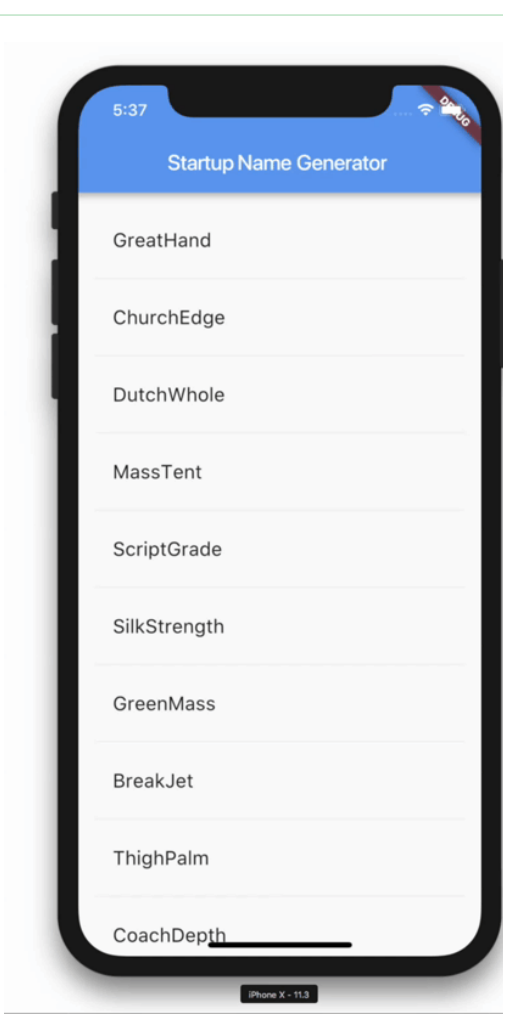
Write your first Flutter app, part 1

Contents

Tip: This codelab walks you through writing your first Flutter app on mobile. You might prefer to try writing your first Flutter app on the web. **Note that if you have enabled web, the completed app just works on all of these devices!**

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, or web programming.

This guide is part 1 of a two-part codelab. You can find part 2 on the Google Developers Codelabs, as well as part 1.



What you'll build in part 1

You'll implement a simple mobile app that generates proposed names for a startup company. The user can select and unselect names, saving the best ones. The code lazily generates names. As the user scrolls, more names are generated. There is no limit t 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, 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 (callec 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 using any of the following devices:

- A physical device (Android or iOS) connected to your computer and set to developer mode
- The iOS simulator
- The Android emulator
- A browser (Chrome is required for debugging)

Step 1: Create the starter Flutter app

Create a simple, templated Flutter app, using the instructions in Getting Started with your first Flutter app. Name the project **startup_namer** (instead of *myapp*).

> **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.

In this codelab, you'll mostly be editing **lib/main.dart**, where the Dart code lives.

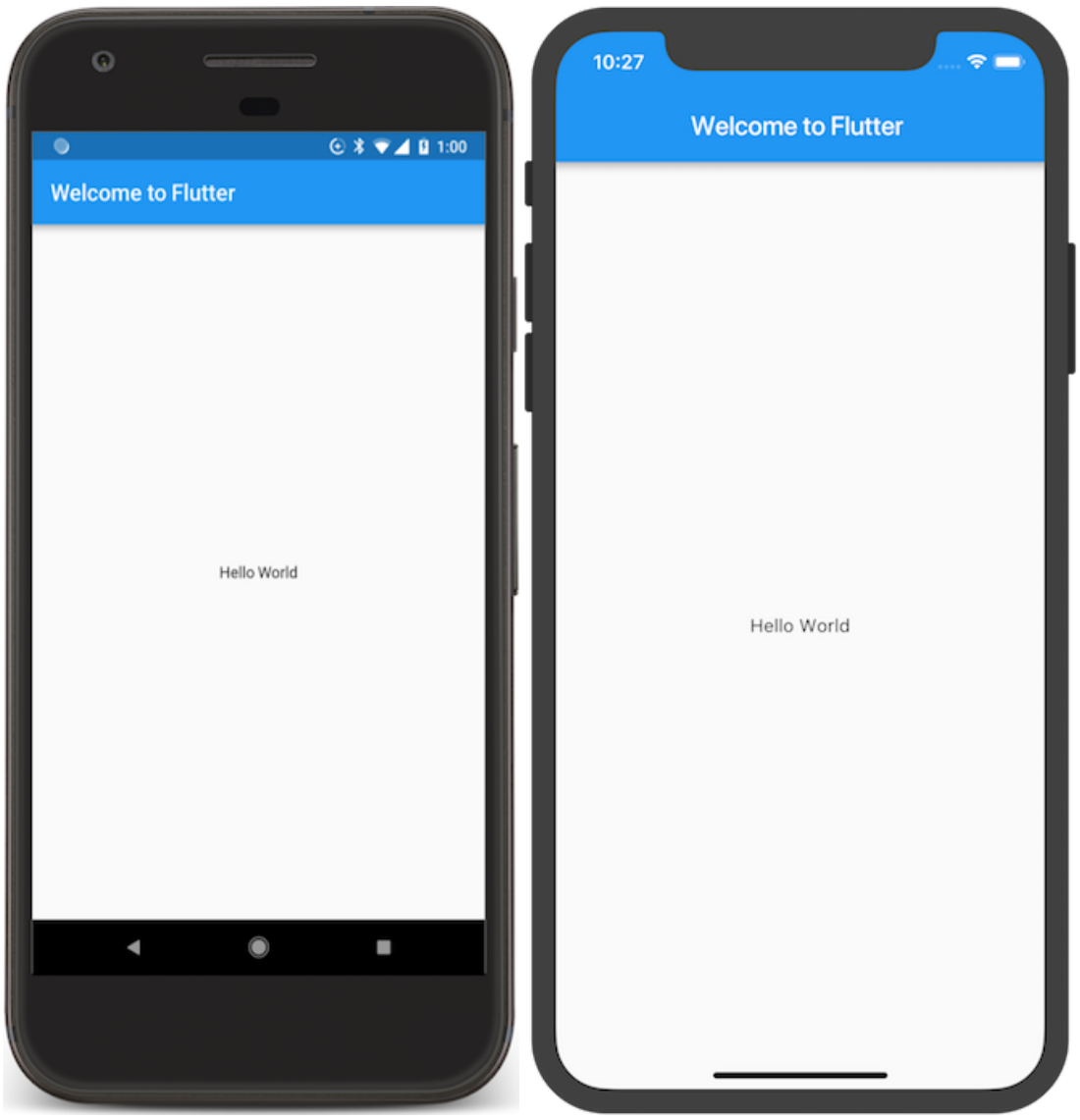
1. Replace the contents of `lib/main.dart`.
   Delete all of the code from **lib/main.dart**. Replace with the following code, which displays "Hello World" in the center of the screen.

lib/main.dart

```
// 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(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'),
        ),
      ),
    );
  }
}
```

> **Tip:** When pasting code into your app, indentation can become skewed. You can fix this automatically with the Flutter 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, or web output, depending on your device.



Android iOS

Tip: The first time you run on a physical device, it can take awhile to load. After this, you can use hot reload for quick updates. **Save** also performs a hot reload if the app is running.

Observations

- This example creates a Material app. Material is a visual design language that is standard on mobile and the web. Flutter of 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. In Flutter, 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 tr 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 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.

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. The `pubspec.yaml` file manages the assets and dependencies for a Flutter app. In `pubspec.yaml`, add `english_words` (3.1.5 higher) to the dependencies list:

{step1_base → step2_use_package}/pubspec.yaml

```
@@ -5,4 +5,5 @@
 dependencies:
   flutter:
     sdk: flutter
   cupertino_icons: ^0.1.2
+  english_words: ^3.1.0
```

2. While viewing the `pubspec.yaml` file in Android Studio's editor view, click **Packages get**. This pulls the package into your pro 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 `Packages get` also auto-generates the `pubspec.lock` file with a list of all packages pulled into the project and tl version numbers.

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

lib/main.dart

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

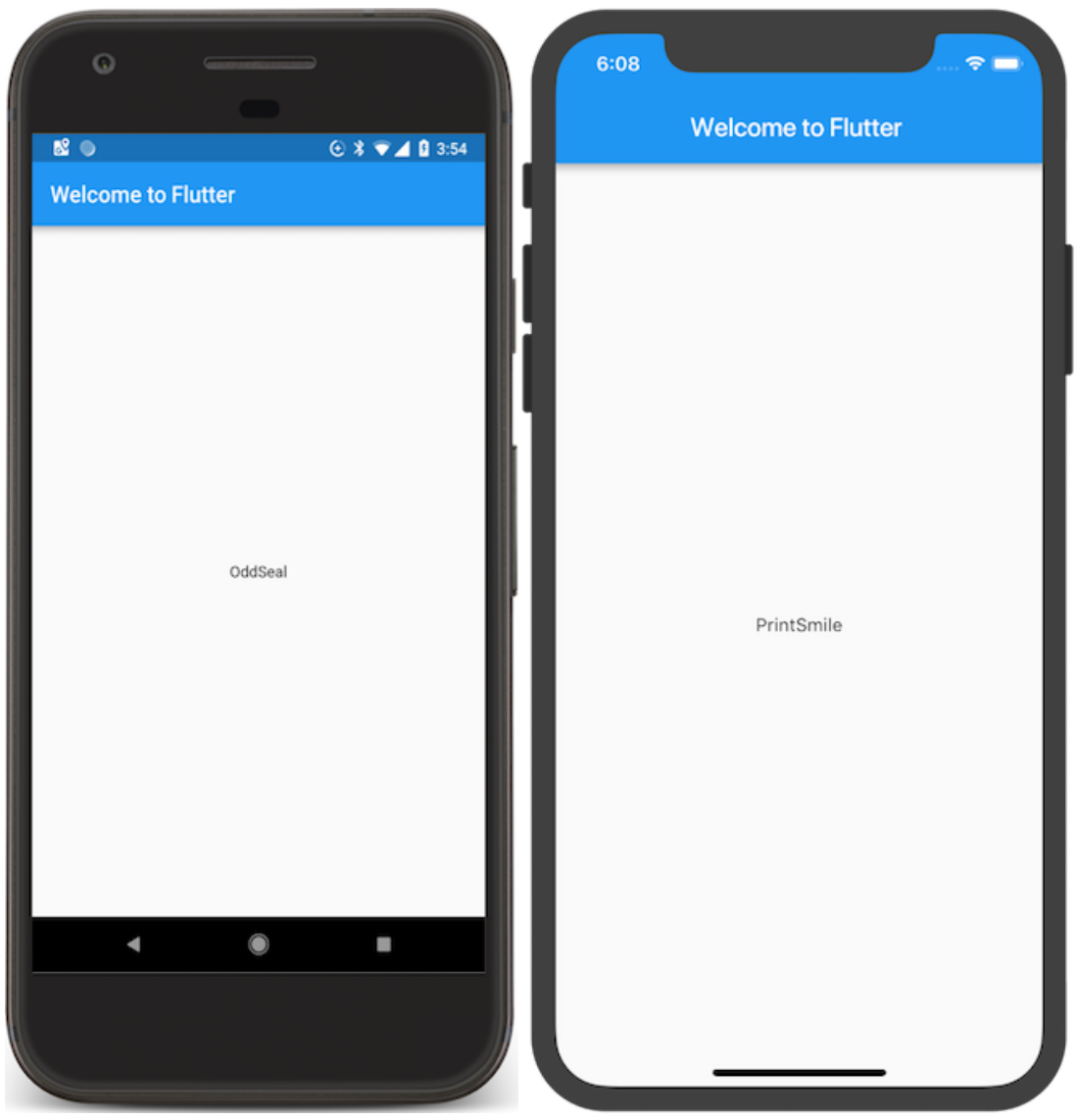
4. Use the English words package to generate the text instead of using the string “Hello World”:

{step1_base → step2_use_package}/lib/main.dart

```
@@ -9,6 +10,7 @@
 class MyApp extends StatelessWidget {
   @override
   Widget build(BuildContext context) {
+    final wordPair = WordPair.random();
     return MaterialApp(
       title: 'Welcome to Flutter',
       home: Scaffold(
@@ -16,7 +18,7 @@
           title: Text('Welcome to Flutter'),
         ),
         body: Center(
-          child: Text('Hello World'),
+          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 se different word pair, chosen at random, in the running app. This is because the word pairing is generated inside the build met which is run each time the `MaterialApp` requires rendering, or when toggling the Platform in Flutter Inspector.



Android 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 sui of debugging and profiling tools. If needed, use the code at the following links to get back on track.

- 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 le two classes: 1) a `StatefulWidget` class that creates an instance of 2) a `State` class. The `StatefulWidget` class is, itself, immuta 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 `RandomWo` as a child inside the existing `MyApp` stateless widget.

1. Create a minimal state class. Add the following to the bottom of `main.dart`:

lib/main.dart (RandomWordsState)

```
class RandomWordsState extends State<RandomWords> {
  // TODO Add build() method
}
```

Notice the declaration `State<RandomWords>`. This indicates that we're using the generic `State` class specialized for use with `RandomWords`. Most of the app's logic and state resides here—it maintains the state for the `RandomWords` widget. This class saves the generated word pairs, which grows infinitely as the user scrolls, and favorite word pairs (in part 2), as the user ad removes them from the list by toggling the heart icon.

`RandomWordsState` depends on the `RandomWords` class. You'll add that next.

2. Add the stateful `RandomWords` widget to `main.dart`. The `RandomWords` widget does little else beside creating its `State` class

lib/main.dart (RandomWords)

```
class RandomWords extends StatefulWidget {
  @override
  RandomWordsState createState() => RandomWordsState();
}
```

After adding the state class, the IDE complains that the class is missing a build method. Next, you'll add a basic build meth that generates the word pairs by moving the word generation code from `MyApp` to `RandomWordsState`.

3. Add the `build()` method to `RandomWordsState`:

lib/main.dart (RandomWordsState)

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

4. Remove the word generation code from `MyApp` by making the changes shown in the following diff:

{step2_use_package → step3_stateful_widget}/lib/main.dart

```
@@ -10,7 +10,6 @@
 class MyApp extends StatelessWidget {
   @override
   Widget build(BuildContext context) {
-    final wordPair = WordPair.random();
     return MaterialApp(
       title: 'Welcome to Flutter',
       home: Scaffold(
@@ -18,8 +17,8 @@
           title: Text('Welcome to Flutter'),
         ),
         body: Center(
-          child: Text(wordPair.asPascalCase),
+          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 the following warning on a hot reload, consider restarting the app:

Reloading…
Some program elements were changed during reload but did not run when the view was reassembled; you might need to restart the app (by pressing "R") for the changes to have an effect.

It might be a false positive, but restarting 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 sui 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 `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` varia for making the font size larger.

lib/main.dart

```
class RandomWordsState extends State<RandomWords> {
  final _suggestions = <WordPair>[];
  final _biggerFont = const TextStyle(fontSize: 18.0);
  // ...
}
```

Note: Prefixing an identifier with an underscore enforces privacy in the Dart language.

Next, you'll add a `_buildSuggestions()` function to the `RandomWordsState` class. This method builds 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 be at 0 and increments each time the function is called. It increments twice for every suggested word pairing: once for the List and once for the Divider. This model allows the suggested list to grow infinitely as the user scrolls.

2. Add a `_buildSuggestions()` function to the `RandomWordsState` class:

lib/main.dart (_buildSuggestions)

```
Widget _buildSuggestions() {
  return ListView.builder(
      padding: const EdgeInsets.all(16.0),
      itemBuilder: /*1*/ (context, i) {
        if (i.isOdd) return Divider(); /*2*/
        final index = i ~/ 2; /*3*/
        if (index >= _suggestions.length) {
          _suggestions.addAll(generateWordPairs().take(10)); /*4*/
        }
        return _buildRow(_suggestions[index]);
      });
}
```

`/*1*/` The `itemBuilder` callback is called once per suggested word pairing, and places each suggestion into a `ListTile` r 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. Th 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 lis

The `_buildSuggestions()` function calls `_buildRow()` once per word pair. This function displays each new pair in a `ListTi` which allows you to make the rows more attractive in the next step.

3. Add a `_buildRow()` function to `RandomWordsState`:

lib/main.dart (_buildRow)

```
Widget _buildRow(WordPair pair) {
  return ListTile(
    title: Text(
      pair.asPascalCase,
      style: _biggerFont,
    ),
  );
}
```

4. In the `RandomWordsState` class, update the `build()` method to use `_buildSuggestions()`, rather than directly calling the w generation library. (`Scaffold` implements the basic Material Design visual layout.) Replace the method body with the highlighted code:

lib/main.dart (build)

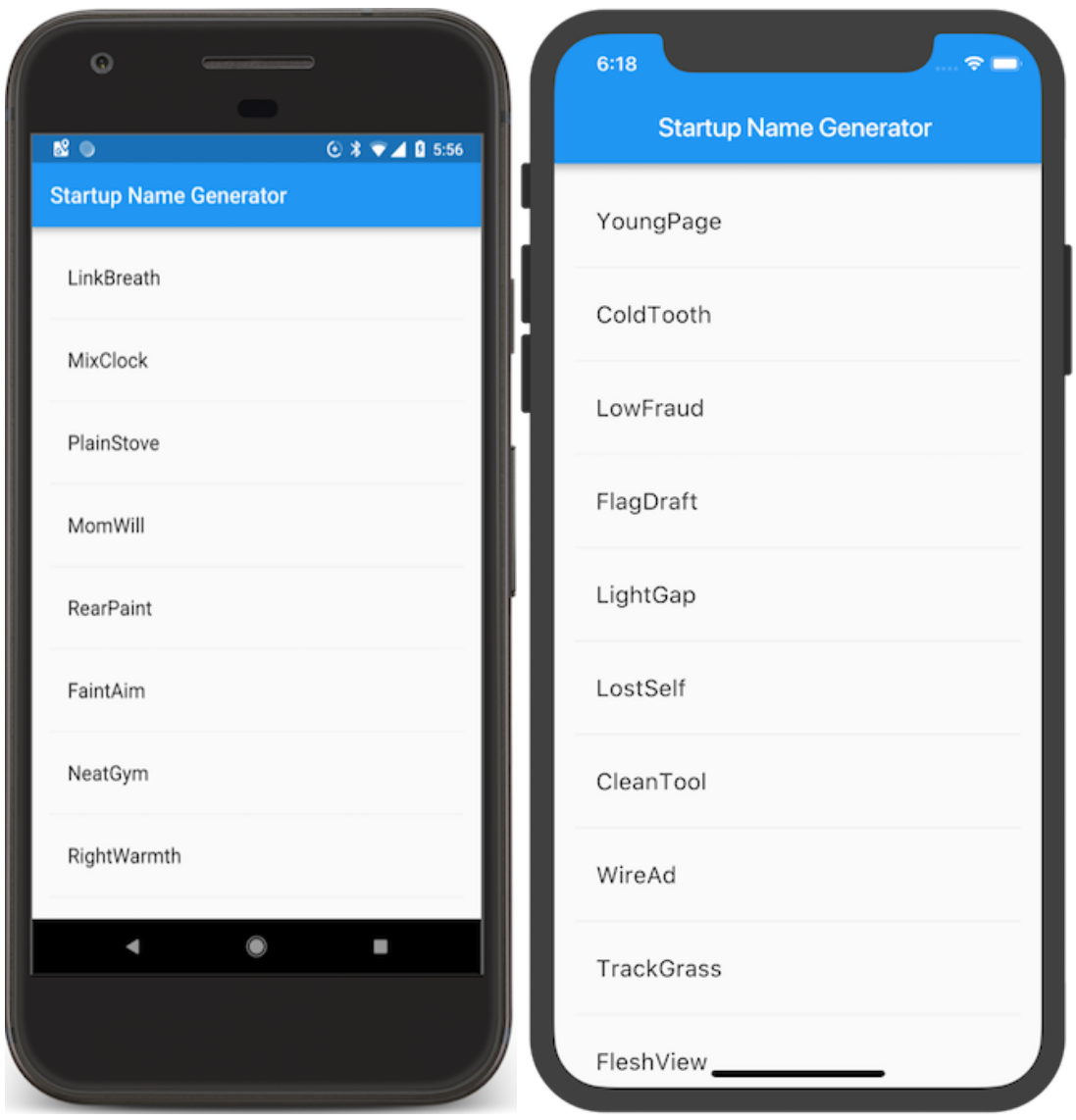
```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Startup Name Generator'),
    ),
    body: _buildSuggestions(),
  );
}
```

5. In the `MyApp` class, update the `build()` method by changing the title, and changing the home to be a `RandomWords` widget:

{step3_stateful_widget → step4_infinite_list}/lib/main.dart

```
@@ -10,15 +10,8 @@
 class MyApp extends StatelessWidget {
   @override
   Widget build(BuildContext context) {
     return MaterialApp(
-      title: 'Welcome to Flutter',
-      home: Scaffold(
+      title: 'Startup Name Generator',
+      home: RandomWords(),
-        appBar: AppBar(
-          title: Text('Welcome to Flutter'),
-        ),
-        body: Center(
-          child: RandomWords(),
-        ),
-      ),
     );
   }
```

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



Android 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 sui 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 read analyze performance or release your app, you'll want to use Flutter's "profile" or "release" build modes. For more details, see Flutt build modes.

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

Next steps

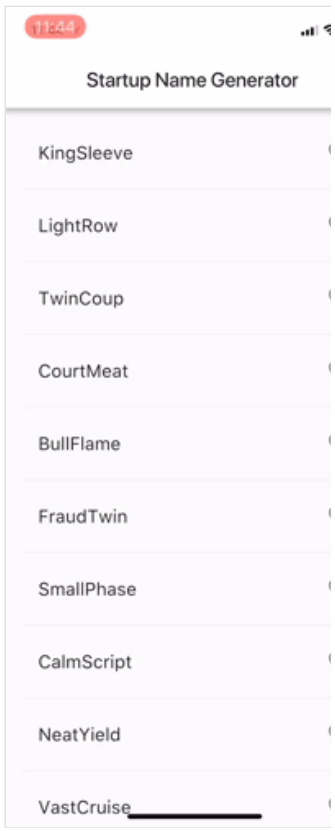
Congratulations!

You've written an interactive Flutter app that runs on both iOS and Android. In this codelab, you've:

- Created a Flutter app from the ground up.
- 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.



The app from part 2

〈 Test drive
Learn more