‹ Test drive Learn more ›

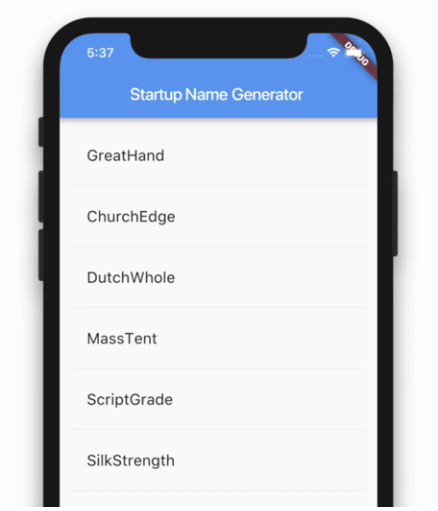
Write your first Flutter app, part 1

Get started > Write your first app

Contents

Tip: This codelab walks you through writing your first Flutter app. You might prefer to try writing your first Flutter app on the web.

If you prefer an instructor-led version of this codelab, check out the following workshop:



This is a guide to creating your first Flutter app. If you are familiar with object-oriented code and basic programming concepts such as variables, loops, and conditionals, you can complete this tutorial. You don't need previous experience with Dart, mobile, desktop, or web programming.

This codelab is part 1 of a two-part codelab. You can find part 2 on Google Developers Codelabs (as well as a copy of this codelab, part 1).

What you'll build in part 1

You'll implement a simple app that generates proposed names for a startup company. The user can select and unselect names, saving the best ones. The code lazily generates 10 names at a time. As the user scrolls, more names are generated. There is no limit to how far a user can scroll.

The animated GIF shows how the app works at the completion of part 1.

What you'll learn in part 1

- How to write a Flutter app that looks natural on iOS, Android, desktop (Windows, for example), and the web
- Basic structure of a Flutter app
- Finding and using packages to extend functionality
- Using hot reload for a quicker development cycle
- How to implement a stateful widget
- How to create an infinite, lazily loaded list

In part 2 of this codelab, you'll add interactivity, modify the app's theme, and add the ability to navigate to a new screen (called a *route* in Flutter).

What you'll use

You need two pieces of software to complete this lab: the Flutter SDK and an editor. This codelab assumes Android Studio, but you can use your preferred editor.

You can run this codelab by using any of the following devices:

- A physical device (Android or iOS) connected to your computer and set to developer mode
- The iOS simulator (requires installing Xcode tools)
- The Android emulator (requires setup in Android Studio)
- A browser (Chrome is required for debugging)
- As a Windows, Linux, or macOS desktop application

Every Flutter app you create also compiles for the web. In your IDE under the **devices** pulldown, or at the command line using `flutter devices`, you should now see **Chrome** and **Web server** listed. The **Chrome** device automatically starts Chrome. The **Web server** starts a server that hosts the app so that you can load it from any browser. Use the Chrome device during development so that you can use DevTools, and the web server when you want to test on other browsers. For more information, see Building a web application with Flutter and Write your first Flutter app on the web.

Also, Flutter apps can compile for desktop. You should see your operating system listed in your IDE under **devices**, for example: **Windows (desktop)**, or at the command line using `flutter devices`. For more information on building apps for desktop, see Write a Flutter desktop application.

startup_namer (instead of *flutter_app*).

> **Tip:** If you don't see "New Flutter Project" as an option in your IDE, make sure you have the plugins installed for Flutter and Dart.

You'll mostly edit **lib/main.dart**, where the Dart code lives.

1. Replace the contents of `lib/main.dart`.
   Delete all of the code from **lib/main.dart**. Replace with the following code, which displays "Hello World" in the center of the screen.

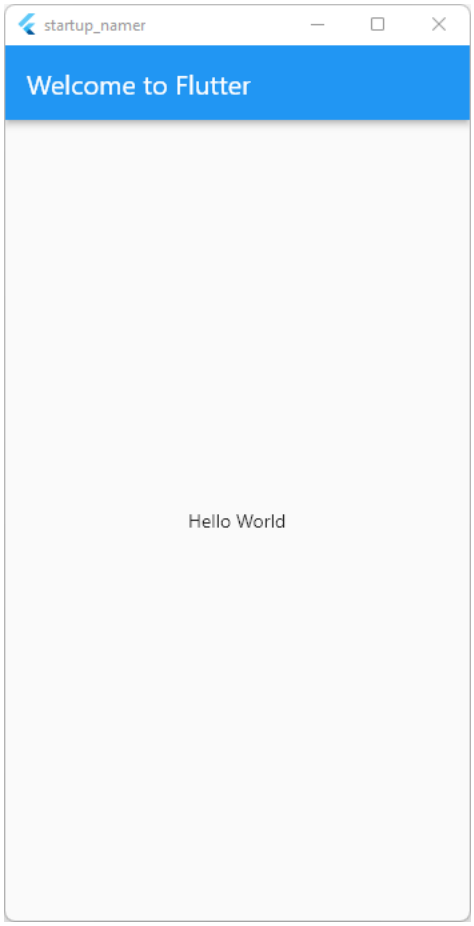
lib/main.dart

```dart
// Copyright 2018 The Flutter team. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Welcome to Flutter'),
        ),
        body: const Center(
          child: Text('Hello World'),
        ),
      ),
```

tools:

- Android Studio and IntelliJ IDEA: Right-click the code and select **Reformat Code with dartfmt**.
- VS Code: Right-click and select **Format Document**.
- Terminal: Run `flutter format <filename>`.

2. Run the app in the way your IDE describes. You should see either Android, iOS, Windows, Linux, macOS, or web output, depending on your device.



Windows

iOS

Tip: The first time you run on a physical device, it can take a while to load. Afterward, you can use hot reload for quick updates. **Save** also performs a hot reload if the app is running. When running an app directly from the console using `flutter run`, enter `r` to perform hot reload.

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. It's a good idea to have a `uses-material-design: true` entry in the `flutter` section of your `pubspec.yaml` file. This will allow you to use more features of Material, such as their set of predefined Icons.
- The app extends `StatelessWidget`, which makes the app itself a widget. In Flutter, almost everything is a widget, including alignment, padding, and layout.
- The `Scaffold` widget, from the Material library, provides a default app bar, and a body property that holds the widget tree for the home screen. The widget subtree 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

Step 2: Use an external package

In this step, you'll start using an open-source package named english_words, which contains a few thousand of the most used English words plus some utility functions.

You can find the `english_words` package, as well as many other open source packages, on pub.dev.

1. Add `english_words` package to your project as follows:

```
$ flutter pub add english_words
Resolving dependencies...
+ english_words 4.0.0
  path 1.8.0 (1.8.1 available)
  source_span 1.8.1 (1.8.2 available)
  test_api 0.4.3 (0.4.9 available)
Downloading english_words 4.0.0...
Changed 1 dependency!
```

   The `pubspec.yaml` file manages the assets and dependencies for a Flutter app. In `pubspec.yaml`, you will see that the `english_words` dependency has been added:

{step1_base → step2_use_package}/pubspec.yaml
☐ Viewed

```
@@ -9,4 +9,5 @@
  dependencies:
    flutter:
      sdk: flutter
    cupertino_icons: ^1.0.2
+   english_words: ^4.0.0
```

2. While viewing the `pubspec.yaml` file in Android Studio's editor view, click **Pub get**. This pulls the package into your project. You should see the following in the console:

```
$ flutter pub get
Running "flutter pub get" in startup_namer...
Process finished with exit code 0
```

   Performing `Pub get` also auto-generates the `pubspec.lock` file with a list of all packages pulled into the project and their version numbers.

3. In `lib/main.dart`, import the new package:

lib/main.dart

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

   As you type, Android Studio gives you suggestions for libraries to import. It then renders the import string in gray, letting you know that the imported library is unused (so far).

4. Use the English words package to generate the text instead of using the string "Hello World":

{step1_base → step2_use_package}/lib/main.dart
☐ Viewed

```
@@ -2,6 +2,7 @@
```

```
@@ -13,14 +14,15 @@
    @override
    Widget build(BuildContext context) {
+     final wordPair = WordPair.random();
      return MaterialApp(
        title: 'Welcome to Flutter',
        home: Scaffold(
          appBar: AppBar(
            title: const Text('Welcome to Flutter'),
          ),
-         body: const Center(
-           child: Text('Hello World'),
+         body: Center(
+           child: Text(wordPair.asPascalCase),
          ),
        ),
      );
```

Note: "Pascal case" (also known as "upper camel case"), means that each word in the string, including the first one, begins with an uppercase letter. So, "uppercamelcase" becomes "UpperCamelCase".

5. If the app is running, hot reload to update the running app. Each time you click hot reload, or save the project, you should see a different word pair, chosen at random, in the running app. This is because the word pairing is generated inside the build method, which is run each time the `MaterialApp` requires rendering, or when toggling the Platform in Flutter Inspector.

Windows

iOS

- pubspec.yaml
- lib/main.dart

Step 3: Add a Stateful widget

State*less* widgets are immutable, meaning that their properties can't change—all values are final.

State*ful* widgets maintain state that might change during the lifetime of the widget. Implementing a stateful widget requires at least two classes: 1) a `StatefulWidget` class that creates an instance of 2) a `State` class. The `StatefulWidget` class is, itself, immutable and can be thrown away and regenerated, but the `State` class persists over the lifetime of the widget.

In this step, you'll add a stateful widget, `RandomWords`, which creates its `State` class, `_RandomWordsState`. You'll then use `RandomWords` as a child inside the existing `MyApp` stateless widget.

1. Create the boilerplate code for a stateful widget.
   In `lib/main.dart`, position your cursor after all of the code, enter **Return** a couple times to start on a fresh line. In your IDE, start typing `stful`. The editor asks if you want to create a `Stateful` widget. Press **Return** to accept. The boilerplate code for two classes appears, and the cursor is positioned for you to enter the name of your stateful widget.

2. Enter `RandomWords` as the name of your widget.
   The `RandomWords` widget does little else beside creating its `State` class.

   Once you've entered `RandomWords` as the name of the stateful widget, the IDE automatically updates the accompanying `State` class, naming it `_RandomWordsState`. By default, the name of the `State` class is prefixed with an underbar. Prefixing an identifier with an underscore enforces privacy in the Dart language and is a recommended best practice for `State` objects.

   The IDE also automatically updates the state class to extend `State<RandomWords>`, indicating that you're using a generic `State` class specialized for use with `RandomWords`. Most of the app's logic resides here—it maintains the state for the `RandomWords` widget. This class saves the list of generated word pairs, which grows infinitely as the user scrolls and, in part 2 of this lab, favorites word pairs as the user adds or removes them from the list by toggling the heart icon.

   Both classes now look as follows:

```dart
class RandomWords extends StatefulWidget {
  const RandomWords({ Key? key }) : super(key: key);

  @override
  State<RandomWords> createState() => _RandomWordsState();
}

class _RandomWordsState extends State<RandomWords> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

3. Update the `build()` method in `_RandomWordsState`:

lib/main.dart (_RandomWordsState)

```dart
class _RandomWordsState extends State<RandomWords> {
    @override
    Widget build(BuildContext context) {
      final wordPair = WordPair.random();
      return Text(wordPair.asPascalCase);
    }
  }
```

Viewed

```
@@ -14,16 +14,15 @@
    @override
    Widget build(BuildContext context) {
-     final wordPair = WordPair.random();
      return MaterialApp(
        title: 'Welcome to Flutter',
        home: Scaffold(
          appBar: AppBar(
            title: const Text('Welcome to Flutter'),
          ),
-         body: Center(
-           child: Text(wordPair.asPascalCase),
+         body: const Center(
+           child: RandomWords(),
          ),
        ),
      );
    }
```

5. Restart the app. The app should behave as before, displaying a word pairing each time you hot reload or save the app.

> **Tip:** If you see a warning on a hot reload that you might need to restart the app, consider restarting it. The warning might be a false positive, but restarting your app ensures that your changes are reflected in the app's UI.

Problems?

If your app is not running correctly, look for typos. If you want to try some of Flutter's debugging tools, check out the DevTools suite of debugging and profiling tools. If needed, use the code at the following link to get back on track.

- lib/main.dart

Step 4: Create an infinite scrolling ListView

In this step, you'll expand `_RandomWordsState` to generate and display a list of word pairings. As the user scrolls the list (displayed in a `ListView` widget) grows infinitely. `ListView`'s `builder` factory constructor allows you to build a list view lazily, on demand.

1. Add a `_suggestions` list to the `_RandomWordsState` class for saving suggested word pairings. Also, add a `_biggerFont` variable for making the font size larger.

lib/main.dart

```
class _RandomWordsState extends State<RandomWords> {
  final _suggestions = <WordPair>[];
  final _biggerFont = const TextStyle(fontSize: 18);
  // ···
}
```

2. Next, you'll add a `ListView` widget to the `_RandomWordsState` class with the `ListView.builder` constructor. This method creates the `ListView` that displays the suggested word pairing.

   The `ListView` class provides a builder property, `itemBuilder`, that's a factory builder and callback function specified as an anonymous function. Two parameters are passed to the function—the `BuildContext`, and the row iterator, `i`. The iterator begins at 0 and increments each time the function is called. It increments twice for every suggested word pairing: once for the ListTile,

```
return ListView.builder(
  padding: const EdgeInsets.all(16.0),
  itemBuilder: /*1*/ (context, i) {
    if (i.isOdd) return const Divider(); /*2*/

    final index = i ~/ 2; /*3*/
    if (index >= _suggestions.length) {
      _suggestions.addAll(generateWordPairs().take(10)); /*4*/
    }
    return Text(_suggestions[index].asPascalCase);
  },
);
```

`/*1*/` The `itemBuilder` callback is called once per suggested word pairing, and places each suggestion into a `ListTile` row. For even rows, the function adds a `ListTile` row for the word pairing. For odd rows, the function adds a `Divider` widget to visually separate the entries. Note that the divider might be difficult to see on smaller devices.

`/*2*/` Add a one-pixel-high divider widget before each row in the `ListView`.

`/*3*/` The expression `i ~/ 2` divides `i` by 2 and returns an integer result. For example: 1, 2, 3, 4, 5 becomes 0, 1, 1, 2, 2. This calculates the actual number of word pairings in the `ListView`, minus the divider widgets.

`/*4*/` If you've reached the end of the available word pairings, then generate 10 more and add them to the suggestions list.

The `ListView.builder` constructor creates and displays a `Text` widget once per word pairing. In the next step, you'll instead return each new pair as a `ListTile`, which allows you to make the rows more attractive in the next step.

3. Replace the returned `Text` in the `itemBuilder` body of the `ListView.builder` in `_RandomWordsState` with a `ListTile` displaying the suggestion:

lib/main.dart (listTile)

```
return ListTile(
  title: Text(
    _suggestions[index].asPascalCase,
    style: _biggerFont,
  ),
);
```

A `ListTile` is a fixed height row that contains text as well as leading or trailing icons or other widgets.

4. Once complete, the `build()` method in the `_RandomWordsState` class should match the following highlighted code:

lib/main.dart (build)

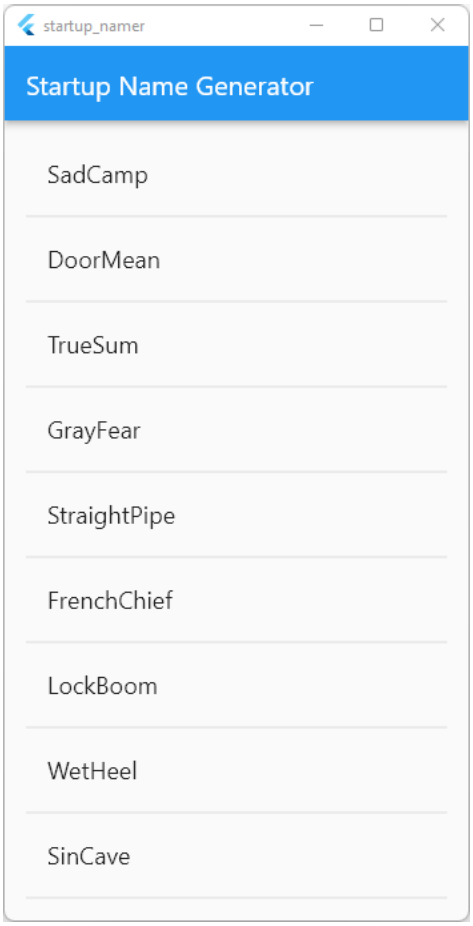
```
@override
Widget build(BuildContext context) {
  return ListView.builder(
    padding: const EdgeInsets.all(16.0),
    itemBuilder: /*1*/ (context, i) {
      if (i.isOdd) return const Divider(); /*2*/

      final index = i ~/ 2; /*3*/
      if (index >= _suggestions.length) {
        _suggestions.addAll(generateWordPairs().take(10)); /*4*/
      }
      return ListTile(
        title: Text(
          _suggestions[index].asPascalCase,
          style: _biggerFont,
        ),
      );
    },
  );
}
```

{step3_stateful_widget → step4_infinite_list}/lib/main.dart

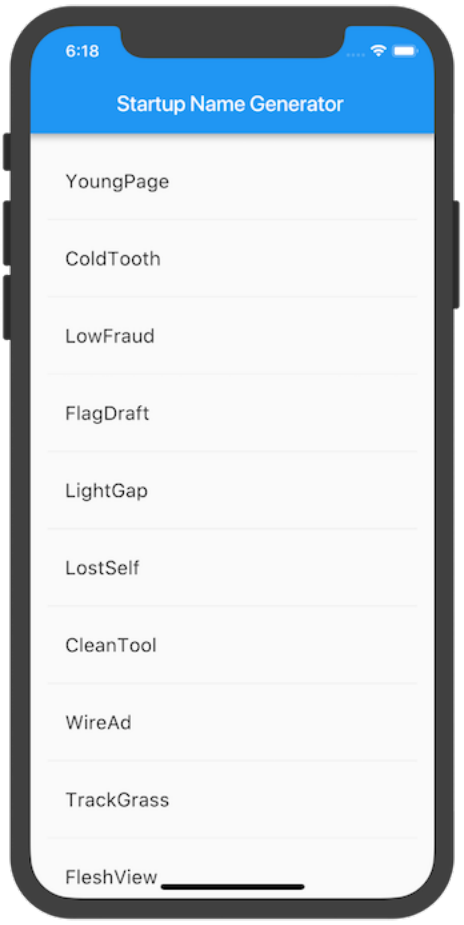
Viewed

```
@@ -14,12 +14,12 @@
    @override
    Widget build(BuildContext context) {
      return MaterialApp(
-       title: 'Welcome to Flutter',
+       title: 'Startup Name Generator',
        home: Scaffold(
          appBar: AppBar(
-           title: const Text('Welcome to Flutter'),
+           title: const Text('Startup Name Generator'),
          ),
          body: const Center(
            child: RandomWords(),
          ),
@@ -28,12 +28,30 @@
    }
  }
  class _RandomWordsState extends State<RandomWords> {
+   final _suggestions = <WordPair>[];
+   final _biggerFont = const TextStyle(fontSize: 18);
+
    @override
    Widget build(BuildContext context) {
-     final wordPair = WordPair.random();
-     return Text(wordPair.asPascalCase);
+     return ListView.builder(
+       padding: const EdgeInsets.all(16.0),
+       itemBuilder: /*1*/ (context, i) {
+         if (i.isOdd) return const Divider(); /*2*/
+
+         final index = i ~/ 2; /*3*/
+         if (index >= _suggestions.length) {
+           _suggestions.addAll(generateWordPairs().take(10)); /*4*/
+         }
+         return ListTile(
+           title: Text(
+             _suggestions[index].asPascalCase,
+             style: _biggerFont,
+           ),
+         );
+       },
+     );
    }
  }
  class RandomWords extends StatefulWidget {
```

6. Restart the app. You should see a list of word pairings no matter how far you scroll.



Windows



iOS

Problems?

If your app is not running correctly, look for typos. If you want to try some of Flutter's debugging tools, check out the DevTools suite of debugging and profiling tools. If needed, use the code at the following link to get back on track.

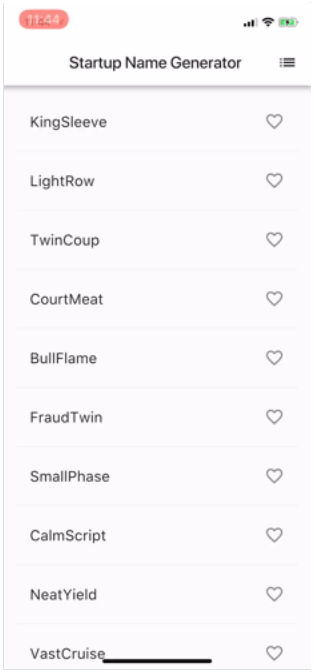
- lib/main.dart

Profile or release runs

> **Important:** Do *not* test the performance of your app with debug and hot reload enabled.

So far you've been running your app in *debug* mode. Debug mode trades performance for useful developer features such as hot reload and step debugging. It's not unexpected to see slow performance and janky animations in debug mode. Once you are ready to analyze performance or release your app, you'll want to use Flutter's "profile" or "release" build modes. For more details, see Flutter's build modes.

> **Important:** If you're concerned about the package size of your app, see Measuring your app's size.



- Written Dart code.
- Leveraged an external, third-party library.
- Used hot reload for a faster development cycle.
- Implemented a stateful widget.
- Created a lazily loaded, infinite scrolling list.

If you would like to extend this app, proceed to part 2 on the Google Developers Codelabs site, where you add the following functionality:

- Implement interactivity by adding a clickable heart icon to save favorite pairings.
- Implement navigation to a new route by adding a new screen containing the saved favorites.
- Modify the theme color, making an all-white app.

‹ Test drive Learn more ›