widget, containing all other widgets in the application.

Typical structure of a Flutter application:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'My Flutter App',
      home: MyHomePage(),
    );
  }
}

class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home Page'),
      ),
      body: Center(
        child: Text('Hello, Flutter!'),
      ),
    );
  }
}
```

In this example, `main()` is the entry point of the Flutter app, and it calls `runApp()` with an instance of `MyApp` as its argument. `MyApp` is a widget that extends `MaterialApp` and serves as the root of the widget tree. The `MaterialApp` widget then contains `MyHomePage` as its `home` property, and `MyHomePage` contains the UI elements of the home page.

UI

The user interface (UI) is built using a variety of widgets that represent different elements, such as layouts, controls, text, images, and more. Some common UI widgets in Flutter:

1. **Scaffold**:
    - Scaffold is a layout widget that provides a basic structure for implementing material design layouts.
    - It typically includes features like an app bar, a body, floating action buttons, drawers, and bottom navigation bars.
    - Scaffold simplifies the process of creating common app layouts by providing pre-defined components and layout structure.

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

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

    class MyApp extends StatelessWidget {
      @override
      Widget build(BuildContext context) {
        return MaterialApp(
          home: Scaffold(
            appBar: AppBar(
              title: Text('Scaffold Example'),
            ),
            body: Center(
              child: Text('Hello, Flutter!'),
            ),
          ),
        );
      }
    }
    ```
2. **Row**:
    - Row is a layout widget that arranges its children widgets horizontally in a single row.
    - Children widgets are positioned side by side, and their sizes can be adjusted using properties like mainAxisAlignment and crossAxisAlignment.

    ```
    Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text('Hello'),
        Text('World'),
      ],
    )
    ```
3. **Column**:
    - Column is a layout widget that arranges its children widgets vertically in a single column.
    - Children widgets are stacked one on top of the other, and their sizes can be adjusted using properties like mainAxisAlignment and crossAxisAlignment.

    ```
    Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text('Hello'),
        Text('World'),
      ],
    )
    ```
4. **Container**:
    - Container is a versatile layout widget that allows you to customize its appearance, size, padding, margin, and decoration.
    - It can contain a single child widget and is often used to apply styling and layout properties to its child.

    ```
    Container(
      color: Colors.blue,
      padding: EdgeInsets.all(16.0),
    ```

```
    child: Text('This is a container'),
  )
```

A list of common widgets in Flutter:

Of course! Here’s the list of Flutter widgets and their descriptions formatted into a 2-column table:

Widget Name	Brief Description
AbsorbPointer	Ignores pointer events but passes them along
ActionChip	Represents a chip that performs an action
AlertDialog	Dialog with a title, content, and actions
Align	Aligns its child within itself
Alignment	Represents a point in a 2D coordinate system
AlignmentDirectional	Represents a point in a 2D coordinate system (RTL support)
AnimatedBuilder	Animates a widget over a period of time
AnimatedContainer	Animates the transformation of a container
AnimatedCrossFade	Cross-fades between two children
AnimatedDefaultTextStyle	Animates the default text style
AnimatedList	A scrolling container that animates items
AnimatedOpacity	Animates the opacity of a widget
AnimatedPadding	Animates its padding
AnimatedPhysicalModel	Animates the parameters of a physical model
AnimatedPositioned	Animates the position of a widget
AnimatedSize	Animates the size of a widget
AnimatedSwitcher	Switches between two children with a fade
AnimatedTheme	Animates the properties of a theme
AppBar	Material design app bar
AspectRatio	Adjusts its child to match a given aspect ratio
AssetsImage	Displays an image from an asset
BackButton	A button with a “back” icon
BottomAppBar	An app bar that integrates with a bottom navigation bar
BottomNavigationBar	A bottom navigation bar
BottomNavigationBarItem	An item within a bottom navigation bar
BottomSheet	A panel that slides in from the bottom
ButtonBar	A horizontal arrangement of buttons
ButtonBarThemeData	Defines the theme for ButtonBar widgets
ButtonTheme	Applies a button theme to descendant widgets
ButtonThemeData	Defines the theme for buttons
Card	Material design card
Center	Aligns its child to the center of the parent
Checkbox	A material design checkbox
Chip	Represents a small, interactive, material design element
ChipThemeData	Defines the theme for Chip widgets
CircleAvatar	A circular user profile picture
ClipOval	Clips its child to an oval shape
ClipPath	Clips its child using a custom clipper
ClipRect	Clips its child to a rectangular shape
CloseButton	A button with a “close” icon
CloseButtonIcon	Icon used by the CloseButton
CloseButtonTheme	Theme for CloseButton
ColorScheme	Defines a color scheme for a set of UI components
Column	A vertical arrangement of widgets
ConstrainedBox	Forces its child to have a specific width and/or height
Container	A convenience widget for styling and positioning
CrossAxisAlignment	Aligns children along the cross axis
CupertinoActionSheet	A sheet of options displayed to the user
CupertinoActivityIndicator	An iOS-style activity indicator

Widget Name	Brief Description
CupertinoAlertDialog	An iOS-style alert dialog
CupertinoButton	An iOS-style button
CupertinoColors	Cupertino color constants
CupertinoContextMenuAction	An action in a context menu
CupertinoContextMenuActionStyle	Style for CupertinoContextMenuAction
CupertinoContextMenuDivider	Divider in a context menu
CupertinoContextMenuSection	A section in a context menu
CupertinoContextMenuSheet	A context menu sheet
CupertinoDatePicker	An iOS-style date picker
CupertinoDialogAction	A button used in a Cupertino dialog
CupertinoDialogThemeData	Defines the theme for CupertinoDialog widgets
CupertinoIcons	Cupertino icon constants
CupertinoPageScaffold	An iOS-style page layout
CupertinoPicker	An iOS-style picker control
CupertinoPopupSurface	A rounded surface with an elevation, typically used for pop-up menus
CupertinoScrollbar	A scrollbar with a Cupertino style
CupertinoSlider	An iOS-style slider control
CupertinoSwitch	An iOS-style switch control
CupertinoTabBar	An iOS-style tab bar
CupertinoTabScaffold	An iOS-style tabbed navigation layout
CupertinoTabView	An iOS-style tab view
CupertinoTextField	An iOS-style text field
CupertinoTheme	Cupertino theme data
CupertinoTimerPicker	An iOS-style timer picker
CustomMultiChildLayout	A widget that uses a delegate to size and position multiple children
CustomPaint	Provides a canvas on which to draw during the paint phase
CustomScrollView	A ScrollView that creates custom scroll effects using slivers
DataTable	Displays data in a tabular format
DateUtils	Date utilities for working with dates
DecoratedBox	Decorates its child with a decoration
DefaultTabController	Manages the state of a TabBar and TabBarView
Dismissible	A widget that can be dismissed by dragging
Divider	A thin horizontal line
Drawer	A material design drawer
DrawerHeader	A drawer header with an optional account name, email, and current account picture

State management (Provider, BloC)

State management refers to the management of data and UI state within an application. There are various approaches to state management in Flutter, two of the most popular being Provider and BLoC (Business Logic Component).

1. **Provider**:
    - Simple, flexible, and lightweight state management solution that is built into Flutter. It follows the InheritedWidget pattern to propagate data down the widget tree.
    - With Provider, you can use a Provider class to expose data to descendant widgets and rebuild UI components whenever the underlying data changes.
    - Provider is often favored for its simplicity and ease of use, especially for smaller to medium-sized applications or when you have a relatively small amount of shared state.
    - It's worth noting that Provider can be used in conjunction with other state management techniques, such as ChangeNotifier or Riverpod, to handle more complex state scenarios.
2. **BLoC (Business Logic Component)**:
    - BLoC is a pattern for managing state in Flutter applications, focusing on separating business logic from presentation logic.
    - With BLoC, you create classes called "BLoCs" that are responsible for managing the application's state

and business logic. These BLoCs typically expose Streams or StreamControllers to communicate changes in state to UI components.

- BLoC often works hand-in-hand with Streams and StreamBuilder widgets to listen for changes in state and update the UI accordingly.
- BLoC is favored for its separation of concerns and scalability, making it suitable for larger and more complex applications where state management needs to be more structured and maintainable.