

ステップ1：ハートのアイコンを追加する

このステップでは、各行にハートのアイコンを追加します。次のステップでは、それらをタップ可能にして、お気に入りを保存します。

1. `_saved`セットを`_RandomWordsState`に追加します。このセットには、ユーザーがお気に入りにした単語ペアを格納します。適切に実装されたセットは重複エントリを許可しないため、セットはListよりも優先されます。

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

2. `_buildRow`関数で、単語ペアがまだお気に入りに追加されていないことを確認するために、`alreadySaved`チェックを追加します。

```
Widget _buildRow(WordPair pair) {  
  final alreadySaved = _saved.contains(pair); // NEW  
  ...  
}
```

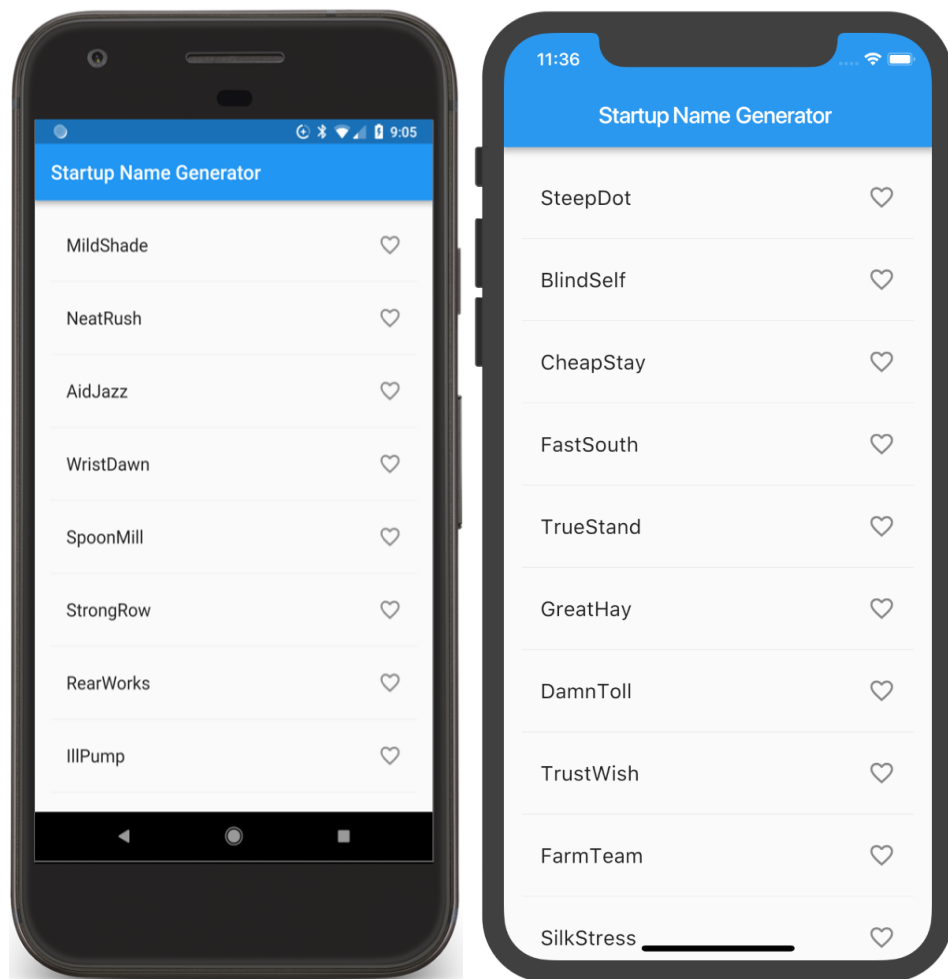
`_buildRow()`では、ハート型のアイコンをリストアイテムオブジェクトに追加して、お気に入りにします。次のステップでは、ハートのアイコンを操作する機能を追加します。

3. 以下に示すように、テキストの後にアイコンを追加します。

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

4. アプリをホットリロードします。

これで、各行に色のないハートが表示されるはずですが、まだタッチしても反応はありません。



問題？

アプリが正しく実行されていない場合は、次のリンクのコードを使用して、このチュートリアルに戻ることができます。

- [lib / main.dart](#)

ステップ2：ハートのアイコンとタッチできるようにする

このステップでは、ハートのアイコンをタップ可能にします。ユーザーがリスト内のアイコンをタップしてお気に入りの状態を切り替えると、その単語ペアが保存されたお気に入りのセットに追加または削除されます。

これを行うには、`_buildRow`関数を変更します。単語ペアがすでにお気に入りの追加されている場合は、もう一度タップするとお気に入りから削除されます。アイテムがタップされると、関数は`setState()`を呼び出して、お気に入りの状態が変更されたことをフレームワークに通知します。

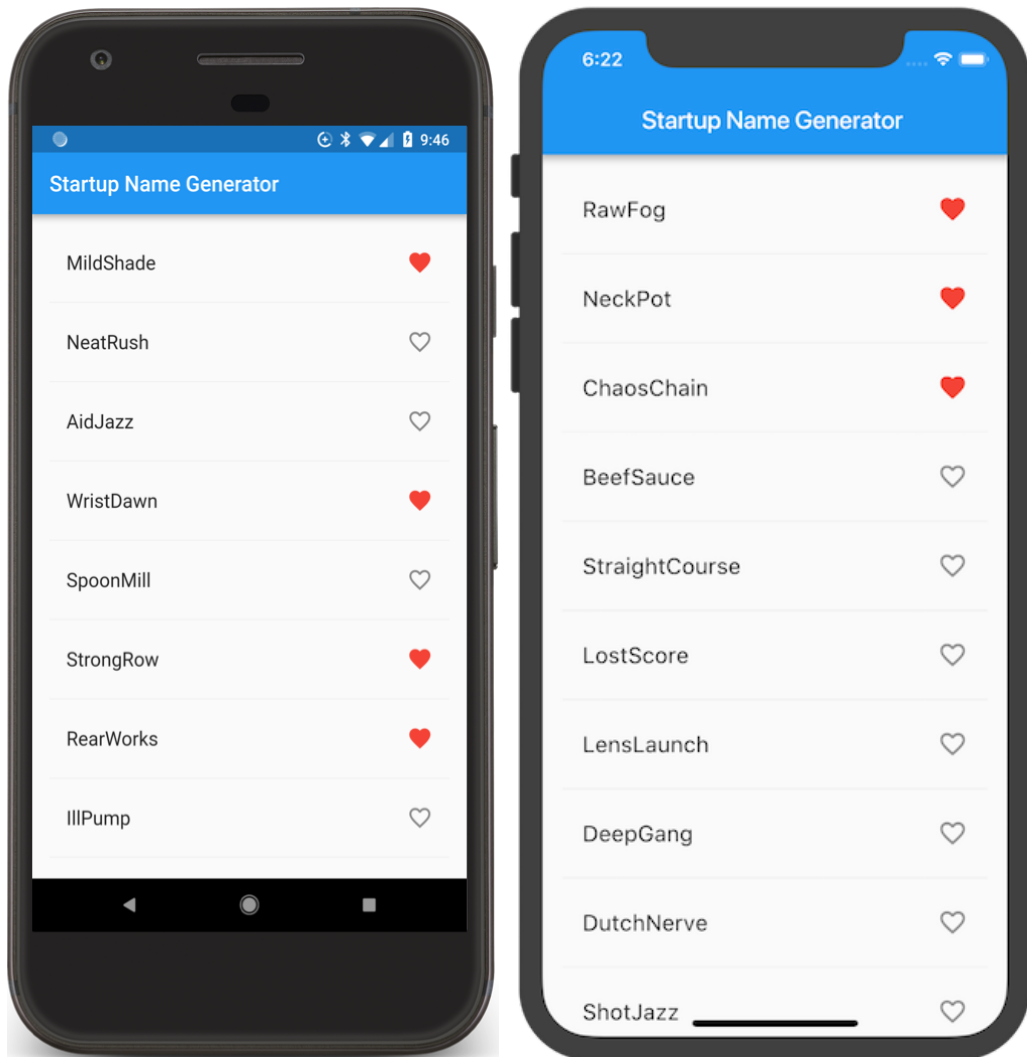
1. 以下に示すように、`onTap`を`_buildRow`メソッドに追加します。

```
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: () {          // NEW lines from here...
      setState(() {
        if (alreadySaved) {
          _saved.remove(pair);
        } else {
          _saved.add(pair);
        }
      });
    },                    // ... to here.
  );
}
```

ヒント：Flutterのリアクティブスタイルフレームワークでは、`setState()`を呼び出すとStateオブジェクトの`build()`メソッドが呼び出され、UIが更新されます。

2. アプリをホットリロードします。

アイテムをタップして、単語ペアをお気に入りにしたり、お気に入りから外すことができるはずです。タイルをタップすると、タップポイントから発せられる暗黙のインクスブラッシュアニメーションが生成されます。



問題？

アプリが正しく実行されていない場合は、次のリンクのコードを使用して、このチュートリアルに戻ることができます。

- [lib / main.dart](#)

ステップ3：お気に入りを表示するページを追加する

このステップでは、お気に入りを表示する新しい画面（Flutterではルートと呼ぶ）を追加します。ホーム画面と新しい画面の間を移動する方法を学習します。

Flutterでは、ナビゲーターはアプリのルートを含むスタックを管理します。ルートをナビゲーターのスタックにプッシュすると、その画面の表示が更新されます。ナビゲーターのスタックからルートをポップすると、表示が前の画面に戻ります。

次に、`_RandomWordsState`の`build`メソッドの`AppBar`にリストアイコンを追加します。ユーザーがリストアイコンをクリックすると、保存されたお気に入りを含む新しいルートがナビゲーターにプッシュされ、アイコンが表示されます。


1. アイコンとそれに対応するアクションを`build`メソッドに追加します。

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

ヒント：一部のウィジェットプロパティは単一のウィジェット（子）を取り、`action`などの他のプロパティは、角括弧（`[]`）で示されているように、ウィジェットの配列（子）を取ります。

2. `_pushSaved()`関数を`_RandomWordsState`クラスに追加します。

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

3. アプリをホットリロードします。リストアイコン  がアプリバーに表示されます。`_pushSaved`関数が空であるため、タップしてもまだ何も起こりません。

次に、ルートを作成し、ナビゲーターのスタックにプッシュします。このアクションにより、画面が変更され、新しいルートが表示されます。新しいページのコンテンツは、無名関数`MaterialPageRoute`の`builder`プロパティに組み込まれています。

4. 以下に示すように、`Navigator.push`を呼び出します。これにより、ルートがナビゲーターのスタックにプッシュされます。IDEは無効なコードについて文句を言いますが、次のセクションで修正します。

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

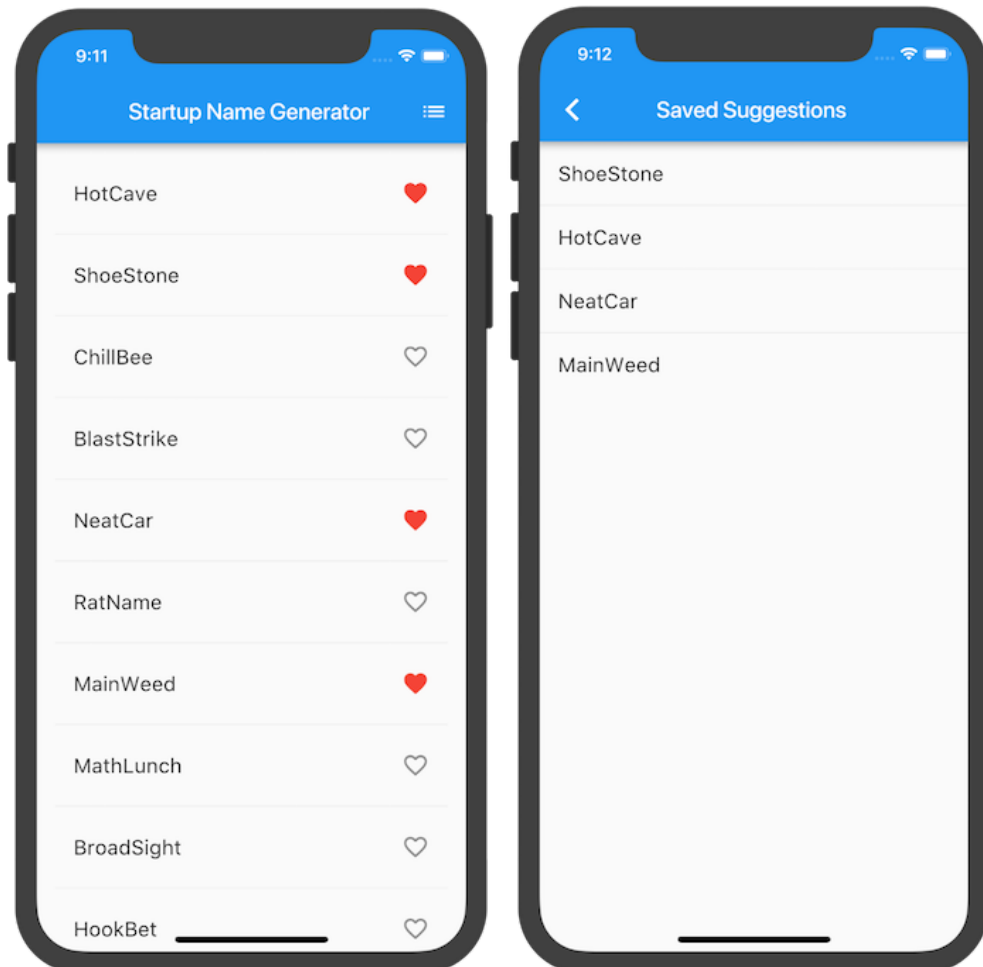
次に、`MaterialPageRoute`とそのビルダーを追加します。とりあえず、リストアイテムを生成するコードを追加します。`ListTile`の`divideTiles()`メソッドは、各リストアイテム間の水平方向の間隔を追加します。変数`divided`は、便利な関数`toList()`によってリストに変換された最終行を保持しています。

5. 次のコードスニペットに示すように、コードを追加します。

```
void _pushSaved() {  
  Navigator.of(context).push(  
    MaterialPageRoute<void>(  
      // NEW lines from here...  
      builder: (BuildContext context) {  
        final tiles = _saved.map(  
          (WordPair pair) {  
            return ListTile(  
              title: Text(  
                pair.asPascalCase,  
                style: _biggerFont,  
              ),  
            );  
          },  
        );  
        final divided = ListTile.divideTiles(  
          context: context,  
          tiles: tiles,  
        ).toList();  
  
        return Scaffold(  
          appBar: AppBar(  
            title: Text('Saved Suggestions'),  
          ),  
          body: ListView(children: divided),  
        );  
      }, // ...to here.  
    ),  
  );  
}
```

`builder`プロパティは、`SavedSuggestions`という新しいルートのアプリバーを含む`Scaffold`を返します。新しいルートの本体は、リストアイテムを含む`ListView`で構成されます。各行は仕切りで区切られています。

6. アプリをホットリロードします。いくつかの単語ペアをお気に入りに追加し、アプリバーのリストアイコンをタップします。お気に入りを含む新しい画面が表示されます。ナビゲーターはアプリバーに「戻る」ボタンを追加することに注意してください。Navigator.popを明示的に実装する必要はありません。戻るボタンをタップして、ホームルートに戻ります。



問題？

アプリが正しく実行されていない場合は、次のリンクのコードを使用して、このチュートリアルに戻ることができます。

- [lib / main.dart](#)

ステップ4：アプリのテーマを変更する

このステップでは、アプリのテーマを変更します。テーマは、アプリケーションの見た目と雰囲気を制御します。物理デバイスまたはエミュレーターに依存するデフォルトのテーマを使用するか、ブランドを反映するようにテーマをカスタマイズできます。

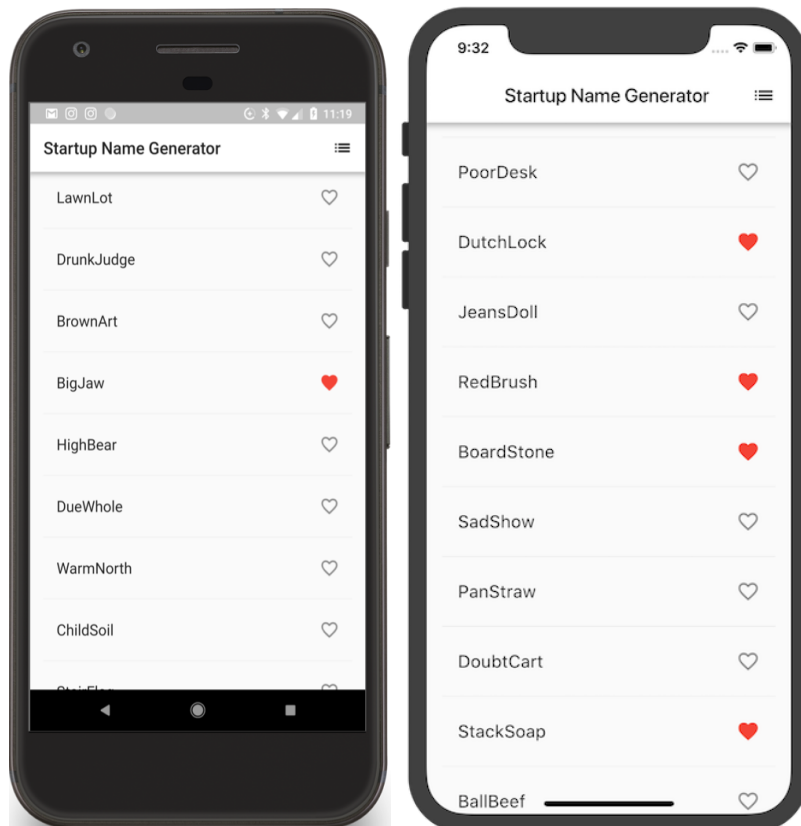
[ThemeData](#)クラスを設定することで、アプリのテーマを簡単に変更できます。アプリはデフォルトのテーマを使用しますが、アプリのテーマカラーを白に変更します。

1. MyAppクラスの色を変更します。

```
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(),  
    );  
  }  
}
```

2. アプリをホットリロードします。アプリバーも含め、背景全体が白になりました。

練習として、ThemeDataを使用してUIの他の側面を変更します。Materialライブラリの[Colors](#)クラスは、さまざまな色定数を提供しています。ホットリロードにより、UIの実験がすばやく簡単になります。



問題？

アプリが正しく実行されていない場合は、次のリンクのコードを使用して、最終的なアプリのコードを確認してください。

- [lib / main.dart](#)

下記の手順で、iOSとAndroidで実行されるインタラクティブなFlutterアプリを作成しました。

- Dartコードの記述
- 開発サイクルを短縮するためのホットリロードの使用
- ステートフルウィジェットを実装し、アプリに双方向性を追加
- ルートを作成し、ホーム画面と新しい画面の間を移動するためのロジックを追加
- テーマを使用してアプリのUIの外観を変更する方法を学ぶ

次のリソースからFlutterSDKの詳細をご覧ください。

- [Flutterのレイアウト](#)
- [Flutterアプリにインタラクティブ機能を追加する](#)
- [ウィジェットの紹介](#)
- [Android開発者向けのフラッター](#)
- [ReactNative開発者のためのフラッター](#)
- [Web開発者向けのフラッター](#)
- [FlutterのYouTubeチャンネル](#)

その他のリソースは次のとおりです。

- [クックブック](#)
- [ダートへのブートストラップ](#)

また、[Flutterコミュニティに接続してください](#)！