

# Write Your First Flutter App

## part 2

### 1. Introduction

Flutter is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase. Flutter works with existing code, is used by developers and organizations around the world, and is free and open source.

In this codelab, you'll extend a basic Flutter mobile app to include interactivity. You'll also create a second page (called a *route*) that the user can navigate to. Finally, you'll modify the app's theme (color). This codelab extends part 1, [Create a Lazily Loaded List](#), but we'll provide the starting code, if you'd like to start with part 2.

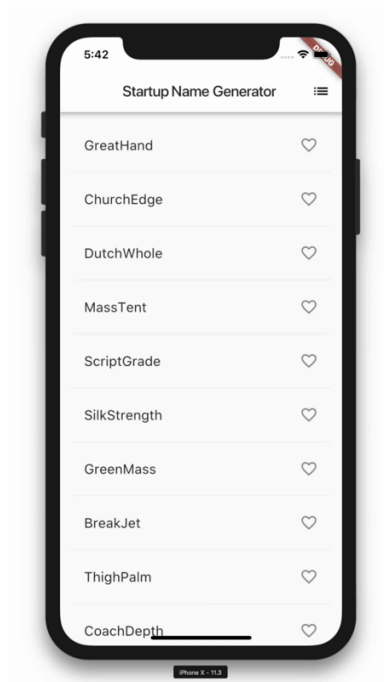
### What you'll learn in part 2

- How to write a Flutter app that looks natural on both iOS and Android.
- Using hot reload for a quicker development cycle.
- How to add interactivity to a stateful widget.
- How to create and navigate to a second screen.
- How to change the look of an app using themes.

### What you'll build in part 2

You'll start with a simple mobile app that generates an endless list of proposed names for a startup company. By the end of this codelab, the user can select and unselect names, saving the best ones. Tapping the list icon in the upper right of the app bar navigates to a new page (called a *route*) that lists only the favorited names.

The animated GIF shows how the finished app works.



## 2. Set up your Flutter environment

If you haven't completed Part 1, see [Set up Your Flutter Environment](#), in [Write Your First Flutter App, part 1](#), to set up your environment for Flutter development.

## 3. Get the Starting App

If you have worked through part 1 of this codelab, you already have the starting app, **startup\_namer**. You can proceed to the next step.

If you don't have **startup\_namer**, no fear, you can get it using the following instructions.



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*).



Delete all of the code from **lib/main.dart**. Replace with the code from [this file](#), which displays an infinite, lazily loaded list of proposed startup names.



Update **pubspec.yaml** by adding the English Words package, as shown below:

```
dependencies:  
  flutter:
```

```
sdk: flutter
```

```
cupertino_icons: ^0.1.2  
english_words: ^3.1.0 // Add this line.
```

The English Words package generates pairs of random words, which are used as potential startup names.

While viewing the pubspec in Android Studio's editor view, click **Packages get** upper right. This pulls the package into your project. You should see the following in the console:

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

Run the app. Scroll as far as you want, viewing a continual supply of proposed startup names.

#### 4. Add icons to the list

In this step, you'll add heart icons to each row. In the next step, you'll make them tappable and save the favorites..

Add a `_saved` Set to `RandomWordsState`. This Set stores the word pairings that the user favorited. Set is preferred to List because a properly implemented Set does not allow duplicate entries.

```
class RandomWordsState extends State<RandomWords> {  
  final List<WordPair> _suggestions = <WordPair>[];  
  final Set<WordPair> _saved = Set<WordPair>(); // Add this line.  
  final TextStyle _biggerFont = TextStyle(fontSize: 18.0);  
  ...  
}
```

In the `_buildRow` function, add an `alreadySaved` check to ensure that a word pairing has not already been added to favorites.

```
Widget _buildRow(WordPair pair) {  
  final bool alreadySaved = _saved.contains(pair); // Add this line.
```

```
...
}
```

In `_buildRow()` you'll also add heart-shaped icons to the `ListTile` objects to enable favoriting. In the next step, you'll add the ability to interact with the heart icons.



Add the icons, as shown below:

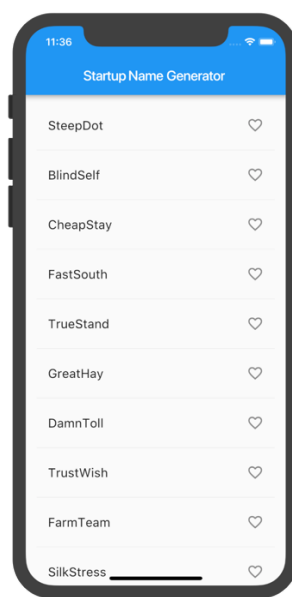
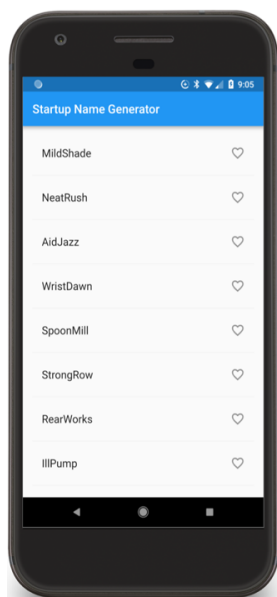
```
Widget _buildRow(WordPair pair) {
  final bool alreadySaved = _saved.contains(pair);
  return ListTile(
    title: Text(
      pair.asPascalCase,
      style: _biggerFont,
    ),
    trailing: Icon( // Add the lines from here...
      alreadySaved ? Icons.favorite : Icons.favorite_border,
      color: alreadySaved ? Colors.red : null,
    ), // ... to here.
  );
}
```



Hot reload the app. You should now see open hearts on each row, but they are not yet interactive.

Android

iOS



## Problems?

If your app is not running correctly, you can use the code at the following link to get back on track.

- [lib/main.dart](#)

## 5. Add interactivity

In this step, you'll make the heart icons tappable. When the user taps an entry in the list, toggling its "favorited" state, that word pairing is added or removed from a set of saved favorites.

To do this, you'll modify the `_buildRow` function. If a word entry has already been added to favorites, tapping it again removes it from favorites. When a tile has been tapped, the function calls `setState()` to notify the framework that state has changed.



Add `onTap`, as shown below:

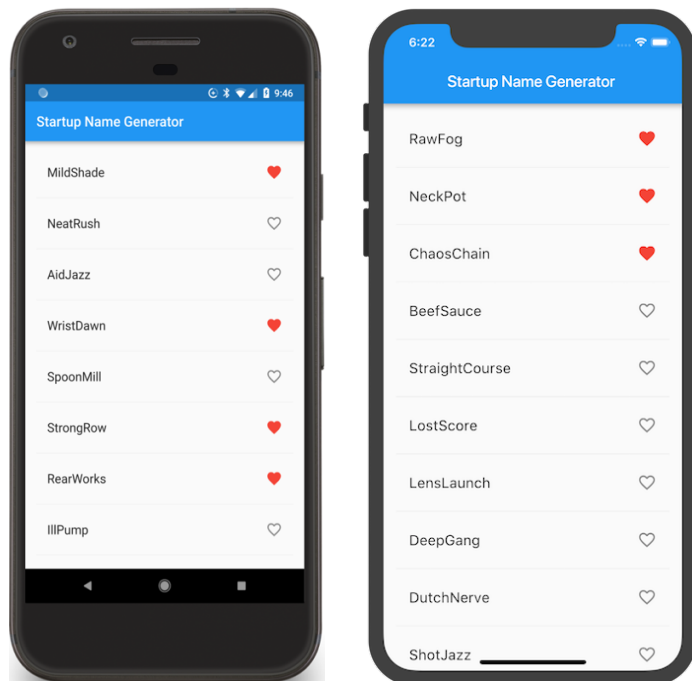
```
Widget _buildRow(WordPair pair) {
  final alreadySaved = _saved.contains(pair);
  return ListTile(
    title: Text(
      pair.asPascalCase,
      style: _biggerFont,
    ),
    trailing: Icon(
      alreadySaved ? Icons.favorite : Icons.favorite_border,
      color: alreadySaved ? Colors.red : null,
    ),
    onTap: () {          // Add 9 lines from here...
      setState(() {
        if (alreadySaved) {
          _saved.remove(pair);
        } else {
          _saved.add(pair);
        }
      });
    },                    // ... to here.
  );
}
```

**Tip:** In Flutter's reactive style framework, calling `setState()` triggers a call to the `build()` method for the State object, resulting in an update to the UI.

Hot reload the app. You should be able to tap any tile to favorite, or unfavorite, the entry. Tapping a tile generates an implicit ink splash animation emanating from the tap point.

Android

iOS



## Problems?

If your app is not running correctly, you can use the code at the following link to get back on track.

- [lib/main.dart](#)

## 6. Navigate to a new screen

In this step, you'll add a new page (called a *route* in Flutter) that displays the favorites. You'll learn how to navigate between the home route and the new route.

In Flutter, the Navigator manages a stack containing the app's routes. Pushing a route onto the Navigator's stack, updates the display to that route. Popping a route from the Navigator's stack, returns the display to the previous route.

Next, you'll add a list icon to the AppBar in the build method for RandomWordsState. When the user clicks the list icon, a new route that contains the saved favorites is pushed to the Navigator, displaying the icon.



Add the icon and its corresponding action to the `build` method:

```
class RandomWordsState extends State<RandomWords> {  
  ...  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('Startup Name Generator'),  
        actions: <Widget>[           // Add 3 lines from here...  
          IconButton(icon: Icon(Icons.list), onPressed: _pushSaved),  
          // ... to here.  
        ],  
      ),  
      body: _buildSuggestions(),  
    );  
  }  
  ...  
}
```


Tip: Some widget properties take a single widget (`child`), and other properties, such as `action`, take an array of widgets (`children`), as indicated by the square brackets (`[]`).



Add a `_pushSaved()` function to the RandomWordsState class.

```
void _pushSaved() {  
}
```



Hot reload the app. The list icon () appears in the app bar. Tapping it does nothing yet, because the `_pushSaved` function is empty.

Next, you'll build a route and push it to the Navigator's stack. This action changes the screen to display the new route. The content for the new page is built in MaterialPageRoute's `builder` property, in an anonymous function.



Call `Navigator.push`, as shown below, which pushes the route to the Navigator's stack. The IDE will complain about invalid code, but you will fix that in the next section.

```
void _pushSaved() {
  Navigator.of(context).push(
  );
}
```

Next, you'll add the `MaterialPageRoute` and its builder. For now, add the code that generates the `ListTile` rows. The `divideTiles()` method of `ListTile` adds horizontal spacing between each `ListTile`. The `divided` variable holds the final rows, converted to a list by the convenience function, `toList()`.



Add the code, as shown below:

```
void _pushSaved() {
  Navigator.of(context).push(
    MaterialPageRoute<void>( // Add 20 lines from here...
      builder: (BuildContext context) {
        final Iterable<ListTile> tiles = _saved.map(
          (WordPair pair) {
            return ListTile(
              title: Text(
                pair.asPascalCase,
                style: _biggerFont,
              ),
            );
          },
        );
        final List<Widget> divided = ListTile
          .divideTiles(
            context: context,
            tiles: tiles,
          )
          .toList();
      },
    ),
  );
}
```

The builder property returns a `Scaffold`, containing the app bar for the new route, named "Saved Suggestions." The body of the new route consists of a `ListView` containing the `ListTiles` rows; each row is separated by a divider.



Add horizontal dividers, as shown below:



```

void _pushSaved() {
  Navigator.of(context).push(
    MaterialPageRoute<void>(
      builder: (BuildContext context) {
        final Iterable<ListTile> tiles = _saved.map(
          (WordPair pair) {
            return ListTile(
              title: Text(
                pair.asPascalCase,
                style: _biggerFont,
              ),
            );
          },
        );
        final List<Widget> divided = ListTile
          .divideTiles(
            context: context,
            tiles: tiles,
          )
          .toList();

        return Scaffold(           // Add 6 lines from here...
          appBar: AppBar(
            title: Text('Saved Suggestions'),
          ),
          body: ListView(children: divided),
        );                          // ... to here.
      },
    ),
  );
}

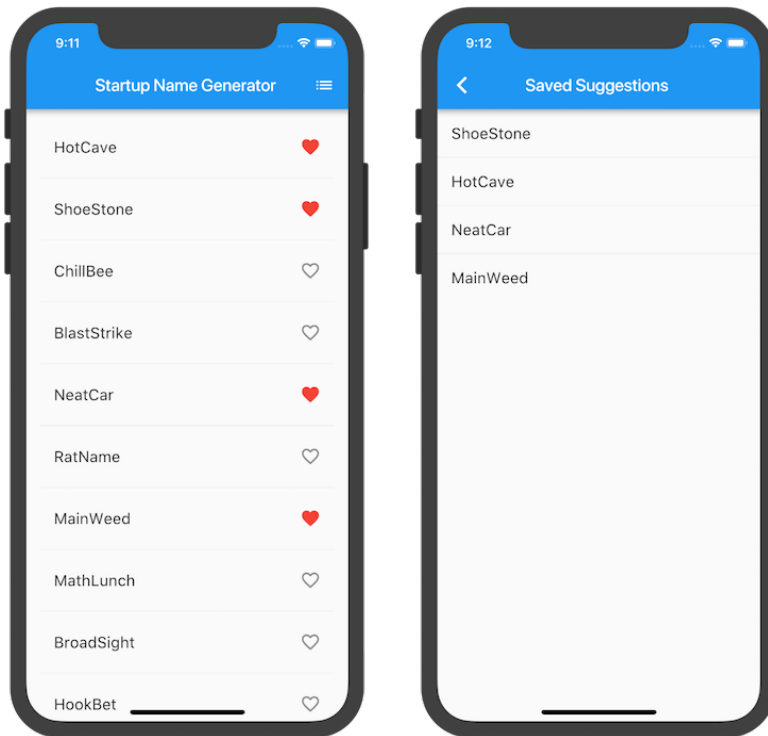
```



Hot reload the app. Favorite some of the selections and tap the list icon in the app bar. The new route appears containing the favorites. Note that the Navigator adds a "Back" button to the app bar. You did not have to explicitly implement Navigator.pop. Tap the back button to return to the home route.

iOS - Main route

iOS - Saved suggestions route



## Problems?

If your app is not running correctly, you can use the code at the following link to get back on track.

- [lib/main.dart](#)

## 7. Change the UI using Themes

In this step, you'll modify the app's theme. The *theme* controls the look and feel of your app. You can either use the default theme, which is dependent on the physical device or emulator, or customize the theme to reflect your branding.

You can easily change an app's theme by configuring the [ThemeData](#) class. This app currently uses the default theme, but you'll change the app's primary color to white.



Change the color in the MyApp class:

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Startup Name Generator',  
    );  
  }  
}
```

```

theme: ThemeData( // Add the 3 lines from here...
  primaryColor: Colors.white,
), // ... to here.
home: RandomWords(),
);
}
}

```

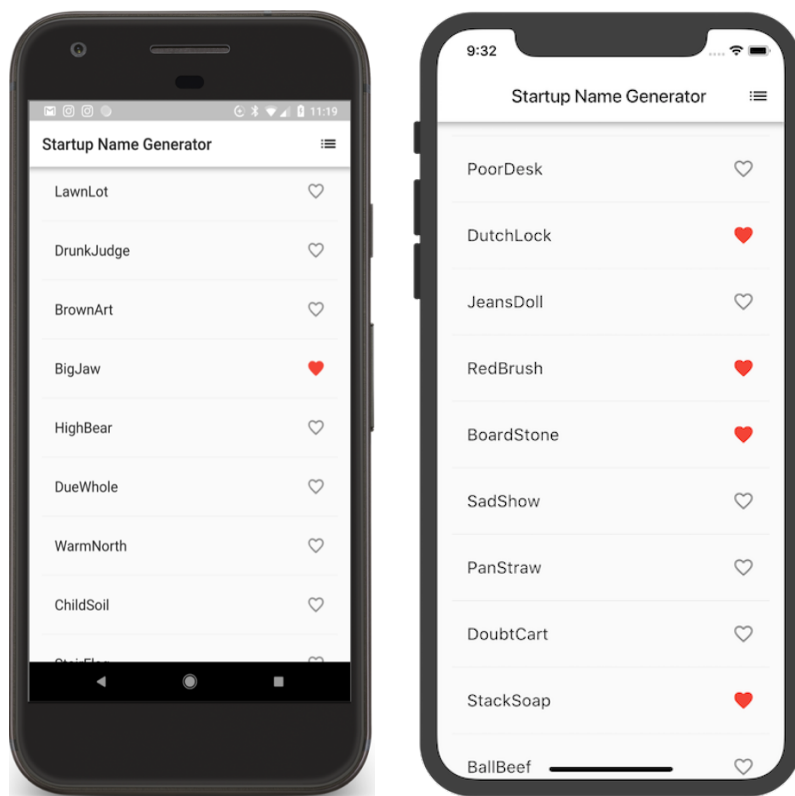


Hot reload the app. The entire background is now white, even the app bar.

As an exercise for the reader, use ThemeData to change other aspects of the UI. The [Colors](#) class in the Material library provides many color constants you can play with, and hot reload makes experimenting with the UI quick and easy.

Android

iOS



## Problems?

If you've gotten off track, use the code from the following link to see the code for the final app.

- [lib/main.dart](#)

## 8. Well done!

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

- Written Dart code.
- Used hot reload for a faster development cycle.
- Implemented a stateful widget, adding interactivity to your app.
- Created a route and added logic for moving between the home route and the new route.
- Learned about changing the look of your app's UI using themes.

## 9. Next steps

Learn more about the Flutter SDK:

- [Building layouts in Flutter](#) tutorial
- [Add interactivity](#) tutorial
- [Introduction to widgets](#)
- [Flutter for Android developers](#)
- [Flutter for React Native developers](#)
- [Flutter for web developers](#)
- [Flutter's Youtube channel](#)

Other resources include:

- [Build Native Mobile Apps with Flutter](#) (a free Udacity course)
- [Flutter cookbook](#)
- [From Java to Dart](#) codelab
- [Bootstrap into Dart: learn more about the language](#)

Please reach out to us at our [mailing list](#). We'd love to hear from you!