Local Persistence

Introduction

In computer science, persistence refers to the characteristic of state that outlives the process that created it. This is achieved in practice by storing the state as data in computer data storage.
So, it means the storage of data for later use, even after the program that created it has been closed.
In the context of this book, there are two main types of persistence:

- Remote Persistence.
    - This would be achieved by communicating with a remote computer using a protocol like Http. We have already covered Http in another chapter.
- Local Persistence
    - Persisting data to the device running the Flutter app.

The purpose of this chapter is to cover Local Persistence.

Your Options

In regard to local persistence, you have the following options:

- Using a sql database
    - This is (obviously) the most powerful option, especially for querying data.
    - We will cover the SQLite database in this chapter. It is recommended for Flutter as it is an easy-to-use package for Flutter and it works on both Android & iOS.
- Using local files.
    - Not good for querying data.
    - Good for complicated objects and large amounts of data.
    - You have to write the code that reads the data from the files, as well as the code that writes the data to the files.
    - You have full control over the file format.
    - Easy to copy this data to another device as a file.
- Using shared preferences.
    - This is using the shared_preferences package.
    - This is great for simple data, it's very easy to use.
    - Probably not the best way to store complicated objects or large amounts of data.

SQLite Database

This Flutter package is available here: https://pub.dartlang.org/packages/sqflite

Introduction

- This database runs amazingly fast. Note that there is no 'please wait' code in the example. It was just not required as all of the database operations were instantaneous.
- It was also simple to setup and get working.
- It also had versioning built-in out of the box. You could write code to the database object to handle initial database creation, when the database version changed etc.
- It had the ability to use 'data objects' (in the example this is a Word object).
- It had transaction handling.

Step 1 – Add Dependencies to Project

Add the following dependencies to your 'pubspec.yaml' file. After that you will need to do a 'flutter packages get' on the command line in the root of your project to download the dependencies.

- The sqflite package provides classes and functions that allow you to interact with a SQLite database.
- The path package provides functions that allow you to correctly define the location to store the database on disk.

```
dependencies:
 flutter:
   sdk: flutter
 sqflite:
 path:
```

Step 2 – Define the Data Model

At this point you should create the Dart classes that represent entities in your database. In my example, I create a 'Word' class. Note how I implemented the 'equals' and 'hashcode' so that the Word could be compared with other Words using an '=='.

```
class Word {
 final int _id;
 final String _english;
 final String _spanish;

 Word(this._id, this._english, this._spanish);

 Map<String, dynamic> toMap() {
  return {'id': _id, 'english': _english, 'spanish': _spanish};
 }

 String get spanish => _spanish;
```

```
  String get english => _english;

  int get id => _id;

  operator ==(other) =>
     (other != null) && (other is Word) && (_id == other._id);

  int get hashCode => _id.hashCode;
}
```

Step 3 – Open the Database

You should open the database when the app runs. It is two-step procedure and each step is asynchronous:

- Load database path.
- Open database.

Load Database Path

```
Future<bool> loadDatabasesPath() async {
   _databasesPath = await getDatabasesPath();
   return true;
  }
```

Open Database

Note how the 'openAndInitDatabase' method in the example code both initializes (only once) and returns the database. The database initialization is performed when it is fired by 'onCreate'.

```
Future<bool> openAndInitDatabase() async {
   _database = await openDatabase(
     join(_databasesPath, 'vocabulary.db'),
     onCreate: (db, version) {
       debugPrint("creating database...");
       db.execute("CREATE TABLE word(id INTEGER PRIMARY KEY, english TEXT, "
          "spanish TEXT, correct INTEGER, incorrect INTEGER)");
       db.execute("INSERT INTO word(english, spanish) "
          "VALUES ('uncle', 'tio')");
       db.execute("INSERT INTO word(english, spanish) "
          "VALUES ('reader', 'lector')");
       db.execute("INSERT INTO word(english, spanish) "
          "VALUES ('to keep vigil over', 'velar')");
       db.execute("INSERT INTO word(english, spanish) "
          "VALUES ('to remove', 'quitar')");
       db.execute("INSERT INTO word(english, spanish) "
          "VALUES ('to continue', 'reanudar')");
       db.execute("INSERT INTO word(english, spanish) "
          "VALUES ('until', 'hasta')");
       debugPrint("done");
     },
     version: 1,
   );
```

```
  return true;
 }
```

Retrieve Rows from Database

You use the 'query' method to retrieve data from the database.

```
final List<Map<String, dynamic>> words = await _database.query('word');
final List<Word> list = List.generate(words.length, (i) {
 return Word(words[i]['id'], words[i]['english'], words[i]['spanish']);
});
```

Executing SQL

The database object provides a 'execute' method in case you need to execute an SQL commands.

```
db.execute("INSERT INTO word(english, spanish) "
        "VALUES ('uncle', 'tio')");
```

Insert into Database

The database object provides an 'insert' method in case you need to insert rows into the database. Make sure that the primary key field is null if you want the SQLite to insert a new id for you.

```
 Future<int> addWord(Word word) async {
  return await _database.insert(
   'word',
   word.toMap(),
   conflictAlgorithm: ConflictAlgorithm.replace,
  );
 }
```

Update Row in Database

The database object provides an 'update' method in case you need to insert rows into the database.

```
Future<void> updateDog(Dog dog) async {
 // Get a reference to the database
 final db = await database;

 // Update the given Dog
 await db.update(
  'dogs',
  dog.toMap(),
  // Ensure we only update the Dog with a matching id
  where: "id = ?",
  // Pass the Dog's id through as a whereArg to prevent SQL injection
  whereArgs: [dog.id],
 );
```

```
}
```

Delete Row in Database

The database object provides an 'delete' method in case you need to delete rows into the database.

```
Future<void> deleteWord(Word word) async {
  return await _database.delete(
    'word',
    where: "id = ?",
    whereArgs: [word.id],
  );
}
```

Example – 'sqlite_vocabulary'

This app was written to help either an English-speaking person learn Spanish or a Spanish-speaking person learn English. The UI could definitely be improved but really the purpose of this app is to show how Flutter can work with a database.

It has three buttons at the top:

- Change mode from English -> Spanish to Spanish -> English (and back again).
- Add a new word.
- Delete the current word.

It has two floating buttons at the bottom:

- The button in the middle reveals the answer for the current word. For example, if you are asked 'Word in English is reader. What is the word in Spanish?' then it will reveal 'lector'.
- The button on the right moves onto the next word, randomly chosen.

Dependencies

Add the following dependencies to your 'pubspec.yaml' file. After that you will need to do a 'flutter packages get' on the command line in the root of your project to download the dependencies.

```
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

 sqflite:
 path:
```

Source Code

All of the words are stored in the database and all of the database code is contained in the 'DbWidget' inherited widget, at the top of the Widget tree so it can be accessed from any other Widget.

```
import 'dart:async';
import 'dart:math';

import 'package:flutter/material.dart';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';

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

```
enum Language { english, spanish }

class Word {
  final int _id;
  final String _english;
  final String _spanish;

  Word(this._id, this._english, this._spanish);

  Map<String, dynamic> toMap() {
    return {'id': _id, 'english': _english, 'spanish': _spanish};
  }

  String get spanish => _spanish;

  String get english => _english;

  int get id => _id;

  operator ==(other) =>
      (other != null) && (other is Word) && (_id == other._id);

  int get hashCode => _id.hashCode;
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return DbWidget(
        child: MaterialApp(
          title: 'Flutter Demo',
          theme: ThemeData(
            primarySwatch: Colors.blue,
          ),
          home: HomeWidget()));
  }
}

class DbWidget extends InheritedWidget {
  final _random = new Random();
  Database _database;
  String _databasesPath;

  DbWidget({Key key, @required Widget child})
      : assert(child != null),
        super(key: key, child: child);

  Future<bool> loadDatabasesPath() async {
    _databasesPath = await getDatabasesPath();
    return true;
  }

  Future<bool> openAndInitDatabase() async {
    _database = await openDatabase(
      join(_databasesPath, 'vocabulary.db'),
      onCreate: (db, version) {
```

```
    debugPrint("creating database...");
    db.execute("CREATE TABLE word(id INTEGER PRIMARY KEY, english TEXT, "
      "spanish TEXT, correct INTEGER, incorrect INTEGER)");
    db.execute("INSERT INTO word(english, spanish) "
      "VALUES ('uncle', 'tio')");
    db.execute("INSERT INTO word(english, spanish) "
      "VALUES ('reader', 'lector')");
    db.execute("INSERT INTO word(english, spanish) "
      "VALUES ('to keep vigil over', 'velar')");
    db.execute("INSERT INTO word(english, spanish) "
      "VALUES ('to remove', 'quitar')");
    db.execute("INSERT INTO word(english, spanish) "
      "VALUES ('to continue', 'reanudar')");
    db.execute("INSERT INTO word(english, spanish) "
      "VALUES ('until', 'hasta')");
    debugPrint("done");
   },
   version: 1,
  );
  return true;
}

Future<Word> loadNextWord(Word priorWord) async {
  final List<Map<String, dynamic>> words = await _database.query('word');
  final List<Word> list = List.generate(words.length, (i) {
   return Word(words[i]['id'], words[i]['english'], words[i]['spanish']);
  });

  Word nextWord = null;
  do {
   int nextWordIndex = _nextRandom(0, list.length);
   nextWord = list[nextWordIndex];
  } while (nextWord == priorWord);
  return nextWord;
}

Future<int> addWord(Word word) async {
  return await _database.insert(
   'word',
   word.toMap(),
   conflictAlgorithm: ConflictAlgorithm.replace,
  );
}

Future<void> deleteWord(Word word) async {
  return await _database.delete(
   'word',
   where: "id = ?",
   whereArgs: [word.id],
  );
}

static DbWidget of(BuildContext context) {
  return context.inheritFromWidgetOfExactType(DbWidget) as DbWidget;
}

@override
bool updateShouldNotify(covariant InheritedWidget oldWidget) {
```

```
    return false;
  }

  int _nextRandom(int min, int max) => min + _random.nextInt(max - min);
}

class HomeWidget extends StatefulWidget {
  HomeWidget({Key key}) : super(key: key);

  @override
  _HomeWidgetState createState() => _HomeWidgetState();
}

class _HomeWidgetState extends State<HomeWidget> {
  final GlobalKey<ScaffoldState> _scaffoldKey = GlobalKey<ScaffoldState>();

  bool _loadedDatabasePath = false;
  bool _openedDatabase = false;
  Language _language = Language.spanish;
  Word _priorWord;
  Word _word;

  _showSnackBar(String content, {bool error = false}) {
    _scaffoldKey.currentState.showSnackBar(SnackBar(
      content:
          Text('${error ? "An unexpected error occurred: " : ""}${content}'),
    ));
  }

  _loadDatabasesPath(BuildContext context) {
    try {
      DbWidget.of(context).loadDatabasesPath().then((b) {
        setState(() {
          _loadedDatabasePath = true;
        });
      }).catchError((error) {
        _showSnackBar(error.toString(), error: true);
      });
    } catch (e) {
      _showSnackBar(e.toString(), error: true);
    }
  }

  _openAndInitDatabase(BuildContext context) {
    try {
      DbWidget.of(context).openAndInitDatabase().then((b) {
        setState(() {
          _openedDatabase = true;
        });
      }).catchError((error) {
        _showSnackBar(error.toString(), error: true);
      });
    } catch (e) {
      _showSnackBar(e.toString(), error: true);
    }
  }

  _loadWord(BuildContext context) {
```

```
  try {
    DbWidget.of(context).loadNextWord(_priorWord).then((word) {
      setState(() {
        _word = word;
      });
    }).catchError((error) {
      _showSnackBar(error.toString(), error: true);
    });
  } catch (e) {
    _showSnackBar(e.toString(), error: true);
  }
}

@override
Widget build(BuildContext context) {
  if (!_loadedDatabasePath) {
    _loadDatabasesPath(context);
  } else if (!_openedDatabase) {
    _openAndInitDatabase(context);
  } else if (_word == null) {
    _loadWord(context);
  }

  WordWidget englishWordWidget =
      WordWidget(Language.english, _language, _word);
  WordWidget spanishWordWidget =
      WordWidget(Language.spanish, _language, _word);

  Column wordWidgets = _language == Language.spanish
      ? Column(children: [englishWordWidget, spanishWordWidget])
      : Column(children: [spanishWordWidget, englishWordWidget]);

  AppBar appBar = AppBar(title: Text("Vocabulary"), actions: <Widget>[
    IconButton(icon: Icon(Icons.shuffle), onPressed: () => _switchLanguage()),
    IconButton(icon: Icon(Icons.add), onPressed: () => _addWord(context)),
    IconButton(
        icon: Icon(Icons.remove), onPressed: () => _deleteWord(context))
  ]);

  return Scaffold(
      key: _scaffoldKey,
      appBar: appBar,
      body: wordWidgets,
      floatingActionButton: FloatingActionButton(
          child: Icon(Icons.refresh), onPressed: () => _loadNextWord()));
}

_loadNextWord() {
  setState(() {
    _priorWord = _word;
    _word = null;
  });
}

_switchLanguage() {
  Language newLanguage =
      _language == Language.spanish ? Language.english : Language.spanish;
  setState(() => _language = newLanguage);
```

```
  }

  _addWord(BuildContext context) async {
    Word word = await showDialog<Word>(
        context: context,
        builder: (BuildContext context) {
          return Dialog(child: AddDialogWidget());
        });
    if (word != null) {
      try {
        DbWidget.of(context).addWord(word).then((_) {
          _loadNextWord();
          _showSnackBar("Added word.");
        }).catchError((e) => _showSnackBar(e.toString(), error: true));
      } catch (e) {
        _showSnackBar(e.toString(), error: true);
      }
    }
  }

  _deleteWord(BuildContext context) {
    _showConfirmDialog(context, _word).then((result) {
      if (result == true) {
        try {
          DbWidget.of(context).deleteWord(_word).then((_) {
            _loadNextWord();
            _showSnackBar("Deleted word.");
          }).catchError((e) => _showSnackBar(e.toString(), error: true));
        } catch (e) {
          _showSnackBar(e.toString(), error: true);
        }
      }
    });
  }
}

class WordWidget extends StatefulWidget {
  WordWidget(this._widgetLanguage, this._language, this._word) {}

  final Language _widgetLanguage;
  final Language _language;
  final Word _word;

  @override
  _WordWidgetState createState() => _WordWidgetState();
}

class _WordWidgetState extends State<WordWidget> {
  bool _revealed = false;

  _WordWidgetState() {}

  @override
  void didUpdateWidget(Widget oldWidget) {
    _revealed = false;
  }

  @override
```

```
Widget build(BuildContext context) {
 bool isReveal = widget._widgetLanguage == widget._language;

 List<Widget> widgets = [];

 String titleText = isReveal
    ? "What's the word in ${getLanguageName(widget._widgetLanguage)}?"
    : "Word in ${getLanguageName(widget._widgetLanguage)} is:";

 widgets.add(Padding(
    padding: EdgeInsets.only(bottom: 20.0),
    child: Text(titleText,
       style: const TextStyle(fontSize: 30.0, fontWeight: FontWeight.bold),
       textAlign: TextAlign.center)));

 if ((isReveal) && (!_revealed)) {
  widgets.add(FloatingActionButton(
     child: Icon(Icons.remove_red_eye),
     onPressed: () => {setState(() => _revealed = true)}));
 } else {
  String word = widget._word == null
     ? ""
     : widget._widgetLanguage == Language.english
        ? widget._word._english
        : widget._word._spanish;
  widgets.add(Text(
   word,
   style: const TextStyle(
      fontSize: 30.0,
      fontWeight: FontWeight.bold,
      fontStyle: FontStyle.italic),
   textAlign: TextAlign.center,
  ));
 }

 return Expanded(
    child: Container(
  child: Column(
     mainAxisAlignment: MainAxisAlignment.center,
     crossAxisAlignment: CrossAxisAlignment.stretch,
     children: widgets),
  decoration: BoxDecoration(
   image: DecorationImage(
    colorFilter: new ColorFilter.mode(
       Colors.white.withOpacity(0.3), BlendMode.dstATop),
    image: NetworkImage(widget._widgetLanguage == Language.english
       ? "https://upload.wikimedia.org/wikipedia/en/thumb/a/ae/" +
          "Flag_of_the_United_Kingdom.svg/" +
          "510px-Flag_of_the_United_Kingdom.svg.png"
       : "https://upload.wikimedia.org/wikipedia/en/thumb/9/9a/" +
          "Flag_of_Spain.svg/400px-Flag_of_Spain.svg.png"),
    fit: BoxFit.cover,
   ),
  ),
  padding: EdgeInsets.all(10.0),
 ));
}
```

```
  String getLanguageName(Language language) {
   return widget._widgetLanguage == Language.spanish ? "Spanish" : "English";
  }
}

class AddDialogWidget extends StatelessWidget {
  static final _formKey = GlobalKey<FormState>();
  static final TextEditingController _englishTextController =
     new TextEditingController();
  static final TextEditingController _spanishTextController =
     new TextEditingController();

  AddDialogWidget() : super();

  @override
  Widget build(BuildContext context) {
   return Container(
      height: 260.0,
      width: 250.0,
      child: Padding(
         padding: EdgeInsets.all(10.0),
         child: Form(
            key: _formKey,
            child: Column(
               mainAxisAlignment: MainAxisAlignment.spaceAround,
               children: [
                Text("Add Word",
                   style: TextStyle(
                      fontSize: 20.0, fontWeight: FontWeight.bold)),
                TextFormField(
                   validator: (value) {
                    if (value.isEmpty) {
                     return 'Please enter the word in English.';
                    }
                   },
                   decoration: InputDecoration(
                      icon: const Icon(Icons.location_city),
                      hintText: 'English',
                      labelText: 'Enter the word in English'),
                   onSaved: (String value) {},
                   controller: _englishTextController),
                TextFormField(
                   validator: (value) {
                    if (value.isEmpty) {
                     return 'Please enter the word in Spanish.';
                    }
                   },
                   decoration: InputDecoration(
                      icon: const Icon(Icons.location_city),
                      hintText: 'Spanish',
                      labelText: 'Enter the word in Spanish'),
                   onSaved: (String value) {},
                   controller: _spanishTextController),
                FlatButton(
                   child: Text("Add"),
                   onPressed: () {
                    if (_formKey.currentState.validate()) {
                     _formKey.currentState.save();
```

```
                    Navigator.pop(
                      context,
                      Word(null, _englishTextController.text,
                        _spanishTextController.text));
                    _englishTextController.text = "";
                    _spanishTextController.text = "";
                  }
                })
            ])));
  }
}

Future<bool> _showConfirmDialog(BuildContext context, Word word) async {
  return await showDialog<bool>(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: const Text('Confirm'),
        content: Text(
          'Are you sure you want to delete the word "${word.english}?'),
        actions: <Widget>[
          FlatButton(
            onPressed: () {
              Navigator.pop(context, true);
            },
            child: const Text('Yes'),
          ),
          FlatButton(
            onPressed: () {
              Navigator.pop(context, false);
            },
            child: const Text('No'),
          )
        ],
      );
    });
}
```

Further Reading

https://medium.com/flutter-community/using-sqlite-in-flutter-187c1a82e8b
https://flutter.dev/docs/cookbook/persistence/sqlite
https://proandroiddev.com/flutter-bookshelf-app-part-3-managing-data-the-right-way-30569abf9487

Local Files

Introduction

If you don't need to query but you need to store possibly complex objects and lots of data with full-control then this is probably the best way to do it.

Flutter provides a core package 'dart.io' to help you with input and output at the device level. Remember that this may be different for different devices (platforms). For example, some of the file details may be different for an Android than iOS. That is why the Platform class is covered below.

The Flutter 'dart.io' core package includes Directory and File objects for the purpose of working with Directories and Files. These objects are excellent because they can work both synchronously and asynchronously, allowing you to maintain a responsive app even when dealing with large amounts of data.

However, this package does not tell you how to store the data in the files, what file format to use and how to serialize and deserialize objects into files. That is both good and bad but it requires some work on your part.

Platform

When you are coding with local files and directories, sometimes you need information about the device platform:

- Number of processors.
- Path separator.
- Operating System.
- Operating System version.
- Local hostname.
- Version.

The Platform class exists to provide this information to you.

Path Separator

Very useful when you want to separate elements from the path, such as the directory and the filename.

In the example below, I create a 'Directory' object and use it to query local files in the 'Application Documents' directory. When I do this, I get a list of files and each file has a path, which includes the filename at the end. I parse out the filename by finding the last path file separator (using Platform.pathSeparator) and calculating the filename as the rest of the path from there onward.

```
Directory(_path).listSync().forEach((FileSystemEntity fse) {
    String path = fse.path;
    if (path.endsWith(".themeColor")) {
      int startIndex = path.lastIndexOf(Platform.pathSeparator) + 1;
      int endIndex = path.lastIndexOf(".themeColor");
      filenameList.add(path.substring(startIndex, endIndex));
    }
```

```
  });
```

Path Provider Package

This is a package that (obviously) provides information about commonly used locations on the filesystem:

```
Directory tempDir = await getTemporaryDirectory();
Directory appDocDir = await getApplicationDocumentsDirectory();
```

It supports iOS and Android. More information here:
https://pub.dartlang.org/packages/path_provider

We use it in the example below, as it involves files in the Application Documents directory.

Application Documents Directory

This is a directory that your app has access to, as a place to store local files. Remember that you can create subdirectories within this directory as well as files. If you look at the constructor for the BLoC in the example code below, you will see that you get its value using an asynchronous method call to 'getApplicationDocumentsDirectory' in the path provider package (see above).

```
ThemeBLOC({Key key, @required Widget child})
  : assert(child != null),
    super(key: key, child: child) {
 getApplicationDocumentsDirectory()
    .then((directory) => _path = directory.path);
```

Directories

In order to work with Directories, the core Flutter package 'dart.io' provides a Directory object. You can create Directory objects from paths or uris. It provides methods for getting information about the directory, as well as methods for modifying it. It also has properties for providing more information.

Files

In order to work with Files, the core Flutter package 'dart.io' provides a File object. You can create File objects from paths or uris. It provides methods for getting information about the file, as well as methods for opening it, reading from it, writing to it and setting file information (such as when it was last accessed or modified). It also has properties for providing more information.
Note that you can open files in the following modes:

Mode	Description
READ	Mode for opening a file only for reading.

WRITE	Mode for opening a file for reading and writing.
APPEND	Mode for opening a file for reading and writing to the end of it.
WRITE ONLY	Mode for opening a file for writing only.
WRITE ONLY APPEND	Mode for opening a file for writing only to the end of it.

Directory & File Methods

Note that the Directory and File objects provide both synchronous and asynchronous methods. Obviously, you should consider asynchronous methods if you think these methods could take some time to complete.

Reading & Writing Data to a File

You need to decide the file format before you write code to read & write the data in the file. You can choose a text format or a binary file format.

Text & Binary Files

A text file stores data in the form of alphabets, digits and other special symbols by storing their ASCII values and are in a human readable format.
A binary file contains a sequence or a collection of bytes which are not in a human readable format.
A small error in a textual file can be recognized and eliminated when seen. Whereas, a small error in a binary file corrupts the file and is not easy to detect.

Text / JSON Format

When I wrote this example, I had just covered the working on the Flutter JSON example here: Serializing & Deserializing JSON. So JSON was fresh in my mind and I chose that format, working with the Flutter 'convert' package methods 'jsonEncode' and 'jsonDecode'.

Within the JSON encoding, the example uses two methods to serialize/deserialize the color: 'colorToJson' and 'jsonToColor'.

- 'colorToJson' works by matching the color from the list of colors using the color value, then returning the text.
- 'jsonToColor' works by matching the color from the list of colors using the text value, then returning the color.

Write Data to a File

Note that there are different ways to write data to a file:

- Write as bytes.

- Write as string.

Note that you can perform this operation synchronously or asynchronously.
Code from the example below:

```
saveAs(String filename) {
  String json = jsonEncode(_colorOptions.toJson());
  File("${_path}/${filename}.themeColor").writeAsString(json);
}
```

Read Data from a File

Note that there are different ways to read a file:

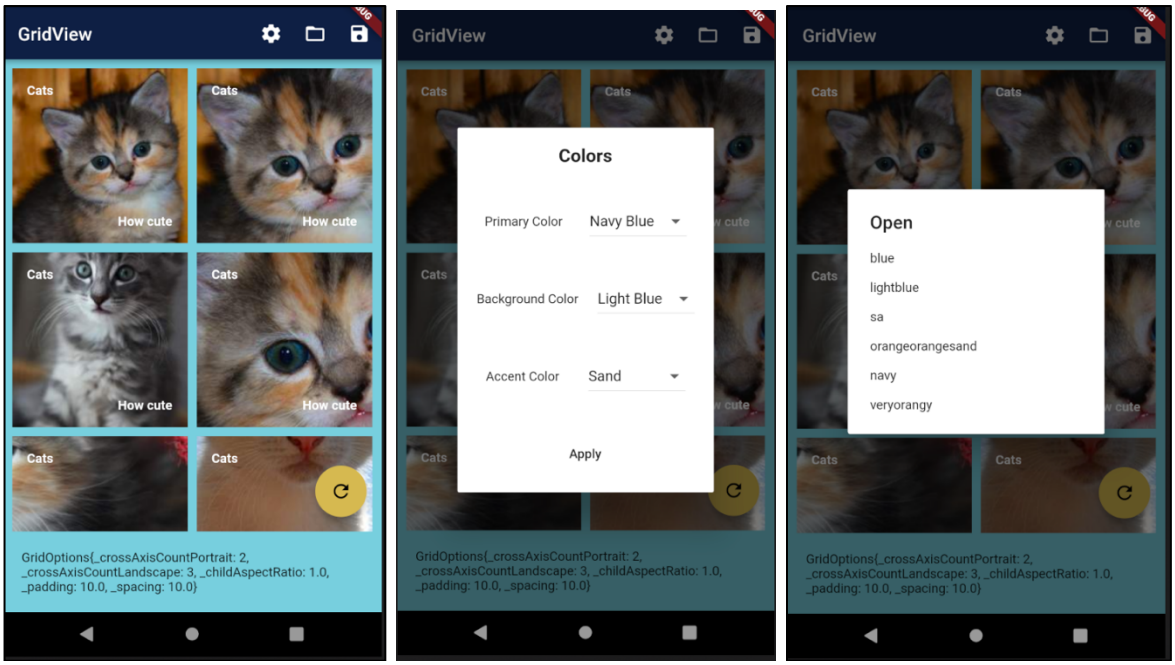
- Read as bytes.
- Read as lines.
- Read as strings.

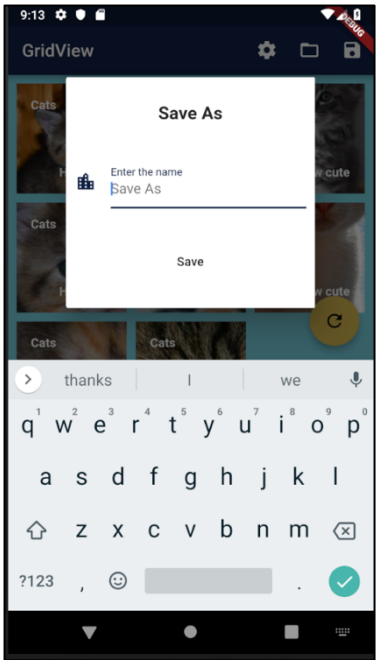
Note that you can perform this operation synchronously or asynchronously.
Code from the example below:

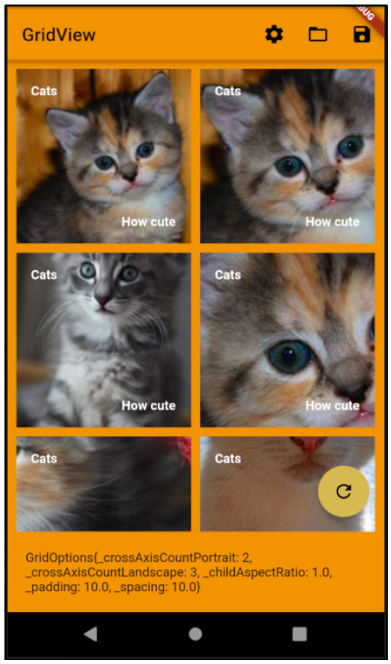
```
File("${fse.path}").readAsString().then((str) {
      ColorOptions newColorOptions = ColorOptions.fromJson(jsonDecode(str));
      this.colorOptions = newColorOptions;
    });
```

Example ‘persistence_files’

This app shows the grid of cat pictures but it also has toolbar options to configure the colors, open a color theme and save a color theme. It stores the color themes as local files (with the file extension ‘.themeColor’).







This example uses the BLoC pattern for the theme color state: [State & BLoCs w/Streams Approach] .

This example also has some useful keyboard code that only allows the user to enter names with letters a-z.

Dependencies

Add the following dependencies to your 'pubspec.yaml' file. After that you will need to do a 'flutter packages get' on the command line in the root of your project to download the dependencies.

```
dependencies:
  flutter:
    sdk: flutter
  rxdart: 0.18.1
  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^0.1.2
  path_provider: ^0.5.0+1
dev_dependencies:
  flutter_test:
    sdk: flutter
```

Source Code:

```
import 'dart:convert';
import 'dart:io';

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

```
import 'package:path_provider/path_provider.dart';
import 'package:rxdart/rxdart.dart';

void main() => runApp(ThemeBLOC(child: new GridViewApp()));

//TODO Fix horrible color choices. :)
const COLOR_COFFEE = Color.fromARGB(0xFF, 112, 80, 80);
const COLOR_DARK_BROWN = Color.fromARGB(0xFF, 59, 20, 18);
const COLOR_GREY = Color.fromARGB(0xFF, 68, 68, 68);
const COLOR_LIGHT_BLUE = Color.fromARGB(0xFF, 122, 207, 221);
const COLOR_MAROON = Color.fromARGB(0xFF, 86, 18, 16);
const COLOR_NAVY_BLUE = Color.fromARGB(0xFF, 15, 32, 67);
const COLOR_ORANGE = Color.fromARGB(0xFF, 240, 146, 34);
const COLOR_SAND = Color.fromARGB(0xFF, 213, 184, 88);
const COLOR_YELLOW = Color.fromARGB(0xFF, 246, 236, 32);

const COLOR_DROPDOWN_MENU_ITEMS = [
  DropdownMenuItem(value: COLOR_COFFEE, child: const Text("Coffee")),
  DropdownMenuItem(value: COLOR_DARK_BROWN, child: const Text("Dark Brown")),
  DropdownMenuItem(value: COLOR_GREY, child: const Text("Grey")),
  DropdownMenuItem(value: COLOR_LIGHT_BLUE, child: const Text("Light Blue")),
  DropdownMenuItem(value: COLOR_MAROON, child: const Text("Maroon")),
  DropdownMenuItem(value: COLOR_NAVY_BLUE, child: const Text("Navy Blue")),
  DropdownMenuItem(value: COLOR_ORANGE, child: const Text("Orange")),
  DropdownMenuItem(value: COLOR_SAND, child: const Text("Sand")),
  DropdownMenuItem(value: COLOR_YELLOW, child: const Text("Yellow")),
];

class ColorOptions {
  Color primaryColor;
  Color scaffoldBackgroundColor;
  Color accentColor;

  ColorOptions(
      {@required this.primaryColor,
      @required this.scaffoldBackgroundColor,
      @required this.accentColor});

  ColorOptions.copyOf(ColorOptions other) {
    this.primaryColor = other.primaryColor;
    this.scaffoldBackgroundColor = other.scaffoldBackgroundColor;
    this.accentColor = other.accentColor;
  }

  Map<String, dynamic> toJson() {
    Map<String, dynamic> map = {
      'primaryColor': '${colorToJson(primaryColor)}',
      'scaffoldBackgroundColor': '${colorToJson(scaffoldBackgroundColor)}',
      'accentColor': '${colorToJson(accentColor)}'
    };
    return map;
  }

  ColorOptions.fromJson(Map<String, dynamic> json)
      : primaryColor = jsonToColor(json['primaryColor']),
        scaffoldBackgroundColor = jsonToColor(json['scaffoldBackgroundColor']),
        accentColor = jsonToColor(json['accentColor']);
```

```
  static String colorToJson(Color color) {
    DropdownMenuItem menuItemForColor =
        COLOR_DROPDOWN_MENU_ITEMS.firstWhere((item) => item.value == color);
    return (menuItemForColor.child as Text).data;
  }

  static Color jsonToColor(String json) {
    DropdownMenuItem menuItemForColor = COLOR_DROPDOWN_MENU_ITEMS
        .firstWhere((item) => (item.child as Text).data == json);
    return menuItemForColor.value;
  }
}

class GridOptions {
  int crossAxisCountPortrait;
  int crossAxisCountLandscape;
  double childAspectRatio;
  double padding;
  double spacing;

  GridOptions(
      {@required this.crossAxisCountPortrait,
      @required this.crossAxisCountLandscape,
      @required this.childAspectRatio,
      @required this.padding,
      @required this.spacing});

  @override
  String toString() {
    return 'GridOptions{_crossAxisCountPortrait: $crossAxisCountPortrait, _crossAxisCountLandscape:
$crossAxisCountLandscape, _childAspectRatio: $childAspectRatio, _padding: $padding, _spacing:
$spacing}';
  }
}

class ThemeBLOC extends InheritedWidget {
  String _path;

  ThemeBLOC({Key key, @required Widget child})
      : assert(child != null),
        super(key: key, child: child) {
    getApplicationDocumentsDirectory()
        .then((directory) => _path = directory.path);
  }

  ColorOptions _colorOptions = ColorOptions(
      primaryColor: COLOR_NAVY_BLUE,
      scaffoldBackgroundColor: COLOR_LIGHT_BLUE,
      accentColor: COLOR_SAND);

  static ThemeBLOC of(BuildContext context) {
    return context.inheritFromWidgetOfExactType(ThemeBLOC) as ThemeBLOC;
  }

  ThemeData get startingThemeData {
    return createThemeDataFromColorOptions();
  }
```

```
ThemeData createThemeDataFromColorOptions() {
  return ThemeData(
      primaryColor: _colorOptions.primaryColor,
      scaffoldBackgroundColor: _colorOptions.scaffoldBackgroundColor,
      accentColor: _colorOptions.accentColor);
}

@override
bool updateShouldNotify(covariant InheritedWidget oldWidget) {
  // We are going to use a stream for updating widget tree (see StreamBuilder).
  return false;
}

// Used to update widget tree (see StreamBuilder).
Stream<ThemeData> get themeStream => _themeSubject.stream;
final _themeSubject = BehaviorSubject<ThemeData>();

ColorOptions get colorOptions => _colorOptions;

set colorOptions(ColorOptions value) {
  _colorOptions = value;
  _themeSubject.add(createThemeDataFromColorOptions()); // update widget tree
}

List<String> get filenames {
  List<String> filenameList = [];
  Directory(_path).listSync().forEach((FileSystemEntity fse) {
    String path = fse.path;
    if (path.endsWith(".themeColor")) {
      int startIndex = path.lastIndexOf(Platform.pathSeparator) + 1;
      int endIndex = path.lastIndexOf(".themeColor");
      filenameList.add(path.substring(startIndex, endIndex));
    }
  });
  return filenameList;
}

open(String filename) {
  FileSystemEntity fse =
      Directory(_path).listSync().firstWhere((FileSystemEntity fse) {
    String path = fse.path;
    if (path.endsWith(".themeColor")) {
      int startIndex = path.lastIndexOf(Platform.pathSeparator) + 1;
      if (startIndex != -1) {
        int endIndex = path.lastIndexOf(".themeColor");
        if (endIndex != -1) {
          var pathFilename = path.substring(startIndex, endIndex);
          if (pathFilename == filename) {
            return true;
          }
        }
      }
    }
    return false;
  });
  if (fse != null) {
    File("${fse.path}").readAsString().then((str) {
      ColorOptions newColorOptions = ColorOptions.fromJson(jsonDecode(str));
```

```
      this.colorOptions = newColorOptions;
    });
   }
  }

  saveAs(String filename) {
   String json = jsonEncode(_colorOptions.toJson());
   File("${_path}/${filename}.themeColor").writeAsString(json);
  }
}

class GridViewApp extends StatelessWidget {
 // This widget is the root of your application.
 @override
 Widget build(BuildContext context) {
  ThemeBLOC bloc = ThemeBLOC.of(context);
  return StreamBuilder<ThemeData>(
     // listens to stream in ThemeBLOC to know when to update
     stream: bloc._themeSubject,
     initialData: bloc.startingThemeData,
     builder: (context, snapshot) {
      ThemeData themeData = snapshot.data;
      return MaterialApp(
       title: 'Flutter Demo',
       theme: themeData,
       home: HomeWidget(title: 'Flutter Demo Home Page'),
      );
     });
 }
}

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

 final String title;

 @override
 _HomeWidgetState createState() => new _HomeWidgetState();
}

class _HomeWidgetState extends State<HomeWidget> {
 List<Widget> _kittenTiles = [];
 int _gridOptionsIndex = 0;
 List<GridOptions> _gridOptions = [
  GridOptions(
     crossAxisCountPortrait: 2,
     crossAxisCountLandscape: 3,
     childAspectRatio: 1.0,
     padding: 10.0,
     spacing: 10.0),
  GridOptions(
     crossAxisCountPortrait: 3,
     crossAxisCountLandscape: 4,
     childAspectRatio: 1.5,
     padding: 10.0,
     spacing: 10.0),
  GridOptions(
     crossAxisCountPortrait: 2,
```

```
    crossAxisCountLandscape: 3,
    childAspectRatio: 2.0,
    padding: 10.0,
    spacing: 30.0),
];

_HomeWidgetState() : super() {
 for (int i = 200; i < 1000; i += 100) {
  String imageUrl = "http://placekitten.com/200/${i}";
  _kittenTiles.add(GridTile(
     header: GridTileBar(
        title:
           Text("Cats", style: TextStyle(fontWeight: FontWeight.bold))),
     footer: GridTileBar(
        title: Text("How cute",
           textAlign: TextAlign.right,
           style: TextStyle(fontWeight: FontWeight.bold))),
     child: Image.network(imageUrl, fit: BoxFit.cover)));
 }
}

void _tryMoreGridOptions() {
 setState(() {
  _gridOptionsIndex++;
  if (_gridOptionsIndex >= (_gridOptions.length - 1)) {
   _gridOptionsIndex = 0;
  }
 });
}

@override
Widget build(BuildContext context) {
 GridOptions options = _gridOptions[_gridOptionsIndex];
 return Scaffold(
  appBar: AppBar(title: Text("GridView"), actions: [
   IconButton(
      icon: Icon(Icons.settings),
      tooltip: 'Color Options',
      onPressed: () => _showColorOptionsDialog()),
   IconButton(
      icon: Icon(Icons.folder_open),
      tooltip: 'Open',
      onPressed: () {
       List<String> names = ThemeBLOC.of(context).filenames;
       _showOpenDialog(context, names);
      }),
   IconButton(
      icon: Icon(Icons.save),
      tooltip: 'Save',
      onPressed: () => _showSaveAsDialog(context))
  ]),
  body: OrientationBuilder(builder: (context, orientation) {
   return GridView.count(
      crossAxisCount: (orientation == Orientation.portrait)
         ? options.crossAxisCountPortrait
         : options.crossAxisCountLandscape,
      childAspectRatio: options.childAspectRatio,
      padding: EdgeInsets.all(options.padding),
```

```
        mainAxisSpacing: options.spacing,
        crossAxisSpacing: options.spacing,
        children: _kittenTiles);
    }),
    bottomNavigationBar: Container(
       child: Text(options.toString()), padding: EdgeInsets.all(20.0)),
    floatingActionButton: new FloatingActionButton(
     onPressed: _tryMoreGridOptions,
     tooltip: 'Try more grid options',
     child: new Icon(Icons.refresh),
    ), // This trailing comma makes auto-formatting nicer for build methods.
  );
}

void _showColorOptionsDialog() async {
  ColorOptions colorOptions = await showDialog<ColorOptions>(
     context: context,
     builder: (BuildContext context) {
       return Dialog(
          child: ColorDialogWidget(ThemeBLOC.of(context).colorOptions));
     });
  if (colorOptions != null) {
    ThemeBLOC.of(context).colorOptions = colorOptions;
  }
}

void _showOpenDialog(BuildContext context, List<String> names) async {
  List<SimpleDialogOption> children = names.map((s) {
    return SimpleDialogOption(
     onPressed: () {
       Navigator.pop(context, s);
     },
     child: Text(s),
    );
  }).toList(growable: false);

  String name = await showDialog<String>(
     context: context,
     builder: (BuildContext context) {
       return SimpleDialog(title: const Text('Open'), children: children);
     });

  if (name != null) {
    setState(() {
     ThemeBLOC.of(context).open(name);
    });
  }
}

void _showSaveAsDialog(BuildContext context) async {
  String name = await showDialog<String>(
     context: context,
     builder: (BuildContext context) {
       return Dialog(child: SaveAsDialogWidget());
     });
  if (name != null) {
    ThemeBLOC.of(context).saveAs(name);
  }
```

```
  }
}

class ColorDialogWidget extends StatefulWidget {
  ColorOptions _colorOptions;

  ColorDialogWidget(this._colorOptions) : super();

  @override
  _CustomDialogWidgetState createState() =>
      new _CustomDialogWidgetState(ColorOptions.copyOf(this._colorOptions));
}

class _CustomDialogWidgetState extends State<ColorDialogWidget> {
  ColorOptions _colorOptions;

  _CustomDialogWidgetState(this._colorOptions);

  @override
  Widget build(BuildContext context) {
    return Container(
        height: 400.0,
        width: 250.0,
        child:
            Column(mainAxisAlignment: MainAxisAlignment.spaceAround, children: <
                Widget>[
          Text("Colors",
              style: TextStyle(fontSize: 20.0, fontWeight: FontWeight.bold)),
          Row(mainAxisAlignment: MainAxisAlignment.center, children: <Widget>[
            Spacer(),
            Text("Primary Color"),
            Spacer(),
            new DropdownButton<Color>(
              value: _colorOptions.primaryColor,
              items: COLOR_DROPDOWN_MENU_ITEMS,
              onChanged: (newValue) {
                setState(() {
                  _colorOptions.primaryColor = newValue;
                });
              },
            ),
            Spacer(),
          ]),
          Row(mainAxisAlignment: MainAxisAlignment.center, children: <Widget>[
            Spacer(),
            Text("Background Color"),
            Spacer(),
            new DropdownButton<Color>(
              value: _colorOptions.scaffoldBackgroundColor,
              items: COLOR_DROPDOWN_MENU_ITEMS,
              onChanged: (newValue) {
                setState(() {
                  _colorOptions.scaffoldBackgroundColor = newValue;
                });
              },
            ),
            Spacer(),
          ]),
```

```
      Row(mainAxisAlignment: MainAxisAlignment.center, children: <Widget>[
       Spacer(),
       Text("Accent Color"),
       Spacer(),
       new DropdownButton<Color>(
        value: _colorOptions.accentColor,
        items: COLOR_DROPDOWN_MENU_ITEMS,
        onChanged: (newValue) {
         setState(() {
          _colorOptions.accentColor = newValue;
         });
        },
       ),
       Spacer(),
      ]),
      FlatButton(
        child: Text("Apply"),
        onPressed: () => Navigator.pop(context, _colorOptions))
    ]));
 }
}

class SaveAsDialogWidget extends StatelessWidget {
 static final _formKey = GlobalKey<FormState>();
 static final TextEditingController _nameTextController =
    new TextEditingController();

 SaveAsDialogWidget() : super();

 @override
 Widget build(BuildContext context) {
  return Container(
    height: 260.0,
    width: 250.0,
    child: Padding(
      padding: EdgeInsets.all(10.0),
      child: Form(
        key: _formKey,
        child: Column(
          mainAxisAlignment: MainAxisAlignment.spaceAround,
          children: [
           Text("Save As",
             style: TextStyle(
               fontSize: 20.0, fontWeight: FontWeight.bold)),
           TextFormField(
             autofocus: true,
             validator: (value) {
              if (value.isEmpty) {
               return 'Please enter the name.';
              }
             },
             decoration: InputDecoration(
               icon: const Icon(Icons.location_city),
               hintText: 'Save As',
               labelText: 'Enter the name'),
             keyboardType: TextInputType.text,
             inputFormatters: [
              WhitelistingTextInputFormatter(RegExp(r'[a-z]'))
```

```
                ],
                onSaved: (String value) {},
                controller: _nameTextController),
            FlatButton(
                child: Text("Save"),
                onPressed: () {
                  if (_formKey.currentState.validate()) {
                    _formKey.currentState.save();
                    Navigator.pop(context, _nameTextController.text);
                    _nameTextController.text = "";
                  }
                })
          ])));
  }
}
```

Shared Preferences

Introduction

The 'shared_preferences' package is very useful for providing a local persistent store for simple preference data. This data is lost if the user uninstalls the app or clears the app data.

Each preference item requires its own String key to identify it. In my code example, I use the String key 'themeList' to store the semi-colon delimited list of themes and I use a the theme name as the key for each theme stored as a preference.

More info here: https://pub.dartlang.org/packages/shared_preferences

Methods

Getting a List of All Preferences

This gets a set (similar to a list without duplicates) containing all the keys to local shared preferences.

```
Set<String> getKeys()
```

Getting a Preference

The method you use depends on the type of data stored in the preference.

Method	Description
dynamic get(String key)	Returns a preference for a key, could be any of the types below.
bool getBool(String key)	Returns a boolean preference for a key.
int getInt(String key)	Returns an integer preference for a key.

double getDouble(String key)	Returns a double preference for a key.
String getString(String key)	Returns a string preference for a key.
List<String> getStringList(String key)	Returns a string list preference for a key.

Setting a Preference

The method you use depends on the type of data you want stored in the preference.

Method	Description
Future<bool> setBool(String key)	Sets a boolean preference for a key.
Future<bool> setInt(String key)	Sets an integer preference for a key.
Future<bool> setDouble(String key)	Sets a double preference for a key.
Future<bool> setString(String key)	Sets a string preference for a key.
Future<bool> getStringList(String key)	Sets a string list preference for a key.

Removing a Preference

There is only one method call for all types.

Method	Description
Future<bool> remove(String key)	Removes an entry from persistent storage, whatever the type.

Further Reading

https://medium.com/flutter-community/shared-preferences-how-to-save-flutter-application-settings-and-user-preferences-for-later-554d08671ae9

Example 'persistence_shared_preferences'

This app shows the grid of cat pictures as before and it works in the same way. However, this time it uses the 'shared_preferences' package rather than local files.

Dependencies

Add the following dependencies to your 'pubspec.yaml' file. After that you will need to do a 'flutter packages get' on the command line in the root of your project to download the dependencies.

```
dependencies:
 flutter:
   sdk: flutter
 rxdart: 0.18.1
 # The following adds the Cupertino Icons font to your application.
 # Use with the CupertinoIcons class for iOS style icons.
 cupertino_icons: ^0.1.2
 shared_preferences: ^0.5.1+2
```

Source Code:

Most of the code is the same as the previous example but there are several differences in the ThemeBLOC class:

- The ThemeBLOC loads the SharedPreferences object asynchrously in the constructor.
- The preference 'themeList' is used to store the list of available themes in a single string, delimited by semi-colons.
    - Example of this format: 'themeOne;themeTwo'.
    - In retrospect, it would have been better to use the methods 'getStringList' and 'setStringList' rather than 'getString' and 'setString', instead of storing a list in a single string. It would have made the code less complex.
- Then each theme is stored as its own preference in the same Text / JSON format as in the previous example.

```
import 'dart:convert';
import 'dart:io';

import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:rxdart/rxdart.dart';
import 'package:shared_preferences/shared_preferences.dart';

void main() => runApp(ThemeBLOC(child: new GridViewApp()));

//TODO Fix horrible color choices. :)
const COLOR_COFFEE = Color.fromARGB(0xFF, 112, 80, 80);
const COLOR_DARK_BROWN = Color.fromARGB(0xFF, 59, 20, 18);
const COLOR_GREY = Color.fromARGB(0xFF, 68, 68, 68);
const COLOR_LIGHT_BLUE = Color.fromARGB(0xFF, 122, 207, 221);
const COLOR_MAROON = Color.fromARGB(0xFF, 86, 18, 16);
const COLOR_NAVY_BLUE = Color.fromARGB(0xFF, 15, 32, 67);
const COLOR_ORANGE = Color.fromARGB(0xFF, 240, 146, 34);
const COLOR_SAND = Color.fromARGB(0xFF, 213, 184, 88);
const COLOR_YELLOW = Color.fromARGB(0xFF, 246, 236, 32);

const COLOR_DROPDOWN_MENU_ITEMS = [
  DropdownMenuItem(value: COLOR_COFFEE, child: const Text("Coffee")),
  DropdownMenuItem(value: COLOR_DARK_BROWN, child: const Text("Dark Brown")),
  DropdownMenuItem(value: COLOR_GREY, child: const Text("Grey")),
  DropdownMenuItem(value: COLOR_LIGHT_BLUE, child: const Text("Light Blue")),
  DropdownMenuItem(value: COLOR_MAROON, child: const Text("Maroon")),
  DropdownMenuItem(value: COLOR_NAVY_BLUE, child: const Text("Navy Blue")),
  DropdownMenuItem(value: COLOR_ORANGE, child: const Text("Orange")),
  DropdownMenuItem(value: COLOR_SAND, child: const Text("Sand")),
  DropdownMenuItem(value: COLOR_YELLOW, child: const Text("Yellow")),
];

class ColorOptions {
  Color primaryColor;
  Color scaffoldBackgroundColor;
  Color accentColor;

  ColorOptions(
```

```
    {@required this.primaryColor,
    @required this.scaffoldBackgroundColor,
    @required this.accentColor});

  ColorOptions.copyOf(ColorOptions other) {
   this.primaryColor = other.primaryColor;
   this.scaffoldBackgroundColor = other.scaffoldBackgroundColor;
   this.accentColor = other.accentColor;
  }

  Map<String, dynamic> toJson() {
   Map<String, dynamic> map = {
    'primaryColor': '${colorToJson(primaryColor)}',
    'scaffoldBackgroundColor': '${colorToJson(scaffoldBackgroundColor)}',
    'accentColor': '${colorToJson(accentColor)}'
   };
   return map;
  }

  ColorOptions.fromJson(Map<String, dynamic> json)
     : primaryColor = jsonToColor(json['primaryColor']),
      scaffoldBackgroundColor = jsonToColor(json['scaffoldBackgroundColor']),
      accentColor = jsonToColor(json['accentColor']);

  static String colorToJson(Color color) {
   DropdownMenuItem menuItemForColor =
      COLOR_DROPDOWN_MENU_ITEMS.firstWhere((item) => item.value == color);
   return (menuItemForColor.child as Text).data;
  }

  static Color jsonToColor(String json) {
   DropdownMenuItem menuItemForColor = COLOR_DROPDOWN_MENU_ITEMS
      .firstWhere((item) => (item.child as Text).data == json);
   return menuItemForColor.value;
  }
}

class GridOptions {
  int crossAxisCountPortrait;
  int crossAxisCountLandscape;
  double childAspectRatio;
  double padding;
  double spacing;

  GridOptions(
    {@required this.crossAxisCountPortrait,
    @required this.crossAxisCountLandscape,
    @required this.childAspectRatio,
    @required this.padding,
    @required this.spacing});

  @override
  String toString() {
   return 'GridOptions{_crossAxisCountPortrait: $crossAxisCountPortrait, _crossAxisCountLandscape:
$crossAxisCountLandscape, _childAspectRatio: $childAspectRatio, _padding: $padding, _spacing:
$spacing}';
  }
}
```

```
class ThemeBLOC extends InheritedWidget {
 SharedPreferences _prefs;

 ThemeBLOC({Key key, @required Widget child})
    : assert(child != null),
      super(key: key, child: child) {
  SharedPreferences.getInstance().then((prefs) => _prefs = prefs);
 }

 ColorOptions _colorOptions = ColorOptions(
    primaryColor: COLOR_NAVY_BLUE,
    scaffoldBackgroundColor: COLOR_LIGHT_BLUE,
    accentColor: COLOR_SAND);

 static ThemeBLOC of(BuildContext context) {
  return context.inheritFromWidgetOfExactType(ThemeBLOC) as ThemeBLOC;
 }

 ThemeData get startingThemeData {
  return createThemeDataFromColorOptions();
 }

 ThemeData createThemeDataFromColorOptions() {
  return ThemeData(
     primaryColor: _colorOptions.primaryColor,
     scaffoldBackgroundColor: _colorOptions.scaffoldBackgroundColor,
     accentColor: _colorOptions.accentColor);
 }

 @override
 bool updateShouldNotify(covariant InheritedWidget oldWidget) {
  // We are going to use a stream for updating widget tree (see StreamBuilder).
  return false;
 }

 // Used to update widget tree (see StreamBuilder).
 Stream<ThemeData> get themeStream => _themeSubject.stream;
 final _themeSubject = BehaviorSubject<ThemeData>();

 ColorOptions get colorOptions => _colorOptions;

 set colorOptions(ColorOptions value) {
  _colorOptions = value;
  _themeSubject.add(createThemeDataFromColorOptions()); // update widget tree
 }

 List<String> get themes {
  // Return list of themes.
  String themes = _prefs.getString("themeList");
  return themes == null ? [] : themes.split(";");
 }

 open(String theme) {
  // Open theme preference.
  String themeAsJson = _prefs.getString(theme);
  ColorOptions newColorOptions =
     ColorOptions.fromJson(jsonDecode(themeAsJson));
```

```
    this.colorOptions = newColorOptions;
  }

  saveAs(String theme) {
    // Create new theme preference.
    String themeAsJson = jsonEncode(_colorOptions.toJson());
    _prefs.setString(theme, themeAsJson);

    // Add new theme preference to list of themes.
    String themeList = _prefs.getString('themeList');
    if ((themeList == null) || (themeList.isEmpty)) {
      _prefs.setString("themeList", theme);
    } else if (themeList.indexOf(theme) == -1) {
      _prefs.setString("themeList", themeList + ";" + theme);
    }
  }
}

class GridViewApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    ThemeBLOC bloc = ThemeBLOC.of(context);
    return StreamBuilder<ThemeData>(
        // listens to stream in ThemeBLOC to know when to update
        stream: bloc._themeSubject,
        initialData: bloc.startingThemeData,
        builder: (context, snapshot) {
          ThemeData themeData = snapshot.data;
          return MaterialApp(
            title: 'Flutter Demo',
            theme: themeData,
            home: HomeWidget(title: 'Flutter Demo Home Page'),
          );
        });
  }
}

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

  final String title;

  @override
  _HomeWidgetState createState() => new _HomeWidgetState();
}

class _HomeWidgetState extends State<HomeWidget> {
  List<Widget> _kittenTiles = [];
  int _gridOptionsIndex = 0;
  List<GridOptions> _gridOptions = [
    GridOptions(
        crossAxisCountPortrait: 2,
        crossAxisCountLandscape: 3,
        childAspectRatio: 1.0,
        padding: 10.0,
        spacing: 10.0),
    GridOptions(
```

```
    crossAxisCountPortrait: 3,
    crossAxisCountLandscape: 4,
    childAspectRatio: 1.5,
    padding: 10.0,
    spacing: 10.0),
  GridOptions(
    crossAxisCountPortrait: 2,
    crossAxisCountLandscape: 3,
    childAspectRatio: 2.0,
    padding: 10.0,
    spacing: 30.0),
];

_HomeWidgetState() : super() {
  for (int i = 200; i < 1000; i += 100) {
    String imageUrl = "http://placekitten.com/200/${i}";
    _kittenTiles.add(GridTile(
        header: GridTileBar(
            title:
                Text("Cats", style: TextStyle(fontWeight: FontWeight.bold))),
        footer: GridTileBar(
            title: Text("How cute",
                textAlign: TextAlign.right,
                style: TextStyle(fontWeight: FontWeight.bold))),
        child: Image.network(imageUrl, fit: BoxFit.cover)));
  }
}

void _tryMoreGridOptions() {
  setState(() {
    _gridOptionsIndex++;
    if (_gridOptionsIndex >= (_gridOptions.length - 1)) {
      _gridOptionsIndex = 0;
    }
  });
}

@override
Widget build(BuildContext context) {
  GridOptions options = _gridOptions[_gridOptionsIndex];
  return Scaffold(
    appBar: AppBar(title: Text("GridView"), actions: [
      IconButton(
          icon: Icon(Icons.settings),
          tooltip: 'Color Options',
          onPressed: () => _showColorOptionsDialog()),
      IconButton(
          icon: Icon(Icons.folder_open),
          tooltip: 'Open',
          onPressed: () {
            List<String> names = ThemeBLOC.of(context).themes;
            _showOpenDialog(context, names);
          }),
      IconButton(
          icon: Icon(Icons.save),
          tooltip: 'Save',
          onPressed: () => _showSaveAsDialog(context))
    ]),
```

```
    body: OrientationBuilder(builder: (context, orientation) {
     return GridView.count(
        crossAxisCount: (orientation == Orientation.portrait)
           ? options.crossAxisCountPortrait
           : options.crossAxisCountLandscape,
        childAspectRatio: options.childAspectRatio,
        padding: EdgeInsets.all(options.padding),
        mainAxisSpacing: options.spacing,
        crossAxisSpacing: options.spacing,
        children: _kittenTiles);
    }),
    bottomNavigationBar: Container(
       child: Text(options.toString()), padding: EdgeInsets.all(20.0)),
    floatingActionButton: new FloatingActionButton(
     onPressed: _tryMoreGridOptions,
     tooltip: 'Try more grid options',
     child: new Icon(Icons.refresh),
    ), // This trailing comma makes auto-formatting nicer for build methods.
   );
  }

  void _showColorOptionsDialog() async {
   ColorOptions colorOptions = await showDialog<ColorOptions>(
      context: context,
      builder: (BuildContext context) {
       return Dialog(
          child: ColorDialogWidget(ThemeBLOC.of(context).colorOptions));
      });
   if (colorOptions != null) {
    ThemeBLOC.of(context).colorOptions = colorOptions;
   }
  }

  void _showOpenDialog(BuildContext context, List<String> names) async {
   List<SimpleDialogOption> children = names.map((s) {
    return SimpleDialogOption(
     onPressed: () {
      Navigator.pop(context, s);
     },
     child: Text(s),
    );
   }).toList(growable: false);

   String name = await showDialog<String>(
      context: context,
      builder: (BuildContext context) {
       return SimpleDialog(title: const Text('Open'), children: children);
      });

   if (name != null) {
    setState(() {
     ThemeBLOC.of(context).open(name);
    });
   }
  }

  void _showSaveAsDialog(BuildContext context) async {
   String name = await showDialog<String>(
```

```
      context: context,
      builder: (BuildContext context) {
       return Dialog(child: SaveAsDialogWidget());
      });
   if (name != null) {
    ThemeBLOC.of(context).saveAs(name);
   }
  }
}

class ColorDialogWidget extends StatefulWidget {
 ColorOptions _colorOptions;

 ColorDialogWidget(this._colorOptions) : super();

 @override
 _CustomDialogWidgetState createState() =>
    new _CustomDialogWidgetState(ColorOptions.copyOf(this._colorOptions));
}

class _CustomDialogWidgetState extends State<ColorDialogWidget> {
 ColorOptions _colorOptions;

 _CustomDialogWidgetState(this._colorOptions);

 @override
 Widget build(BuildContext context) {
  return Container(
     height: 400.0,
     width: 250.0,
     child:
        Column(mainAxisAlignment: MainAxisAlignment.spaceAround, children: <
           Widget>[
       Text("Colors",
         style: TextStyle(fontSize: 20.0, fontWeight: FontWeight.bold)),
       Row(mainAxisAlignment: MainAxisAlignment.center, children: <Widget>[
        Spacer(),
        Text("Primary Color"),
        Spacer(),
        new DropdownButton<Color>(
         value: _colorOptions.primaryColor,
         items: COLOR_DROPDOWN_MENU_ITEMS,
         onChanged: (newValue) {
          setState(() {
           _colorOptions.primaryColor = newValue;
          });
         },
        ),
        Spacer(),
       ]),
       Row(mainAxisAlignment: MainAxisAlignment.center, children: <Widget>[
        Spacer(),
        Text("Background Color"),
        Spacer(),
        new DropdownButton<Color>(
         value: _colorOptions.scaffoldBackgroundColor,
         items: COLOR_DROPDOWN_MENU_ITEMS,
         onChanged: (newValue) {
```

```
            setState(() {
              _colorOptions.scaffoldBackgroundColor = newValue;
            });
          },
        ),
        Spacer(),
       ]),
       Row(mainAxisAlignment: MainAxisAlignment.center, children: <Widget>[
        Spacer(),
        Text("Accent Color"),
        Spacer(),
        new DropdownButton<Color>(
          value: _colorOptions.accentColor,
          items: COLOR_DROPDOWN_MENU_ITEMS,
          onChanged: (newValue) {
            setState(() {
              _colorOptions.accentColor = newValue;
            });
          },
        ),
        Spacer(),
       ]),
       FlatButton(
          child: Text("Apply"),
          onPressed: () => Navigator.pop(context, _colorOptions))
     ]));
  }
}

class SaveAsDialogWidget extends StatelessWidget {
  static final _formKey = GlobalKey<FormState>();
  static final TextEditingController _nameTextController =
     new TextEditingController();

  SaveAsDialogWidget() : super();

  @override
  Widget build(BuildContext context) {
   return Container(
      height: 260.0,
      width: 250.0,
      child: Padding(
         padding: EdgeInsets.all(10.0),
         child: Form(
            key: _formKey,
            child: Column(
               mainAxisAlignment: MainAxisAlignment.spaceAround,
               children: [
                Text("Save As",
                   style: TextStyle(
                      fontSize: 20.0, fontWeight: FontWeight.bold)),
                TextFormField(
                   autofocus: true,
                   validator: (value) {
                    if (value.isEmpty) {
                      return 'Please enter the name.';
                    }
                    if (value == "themeList") {
```

```
                    return 'You cannot use this name.';
                  }
                },
                decoration: InputDecoration(
                  icon: const Icon(Icons.location_city),
                  hintText: 'Save As',
                  labelText: 'Enter the name'),
                keyboardType: TextInputType.text,
                inputFormatters: [
                  WhitelistingTextInputFormatter(RegExp(r'[a-z]'))
                ],
                onSaved: (String value) {},
                controller: _nameTextController),
              FlatButton(
                child: Text("Save"),
                onPressed: () {
                  if (_formKey.currentState.validate()) {
                    _formKey.currentState.save();
                    Navigator.pop(context, _nameTextController.text);
                    _nameTextController.text = "";
                  }
                })
            ])));
  }
}
```