A Brief Introduction to Flutter

Luciano Baresi
Politecnico di Milano
Flutter

Fast Development
Paint your app to life in milliseconds with stateful Hot Reload. Use a rich set of fully-customizable widgets to build native interfaces in minutes.

Expressive and Flexible UI
Quickly ship features with a focus on native end-user experiences. Layered architecture allows for full customization, which results in incredibly fast rendering and expressive and flexible designs.

Native Performance
Flutter's widgets incorporate all critical platform differences such as scrolling, navigation, icons and fonts to provide full native performance on both iOS and Android.
A few success stories

**The New York Times**
Flutter helps bring the popular Ken Ken puzzle to life on Android, iOS, Mac, Windows, and the web.

**Square**
The Flutter plugin for our Square Reader SDK enables developers to build apps for merchants that take in-person payments.

**Google Assistant**
Flutter helps power an ever-growing number of Google Assistant apps.

Try it now  
Learn more  
Learn more
How Flutter works
Dart

- Open-source web programming language developed by Google
- Class-based, single-inheritance, object oriented language with C-style syntax
- Supports interfaces, abstract classes, reified generics, optional, and strong typing
- Dart compiles to ARM and x86 code
  - Dart mobile apps can run natively on iOS, Android
  - For web apps, Dart transpiles to JavaScript
Dart

• Everything you can place in a variable is an object
  • Even numbers, functions, and null are objects
• Type annotations are optional
  • If no type is expected, use the special type dynamic
• Dart supports generic types, like List<int>
• Dart supports
  • Top-level functions (such as main()), as well as functions tied to a class or object
  • Top-level variables, as well as variables tied to a class or object
• Public, protected, and private properties do not exist
  • If an identifier starts with an underscore (_), it is private to its library
A gentle introduction to coding

- Dart is a general purpose programming language
- More sophisticated than HTML + Javascript
- We don’t need HTML
- Flutter provides a simple way of composing User Interfaces
- Focus on what we need
  - Appearance
  - Behavior
  - Integration between Appearance and Behavior
Credits and installation

• Heavily based on https://flutter.dev/docs

• First of all, you must
  • Install flutter
  • Install IDE, Simulator, and editor (e.g., VS code)
    • xCode, Android Studio, IntelliJ
  • Test everything
Downloads

- Visual Studio Code
  - https://code.visualstudio.com/
- Flutter
  - https://flutter.dev/docs/get-started/install
- Xcode (Mac OS)
- Android Studio (Linux and Windows)
  - https://developer.android.com/studio
Follow the wizard to install the editor

In Visual Studio Code

- Go to View > Command Palette
- Type and select ‘Install Extension’
- Search for and Install ‘Dart’ and ‘Flutter’ extensions
Download the zip file

Create an installation folder

Unzip the file into the installation folder

Update your Path

- Mac OS: https://flutter.dev/docs/get-started/install/macos#update-your-path
- Windows: https://flutter.dev/docs/get-started/install/windows#update-your-path
- Linux: https://flutter.dev/docs/get-started/install/linux#update-your-path
First app

- flutter create my_app
- cd my_app
- open -a Simulator
  - on my Mac
- flutter run
First App

In VS Code invoke View > Command Palette

Type ‘Flutter’ and select ‘Flutter: New Project’

Select a destination folder to save the project

Open file lib/main.dart

Delete all the code

Copy and paste the code snippet of the next slide in lib/main.dart
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Welcome to Flutter'),
        ),
        body: Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
In VS Code invoke Debug > Start Debugging

Select an emulated device (iOS or Android) or create a new one (Android only)

The application will be installed and executed on the selected device

In the code replace the string “Hello World” with your name

Save and see the changes in the device (hot reload)

Run the App
Observations

• This example creates a Material app
  • Material is a visual design language that is standard on mobile and the web
  • Flutter offers a rich set of Material widgets
• The main() method uses arrow (=>) notation
  • Use arrow notation for one-line functions or methods
• The app extends StatelessWidget which makes the app itself a widget
  • Almost everything is a widget, including alignment, padding, and layout
• The Scaffold widget, from the Material library, provides a default app bar, title, and a body property that holds the widget tree for the home screen
  • The widget tree can be quite complex
• A widget’s main job is to provide a build() method that describes how to display the widget in terms of other, lower level widgets
• The body for this example consists of a Center widget containing a Text child widget
  • The Center widget aligns its widget subtree to the center of the screen
• Flutter is different from other frameworks because its UI is built in code, not (for example) in an XML file or similar
• Widgets are the basic building blocks of a Flutter UI
  • Almost everything in Flutter is a widget
• A widget is an immutable object that describes a specific part of a UI
• Widgets are composable, meaning, that you can combine existing widgets to make more sophisticated widgets
Bundled designs

Material design

To create an iOS-centric design, we can use package Cupertino components
• You build your UI out of widgets
  • Flutter widgets are built using a modern framework that takes inspiration from React

• Widgets describe what their view should look like given their current configuration and state
  • When a widget’s state changes, the widget rebuilds its description
  • The framework computes the diff against the previous description in order to determine the minimal changes needed in the underlying render tree to transition from one state to the next
New widgets

• Are subclasses of either StatelessWidget or StatefulWidget

• A widget’s main job is to implement a `build()` function, which describes the widget in terms of other, lower-level widgets

• The framework builds those widgets in turn until the process bottoms out in widgets that represent the underlying RenderObject, which computes and describes the geometry of the widget
Everything is a widget
Widget catalog

Create beautiful apps faster with Flutter's collection of visual, structural, platform, and interactive widgets. In addition to browsing widgets by category, you can also see all the widgets in the widget index.

- Accessibility
  Make your app accessible.
  Visit

- Animation and Motion
  Bring animations to your app.
  Visit

- Assets, Images, and Icons
  Manage assets, display images, and show icons.
  Visit

- Async
  Async patterns to your Flutter application.

- Basics
  Widgets you absolutely need to know before building your first Flutter app.

- Cupertino (iOS-style widgets)
  Beautiful and high-fidelity widgets for current iOS design language.
Stateful vs. stateless widgets

• If a widget changes (the user interacts with it), it is stateful
  • A widget’s state consists of values that can change, like a slider’s current value or whether a checkbox is checked
• A widget’s state is stored in a State object, separating the widget’s state from its appearance
• When the widget’s state changes, the state object calls setState(), telling the framework to redraw the widget
• A stateless widget has no internal state to manage
Minimal app

- The runApp() function takes the given Widget and makes it the root of the widget tree
- The widget tree consists of two widgets, the Center widget and its child, the Text widget
- The framework forces the root widget to cover the screen, which means the text “Hello, world” ends up centered on screen

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

void main() {
  runApp(
    Center(
      child: Text('Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```
How widgets work

• Widgets are passed as arguments to other widgets
• The Scaffold widget takes a number of different widgets as named arguments, each of which are placed in the Scaffold layout in the appropriate place
• The AppBar widget lets you pass in widgets for the leading widget, and the actions of the title widget
• This pattern recurs throughout the framework and is something you might consider when designing your own widgets
Some basic widgets

• **Text** lets you create a variety of styled text within your application

• **Row** and **Column** let you create flexible layouts in both the horizontal and vertical directions
  • The design of these objects is based on the web’s flexbox layout model

• **Stack** lets you place widgets on top of each other in paint order
  • You can then use the **Positioned** widget on children of a Stack to position them relative to the top, right, bottom, or left edge of the stack
  • Stacks are based on the web’s absolute positioning layout model

• **Container** lets you create a rectangular visual element
  • A container can be decorated with a **BoxDecoration**, such as a background, a border, or a shadow
  • A Container can also have margins, padding, and constraints applied to its size
Rows and Columns

- Row and Column are classes that contain and lay out widgets
  - Widgets inside of a Row or Column are called children
  - Row and Column are referred to as parents
- Row lays out its widgets horizontally, and Column lays out its widgets vertically
Rows and columns

Oeschinene Lake Campground
Kandersteg, Switzerland

CALL
ROUTE
SHARE

Lake Oeschinene lies at the foot of the Blüemlisalp in the Bernese Alps. Situated 1,576 meters above sea level, it is one of the larger Alpine Lakes. A gondola ride from Kandersteg, followed by a half-hour walk through pastures and pine forest, leads you to the lake, which warms to 20 degrees Celsius in the summer. Activities enjoyed here include rowing, and riding the summer toboggan run.
Another example

Strawberry Pavlova

Pavlova is a meringue-based dessert named after the Russian ballerina Anna Pavlova. Pavlova features a crisp crust and soft, light inside, topped with fruit and whipped cream.

<table>
<thead>
<tr>
<th>Rating</th>
<th>Reviews</th>
</tr>
</thead>
<tbody>
<tr>
<td>★★★★★</td>
<td>170</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prep</th>
<th>Cook</th>
<th>Serves</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 min</td>
<td>1 hr</td>
<td>4-6</td>
</tr>
</tbody>
</table>

Row
2 children

Column
4 children

child: new Column
child: new Image
Structure
Our first widget

class BlueBox extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      width: 50,
      height: 50,
      decoration: BoxDecoration(
        color: Colors.blue,
        border: Border.all(),
      ),
    );
  }
}
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Row(
      children: [
        BlueBox(),
        BlueBox(),
        BlueBox(),
      ],
      textDirection: TextDirection.ltr,
    );
  }
}
MainAxisSize

• Determines how much space a Row and Column can occupy on their main axes
  • MainAxisSize.max Row and Column occupy all of the space on their main axes
    • If the combined width of their children is less than the total space on their main axes, their children are laid out with extra space
  • MainAxisSize.min Row and Column only occupy enough space on their main axes for their children
    • Their children are laid out without extra space and at the middle of their main axes
MainAxisAlignment

- `MainAxisAlignment.start` positions children near the beginning of the main axis (Left for Row, top for Column)
- `MainAxisAlignment.end` positions children near the end of the main axis (Right for Row, bottom for Column)
- `MainAxisAlignment.center` positions children at the middle of the main axis
- `MainAxisAlignment.spaceBetween` divides the extra space evenly between children
- `MainAxisAlignment.spaceEvenly` divides the extra space evenly between children and before and after the children
- `MainAxisAlignment.spaceAround` is similar to `MainAxisAlignment.spaceEvenly`, but reduces half of the space before the first child and after the last child to half of the width between the children
Example

```dart
Widget build(BuildContext context) {
  return Row(
    children: [
      BlueBox(),
      BlueBox(),
      BlueBox(),
    ],
    mainAxisAlignment: MainAxisAlignment.spaceBetween,
    textDirection: TextDirection.ltr,
  );
}
```
crossAxisAlignment

- CrossAxisAlignment.start positions children near the top of the cross axis (Row only)
- CrossAxisAlignment.end positions children near the bottom of the cross axis (Row only)
- CrossAxisAlignment.center positions children at the middle of the cross axis (Row only)
- CrossAxisAlignment.stretch stretches children across the cross axis (Top-to-bottom for Row, left-to-right for Column)
- CrossAxisAlignment.baseline aligns children by their character baselines. (Text class only, and requires that the textBaseline property is set to TextBaseline.alphabetic)
Cross axis
class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
        return Row(
            mainAxisAlignment: MainAxisAlignment.spaceAround,
            crossAxisAlignment: CrossAxisAlignment.center,
            children: [
                BlueBox(),
                BiggerBlueBox(),
                BlueBox(),
            ],
            textDirection: TextDirection.ltr,
        );
    }
}
Flexible

- **Flexible** wraps a widget, so the widget becomes resizable
  - When the Flexible widget wraps a widget, the widget becomes the Flexible widget’s child and is considered flexible
  - After inflexible widgets are laid out, the widgets are resized according to their flex and fit properties
  - Flex compares itself against other flex properties before determining what fraction of the total remaining space each Flexible widget receives
  - Fit determines whether a Flexible widget fills all of its extra space
    - FlexFit.loose The widget’s preferred size is used. (Default)
    - FlexFit.tight Forces The widget to fill all of its extra space
Widget build(BuildContext context) {
  return Row(
    children: [
      BlueBox(),
      Flexible(
        fit: FlexFit.loose,
        flex: 1,
        child: BlueBox(),
      ),
      Flexible(
        fit: FlexFit.tight,
        flex: 1,
        child: BlueBox(),
      ),
    ],
    textDirection: TextDirection.ltr,
  );
}
```dart
Widget build(BuildContext context) {
  return Row(
    children: [
      BlueBox(),
      Flexible(
        fit: FlexFit.tight,
        flex: 1,
        child: BlueBox(),
      ),
      Flexible(
        fit: FlexFit.tight,
        flex: 3,
        child: BlueBox(),
      ),
    ],
    textDirection: TextDirection.ltr,
  );
}
```
• Similar to Flexible, Expanded can wrap a widget and force the widget to fill extra space.

• Use Flexible to resize widgets in a Row or Column
  • That way, you can adjust a child widget’s spacing while keeping its size in relation to its parent widget

• Expanded changes the constraints of a child widget, so it fills any empty space
Example

```dart
Widget build(BuildContext context) {
  return Row(
    children: [
      BlueBox(),
      Expanded(child: BlueBox(),),
      BlueBox(),
    ],
    textDirection: TextDirection.ltr,
  );
}
```
SizedBox

- SizedBox can be used in one of two ways when creating exact dimensions
  - It can wrap a widget and resizes it using the height and width properties.
  - If it doesn’t wrap a widget, it uses the height and width to create empty space

```dart
Widget build(BuildContext context) {
  return Row(
    children: [
      BlueBox(),
      SizedBox(
        width: 100,
        child: BlueBox(),
      ),
      BlueBox(),
    ],
    textDirection: TextDirection.ltr,
  );
}
```
Creating space

```dart
Widget build(BuildContext context) {
  return Row(
    children: [
      BlueBox(),
      SizedBox(width: 50),
      BlueBox(),
      BlueBox(),
    ],
    textDirection: TextDirection.ltr,
  );
}
```
Spacer widget

- Spacer can create space between widgets
- Use Spacer to create space using a flex property
- Use SizedBox to create space using a specific number of logical pixels

```dart
Widget build(BuildContext context) {
  return Row(
    children: [
      BlueBox(),
      Spacer(flex: 1),
      BlueBox(),
      BlueBox(),
    ],
    textDirection: TextDirection.ltr,
  );
}
```
Basic widgets

**Container**
A convenience widget that combines common painting, positioning, and sizing widgets.

**Row**
Layout a list of child widgets in the horizontal direction.

**Column**
Layout a list of child widgets in the vertical direction.

**Image**
A widget that displays an image.

**Text**
A run of text with a single style.

**Icon**
A Material Design icon.
Widget build(BuildContext context) {
  return Row(
    crossAxisAlignment: CrossAxisAlignment.center,
    textBaseline: TextBaseline.alphabetic,
    children: [
      Text('Hey!'
        style: TextStyle(
          fontSize: 30,
          fontFamily: 'Futura',
          color: Colors.blue,
        ),
        textDirection: TextDirection.ltr,
      ),
      ... fontSize: 50,
      color: Colors.green,
      ...
      fontSize: 40,
      color: Colors.red,
    ],
    textDirection: TextDirection.ltr,
  );
}
The **Icon** widget displays a graphical symbol that represents an aspect of the UI.

Flutter is preloaded with icon packages for Material and Cupertino applications.

```dart
Widget build(BuildContext context) {
  return Row(
    crossAxisAlignment: CrossAxisAlignment.center,
    textBaseline: TextBaseline.alphabetic,
    children: [
      Icon(
        Icons.widgets,
        size: 50,
        color: Colors.blue,
        textDirection: TextDirection.ltr,
      ),
      Icon(
        Icons.widgets,
        size: 50,
        color: Colors.red,
        textDirection: TextDirection.ltr,
      ),
    ],
    textDirection: TextDirection.ltr,
  );
}
```
Image widget

```dart
Widget build(BuildContext context) {
  return Row(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
    ],
  );
}
```
Text direction problem

• Flutter doesn't know whether the text is LTR or RTL, so you need to tell it the textDirection explicitly
• Or you can just wrap the Text with a Directionality Widget
• A Text widget in the scope of a MaterialApp widget does not need to be given an explicit writing direction because the default localization in the widgets and material libraries is LTR

```dart
Text('Hello', textDirection: TextDirection.ltr,),
Directionality(textDirection: TextDirection.ltr,
child: new Text('Hello'),),
```
Flutter McFlutter
Experienced App Developer
123 Main Street (415) 555-0198
One simple solution
```dart
return MaterialApp(
  title: 'Welcome to Flutter',
  home: Scaffold(
    appBar: AppBar(
      title: Text('Welcome to Flutter'),
    ),
    body: Column(
      mainAxisSize: MainAxisSize.min,
      mainAxisAlignment: MainAxisAlignment.start,
      mainAxisSize: MainAxisSize.min,
      children: [
        Row(
          children: [
            Padding(
              padding: const EdgeInsets.all(8.0),
              child: Icon(Icons.account_circle, size: 50),
            ),
            Column(
              mainAxisAlignment: MainAxisAlignment.start,
              mainAxisSize: MainAxisSize.min,
              children: [
                Text('Flutter McFlutter',
                  style: Theme.of(context).textTheme.headline),
                Text('Experienced App Developer'),
              ],
            ),
          ],
        ),
      ],
    ),
  ),
);```
SizedBox(height: 8),
Row(
    mainAxisAlignment: MainAxisAlignment.spaceBetween,
    children: [
        Text('123 Main Street'),
        Text('415-555-0198'),
    ],
),
SizedBox(height: 16),
Row(
    mainAxisAlignment: MainAxisAlignment.spaceAround,
    children: [
        Icon(Icons.accessibility, size: 30),
        Icon(Icons.timer, size: 30),
        Icon(Icons.phone_android, size: 30),
        Icon(Icons.phone_iphone, size: 30),
    ],
),
)
```dart
body: FlatButton(
  color: Colors.blue,
  textColor: Colors.white,
  disabledColor: Colors.grey,
  disabledTextColor: Colors.black,
  padding: EdgeInsets.all(8.0),
  splashColor: Colors.blueAccent,
  onPressed: () {
    /*...*/
  },
  child: Text(
    'Flat Button',
    style: TextStyle(fontSize: 20.0),
  ),
),
```
body: ListView(
  padding: EdgeInsets.all(8),
  children: [
    Container(
      height: 50,
      color: Colors.amber[600],
      child: Center(child: Text('Entry A')),
    ),
    Container(
      height: 50,
      color: Colors.amber[500],
      child: Center(child: Text('Entry B')),
    ),
    Container(
      height: 50,
      color: Colors.amber[100],
      child: Center(child: Text('Entry C')),
    ),
  ],
),
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Retrieve Text Input',
      home: MyCustomWidget(),
    );
  }
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return CupertinoApp(
      title: 'Flutter SafeArea Cupertino',
      home: CupertinoPageScaffold(
        backgroundColor: CupertinoColors.extraLightBackgroundGray,
        navigationBar: CupertinoNavigationBar(
          backgroundColor: CupertinoColors.lightBackgroundGray,
          middle: Text('SafeArea Test Page',)
        ),
        child: Screen(),
      );
  }
}
class Screen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Row(
        children: [
          BlueBox(),
          BlueBox(),
          BlueBox(),
        ],
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
      ),
    );
  }
}
Wireframes

- press "p" in the console
- "Toggle Debug Painting" command in VS Code
Lake Oeschinensee lies at the foot of the Blüemlisalp in the Bernese Alps. Situated 1,578 meters above sea level, it is one of the larger Alpine Lakes. A gondola ride from Kandersteg, followed by a half-hour walk through pastures and pine forest, leads you to the lake, which warms to 20 degrees Celsius in the summer. Activities enjoyed here include rowing and riding the summer toboggan run.
Assets (pubspec.yaml)

- To include all assets under a directory, specify the directory name with the `/` character at the end
- Only files located directly in the directory are included
- To add files located in subdirectories, create an entry per directory

```yaml
flutter:
  uses-material-design: true
assets:
  - images/lake.jpg
  - assets/background.png
```
Stateful widget

- Stateless widgets are immutable
- Stateful widgets maintain state that might change during the lifetime of the widget
  - A StatefulWidget class that creates an instance of
  - A State class
Stateful widgets

- Implemented by two classes: an extension of StatefulWidget and an extension of State
- Can change their appearance in response to events triggered by user interactions or when they receive data
  - Checkbox, Radio, Slider, InkWell, Form, and TextField are examples
- A widget’s state is stored in a State object
  - State and appearance are kept separated
  - State consists of values that can change, like a slider’s current value
  - State contains the widget’s mutable state and the widget’s build() method
  - When the widget’s state changes, the state object calls setState(), telling the framework to redraw the widget
Initial app
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.indigo,
        fontFamily: 'Arial',
        textTheme: TextTheme(
          headline: TextStyle(fontSize: 72.0, fontWeight: FontWeight.bold),
          title: TextStyle(fontSize: 36.0, fontStyle: FontStyle.italic),
          body1: TextStyle(fontSize: 16.0, fontFamily: 'Hind'),
        ),
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
primarySwatch

• primarySwatch is not a Color, but a MaterialColor
  • It is all the different shades of a color a material app can use
• PrimaryColor is one of those shades
  • It is normally equal to primarySwatch[500]
• It is usually better to define a primarySwatch instead of a primaryColor
  • Some material components may use a different shade of the primaryColor for things such as shadow, border, ...
ThemeData constructor

ThemeData({
    Brightness brightness,
    MaterialColor primarySwatch,
    Color primaryColor,
    Brightness primaryColorBrightness,
    Color primaryColorLight,
    Color primaryColorDark,
    Color accentColor,
    Brightness accentColorBrightness,
    Color canvasColor,
    Color scaffoldBackgroundColor,
    Color bottomAppBarColor,
    Color cardColor,
    Color dividerColor,
    Color focusColor,
    Color hoverColor,
    Color highlightColor,
    Color splashColor,
    InteractiveInkFeatureFactory splashFactory,
    Color selectedRowColor,
    Color unselectedWidgetColor,
    Color disabledColor,
    Color buttonColor,
    ButtonThemeData buttonTheme,
    ToggleButtonsThemeData toggleButtonsTheme,
    Color secondaryHeaderColor,
    Color textSelectionColor,
    Color cursorColor,
    Color textSelectionHandleColor,
    Color backgroundColor,
    Color dialogBackgroundColor,
    Color dialogIndicatorColor,
    Color hintColor,
})
Stateful widget

- This class is the configuration for the state
- It holds the values (title) provided by the parent (App) and used by the build method of the State
- Fields in a Widget subclass are always marked `final`

```dart
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```
State

• The call to setState tells Flutter that something has changed in this State, which causes it to rerun the build method
  • The display can then reflect the updated values
  • If we changed _counter without calling setState(), build would not be called and nothing would appear to happen

```dart
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
}
```
```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Text('You have pushed the button this many times:'),
          Text('${_counter}',
            style: Theme.of(context).textTheme.display1,
          ),
        ],
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: Icon(Icons.add),
    ),
  );
}
```
• Method Theme.of(context) looks up the widget tree and returns the nearest Theme in the tree
  • If you have a standalone Theme defined above your widget, that’s returned
  • If not, the app’s theme is returned
The method is rerun every time setState() is called, for instance as done by method _incrementCounter.

Flutter has been optimized to make rerunning build methods fast.

- You can just rebuild anything that needs updating rather than having to individually change instances of widgets.
Another example

class FavoriteWidget extends StatefulWidget {
  @override
  _FavoriteWidgetState createState() => _FavoriteWidgetState();
}

class _FavoriteWidgetState extends State<FavoriteWidget> {
  bool _isFavorited = true;
  int _favoriteCount = 41;

  void _toggleFavorite() {
    setState(() {
      if (_isFavorited) {
        _favoriteCount -= 1;
        _isFavorited = false;
      } else {
        _favoriteCount += 1;
        _isFavorited = true;
      }
    });
  }
}
@override
Widget build(BuildContext context) {
  return Row(
    mainAxisSize: MainAxisSize.min,
    children: [
      Container(
        padding: EdgeInsets.all(0),
        child: IconButton(
          icon: (_isFavorited ? Icon(Icons.star) : Icon(Icons.star_border)),
          color: Colors.red[500],
          onPressed: _toggleFavorite,
        ),
      ),
      SizedBox(
        width: 18,
        child: Container(
          child: Text('$_favoriteCount'),
        ),
      ),
    ],
  );
}
Who manages the stateful widget’s state?

• If the state is user data, for example the checked or unchecked mode of a checkbox, the state is best managed by the parent widget
• If the state in question is aesthetic, for example an animation, then the state is best managed by the widget itself
• If in doubt, start by managing the state in the parent widget
Widget manages its own state

• Class _TapboxAState
  • Manages state for TapboxA
  • Defines boolean _active that determines the box’s current color
  • Defines method _handleTap() that updates _active when the box is tapped and calls method setState() to update the UI
  • Implements all interactive behavior for the widget
```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter Demo'),
        ),
        body: Center(
          child: TapboxA(),
        ),
      ),
    );
  }
}

class TapboxA extends StatefulWidget {
  @override
  _TapboxAState createState() => _TapboxAState();
}
```
class _TapboxAState extends State<TapboxA> {
  bool _active = false;

  void _handleTap() {
    setState(() {
      _active = !_active;
    });
  }

  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _handleTap,
      child: Container(
        child: Center(
          child: Text(
            _active ? 'Active' : 'Inactive',
            style: TextStyle(fontSize: 32.0, color: Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color: _active ? Colors.lightGreen[700] : Colors.grey[600],
        ),
      );
    }
  }
}
Parent widget manages the widget’s state

• IconButton is a stateless widget because the parent widget needs to know whether the button has been tapped

• TapboxB exports its state to its parent through a callback
  • Because TapboxB doesn’t manage any state, it subclasses StatelessWidget

• Class ParentWidgetState
  • Manages state _active for TapboxB
  • Implements method _handleTapboxChanged() called when the box is tapped
  • When the state changes, calls setState() to update the UI

• Class TapboxB
  • Extends StatelessWidget because all state is handled by its parent
  • When a tap is detected, it notifies the parent
import 'package:flutter/material.dart';
import 'package:flutter/foundation.dart';

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

class ParentWidget extends StatefulWidget {
  @override
  _ParentWidgetState createState() => _ParentWidgetState();
}

class _ParentWidgetState extends State<ParentWidget> {
  bool _active = false;

  void _handleTapboxChanged(bool newValue) {
    setState(() {
      _active = newValue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Container(
      child: TapboxB(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),
    );
  }
}
class TapboxB extends StatelessWidget {
  TapboxB({Key key, this.active: false, @required this.onChanged})
    : super(key: key);

  final bool active;
  final ValueChanged<bool> onChanged;

  void _handleTap() {
    onChanged(!active);
  }

  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _handleTap,
      child: Container(
        child: Center(
          child: Text(
            active ? 'Active' : 'Inactive',
            style: TextStyle(fontSize: 32.0, color: Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color: active ? Colors.lightGreen[700] : Colors.grey[600],
        ),
      );
    }
  }
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Flutter Demo'),
        ),
        body: Center(
          child: ParentWidget(),
        ),
      ),
    );
  }
}
Use an open-source package named `english_words`, which contains a few thousand of the most used English words plus some utility functions

- Available from: https://pub.dev/flutter
- Edit `pubspec.yaml`
While viewing the pubspec execute flutter pub get

- This pulls the package(s) into your project
- It also auto-generates the pubspec.lock file with a list of all packages pulled into the project and their version numbers

```yaml
dependencies:
  flutter:
    sdk: flutter

# The following adds the Cupertino Icons font to your application.
# Use with the CupertinoIcons class for iOS style icons.
cupertino_icons: ^0.1.2
english_words: ^3.1.0
```
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final wordPair = WordPair.random();
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Welcome to Flutter'),
          body: Center(
            child: Text(wordPair.asPascalCase),
          ),
        ),
      ),
    );
  }
}
Flutter Packages

Search Flutter-compatible packages

Flutter makes it easy and fast to build beautiful mobile apps for iOS and Android.

Top Flutter-compatible packages

**shared_preferences**
- **FLUTTER**
  Flutter plugin for reading and writing simple key-value pairs. Wraps NSUserDefaults on iOS and SharedPreferences on Android.

**url_launcher**
- **FLUTTER**
  Flutter plugin for launching a URL on Android and iOS. Supports web, phone, SMS, and email schemes.

**image_picker**
- **FLUTTER**
  Flutter plugin for selecting images from the Android and iOS image library, and taking new pictures with the camera.

**cloud_firestore**
- **FLUTTER**
  Flutter plugin for Cloud Firestore, a cloud-hosted, noSQL database with live synchronization and offline support on Android and iOS.

**path_provider**
- **FLUTTER**
  Flutter plugin for getting commonly used locations on the Android & iOS file systems, such as the temp and app data directories.

**provider**
- **FLUTTER**
  A mixture between dependency injection and state management, built with widgets for widgets.
A curated list of samples

Contains in this list are sample apps, demos, and examples that can help you grow your Flutter skills. Some are maintained here by the Flutter team, but many have been created by the Flutter community and are kept in other repos in and out of GitHub.

This is not an exhaustive list of samples, and just because a project isn’t listed here doesn’t mean that it's not worth exploring. Similarly, while the Flutter team works to keep this list up to date, there are plenty of others created by the community, such as Awesome Flutter from @f0id0.

Please don’t submit pull requests directly updating this file. While we're always happy to learn of new samples from the community, we need to keep this file small. There are plenty of user-maintained indices (like Awesome Flutter) that are meant to be exhaustive, and these are great places for submitting your own work.

How-to Collections

Flutter Examples
A collection of single-topic examples from Nishant Srikantana. These include everything from gradients to JSON to routing and more.

Flutter Example Apps
A few dozen examples from Flutter GDE Pawan Kumar. These cover state management, Firebase, UI design, and many other topics. Each one comes with a YouTube video showing implementation.

Flutter by Example
Twenty-odd samples detailing common tasks with Flutter: how to use text fields, streams and StreamBuilders, and more.

Flutter for web samples (Flutter team)
A collection of Flutter samples structured as Flutter for web projects. You can see them in action at https://flutter.github.io/samples.

Architecture / networking / backend

Flutter Architectural Samples
Almost all applications comprise diverse screens

In Flutter, screens and pages are called routes
  - In Android, a route is equivalent to an Activity
  - In iOS, a route is equivalent to a ViewController
  - In Flutter, a route is just a widget

Flutter provides Navigator to navigate to new routes
Navigate to a new screen and back

- We can create new routes or use `MaterialPageRoute`
  - It transitions to the new route using a platform-specific animation

- Navigate to the second route using `Navigator.push()`
  - `push()` adds a Route to the stack of routes managed by the Navigator

- Return to the first route using `Navigator.pop()`
  - `pop()` removes the current Route from the stack of routes managed by the navigator
Example (FirstRoute)

class FirstRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First Route'),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Open route'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                builder: (context) => SecondRoute(),
              ),
            );
          },
        ),
      ),
    );
  }
}
Example (SecondRoute)

```dart
class SecondRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Second Route"),
      ),
      body: Center(
        child: RaisedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: Text('Go back!'),
        ),
      ),
    );
  }
}
```
Animations

• When a PageRoute is pushed or popped with the Navigator, the entire screen's content is replaced
  • An old route disappears and a new route appears
  • If there's a common visual feature on both routes then it can be helpful for orienting the user for the feature to physically move from one page to the other during the routes' transition
• Such an animation is called a hero animation
Example
Hero

- It’s often helpful to guide users through an app as they navigate from screen to screen
- A common technique to lead users through an app is to animate a widget from one screen to the next
- This creates a visual anchor connecting the two screens
- To connect two screens together with an animation, wrap the widget on both screens in a Hero widget
  - `tag` An object that identifies the `Hero`. It must be the same on both screens
  - `child` The widget to animate across screens
```dart
class MainScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Main Screen'),
      ),
      body: GestureDetector(
        onTap: () {
          Navigator.push(context, MaterialPageRoute(
            builder: (context) => DetailScreen()
          ));
        },
      ),
    );
  }
}
```
class DetailScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: GestureDetector(
        onTap: () {
          Navigator.pop(context);
        },
        child: Center(
        ),
      ),
    );
  }
}
Hero animation (both images)

class DetailScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: GestureDetector(
        onTap: () {
          Navigator.pop(context);
        },
        child: Center(
          child: Hero(
            tag: 'imageHero',
          ),
        ),
      ),
    );
  }
}
Navigate with named routes

• If you need to navigate to the same screen in many parts of your app, you can define a named route, and use the named route for navigation
• Named routes require method Navigator.pushNamed()
Named routes

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

guyễn main()
{  
  runApp(MaterialApp(
      title: 'Named Routes Demo',
      initialRoute: '/',
      routes: {
        '/': (context) => FirstScreen(),
        '/second': (context) => SecondScreen(),
      },
  ));
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Named Routes Demo',
      initialRoute: '/',
      routes: {
        '/': (context) => FirstScreen(),
        '/second': (context) => SecondScreen(),
      },
    );
  }
}
```
class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First Screen'),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Launch screen'),
          onPressed: () {
            Navigator.pushNamed(context, '/second');
          },
        ),
      ),
    );
  }
}
class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Second Screen"),
      ),
      body: Center(
        child: RaisedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: Text('Go back!'),
        ),
      ),
    );
  }
}

Second screen
Working with tabs is a common pattern in apps that follow the Material Design guidelines

- Create a TabController
- Create the tabs
- Create content for each tab
class TabBarDemo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: DefaultTabController(
        length: 3,
        child: Scaffold(
          appBar: AppBar(
            bottom: TabBar(
              tabs: [Tab(icon: Icon(Icons.directions_car)),
                    Tab(icon: Icon(Icons.directions_transit)),
                    Tab(icon: Icon(Icons.directions_bike))],
            ),
          title: Text('Tabs Demo'),
        ),
      ),
      body: TabBarView(
        children: [
          Icon(Icons.directions_car),
          Icon(Icons.directions_transit),
          Icon(Icons.directions_bike),
        ],
      ),
    );
  }
}
Pass arguments to a named route

• In some cases, when navigating you might also need to pass arguments to a named route
  • For example, you might wish to navigate to the /user route and pass information about the user to that route

• Method Navigator.pushNamed() has a parameter argument for this

• You must
  • Extract the arguments by using method ModalRoute.of or inside a method onGenerateRoute() provided by the MaterialApp or CupertinoApp constructor
Step 1

• Define the arguments you need to pass

```java
class ScreenArguments {
    final String title;
    final String message;

    ScreenArguments(this.title, this.message);
}
```
Step 2

Create a widget that extracts the arguments

```dart
class ExtractArgumentsScreen extends StatelessWidget {
  static const routeName = '/extractArguments';

  @override
  Widget build(BuildContext context) {
    final ScreenArguments args = ModalRoute.of(context).settings.arguments;

    return Scaffold(
      appBar: AppBar(
        title: Text(args.title),
      ),
      body: Center(
        child: Text(args.message),
      ),
    );
  }
}
```
Step 3

Register the widget in the routes table

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      routes: {
        ExtractArgumentsScreen.routeName: (context) => ExtractArgumentsScreen(),
      },
      title: 'Navigation with Arguments',
      home: HomeScreen(),
    );
  }
}
Step 4

- Finally, navigate to the ExtractArgumentsScreen when a user taps a button using Navigator.pushNamed()
- Provide the arguments to the route via the arguments property
- The ExtractArgumentsScreen extracts the title and message from these arguments
class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home Screen'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            RaisedButton(
              child: Text('Navigate to screen that extracts arguments'),
              onPressed: () {
                Navigator.pushNamed(
                  context,
                  ExtractArgumentsScreen.routeName,
                  arguments: ScreenArguments(
                    'Extract Arguments Screen',
                    'This message is extracted in the build method.',
                  ),
                );
              },
            ),
          ],
        ),
      ),
    );
  }
}

Alternatively

• extract the arguments using `onGenerateRoute`

• Instead of extracting the arguments directly inside the widget, you can also extract the arguments inside a method `onGenerateRoute()` and pass them to a widget

• Method `onGenerateRoute()` creates the correct route based on the given `RouteSettings`
Return data from a screen

• In some cases, you might want to return data from a new screen
  • For example, you push a new screen that presents two options to a user
  • When the user taps an option, you want to inform the first screen of the user’s selection so that it can act on that information

• You can do this with method Navigator.pop() using the following steps
  • Define the home screen
  • Add a button that launches the selection screen
  • Show the selection screen with two buttons
  • When a button is tapped, close the selection screen
  • Show a snackbar on the home screen with the selection
Define the home screen

The home screen displays a button. When tapped, it launches the selection screen.

class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Returning Data Demo'),
      ),
      body: Center(child: SelectionButton()),
    );
  }
}
Add a button that launches the selection screen

- Now, create the `SelectionButton` that
  - Launches the `SelectionScreen` when it’s tapped
  - Waits for the `SelectionScreen` to return a result

```dart
class SelectionButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return RaisedButton(
      onPressed: () {
        _navigateAndDisplaySelection(context);
      },
      child: Text('Pick an option, any option!'),
    );
  }
}
```
Show the selection screen with two buttons

- Build a selection screen that contains two buttons
- When a user taps a button, the app closes the selection screen and lets the home screen know which button was tapped

```dart
class SelectionScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Pick an option'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Padding(
              padding: const EdgeInsets.all(8.0),
              child: RaisedButton(
                onPressed: () {
                  // Pop here with "Yep"
                },
                child: Text('Yep'),
              ),
            ),
            Padding(
              padding: const EdgeInsets.all(8.0),
              child: RaisedButton(
                onPressed: () {
                  // Pop here with "Nope"
                },
                child: Text('Nope'),
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```
When a button is tapped, close the selection screen

• To return data to the first screen, use method Navigator.pop() that accepts an optional second argument called result
• Any result is then returned to the Future in the SelectionButton

```dart
RaisedButton(
onPressed: () {
  Navigator.pop(context, 'Yep!');
},
  child: Text('Yep!'),
),

RaisedButton(
onPressed: () {
  Navigator.pop(context, 'Nope.');
},
  child: Text('Nope.'),
),
```
Show a snackbar on the home screen with the selection

• Do something with the information that’s returned
  • Show a snackbar displaying the result by using method _navigateAndDisplaySelection() in SelectionButton

```dart
_navigateAndDisplaySelection(BuildContext context) async {
  final result = await Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => SelectionScreen()),
  );

  Scaffold.of(context)
    ..removeCurrentSnackBar()
    ..showSnackBar(SnackBar(content: Text("$result")));
}
```
Send data to a new screen

• Often, you not only want to navigate to a new screen, but also pass data to the screen as well
  • For example, you might want to pass information about the item that’s been tapped.
• For example, you can create a list of todos
  • Define a todo class
  • Display a list of todos
  • Create a detail screen that can display information about a todo
  • Navigate and pass data to the detail screen
Class Todo

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

class Todo {
  final String title;
  final String description;

  Todo(this.title, this.description);
}

void main() {
  runApp(MaterialApp(
    title: 'Passing Data',
    home: TodosScreen(
      todos: List.generate(20, (i) => Todo('Todo $i',
          'A description of what needs to be done for Todo $i',)
          ,
        ),
      )
    );
  }
```

class TodosScreen extends StatelessWidget {
  final List<Todo> todos;

  TodosScreen({Key key, @required this.todos}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Todos'),
      ),
      body: ListView.builder(
        itemCount: todos.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(todos[index].title),
            onTap: () {
              Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (context) => DetailScreen(todo: todos[index]),
                ),
              );
            },
          );
        },
      );
    }
  }
}
class DetailScreen extends StatelessWidget {
  final Todo todo;

  DetailScreen({Key key, @required this.todo}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(todo.title),
      ),
      body: Padding(
        padding: EdgeInsets.all(16.0),
        child: Text(todo.description),
      ),
    );
  }
}
Example
Fetch data from the internet

- Fetching data from the internet is necessary for most apps
- Dart and Flutter provide tools, such as the http package, for this type of work
  - Add the http package
  - Make a network request using the http package
  - Convert the response into a custom Dart object
  - Fetch and display the data with Flutter
import 'dart:async';
import 'dart:convert';

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

void main() => runApp(MyApp(post: fetchPost()));

Future<Post> fetchPost() async {
  final response = await http.get('https://jsonplaceholder.typicode.com/posts/1');

  if (response.statusCode == 200) {
    return Post.fromJson(json.decode(response.body));
  } else {
    throw Exception('Failed to load post');
  }
}
• The http package provides the simplest way to fetch data from the internet
• To install the http package, add it to the dependencies section of the pubspec.yaml

dependencies:
  http: ^0.12.0+2
  flutter:
    sdk: flutter
Dart

• Future is a core Dart class for working with async operations
  • It represents a potential value or error that will be available at some time in the future

• One unnamed constructor
• Many named constructors  
  • <ClassName>.<methodName>

• A factory constructor does not always create a new instance of its class  
  • It might return an instance from a cache, or it might return an instance of a subtype
Example (II)

```java
class Post {
    final int userId;
    final int id;
    final String title;
    final String body;

    Post({this.userId, this.id, this.title, this.body});

    factory Post.fromJson(Map<String, dynamic> json) {
        return Post(
            userId: json['userId'],
            id: json['id'],
            title: json['title'],
            body: json['body'],
        );
    }
}
```
class MyApp extends StatelessWidget {
  final Future<Post> post;

  MyApp({Key key, this.post}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Fetch Data Example',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Scaffold(
        appBar: AppBar(
          title: Text('Fetch Data Example'),
        ),
        body: Center(
          child: FutureBuilder<Post>(
            future: post,
            builder: (context, snapshot) {
              if (snapshot.hasData) {
                return Text(snapshot.data.title);
              } else if (snapshot.hasError) {
                return Text("${snapshot.error}");
              }
              return CircularProgressIndicator();
            },
          ),
        ),
      );
    }
  }
}
State management

• It’s okay to rebuild parts of your UI from scratch instead of modifying it
  • Flutter is fast enough to do that, even on every frame if needed

• Flutter is declarative, that is, it builds its user interface to reflect the current state of your app

• When the state of your app changes, you change the state, and that triggers a redraw of the user interface
Ephemeral state

- Sometimes called *local state* is the state you can neatly contain in a single widget
- There is no need to serialize it, and it doesn’t change in complex ways
- There is no need to use state management techniques on this kind of state
- All you need is a StatefulWidget
App state

• State that you want to share across many parts of your app, and that you want to keep between user sessions
  • User preferences
  • Login info
  • Notifications in a social networking app
  • The shopping cart in an e-commerce app
  • Read/unread state of articles in a news app

• For managing app state, you have many options and your choice depends on many aspects
How can we decide?

Data

Who needs it?

Most widgets

Some widgets

Single widget

App state

Ephemeral state
A few different options

Redux
A predictable state container for JavaScript apps

BLoC
A predictable state management library for Dart

MobX
A simple, scalable and battle tested state management solution