CMPS 312

Flutter Fundamentals

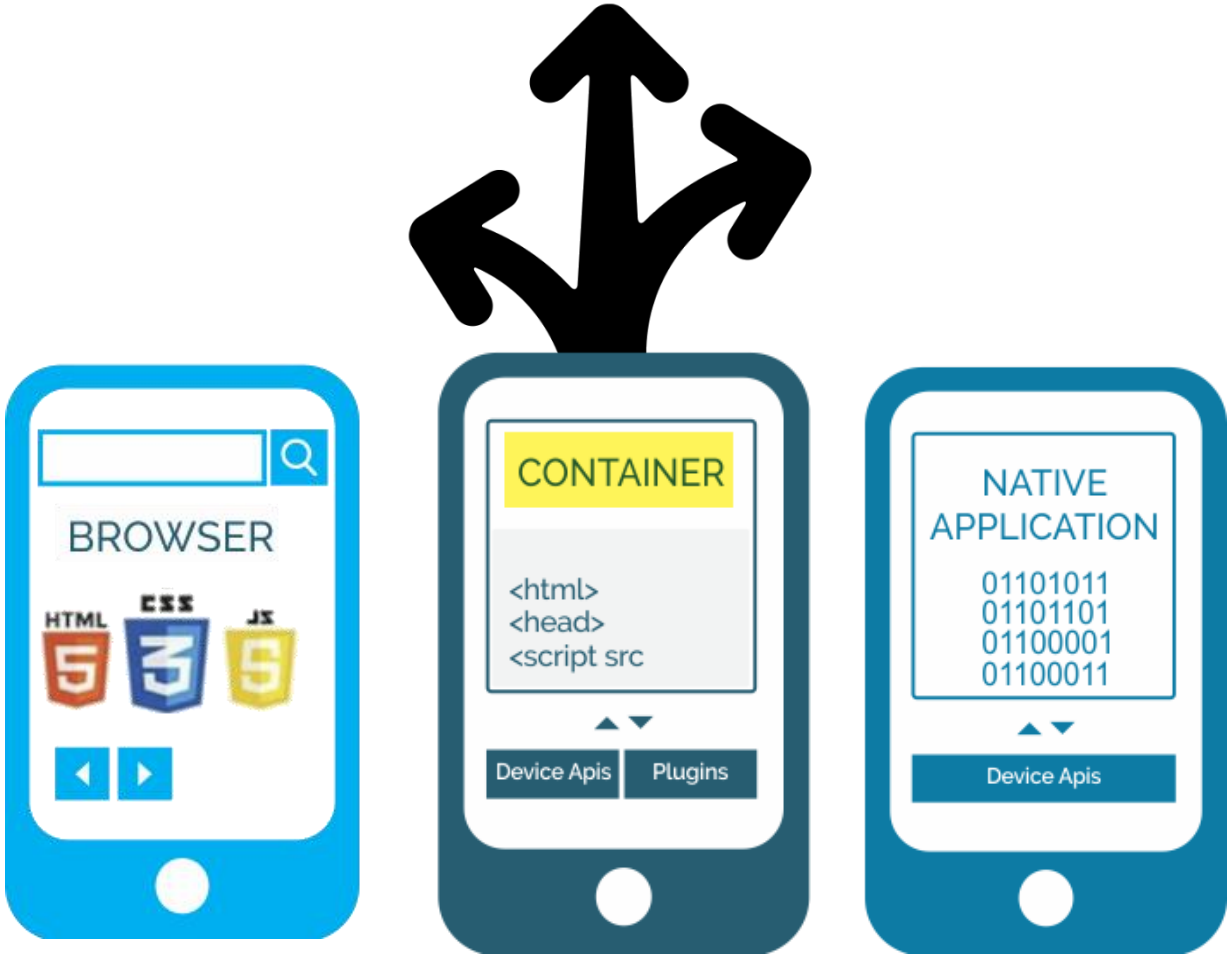
Dr. Abdelkarim Erradi

CSE@QU

Outline

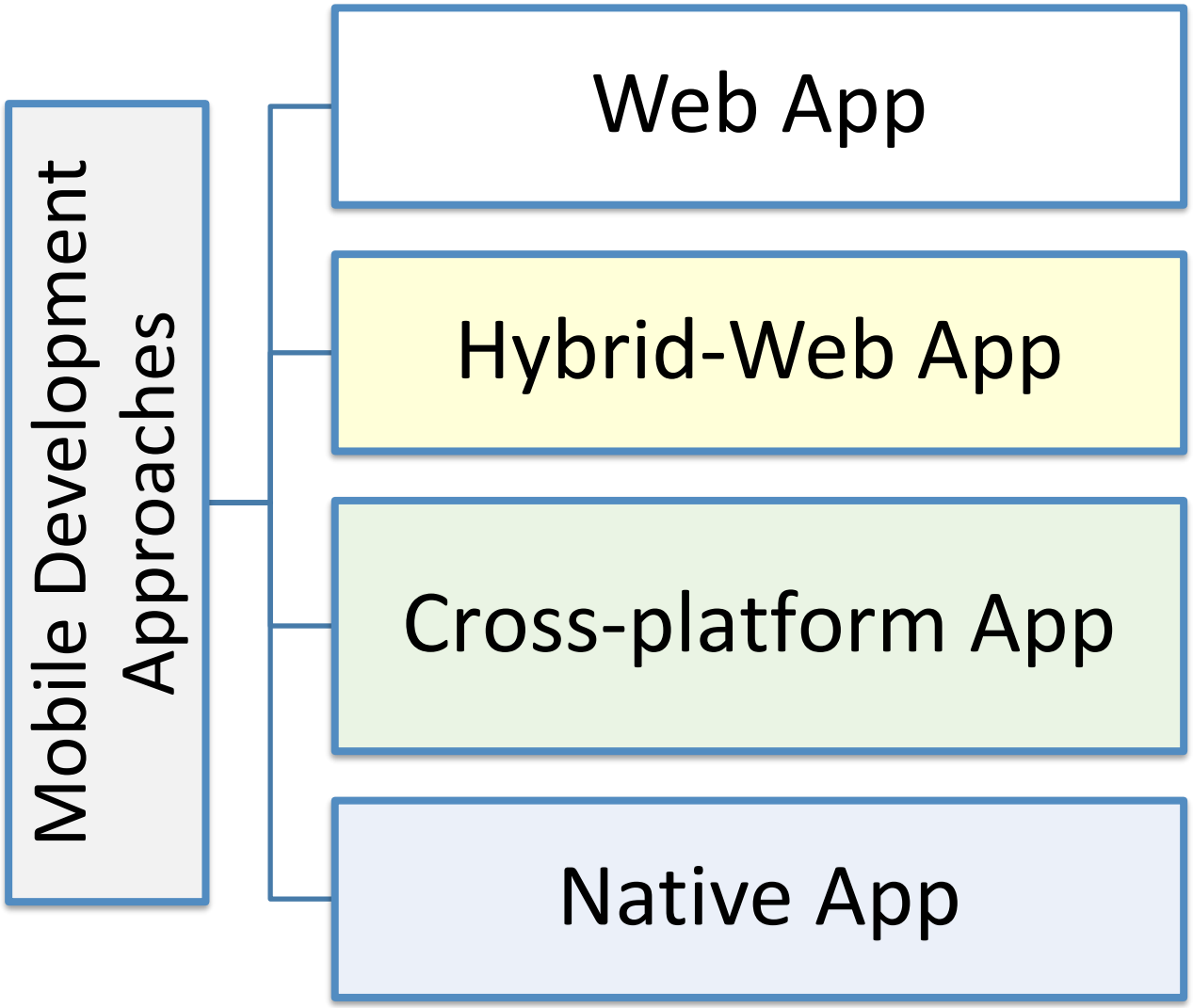
1. Mobile Development Approaches
2. Introduction to Flutter
3. Flutter Key Concepts
4. Widgets
5. Layouts
6. App State Management

Mobile Development Approaches



Back

Mobile Development Approaches



Web App

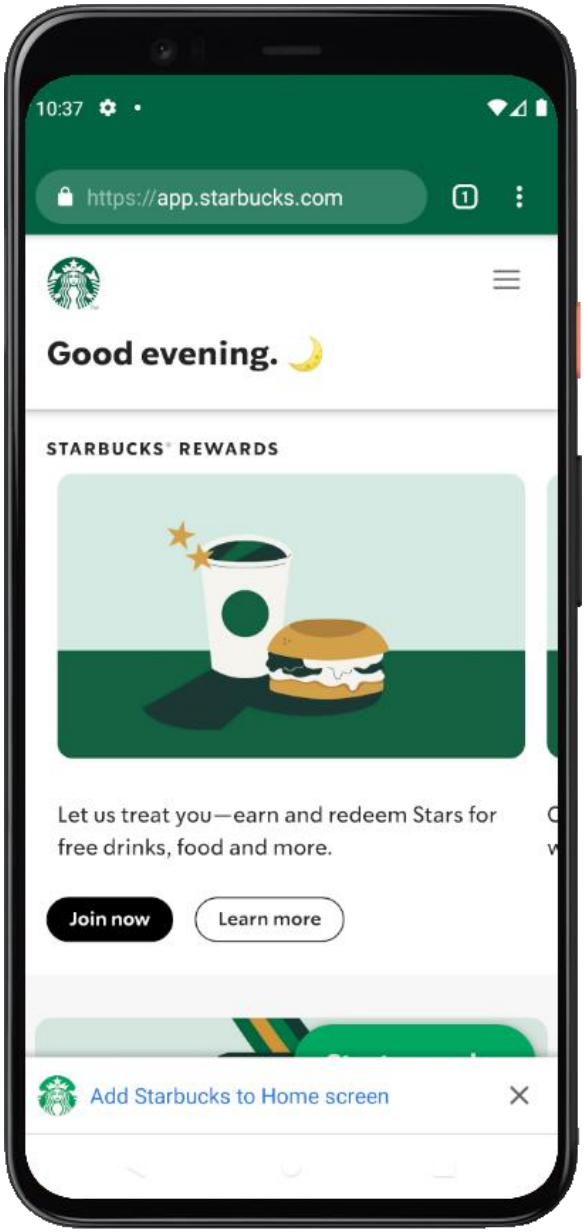
- Responsive Web app adapted to any screen size
- Can be added to Home screen & can work on any platform
- Experience feels like a native app

✓ Can work offline, provide limited access to device's features, such as camera, microphone, location, and notifications

– Slower performance (Run inside a WebView)

– Least access to hardware, sensors, OS

– Not available from the app stores



Hybrid-Web App

- Hybrid-Web Apps: apps blend
  - Mobile-optimized UI components (written using HTML, CSS, and JavaScript) with
  - Native modules or **bridge plugins** for accessing Camera, Geolocation, Bluetooth and other services

✓ Lower development costs (Single codebase)

✓ Multiplatform - Write once, run anywhere

✓ Downloadable from app stores

– Slower performance (not suitable for CPU-intensive apps such as 3D games)

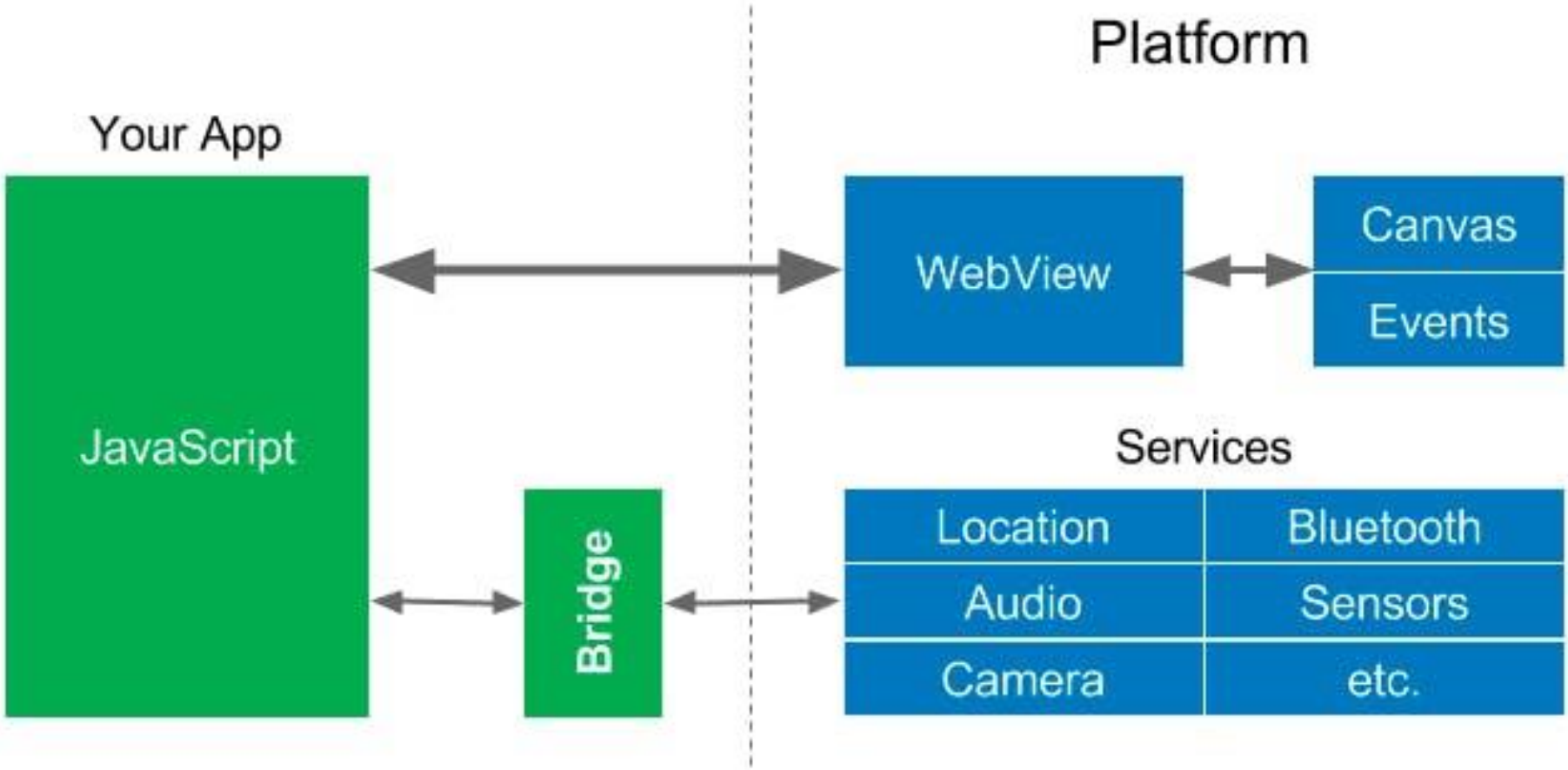
– Highly dependent on libraries and frameworks

IONIC

APACHE CORDOVA™

Hybrid-Web App

- App runs inside a **WebView** responsible for UI Rendering
- App access the platform services via a **bridge**



https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/

Cross-platform App

- Cross-platform mobile development frameworks can be used to build native-looking apps for multiple platforms, such as Android and iOS, using a single codebase

✓ Lower development costs (Multiplatform utilizing a single codebase)

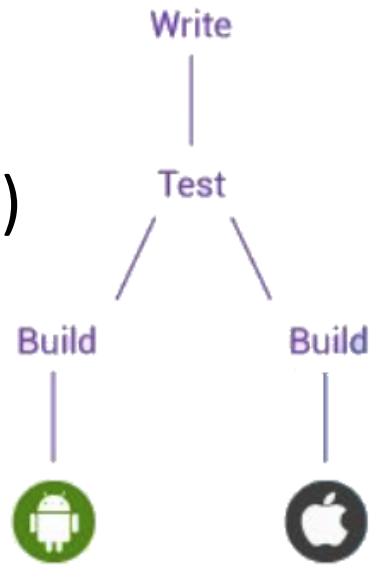
✓ Leverage existing skillset (JavaScript, React, Dart)

✓ UI performance is almost as fast as native

✓ Downloadable from app stores

– Highly dependent on libraries and frameworks

– Delayed update to latest native APIs



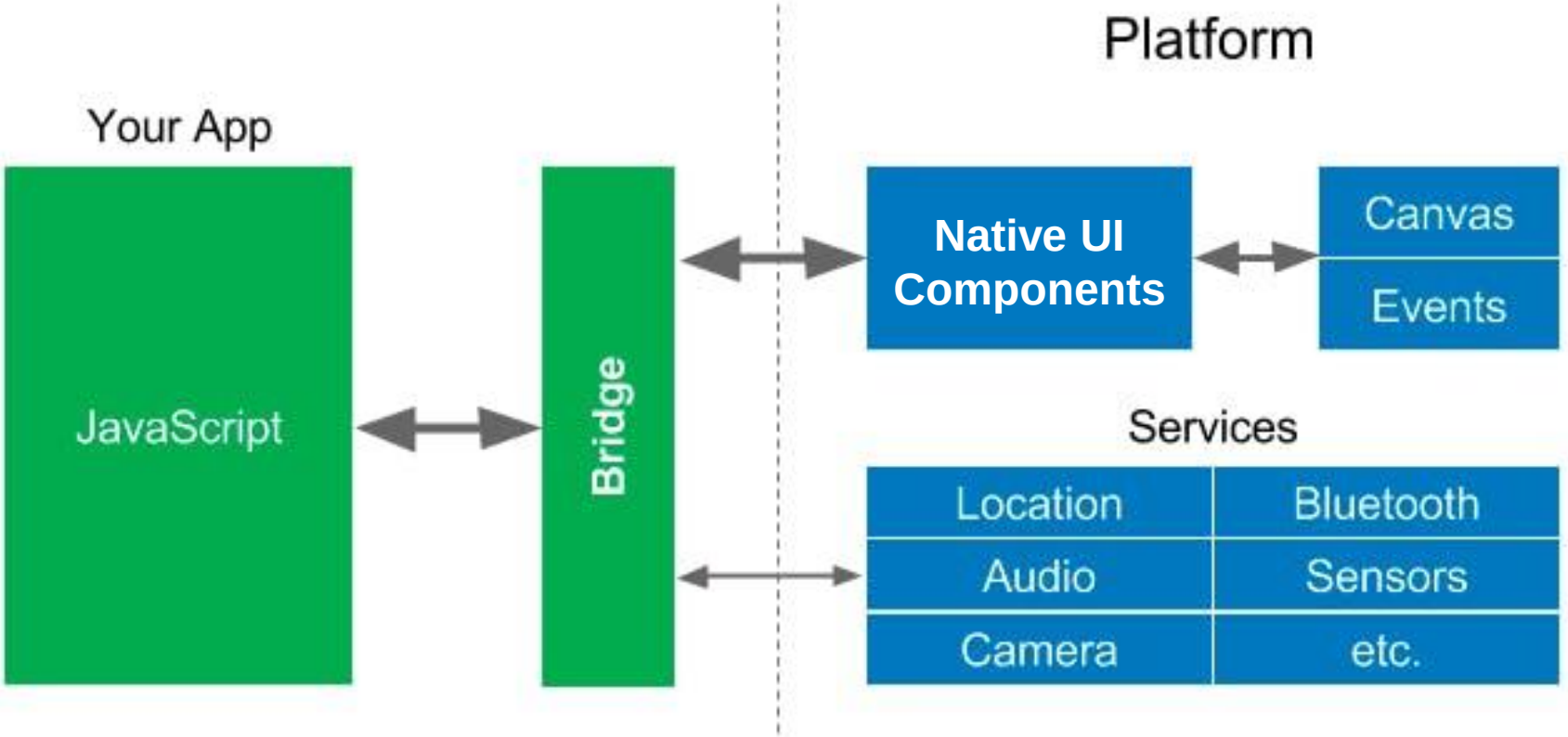
Flutter

React Native

Kotlin Multiplatform

React Native Compiles JavaScript UI components into equivalent **native UI** elements

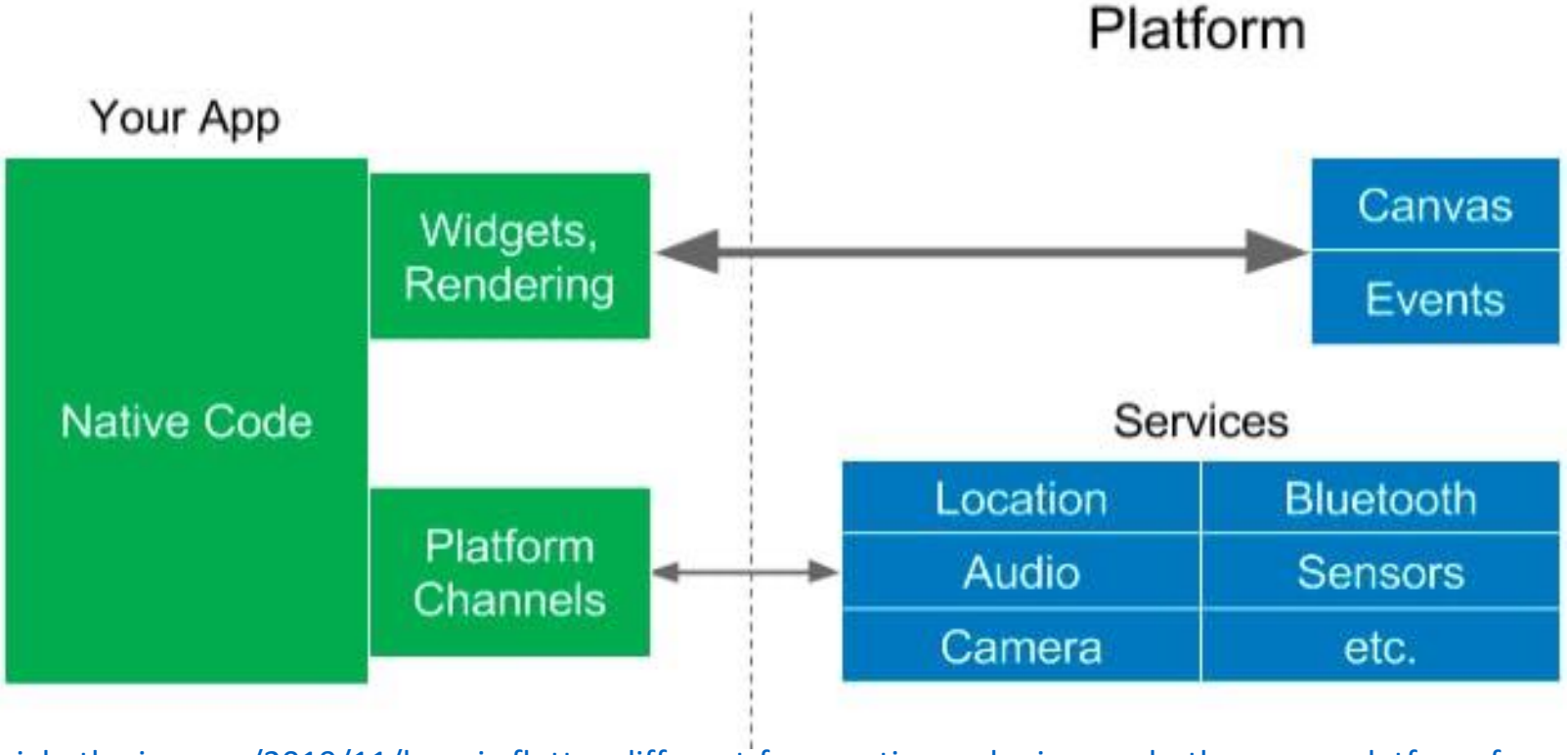
- Remaining code doesn't get compiled, instead runs in a separate JavaScript thread
- App interact with UI and access the platform services via a **bridge**



https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/

Flutter

- Flutter App (written in Dart) is **compiled into native code**, UI uses Flutter own custom widgets rendered by the framework's **graphics engine Impeller or Skia** to work across devices.
- App uses Platform Channels to access the platform services



https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/

Native App

- Uses platform-specific (Android/iOS) UI components and API

✓ Access to all native APIs, hardware, sensors, & OS

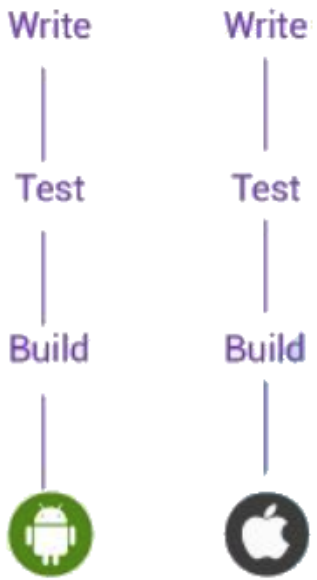
- No third-party dependencies

✓ Fast **performance** as it run directly on OS

✓ High-quality User Experience (UX)

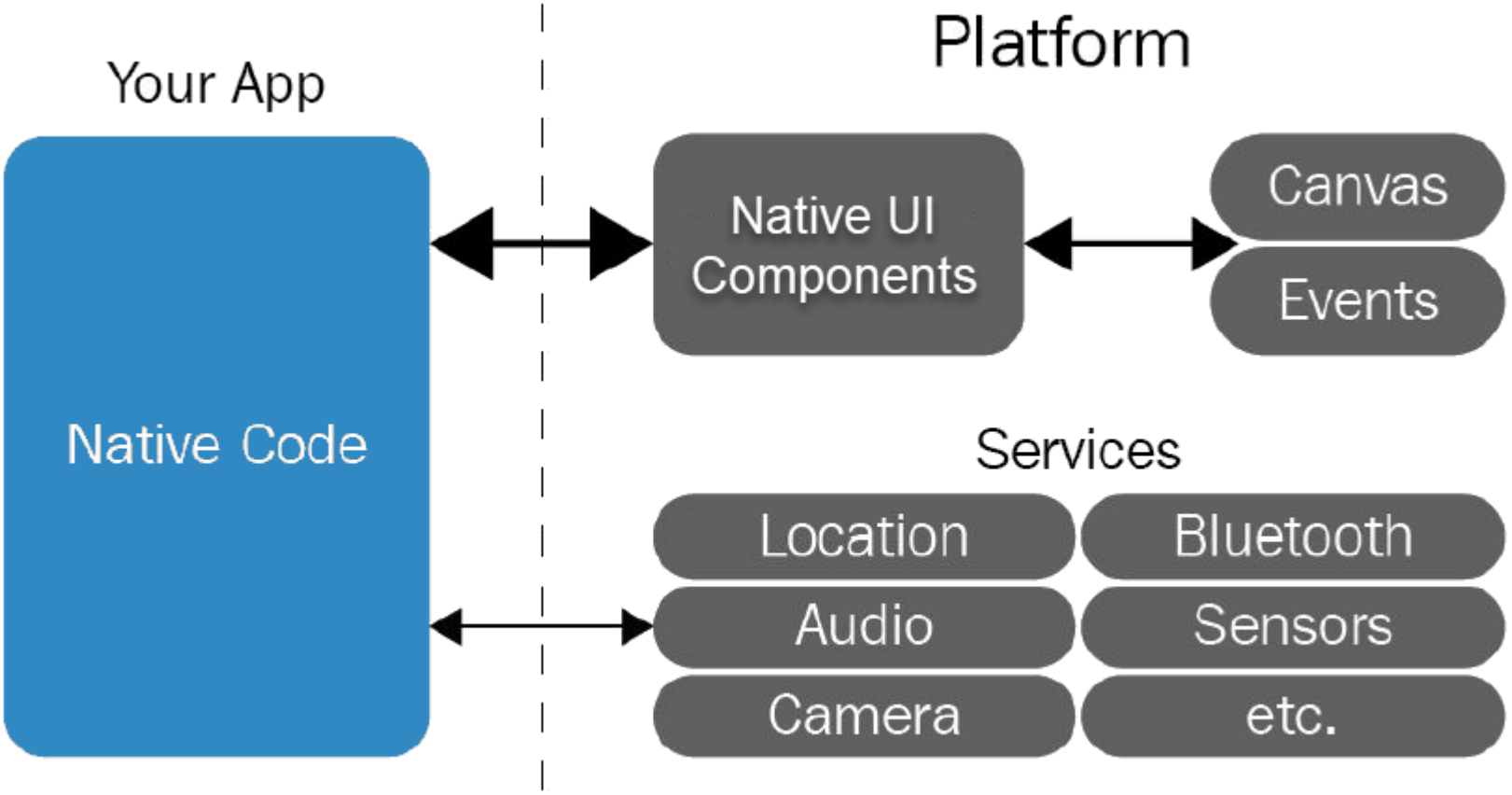
– No codebase reuse

– High dev cost and longer time to market: requires **multiple code bases** and teams



Native Android/iOS Platforms

The **app** has direct access to the platform services

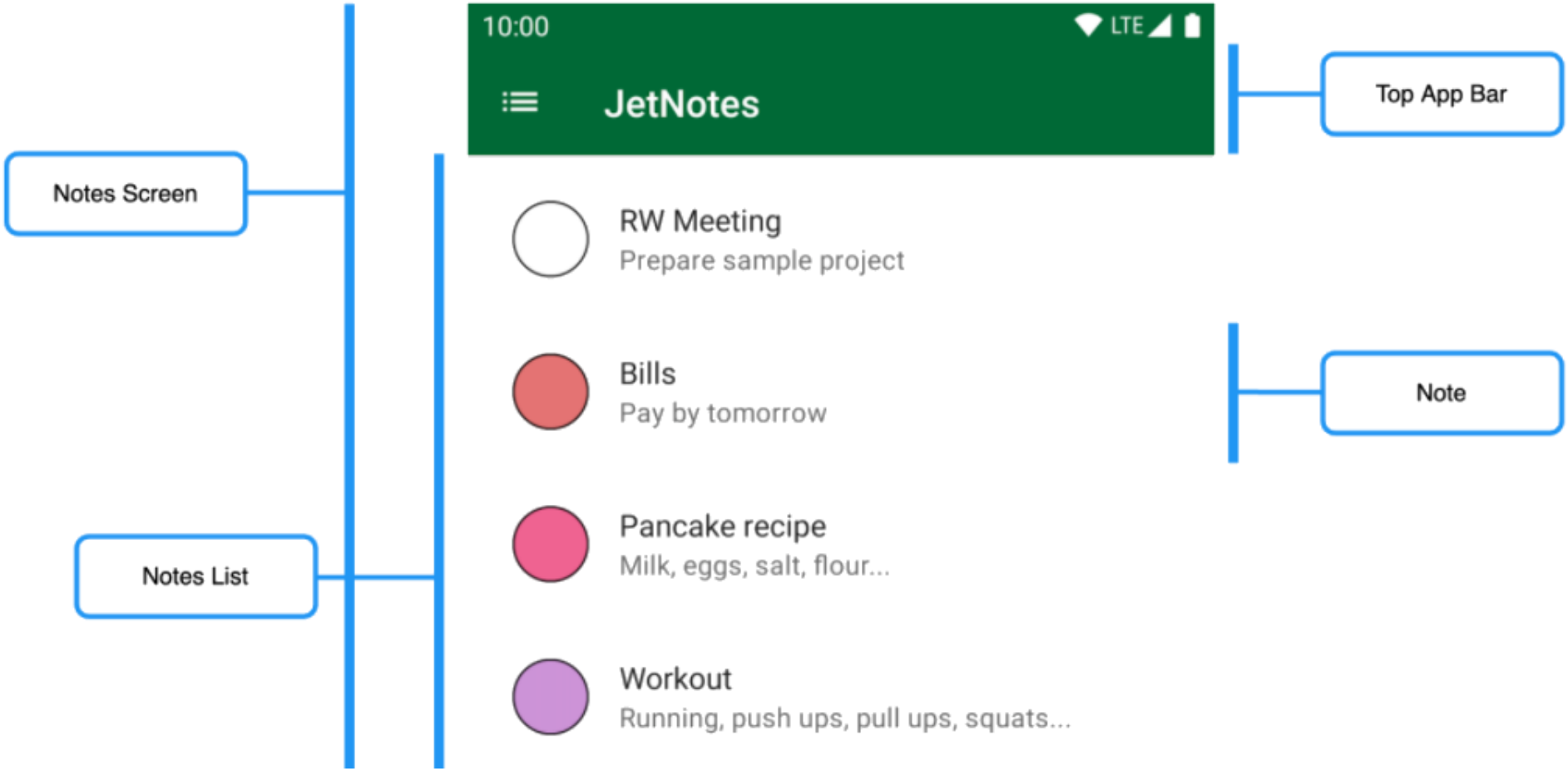


https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/

Mobile App UI Design Process

1. Design the UI wireframe (sketch)
   - Decide what information to present to the user and what input they should supply
   - Decide the UI components and the layout on paper or using a design tool such as Figma
   - Design the app navigation through the screens to achieve the app use cases
2. Breakdown the UI into small reusable UI components (building blocks) that work together to make the whole screen
3. Use a bottom-up approach:
   - Start implementing the smaller UI components and build your way up through the design
   - For each UI component, identify the data needed (app state) and events raised to notify the app logic
   - Manage app state and data exchange between UI components & app logic to respond to the user actions
   - Compose the screens from building block components and arrange them using appropriate layouts

Example - UI decomposition into UI Components



UI Sketch - Example

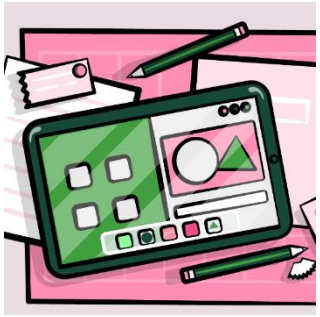


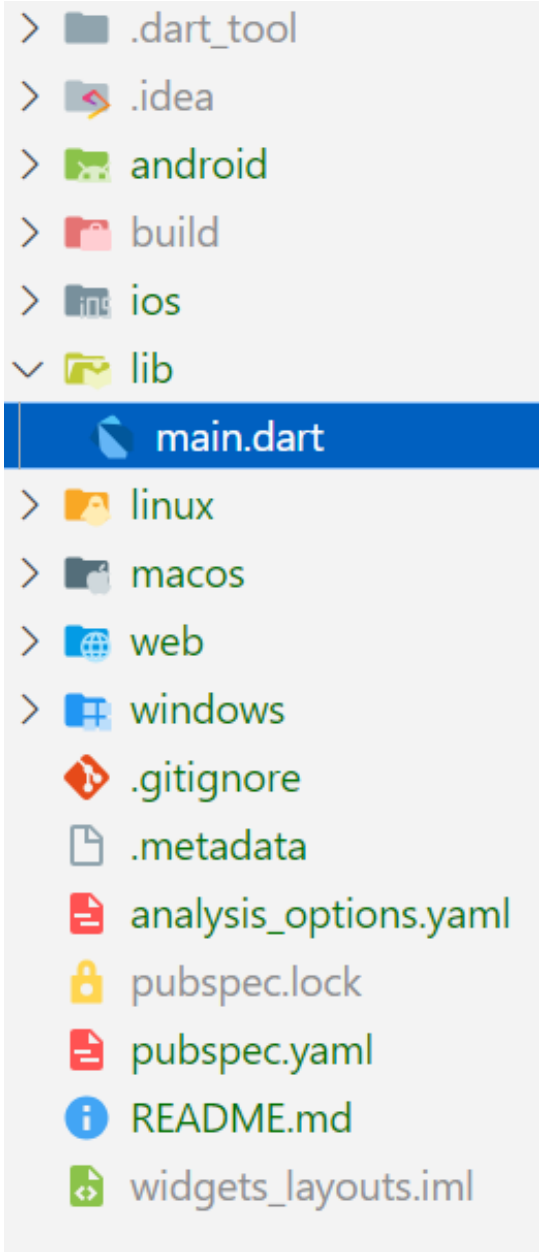
Fig 1. Home screen

Fig 2. Food places

You may design different layouts per screen size

Source: https://www.figma.com/ui-design-tool/

Flutter Project structure



- **`lib/`**: main app code folder, it contains main.dart (the entry point of the app)
  - You can create subdirectories for better organization, such as screens/, models/, widgets/, etc.
- **`android/`**, **`ios/`**, **`web/`**, **`windows/`**, **`macOS/`**, **`linux/`** : platform-specific configuration files and native code
- **`pubspec.yaml`**: a configuration file that lists the app's dependencies, asset declarations, and metadata (like app name, version, etc.)
  - It's essential for managing third-party libraries and resources
- **`build/`**: contains build outputs
  - It is usually excluded from version control
- **`assets/`**: stores external resources like images, fonts, and other files that are included in the app

Introduction to Flutter

Flutter

- Flutter is a **UI toolkit** (including Widgets, Rendering Engine and DevTools) for building applications for mobile, web, and desktop from a single codebase.
- A **declarative component-based programming model**
  - UI is built using composable widgets
    - Each widget define a piece the app's UI programmatically by **describing WHAT to see** (layout/ look and feel) **NOT HOW**
    - Compiler takes care of the HOW and constructs UI elements
  - As state changes the UI automatically updates (Reactive UI)

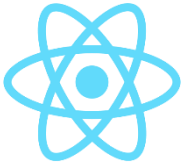
  (without imperatively mutating UI components)
- Inspired by/similar to other declarative UI frameworks such as React and Jetpack Compose

Declarative UI is a major trend

- Describe WHAT to see NOT HOW

Flutter: Google's UI toolkit for building natively compiled applications for mobile, web and desktop from a single codebase

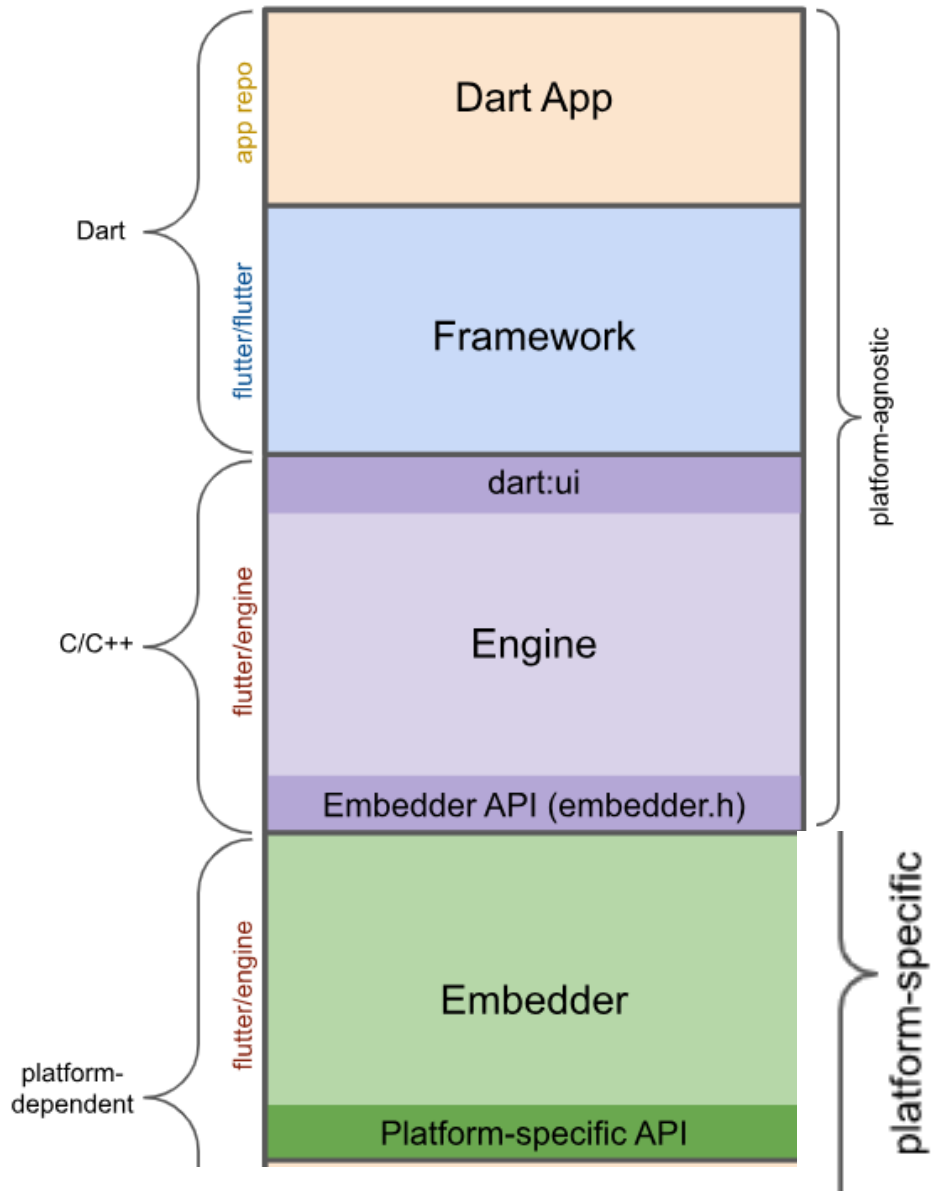
SwiftUI: Apple’s declarative framework for creating apps that run on iOS



React: A JavaScript library for building user interfaces

Jetpack Compose: a **toolkit** for building native Android UI

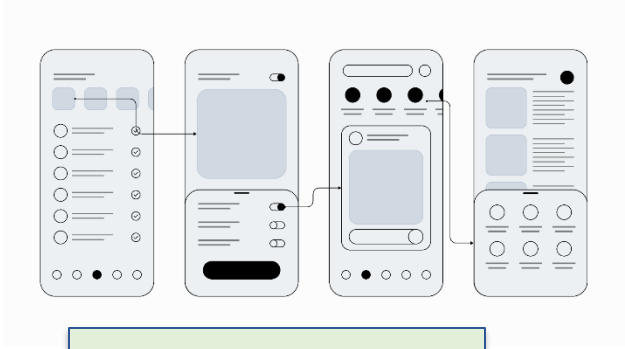
Flutter Software Stack

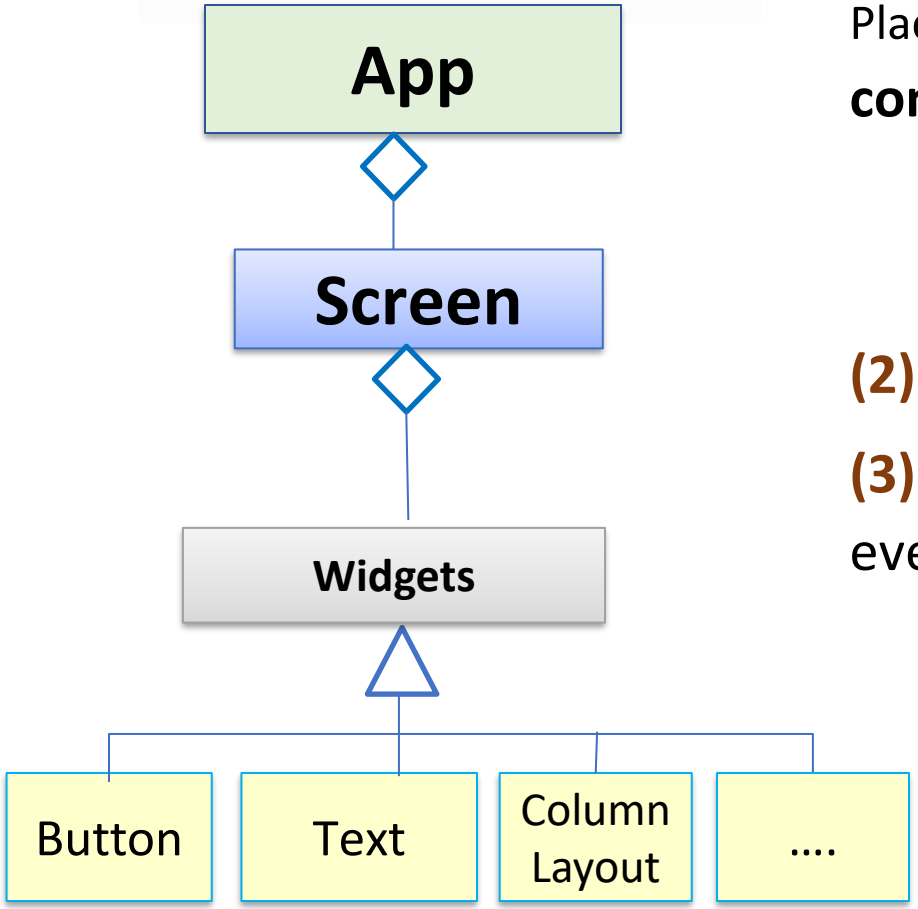


1. **Dart App**: composes widgets into the desired UI
   - Implements business logic
2. **Framework**: provides widgets and higher-level API to build apps
3. **Flutter engine** is responsible for rendering the UI and processing platform events such as touch gestures and keyboard inputs
4. **Embedder** acts as a bridge that handles interaction between the native OS and system resources. More info

Flutter Key Concepts

Declarative UI Programming Model





- App is composed of one or more **screens** (also called pages). A **screen** has:

(1) Widgets (UI Components)

Placed in a **Layout** widgets that acts as a **container** for UI Components

- Layout decides the size and position of widgets placed in it

(2) State objects that provides the data to the UI

(3) Event Handlers to respond to the UI events

- Widgets **raise Events** when the user interacts with them (such as a Pressed event is raised when a button is pressed)
- Connecting user interactions (like button presses) to app behavior

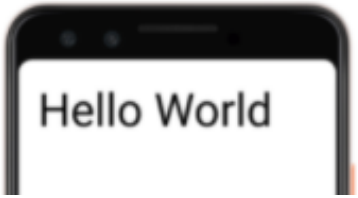
How to define a piece of UI?

- UI is **composed** of small reusable **components** called widgets
- **Widget:** a class that extends StatelessWidget or StatefulWidget depending on whether it manages internal state
  - Each component renders a portion of the UI, transforming the app's data (state) into visual elements
  - **UI = f(state) : UI is a visual representation of state** (e.g., shopping cart in an e-commerce app)
- **State-Driven UI Updates**
  - State changes triggers a redraw of the UI
  - Flutter is declarative: it builds the UI to reflect the current app state

Stateless Widget

Non-interactive UI

String → `void Greeting(String name) { print('Hello, $name'); }` → stdout

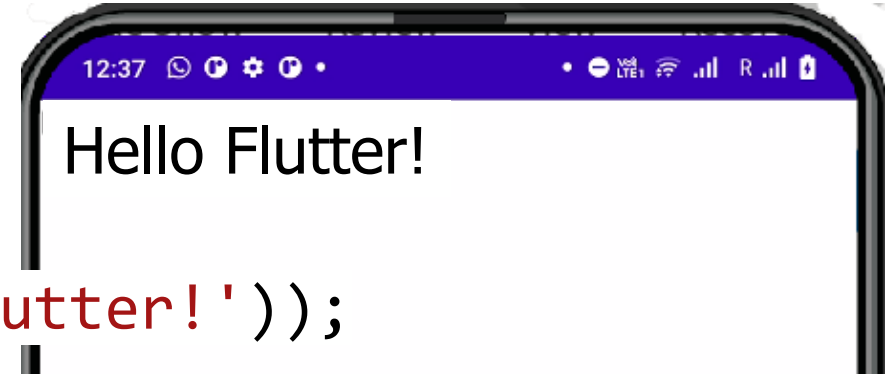



```
class Greeting extends StatelessWidget {
  final String name;
  const Greeting(this.name);

  @override
  Widget build(BuildContext context) {
    return Text('Hello, $name');
  }
}
```

Greeting class uses the input data to render a Text widget on the screen

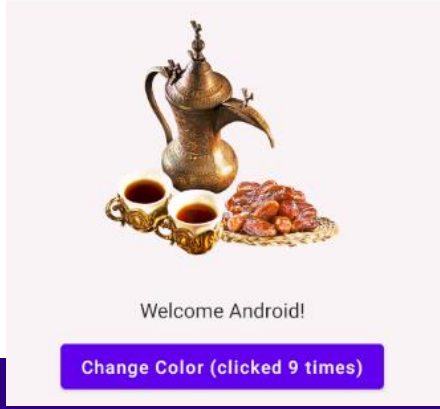
App Entry Point

- The `main()` function is the app entry point
  - Inside it you call the `runApp()` function to launch the app and display the UI on the screen
  - `runApp()` takes a widget (root widget) and displays the app UI
    - The root widget calls other widgets and passing them the appropriate data
  - The root widget can be anything, but typically it's a `MaterialApp` with built-in base theming, navigation, and more

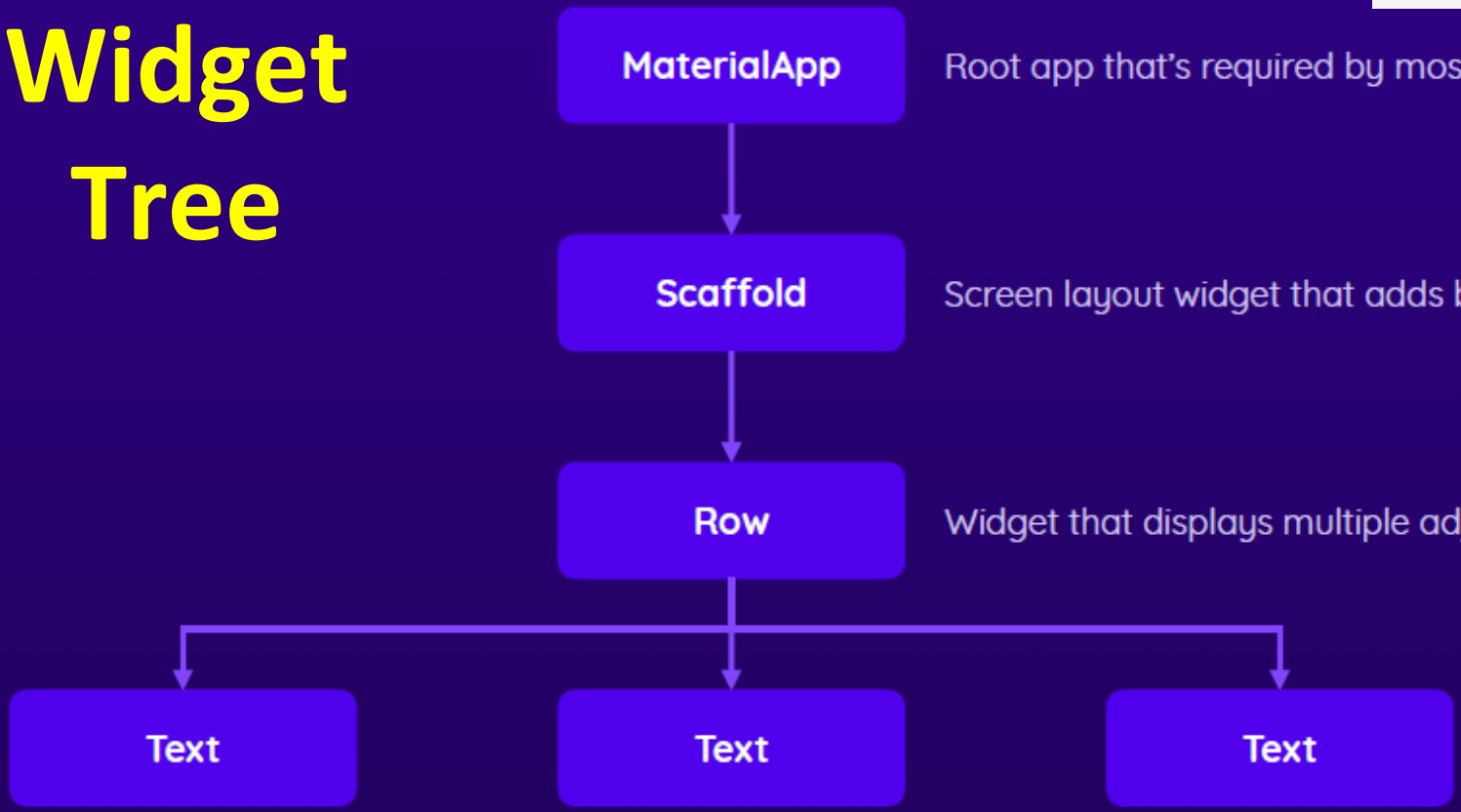



```
void main() {
  runApp(const Greeting('Flutter!'));
}
```

UI = Composition of Widgets



Widget Tree



Source: Flutter Complete Course slides

BuildContext

- `BuildContext` represents the location of a widget within the **widget tree**, serving as a link between the widget and its surrounding environment. It plays a critical role in giving the widget access to:
  - **Theme**: used to customize the app's look and feel, such as colors, fonts.
  - **MediaQuery**: provides information about the screen size, device orientation to enable responsive UI that adapt to different screen sizes
  - **Navigator**: used for navigating between screens

BuildContext usage example

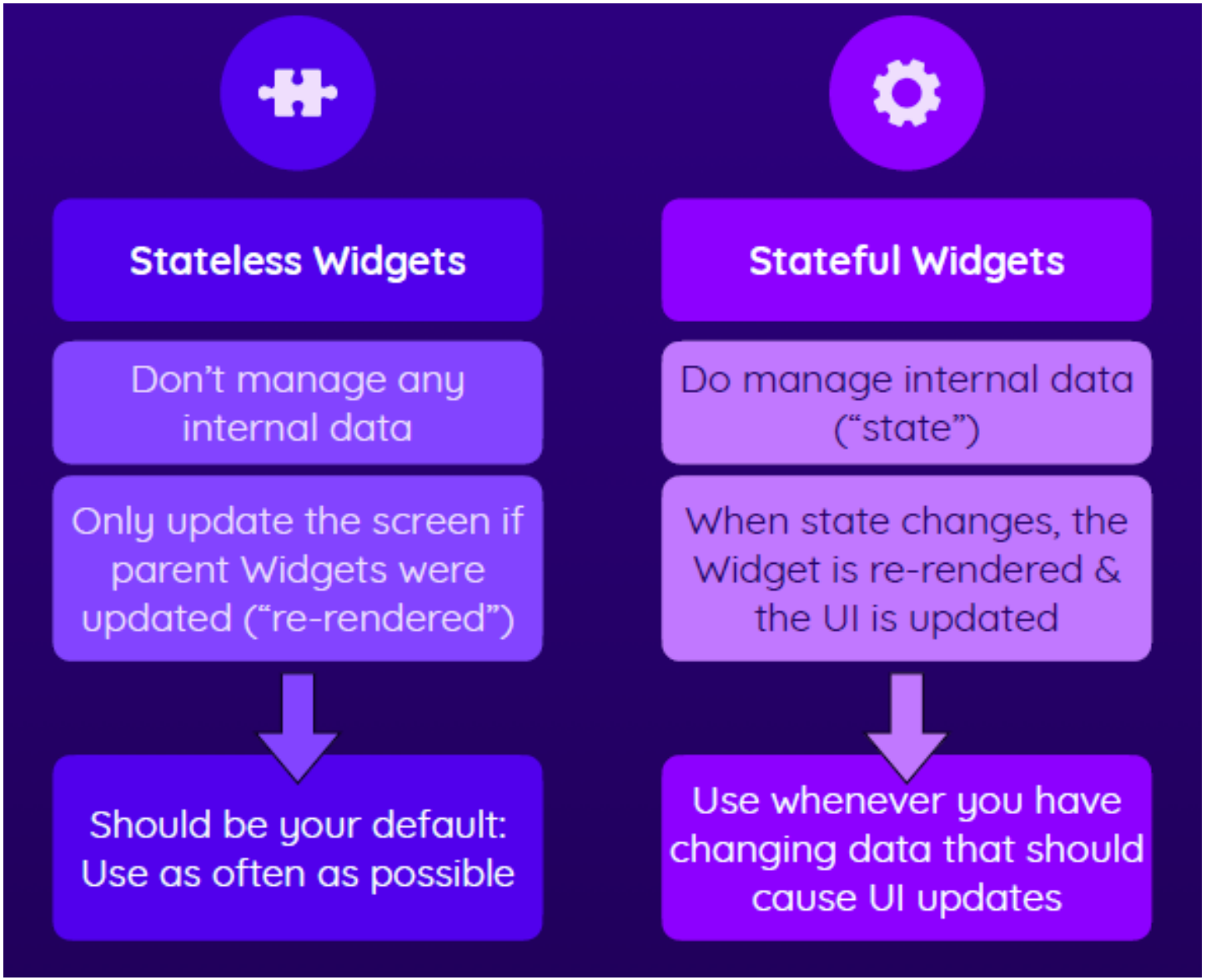
```
class Greeting extends StatelessWidget {
  final String name;

  const Greeting(this.name);

  @override
  Widget build(BuildContext context) {
    return Text(
      'Hello, $name',
      // Using context to access theme data
      style: Theme.of(context).textTheme.headlineLarge,
    );
  }
}
```

Stateless vs Stateful widgets

- A stateless widget doesn't hold any state
  - The caller controls and manages the state
- **Stateful** widgets can hold and manage internal mutable state and update its appearance in response to state changes
  - State variables must be declared in class that extends **State** base class
  - They should be changed inside `setState`(…) method that act as **Change Notifiers** to trigger redrawing the widget

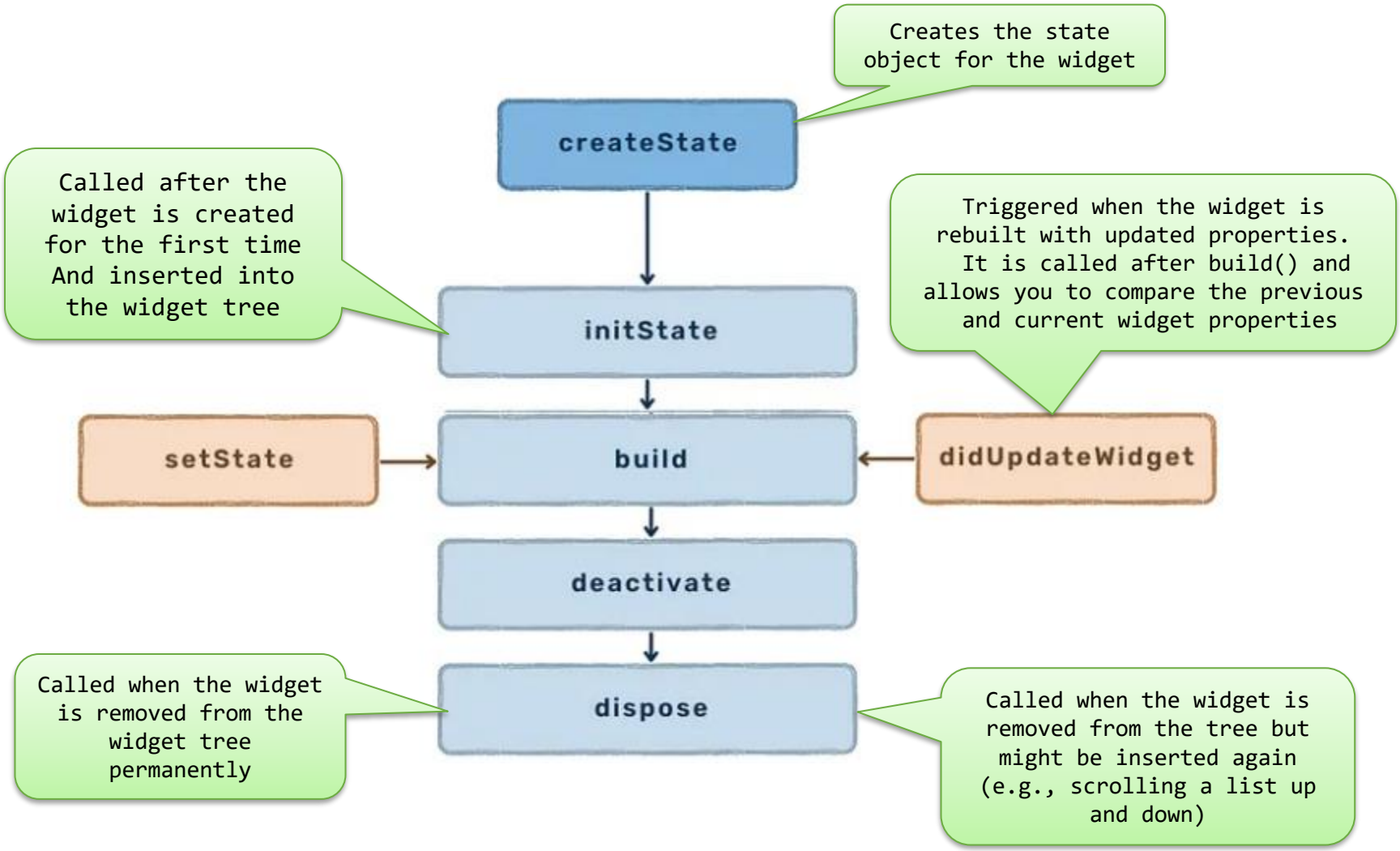
=> UI is **auto-updated** to reflect the updated app state

Stateless vs Stateful

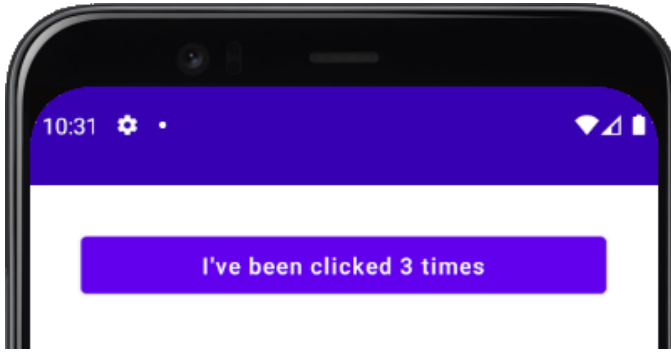


Source: Flutter Complete Course slides

Lifecycle of a Stateful Widget



Stateful Widget Example



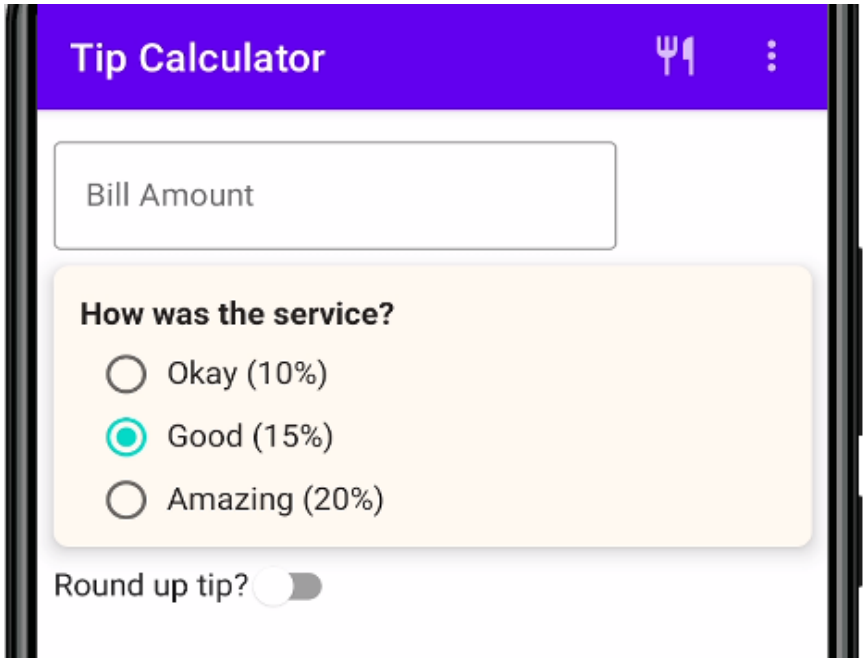
- It extends StatefulWidget base class
- It defined **clicksCount** state variable in a class that extends **State** base class
- Every time the button is clicked, the button widget raises ***onPressed*** event to notify the app logic, which increments **clicksCount** state variable using **setState** method => This causes a **Widget Rebuilding** to take place

```
class ClickCounter extends StatefulWidget {
  const ClickCounter();
  @override
  _ClickCounterState createState()
            => _ClickCounterState();
}
```

```
class _ClickCounterState extends State<ClickCounter> {
  int clicksCount = 0;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Click Counter'),
        centerTitle: true),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
                setState(() {
                  clicksCount += 1;
                });
          },
          child: Text("I've been clicked $clicksCount times"),
        ),
      ));
}}
```

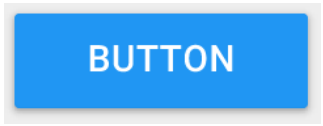
Tip Calculator Example

- In the example below, notice no Compute/OK button, any change of input auto-recomputes and re-displays the tip value
  - Like Excel way: changing a cell value triggers auto-update of formulas and graphs referencing it
- Plus, the code is much more concise and elegant (see posted example)

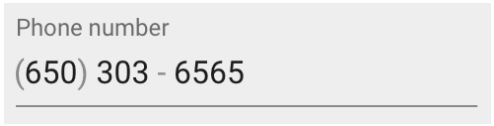


Widgets

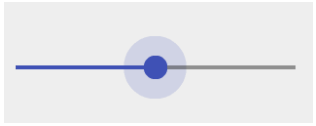
Button



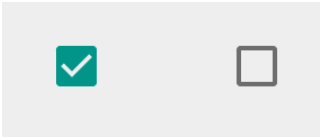
TextField



Slider

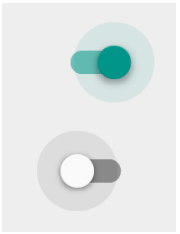


CheckBox



RadioButton

Switch



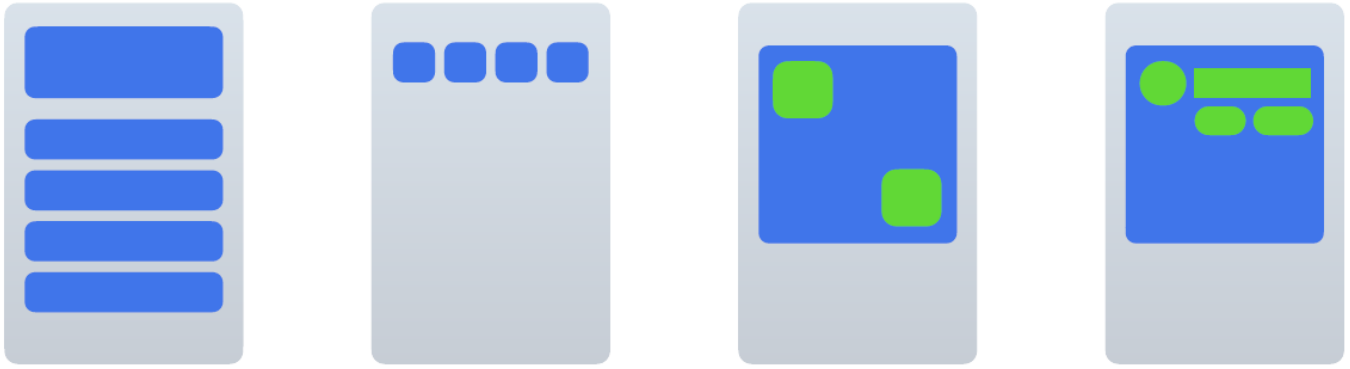
See more details in slides ‘05 Widgets-Layouts’

Full list available at link

Back

Layouts

- Use a Layout to size & **position** UI elements on the screen
- **`Row`** - position elements horizontally
- **`Column`** - position elements vertically
- **`Stack`** - stack elements on top of each other
- Many more…



Column | Row | Stack | Other Layouts

See more details in slides '05 Widgets-Layouts' & this link

Column and Row

`Column()` & `Row()` can be used to place **multiple child widgets next to each other**

`Column()`

Main Axis: **Vertical** Axis

Cross Axis: Horizontal Axis

By default, occupies the **entire available height** but **only the width required** by its content (children)

`Row()`

Main Axis: **Horizontal** Axis

Cross Axis: Vertical Axis

By default, occupies the **entire available width** but **only the height required** by its content (children)

Source: Flutter Complete Course slides

App State Management

https://docs.flutter.dev/data-and-backend/state-mgmt

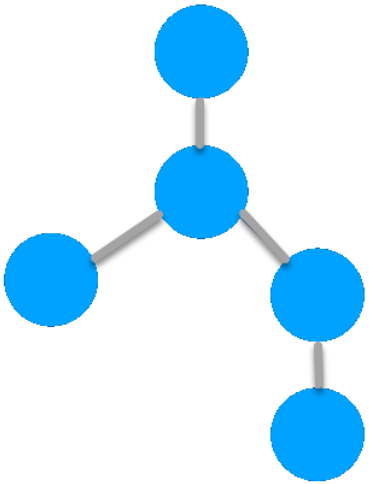
YouTube video

State

- State = whatever **data you need to rebuild the app UI** at any moment in time + its changes during runtime
- UI in Flutter is immutable
  - In Flutter you cannot access/update UI elements directly (as done in the imperative approach)
  - When the user interacts with the UI, the widgets raises events such as onChanged
  - Those events should notify the app logic, which can then change the app's state
  - 👍 When the state changes it causes the build methods of the affected widgets to be automatically called again with the new data
- Flutter intelligently rebuilds only the widgets that changed

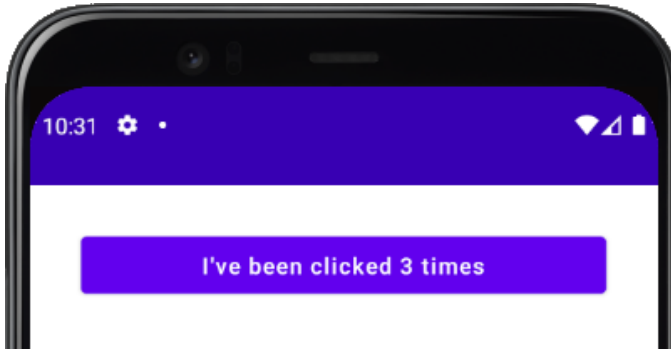
Widget Rebuilding in Flutter

1. **State Change**: When a stateful widget's state changes (e.g., after a user interaction or data update), Flutter triggers a UI rebuild
2. **Widget Tree Reconstruction**: Flutter calls the build() method of the affected widget, reconstructing that widget and its child widgets
    - Flutter does this efficiently by only rebuilding the parts of the widget tree that have changed, minimizing unnecessary work



More details are available at this link and this video

Widget Rebuilding Example



- Every time the button is clicked, the button widget raises ***onPressed*** event to notify the app logic, which increments **clicksCount** state variable
- This causes a **Widget Rebuilding** to take place, i.e., the **ClickCounter** build function is automatically called again to redrawn the widget

```
class ClickCounter extends StatefulWidget {
  const ClickCounter();
  @override
  _ClickCounterState createState()
            => _ClickCounterState();
}
```

```
class _ClickCounterState extends State<ClickCounter> {
  int clicksCount = 0;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Click Counter'),
        centerTitle: true),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
              setState(() {
                clicksCount += 1;
              });
          },
          child: Text("I've been clicked $clicksCount times"),
        ),
      ));
  }
}
```

Stateful versus Stateless

- A **stateless** widget that doesn't hold any state
  - The caller controls and manages the state
- A **stateful** widget can hold and manage internal mutable state
  - Reduced reusability: the state is internal and not exposed, making it hard to reuse the widget in different contexts or with different external state
  - Harder testing: because you need to simulate the state transitions to verify behavior

=> Where possible, **Lift state up** to manage it externally and pass it to widgets to improve reusability and testability

- The widget that previously managed state now takes the state as an input from the parent

Lifting state up (a.ka. State Hoisting)

- To make a widget stateless, **extract** its state and **move it to the parent**
- Then pass the state to the widget as a parameter, along with a callback function that the widget can call to update that state in response to events (e.g., onValueChange, onSelected) e.g.,
  - `String name` : the current value to display
  - `Function(String) onNameChange` : a callback that requests the value to change
- Lifted state variables are owned by the Caller and can passed to other widgets

Lifting state up - Example

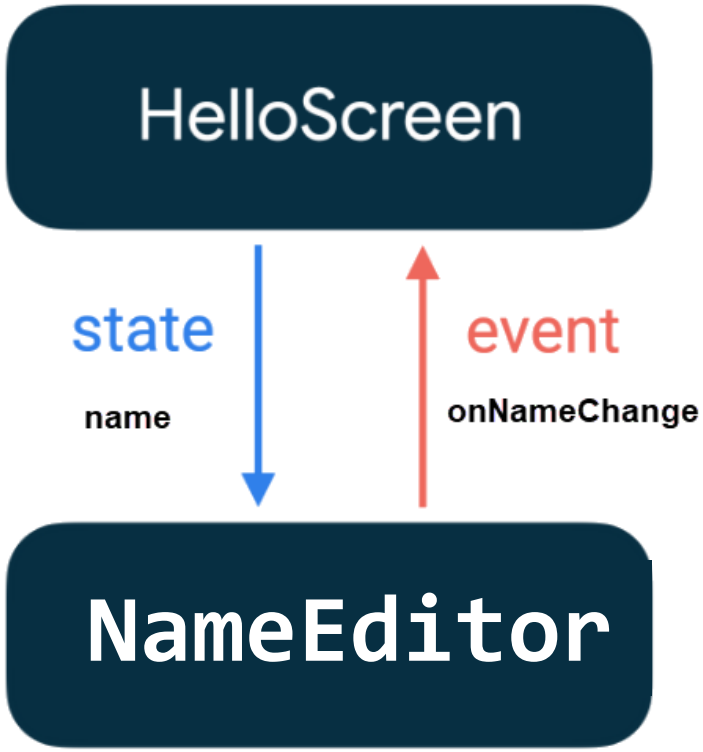

```dart
class NameEditor extends StatelessWidget {
  final String name;
  final Function(String) onNameChange;

  const NameEditor({required this.name,
         required this.onNameChange});

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(16.0),
      child: TextField(
        decoration: const InputDecoration(
          labelText: 'Name',
          border: OutlineInputBorder(),
        ),
        onChanged: onNameChange,
      ),
    );
  }
}
```

```dart
class HelloScreen extends StatefulWidget {
  const HelloScreen();
  @override
  _HelloScreenState createState() => _HelloScreenState();
}

class _HelloScreenState extends State<HelloScreen> {
  String name = '';

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          Text('Hello, $name'),
          const SizedBox(height: 8),
          NameEditor(
            name: name,
            onNameChange: (String newName) {
              setState(() {
                name = newName;
              });
            },
          ),
        ]));
  }
}
```

Unidirectional Data Flow

= a design where **state flows down and events flow up**

```
var name ; // state variable
NameEditor(name : name, onNameChange: (String newName) {
        setState(() { name = newName; }); )
```



State flows down via widget parameter

(e.g., *name*)

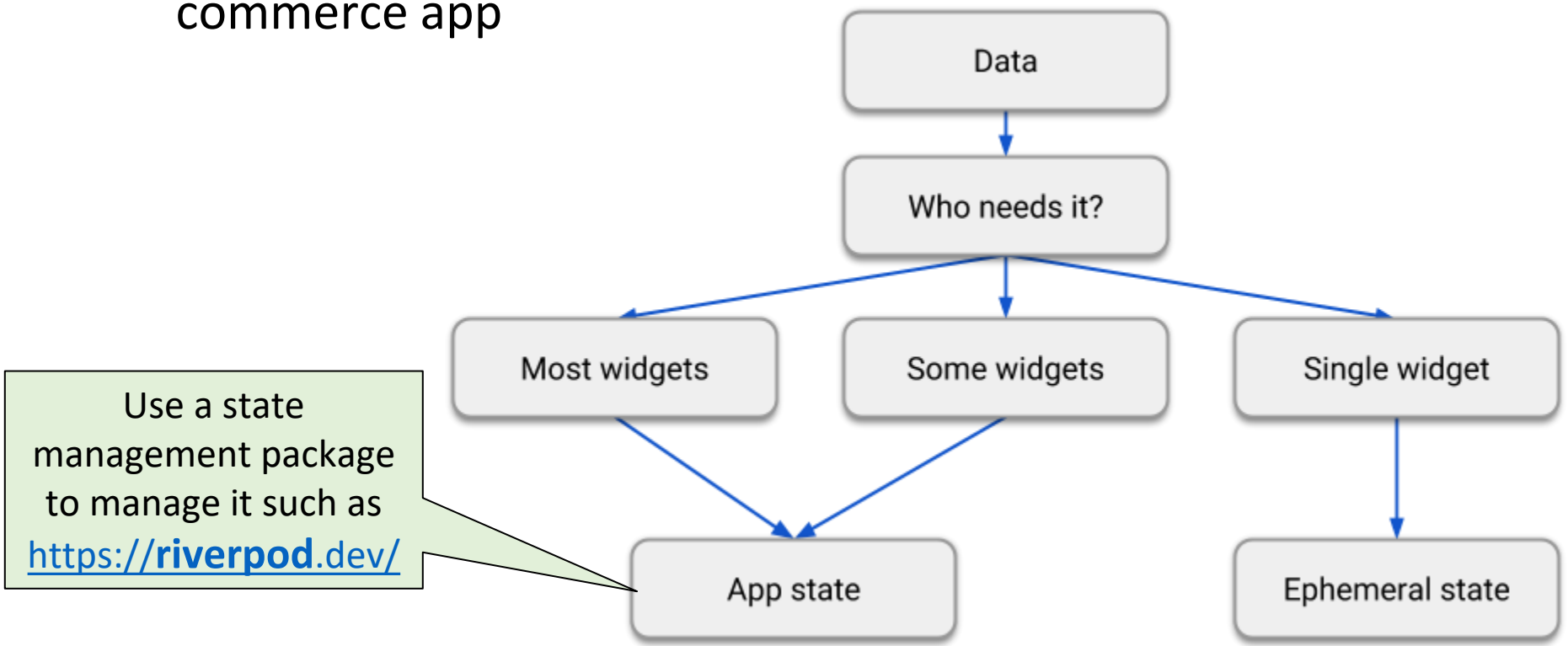
(State change) Event flows up via callback function

(e.g., *onNameChange*)

By hoisting the state out of NameEditor, it can be **reused** in different situations, and it is easier to test

Ephemeral state and App state

- **Ephemeral state** (aka UI state or local state) contained in a single widget (a StatefulWidget can be used to manage it)
  - E.g., current selected option in a BottomNavigationBar
- **App state**: shared across many parts of your app
  - E.g., user preferences, Login info, shopping cart in an e-commerce app



Summary

- Declarative UI is the trend for UI development
  - UI is composed of small reusable widgets
  - **Stateless widgets** don't hold state, making them more reusable and test-friendly
  - **Stateful widgets** manage their own state but are harder to reuse and test
  - **State hoisting** shifts state management to the parent, enhancing the flexibility of child widgets
- Layouts are used to size position widgets on the screen
- Widget is **immutable**
  - It only accepts state & exposes events
  - **Unidirectional Data Flow** pattern:
    - State flows down via parameters
    - Events flow up via callbacks
- .. mastering Flutter will take some time and practice 🤺 🏋️ ...

Resources

- Flutter getting started

https://docs.flutter.dev/get-started/

- Flutter architecture

https://docs.flutter.dev/resources/architectural-overview

- Flutter Code Labs

https://docs.flutter.dev/codelabs

- Widgets

https://docs.flutter.dev/ui/widgets

- Layouts

https://docs.flutter.dev/ui/layout