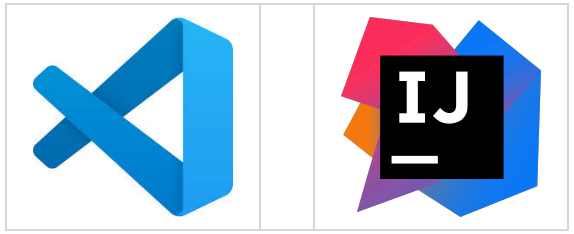
Day 4:
Basics of Flutter

Setting Up Flutter Environment

Recommended IDEs

- Visual Studio Code
- IntelliJ IDEA



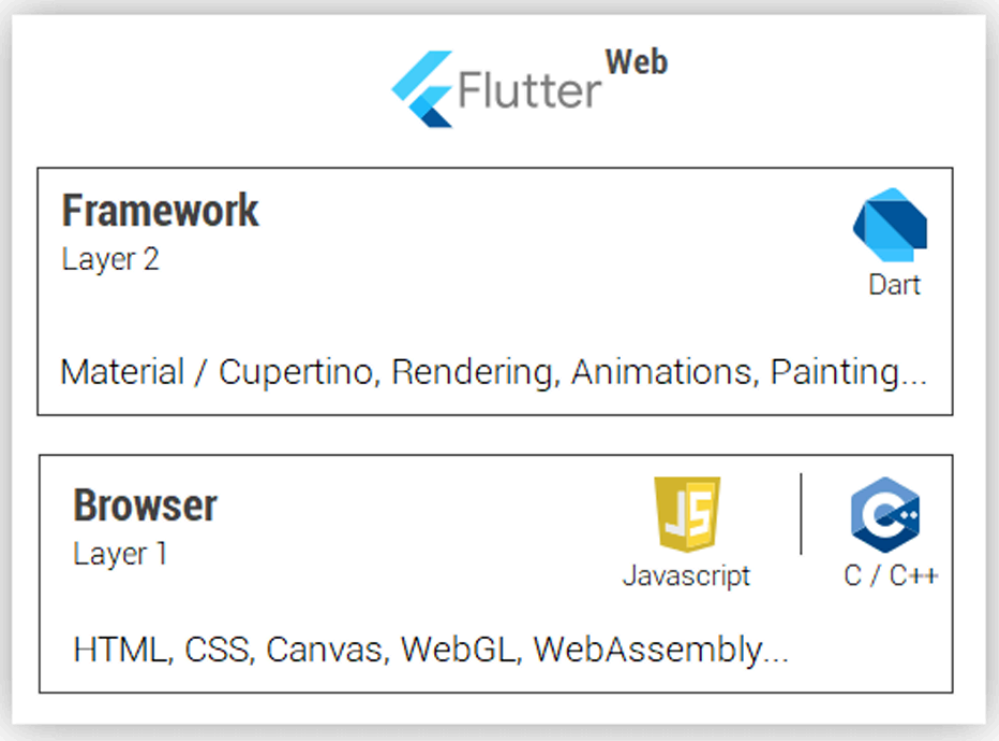
Flutter SDK (Software Development Kit)

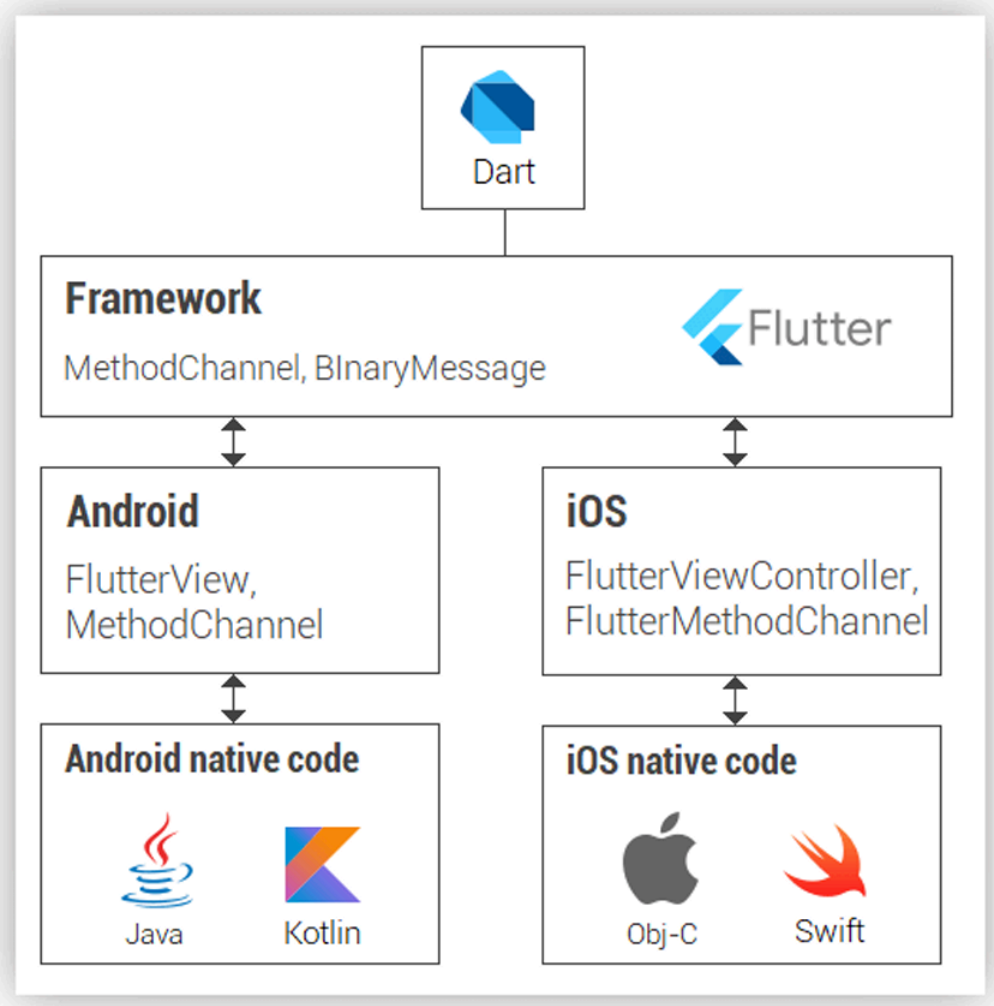
- Flutter SDK

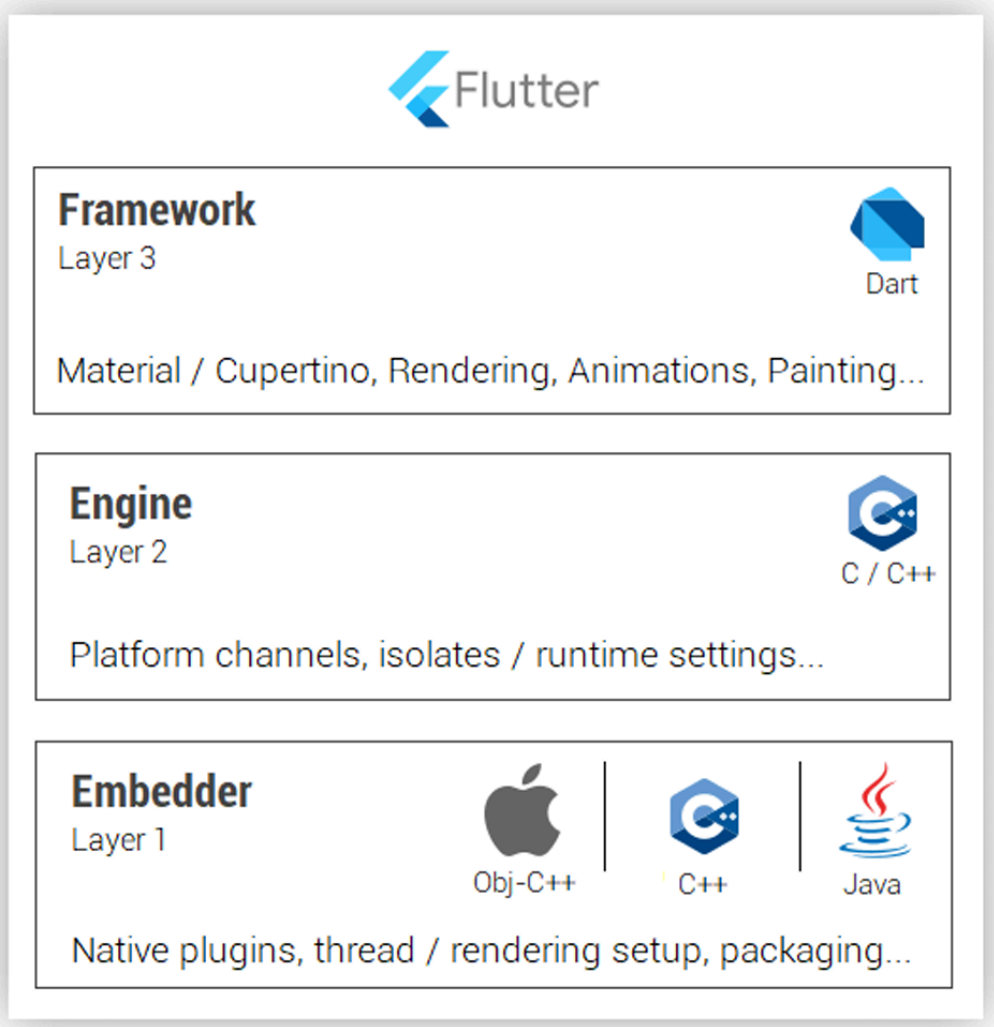
Platform SDKs

- For Android: Android Studio
- For iOS and MacOS: Xcode
- For Linux: GCC (GNU Compiler Collection)
- For Windows: Visual Studio C++ Build Tools

Flutter Framework Architecture

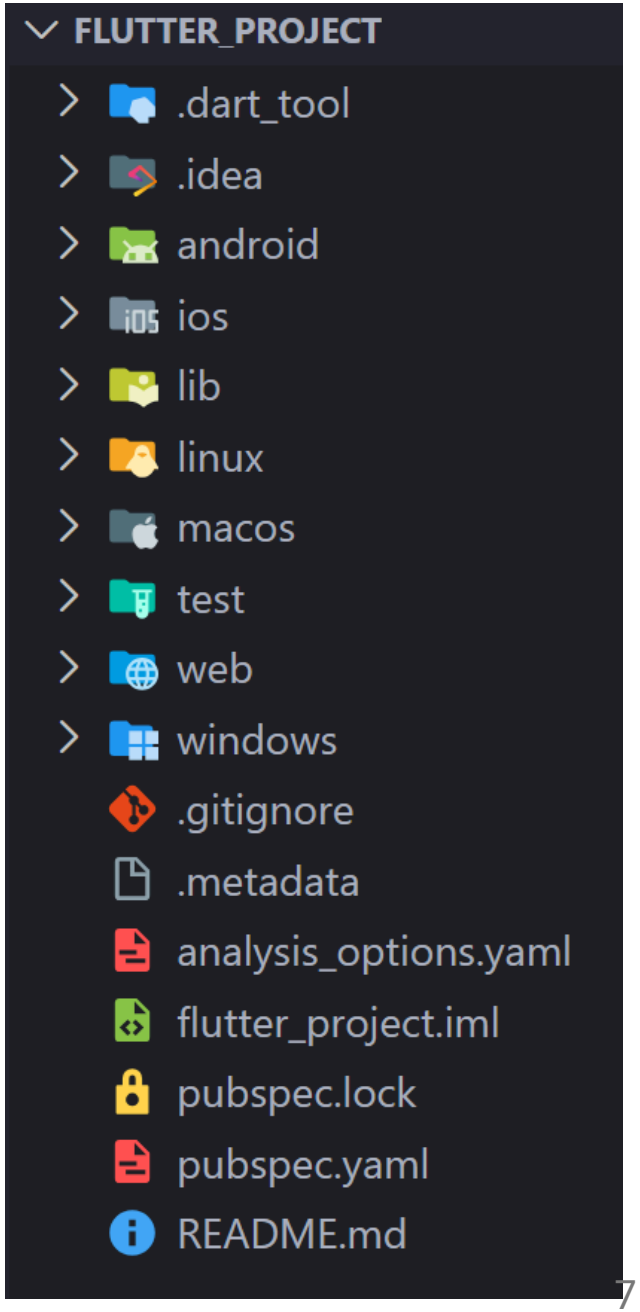



Dart
Framework
MethodChannel, BInaryMessage
Flutter
Android
FlutterView,
MethodChannel
iOS
FlutterViewController,
FlutterMethodChannel
Android native code
Java
Kotlin
iOS native code
Obj-C
Swift


Flutter
Framework
Layer 3
Dart
Material / Cupertino, Rendering, Animations, Painting...
Engine
Layer 2
C / C++
Platform channels, isolates / runtime settings...
Embedder
Layer 1
Obj-C++
C++
Java
Native plugins, thread / rendering setup, packaging...

Flutter Project Structure

- `.dart_tool/` – Auto-generated build files
- `android/` – Android platform code
- `ios/` – iOS platform code
- `web/` – Web platform code (if web enabled)
- `windows/` – Windows platform code (if Windows enabled)
- `macos/` – MacOS platform code (if MacOS enabled)
- `linux/` – Linux platform code (if Linux enabled)
- `lib/` – Main Dart code (app logic & UI)
- `test/` – Unit and widget tests
- `pubspec.yaml` – Project metadata & dependencies
- `analysis_options.yaml` – Linter rules & analysis settings



`.dart_tool/` Folder

`.dart_tool/` is an auto-generated folder created by Dart and Flutter tools. It contains:

- Build artifacts
- Package configurations
- Tool-specific data
- Caches for faster builds
- Dependency resolution files
- Other metadata

`android/` Folder

The `android/` folder contains the Android-specific code and configuration files for your Flutter app, including:

- `app/src/main/` – Main source code and resources
- `build.gradle` – Build configuration files
- `AndroidManifest.xml` – App metadata and permissions
- `res/` – Android resources (images, layouts, etc.)
- `gradle/` – Gradle wrapper files
- `proguard-rules.pro` – Code obfuscation rules
- etc.

`ios/` Folder

The `ios/` folder contains the iOS-specific code and configuration files for your Flutter app, including:

- `Runner.xcodeproj/` – Xcode project file
- `Runner/` – Main source code and resources
- `Info.plist` – App metadata and permissions
- `Assets.xcassets/` – iOS assets (images, icons, etc.)
- `Podfile` – CocoaPods dependencies
- `AppDelegate.swift` or `AppDelegate.m` – App lifecycle management
- `LaunchScreen.storyboard` – Launch screen design
- etc.

`web/` Folder

The `web/` folder contains the web-specific code and configuration files for your Flutter app, including:

- `icons/` – Web app icons
- `favicon.png` – Favicon image
- `index.html` – Main HTML file
- `manifest.json` – Web app manifest
- etc.

`windows/` Folder

The `windows/` folder contains the Windows-specific code and configuration files for your Flutter app, including:

- `runner/` – Main source code and resources
- `CMakeLists.txt` – Build configuration file
- `App.cpp` – App lifecycle management
- `resource.h` – Resource definitions
- etc.

`macos/` Folder

The `macos/` folder contains the MacOS-specific code and configuration files for your Flutter app, including:

- `Runner.xcodeproj/` – Xcode project file
- `Runner/` – Main source code and resources
- `Info.plist` – App metadata and permissions
- `Assets.xcassets/` – MacOS assets (images, icons, etc.)
- `AppDelegate.swift` or `AppDelegate.m` – App lifecycle management
- `MainMenu.xib` – Main menu design
- etc.

`linux/` Folder

The `linux/` folder contains the Linux-specific code and configuration files for your Flutter app, including:

- `runner/` – Main source code and resources
- `CMakeLists.txt` – Build configuration file
- `App.cpp` – App lifecycle management
- `resource.h` – Resource definitions
- etc.

The `pubspec.yaml` File

YAML is a human-readable data-serialization language often used for configuration files, relying on indentation and line breaks instead of punctuation.

Core Sections

- `name` & `version` – App/package identity
- `environment` – SDK constraints (e.g., `>=2.7.0 <3.0.0`)
- `dependencies` – External packages (e.g., `http`, `provider`)
- `assets` – Images, fonts, audio files
- `fonts` – Custom font definitions

Example

```
dependencies:
  flutter:
    sdk: flutter
  http: ^0.12.2
  provider: ^4.3.2+2
```

`analysis_options.yaml` File

`analysis_options.yaml` is a configuration file for the Dart analyzer that allows you to customize linting rules and analysis settings for your Flutter project.

Benefits

- Catches errors early
- Enforces best practices
- Improves code consistency

Example

```
analyzer:
  strong-mode:
    implicit-casts: false
    implicit-dynamic: false
linter:
  rules:
    - avoid_unused_constructor_parameters
    - await_only_futures
```

Managing Assets in Flutter

Declaring Assets

```yaml
flutter:
  assets:
    - assets/images/logo.png
    - assets/data/sample.json
```

Accessing Assets in Code

```dart
Image.asset('assets/images/logo.png');
String data = await rootBundle.loadString('assets/data/sample.json');
```

Managing Fonts in Flutter

Two Approaches

1. **Download & Bundle** – Add `.ttf` files to `fonts/` folder and declare in `pubspec.yaml`
2. **Google Fonts Package** – Use `google_fonts` package for dynamic loading

Recommendation

- Use `google_fonts` during development
- Bundle fonts as assets before publishing for offline support

Hot Reload – Fast Development

Hot reload is a Flutter feature that allows developers to quickly see code changes reflected in the running app without restarting it.

When to Use

- After most code changes
- UI styling adjustments

When to Full Restart

- Changes to `initState()`
- Modifying static fields
- Changing `main()` method

Android Studio

VS Code

Tree Shaking with Build Modes

Tree shaking is an optimization technique that removes unused code from the final app binary, reducing its size.

Constants for Build Modes

- `kDebugMode` – Debug builds
- `kReleaseMode` – Release builds

Example

```
String get name {
  if (kDebugMode) return "Demo"; // Removed in release
  else return _real();
}
```

Why Flutter is Special?

1. Everything in Flutter are Widgets

- Layout, UI elements, even the app itself are widgets
- Widgets can be combined to create complex UIs

2. All Widgets in Flutter form as Tree

- UI built using nested widgets
- Parent-child relationships define layout and behavior

Basic Structure

```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(...);
  }
}
```

Basic Layouts and Widgets

Common Layouts

- `Row` – Horizontal layout
- `Column` – Vertical layout
- `Spacer` – Flexible space
- `SizedBox` – Fixed size box
- etc.

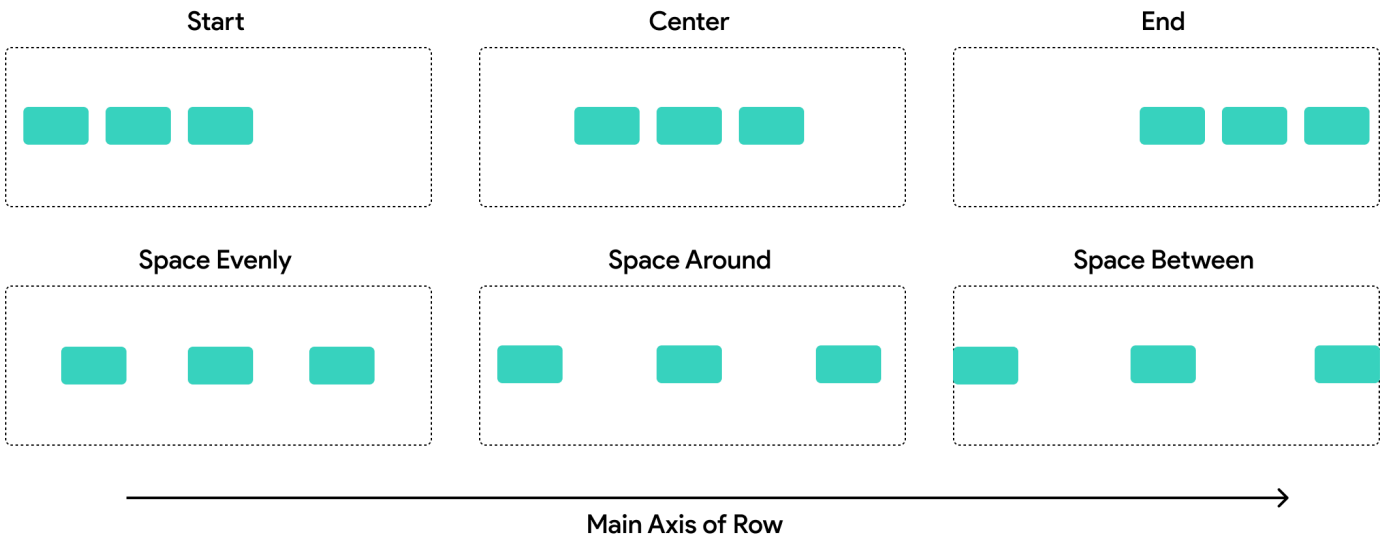
Common Widgets

- `Text` – Display text
- `Image` – Display images
- `Buttons` – Interactive buttons
- `TextField` – Input fields
- etc.

`Row` Widget:

`Row` layout arranges its children horizontally.

```
Row(
  children: const [
    Text("Widget #1"),
    Text("Widget #2"),
  ],
)
```



`Column` Widget:

`Column` layout arranges its children vertically.

```
Column(
  children: const [
    Text("Widget #1"),
    Text("Widget #2"),
  ],
)
```

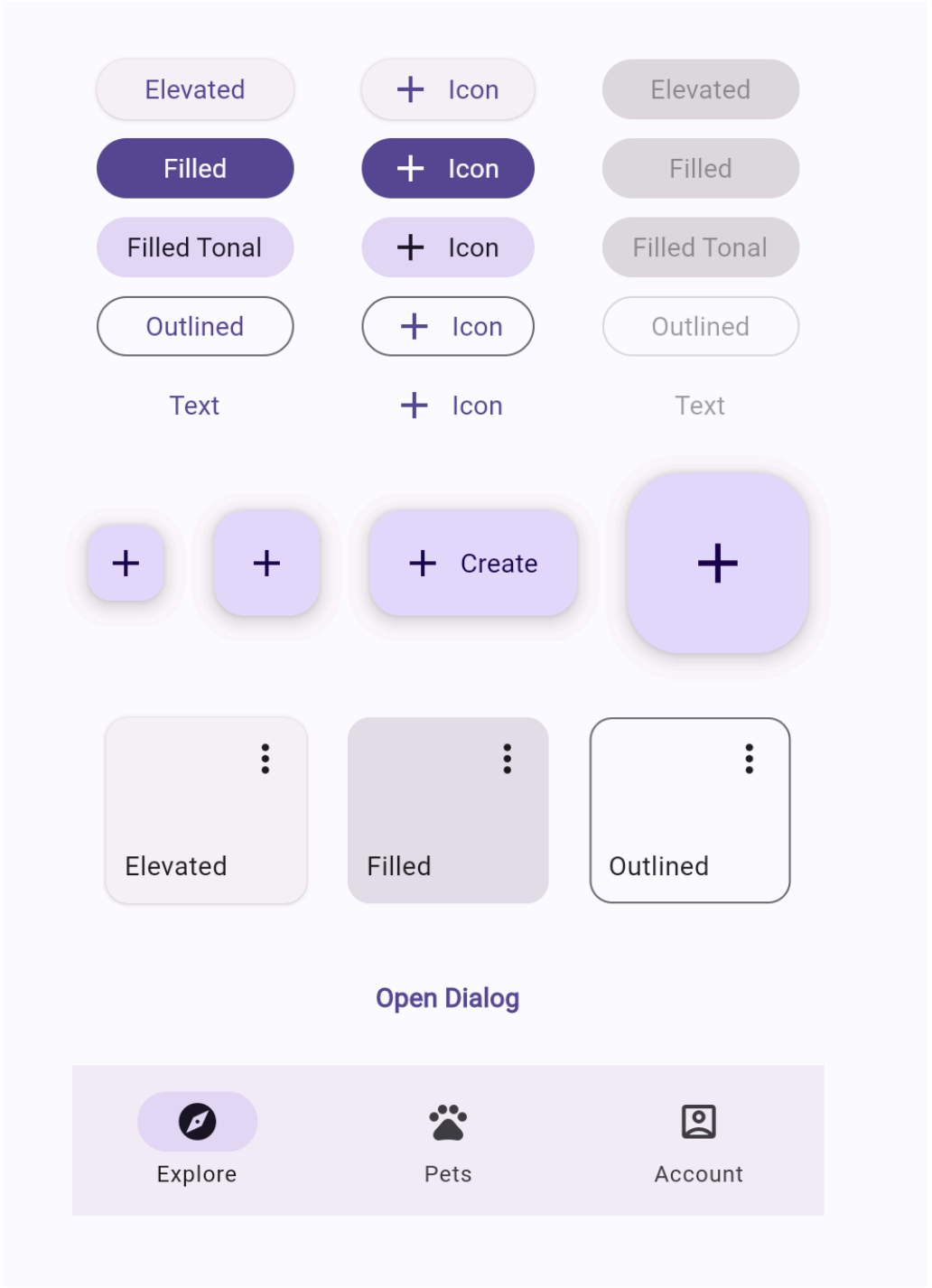
`Text` Widget:

`Text` widget displays a string of text with single style.

```
Text("Hello World"),
```

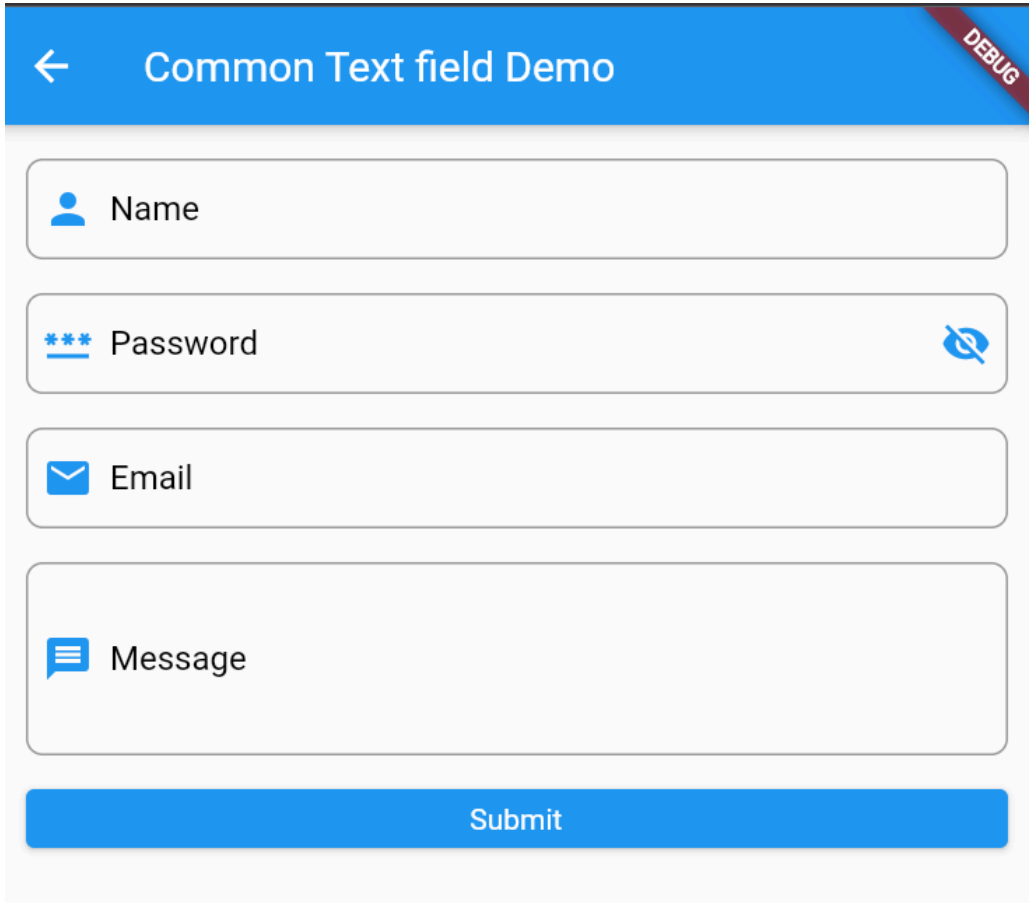
`Button` Widget:

```
ElevatedButton(
  onPressed: () {},
  child: Text("Press Me"),
)
```



`TextField` Widget:

```
TextField(
  decoration: InputDecoration(
    labelText: "Enter your name",
  ),
)
```



Stateless Widgets

Stateless Widgets are widgets that do not require mutable state. It means that once they are built, they cannot change their appearance or behavior based on user interactions or other factors.

- Immutable UI
- No internal state changes
- `final` fields + `const` constructor

Stateful Widgets

Stateful Widgets are widgets that maintain mutable state. They can change their appearance or behavior in response to user interactions or other factors.

- Mutable UI
- Internal state can change
- Uses `setState()` to trigger rebuilds

Stateless Widget Pattern

Stateless widgets have a simple structure with a single class that extends `StatelessWidget` and overrides the `build` method.

```
class ExampleWidget extends StatelessWidget {
  const ExampleWidget();
  @override
  Widget build(BuildContext context) {
    return Text("Hello, Stateless!");
  }
}
```

Stateful Widget Pattern

Stateful widgets consist of two classes: one that extends `StatefulWidget` and another that extends `State<T>` where `T` is the name of the `StatefulWidget` class. The state class holds the mutable state and overrides the `build` method.

```
class Counter extends StatefulWidget {
  const Counter();
  @override
  _CounterState createState() => _CounterState();
}
class _CounterState extends State<Counter> {
  int _count = 0;
  @override
  Widget build(BuildContext context) {
    return Text("$_count");
  }
}
```

Decision Guide

- Use `StatelessWidget` when UI is static
- Use `StatefulWidget` when UI changes dynamically

Widget Keys

Widgets can be assigned **keys** to preserve their state and identity across rebuilds.

There are several types of keys available in Flutter:

- `UniqueKey` – Only equal to itself
- `GlobalKey` – App-wide unique key
- etc.

Example

```
class ExampleWidget extends StatelessWidget {
  final String id; // Unique identifier
  const ExampleWidget({required this.id}) : super(key: ValueKey(id));
  @override
  Widget build(BuildContext context) {
    return Text("Widget with ID: $id");
  }
}
```

Optimization – `const` Constructors

Why Use `const`:

- Widgets built only once
- Cached for subsequent rebuilds
- Improves performance significantly

Example:

```
ListView(
  children: const [
    ExampleWidget(),  // Built once
    ExampleWidget(),  // Reused
    ExampleWidget(),  // Reused
  ],
)
```

Optimization – Widgets vs Functions

Always Use Widget Class

```
class FooterWidget extends StatelessWidget {
  const FooterWidget();
  @override Widget build(...) => Column(...);
}
```

Never Use Function

```
Widget footerWidget() => Column(...); // BAD!
```

Reason

- Widgets can be `const` and cached
- Functions rebuild every time
- Widgets have `BuildContext`

Practical Notes

1. **Organize code** with clear folder structure
2. **Use** `pubspec.yaml` for dependencies & assets
3. **Leverage hot reload** for fast development
4. **Choose correct widget type** (stateless vs stateful)
5. **Optimize with** `const` **constructors**
6. **Understand Flutter's architecture** for better debugging

END