FLUTTER

SUCCINCTLY

BY **ED FREITAS**

SUCCINCTLY EBOOK SERIES

Syncfusion®

www.dbooks.org

Flutter Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj

Copyright © 2019 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

www.dbooks.org

Table of Contents

www.dbooks.org

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the

authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ed Freitas is a consultant on software development applied to customer success, mostly related to financial process automation, accounts payable processing, and data extraction.

He really likes technology and enjoys playing soccer, running, traveling, life hacking, learning, and spending time with his family.

You can reach him at: https://edfreitas.me

Acknowledgements

Many thanks to all the people from the amazing Syncfusion team who contributed to this book and helped it become a reality—especially Jacqueline Bieringer, Tres Watkins, Darren West, and Graham High.

The manuscript manager, Jacqueline Bieringer from Syncfusion, and the technical editor, James McCaffrey from Microsoft Research, thoroughly reviewed the book's organization, code quality, and overall accuracy. Thank you both.

This book is dedicated to *Mi Chelin, Lala,* and *Tita,* who inspire me every day and light up my path ahead—God bless you all, always.

Introduction

With the rapid rise of cross-platform mobile frameworks such as Ionic, React Native, and Xamarin, the folks at Google decided to step into the game and develop their own framework with support for both Android and iOS using the same codebase—this is how Flutter came to be.

Flutter is an open-source mobile application development SDK primarily developed and sponsored by Google, used for developing applications for Android and iOS—as well as being the primary method of creating applications for the Google Fuchsia operating system.

Flutter is written in C, C++, and Dart, and uses the Skia Graphics Engine. It offers a rich set of fully customizable widgets for building native interfaces, including the beautiful Material Design library and Cupertino (iOS-flavored) widgets, rich motion APIs, smooth natural scrolling, platform awareness, and hot reload—which helps to quickly build UIs without losing state on emulators, simulators, and any hardware for iOS and Android.

All these great features have helped Flutter take off very quickly, and developers are flocking to the framework. It's also one of the trending projects in GitHub, which has helped it gain even more popularity.

With Flutter gaining momentum, it seems unlikely that it will fade away anytime soon, so I decided to give it a whirl and write an application with it. My personal experience: I was blown away. I felt immediately productive, even though I had never programmed in Dart before.

Throughout this book, I want to go on that same journey with you. We'll do this by creating a fully functional app, which will allow you to get a good grasp of the framework—whether you are coming from another mobile development framework, or have no previous mobile development experience.

I'm quite excited to embark on this journey with you. I hope that by the end of it, you'll have a great impression of the framework and be able to assess whether Flutter is the right choice for your mobile development needs. So, without further ado, let's get going.

www.dbooks.org

Chapter 1 Setup

Project overview

The application that we'll be building throughout this book is one that we can use to keep track of important personal documents that have an expiration date, such as passports, driver's licenses, or credit cards.

This type of application is handy to have so we know when we need to renew these important documents before they expire. If you have read my other book, *Electron Succinctly*, this is the same application concept we explored back then.

Using the Dart programming language with Flutter, we will take our application one step further—we'll also explore how to use a local SQLite database.

Awesome—let's get our engines ready so we can start setting up our development environment straight away.

Installation

The Flutter setup is incredibly easy, with all the installation steps well documented within the official Flutter documentation site.

I'll be using Windows 10, so I'll be describing setup steps and information related to this operating system; however, there are also easy-to-follow setup guidelines for both macOS and Linux.

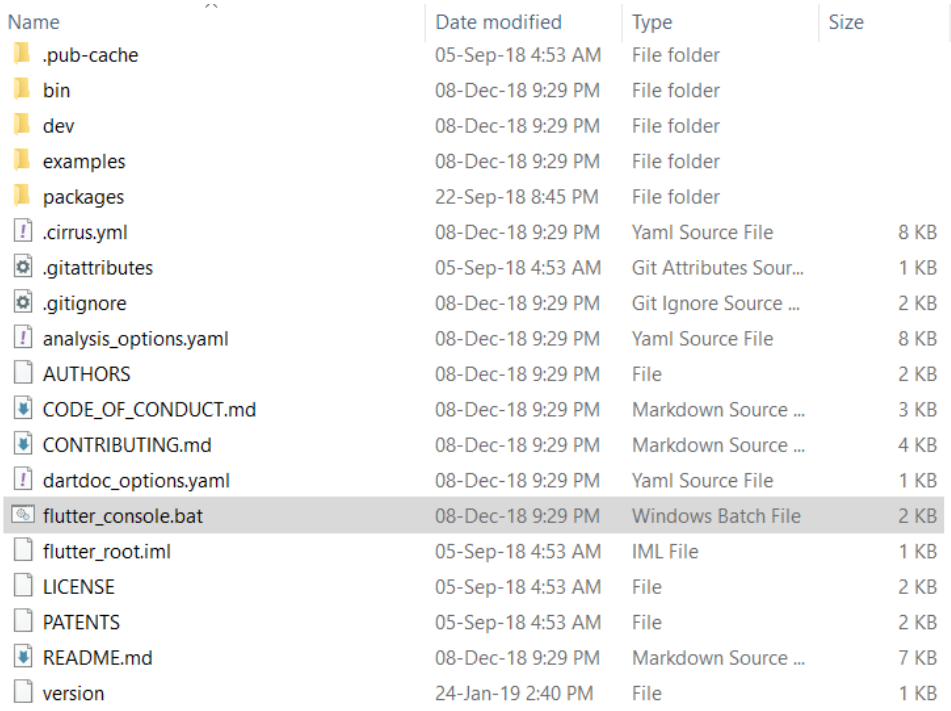
On Windows, there are some essential system requirements that need to be in place, which include having PowerShell 5.0 (or later) and Git for Windows 2.X (or later) installed.

Even though you can write Flutter apps in any editor of your choice—personally, I'm a big fan of Visual Studio Code—Flutter relies on a full installation of Android Studio to supply its Android platform dependencies. You'll also need to set up an Android device emulator. These steps are described in the official documentation.

With the prerequisites in place for Windows, all we need to do is download the installation bundle of the Flutter SDK—at the time of writing, it is Flutter's 1.0.0 stable version for Windows.

Once you've downloaded the zip file, extract it to a desired folder within your drive, such as C:\Flutter. Don't extract the Flutter files to C:\Program Files or C:\Program Files (x86), which require elevated or admin permissions.

Once the files are in the desired folder, locate the file Flutter_console.bat file—this is how it looks on my machine.



Name	Date modified	Type	Size
.pub-cache	05-Sep-18 4:53 AM	File folder	
bin	08-Dec-18 9:29 PM	File folder	
dev	08-Dec-18 9:29 PM	File folder	
examples	08-Dec-18 9:29 PM	File folder	
packages	22-Sep-18 8:45 PM	File folder	
.cirrus.yml	08-Dec-18 9:29 PM	Yaml Source File	8 KB
.gitattributes	05-Sep-18 4:53 AM	Git Attributes Sour...	1 KB
.gitignore	08-Dec-18 9:29 PM	Git Ignore Source ...	2 KB
analysis_options.yaml	08-Dec-18 9:29 PM	Yaml Source File	8 KB
AUTHORS	08-Dec-18 9:29 PM	File	2 KB
CODE_OF_CONDUCT.md	08-Dec-18 9:29 PM	Markdown Source ...	3 KB
CONTRIBUTING.md	08-Dec-18 9:29 PM	Markdown Source ...	4 KB
dartdoc_options.yaml	08-Dec-18 9:29 PM	Yaml Source File	1 KB
flutter_console.bat	08-Dec-18 9:29 PM	Windows Batch File	2 KB
flutter_root.iml	05-Sep-18 4:53 AM	IML File	1 KB
LICENSE	05-Sep-18 4:53 AM	File	2 KB
PATENTS	05-Sep-18 4:53 AM	File	2 KB
README.md	08-Dec-18 9:29 PM	Markdown Source ...	7 KB
version	24-Jan-19 2:40 PM	File	1 KB

Figure 1-a: The Flutter SDK files

In principle, you are now ready to run the Flutter console by executing the **Flutter_console.bat** file. It's recommended—although not strictly necessary—to add the **Flutter\Bin** folder to the **System Path Environment** variable in Windows.

If you are unsure how to add a folder to the Windows Path variable, please refer to this nice article that explains how to do it, step by step, with screenshots.

In my machine, this looks as follows.

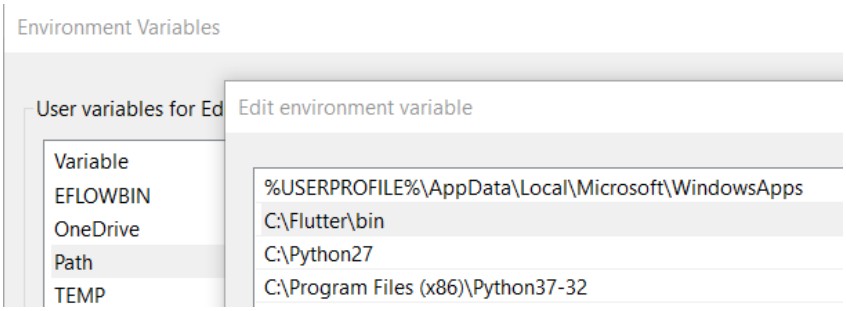


Figure 1-b: Flutter added to the Path variable in Windows

With the SDK file in place, we can now run the **Flutter_console.bat** file—this is what you should see.

www.dbooks.org

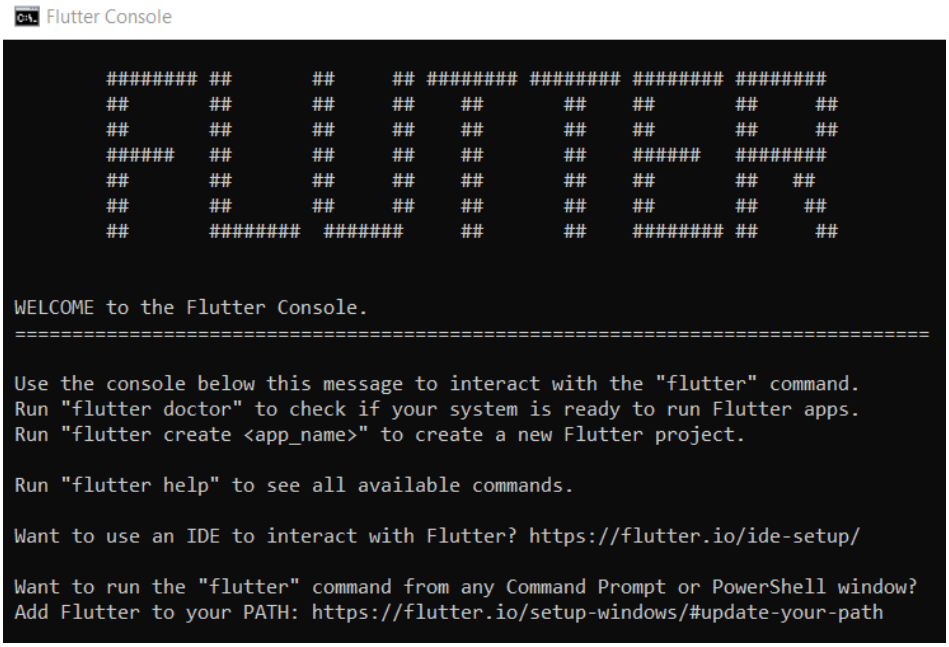


Figure 1-c: The Flutter console running

On the prompt, type the following command to check if Flutter is fully operational.

Code Listing 1-a: The "flutter doctor" command

```
flutter doctor
```

After you execute this command, you will get a result with any issues found—in my case, because I had previously installed Android Studio and Visual Studio Code, I get the following information.

When you run the Android Studio installer, please make sure you follow the official documentation so that you end up with a successful Android Studio and SDK setup.

```
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel beta, v1.0.0, on Microsoft Windows [Version 10.0.17763.253], locale en-US)
[!] Android toolchain - develop for Android devices (Android SDK 28.0.2)
    ! Some Android licenses not accepted.  To resolve this, run: flutter doctor --android-licenses
[√] Android Studio (version 3.2)
[√] VS Code, 64-bit edition (version 1.30.2)
[!] Connected device
    ! No devices available

! Doctor found issues in 2 categories.
```

Figure 1-d: Results from running the flutter doctor command

In my case, Flutter is telling me that I need to run the **`flutter doctor --android-licenses`** command to resolve an issue with some Android licenses not being accepted.

It also highlighted that I don't have a physical device connected, which is fine for now.

Make sure that you resolve all the conflicts highlighted by the **`flutter doctor`** command before proceeding.

Setting up an editor

Once you have completed all the installation steps, it is necessary to set up Flutter to work with your editor of choice. Although I usually use Visual Studio Code for my projects, this time I decided to use Android Studio to code our Flutter application—to me, it felt more natural, and a better fit for mobile development.

The official Flutter documentation describes how to configure Android Studio (IntelliJ) and Visual Studio Code to work with Flutter—please follow these steps.

If you will also be using Android Studio, once you have followed the steps described, you should see the Dart and Flutter plugins installed. On my machine, this looks as follows.

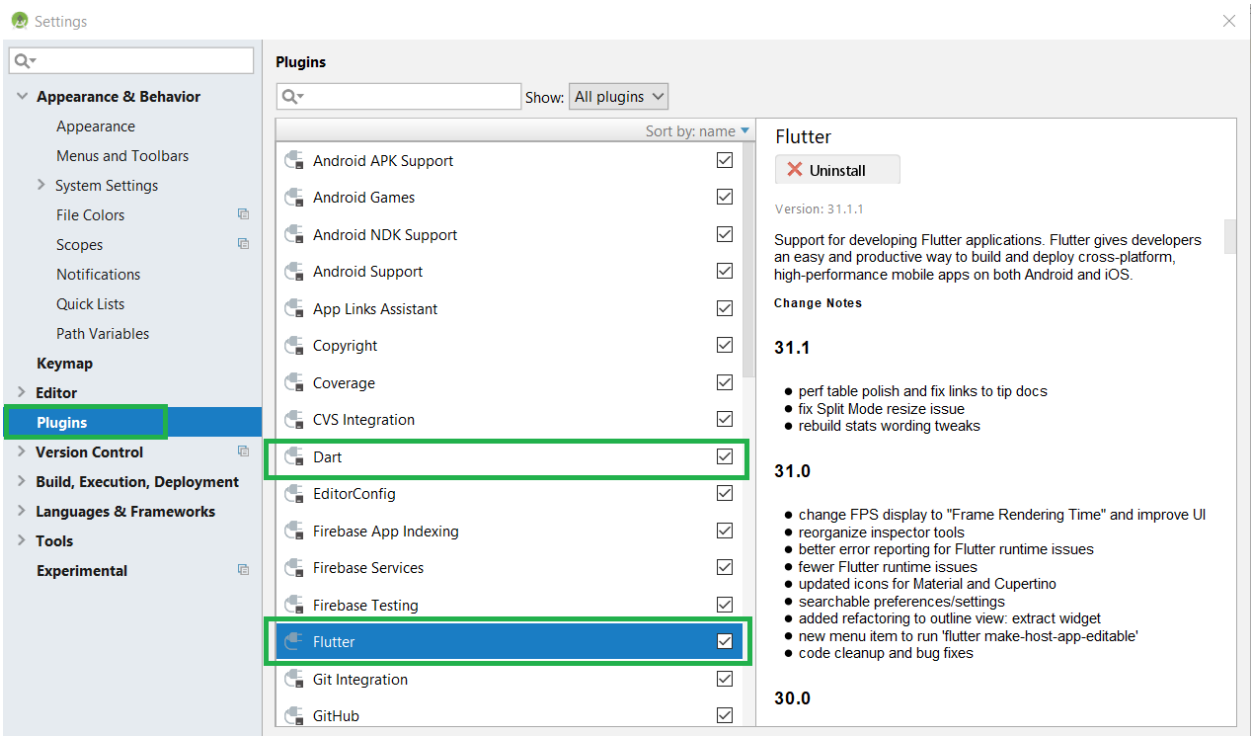


Figure 1-e: The Dart and Flutter plugins installed on Android Studio

Creating the app

Once your editor of choice has been correctly set up following the official documentation guidelines and my previous suggestions, it's time to perform a quick test. We'll create a demo application from one of the predefined templates, and then experience the "hot reload" mechanism after making a change to the app.

www.dbooks.org

The official documentation, which covers the steps that are described and explained in this section, is worth checking.

Open Android Studio and navigate to **File** > **New** > **New Flutter Project**. The following screen will be displayed.

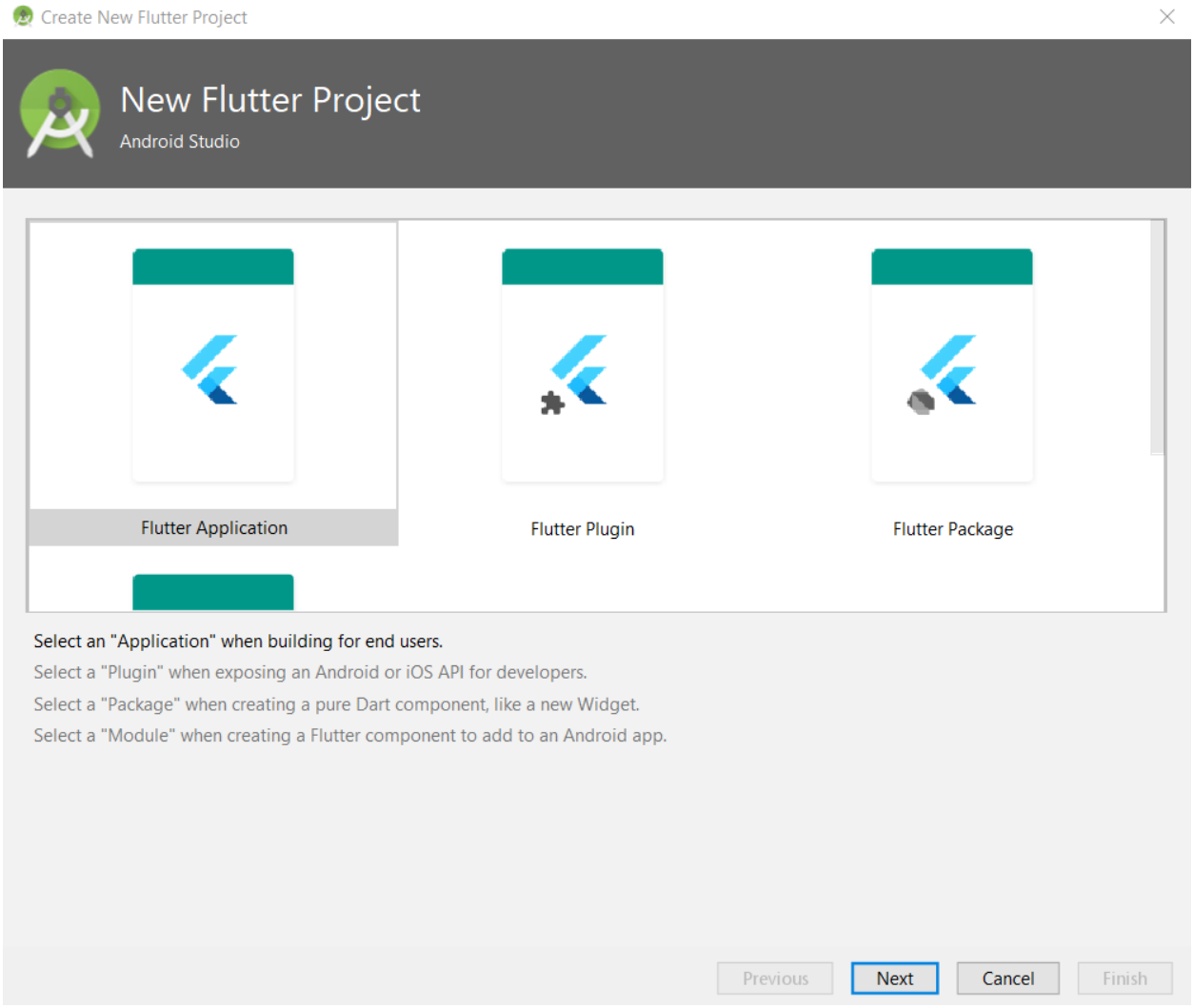


Figure 1-f: Create new Flutter project option (step 1)

Choose the **Flutter Application** option, and then click **Next**. We'll then be presented with a screen where we can enter the **Project name**, **Flutter SDK path**, **Project location**, and a **Description** for the application—we can see this as follows.

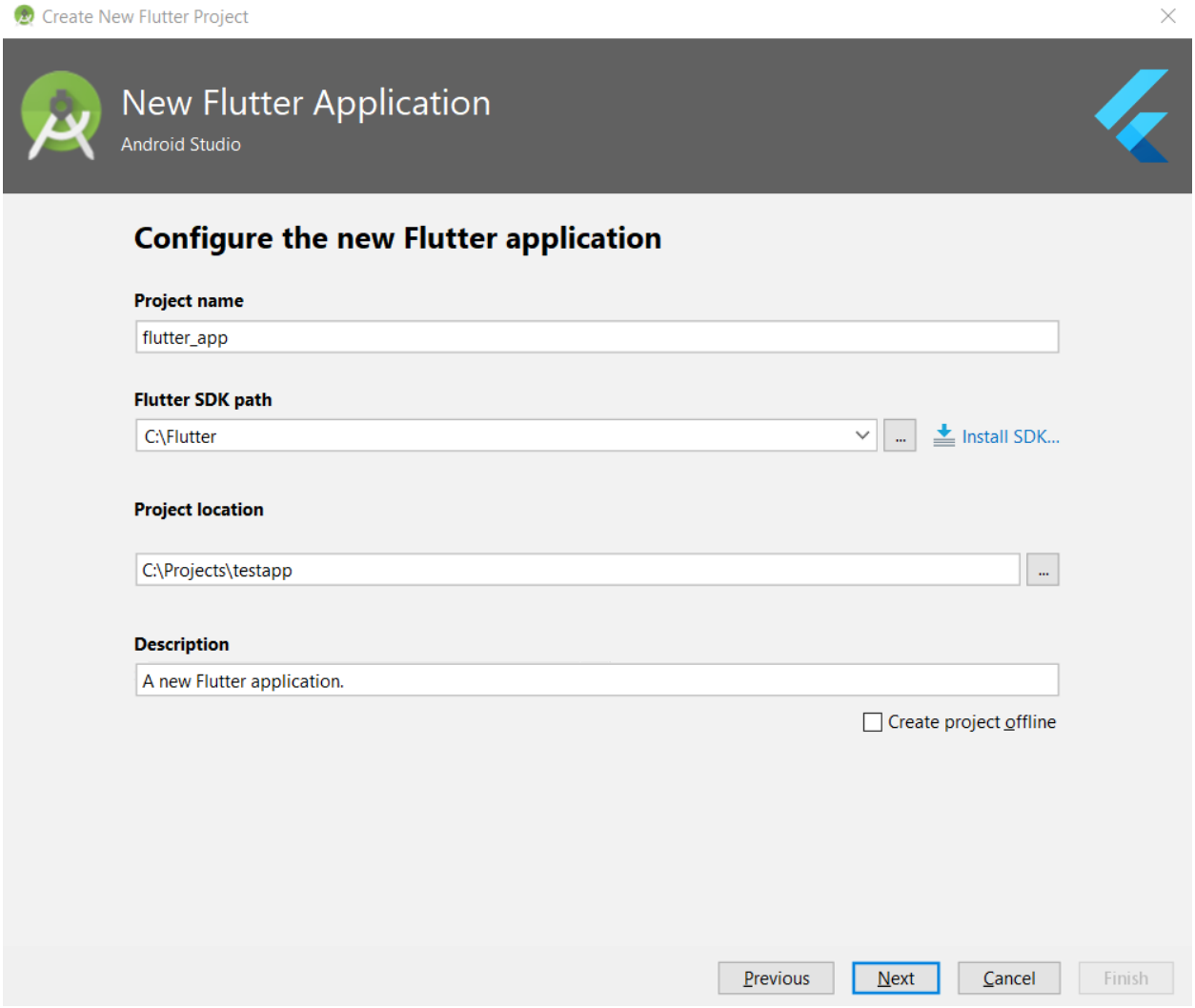


Figure 1-g: Create new Flutter project option (step 2)

Make sure the **Flutter SDK path** text field specifies the correct folder location of the SDK, as previously explained. With those options entered, click **Next**.

In the final step of the app creation process, we are asked to enter the **Company domain** and include (if applicable) **Kotlin support for Android code** and **Swift support for iOS code**—in our case, there's no need to include these options.

The following figure shows the final step of the application creation screen.

www.dbooks.org

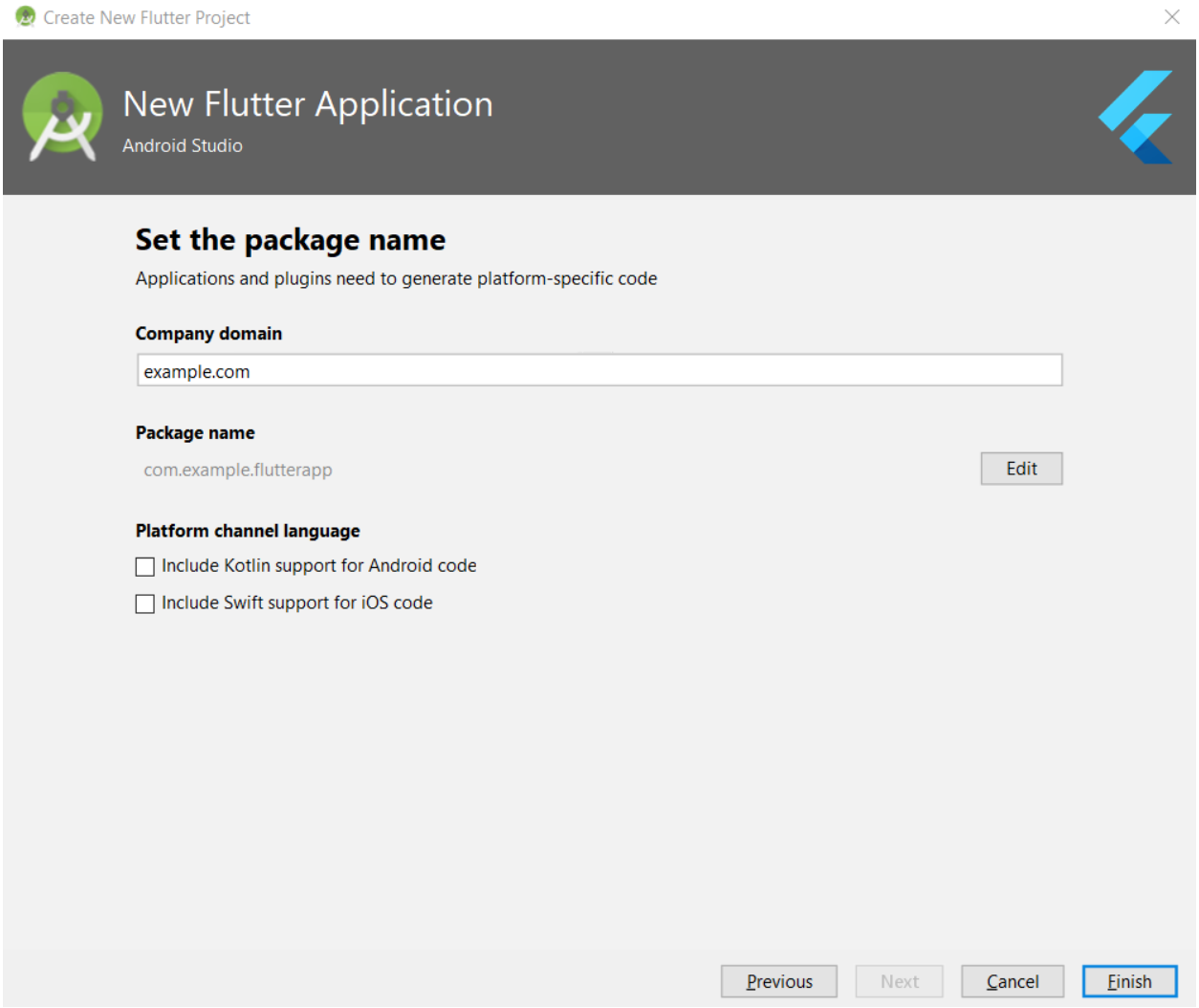


Figure 1-h: Create new Flutter project option (step 3)

To finalize the creation of the demo application, click **Finish**. With the demo application created, make sure you have a virtual device created and ready so we can quickly test the app.

Creating a virtual device

Let's quickly go over the steps required to create a virtual device. With Android Studio opened, go to the **Tools** menu and click on the **AVD Manager** option, which will display the following screen.

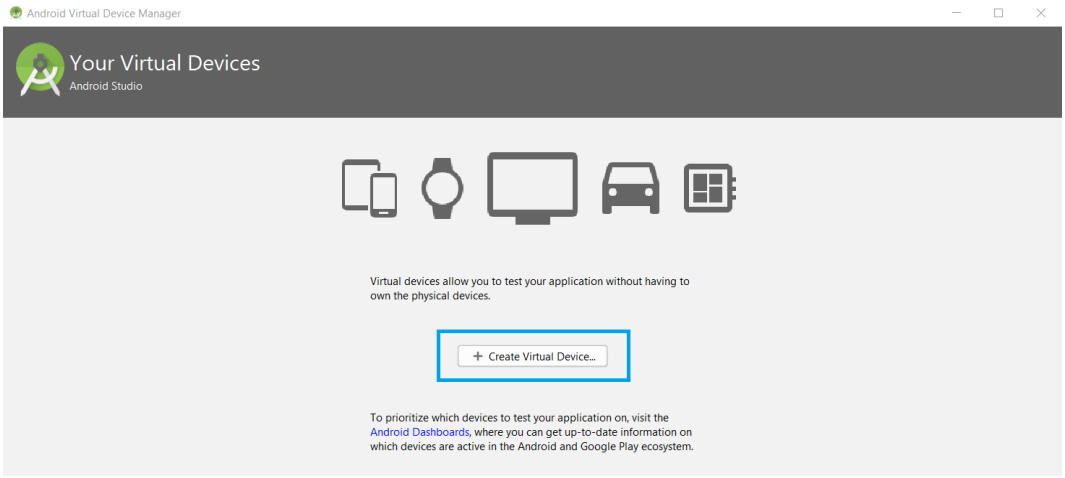


Figure 1-i: Creating a virtual device (step 1)

Then, click **Create Virtual Device**, which will display the following window with all the available virtual devices that can be created for different categories, such as: phones, TV, tablets, and wearable devices.

I'm going to select the **Nexus 6** model from the **Phone** category, but feel free to choose any other.

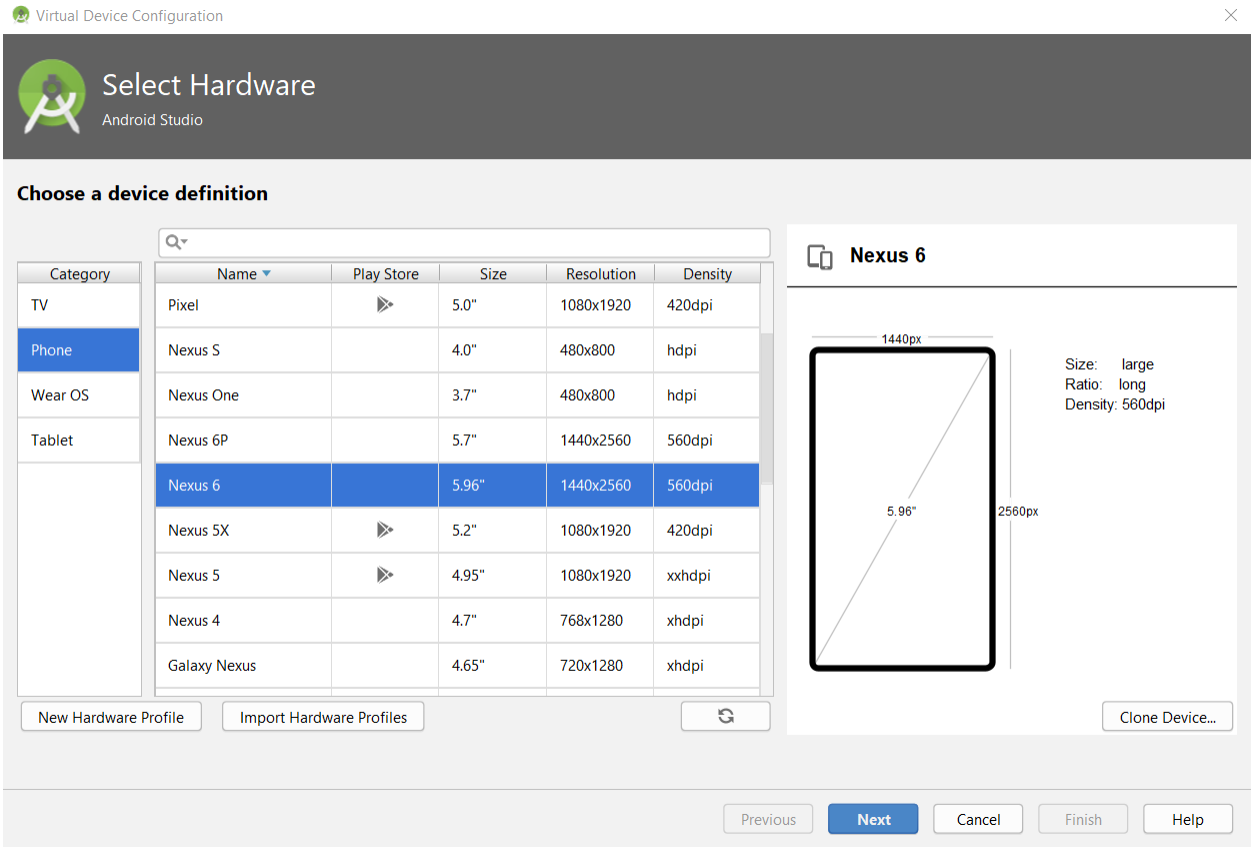


Figure 1-j: Creating a virtual device (step 2)

www.dbooks.org

Once you have your model selected, click **Next**. You will be prompted to select one of the available device images.

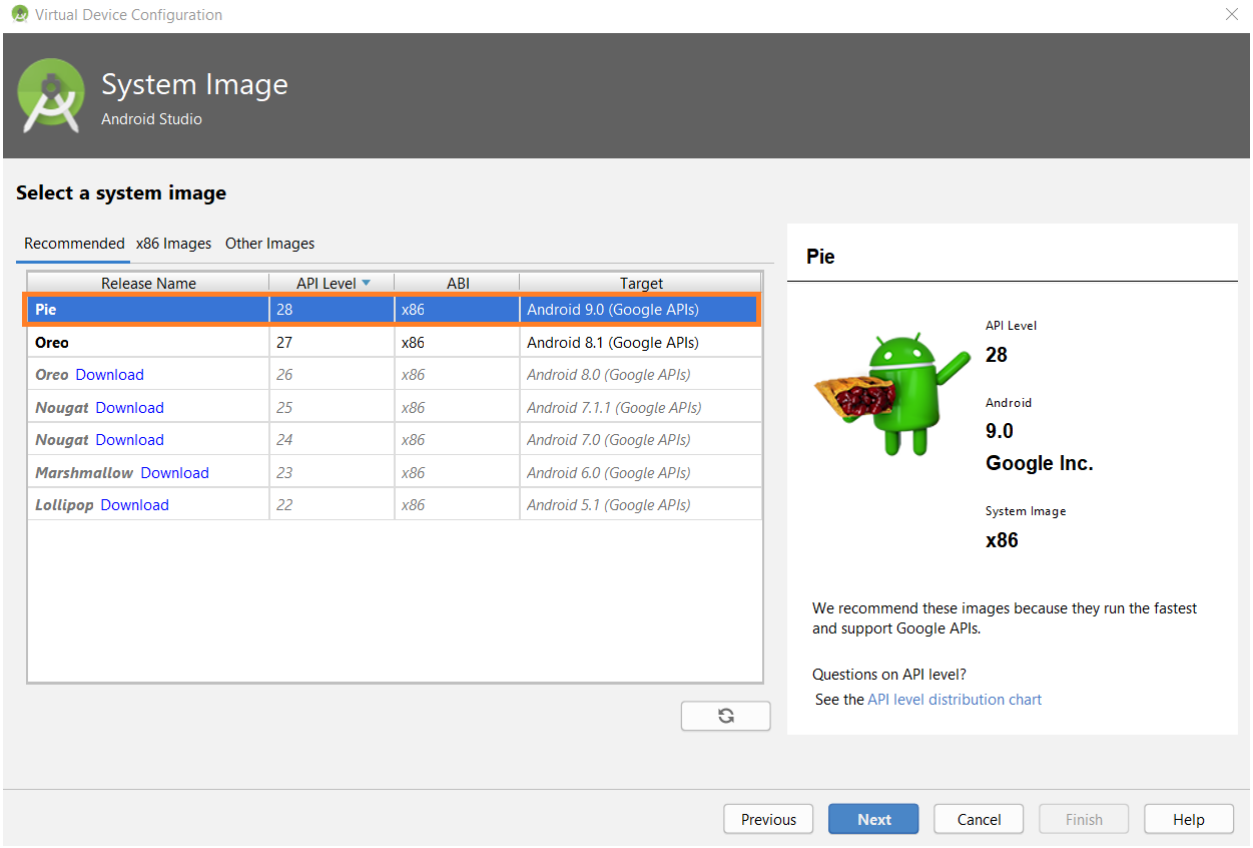


Figure 1-k: Creating a virtual device (step 3)

I'm going with the first recommended option from the available list; however, you can choose any other. It's important to choose an image that plays well with your computer's host operating system. In essence, for emulator performance reasons, it's not recommended to choose an ARM-based image if your computer's host operating system is based on a x86 architecture.

If you've chosen a different image than the one highlighted in Figure 1-k you might have to download the image, using the **Download** link next to the image Release Name field.

Once the image has been selected (and downloaded, if applicable), click **Next** to continue to the last step.

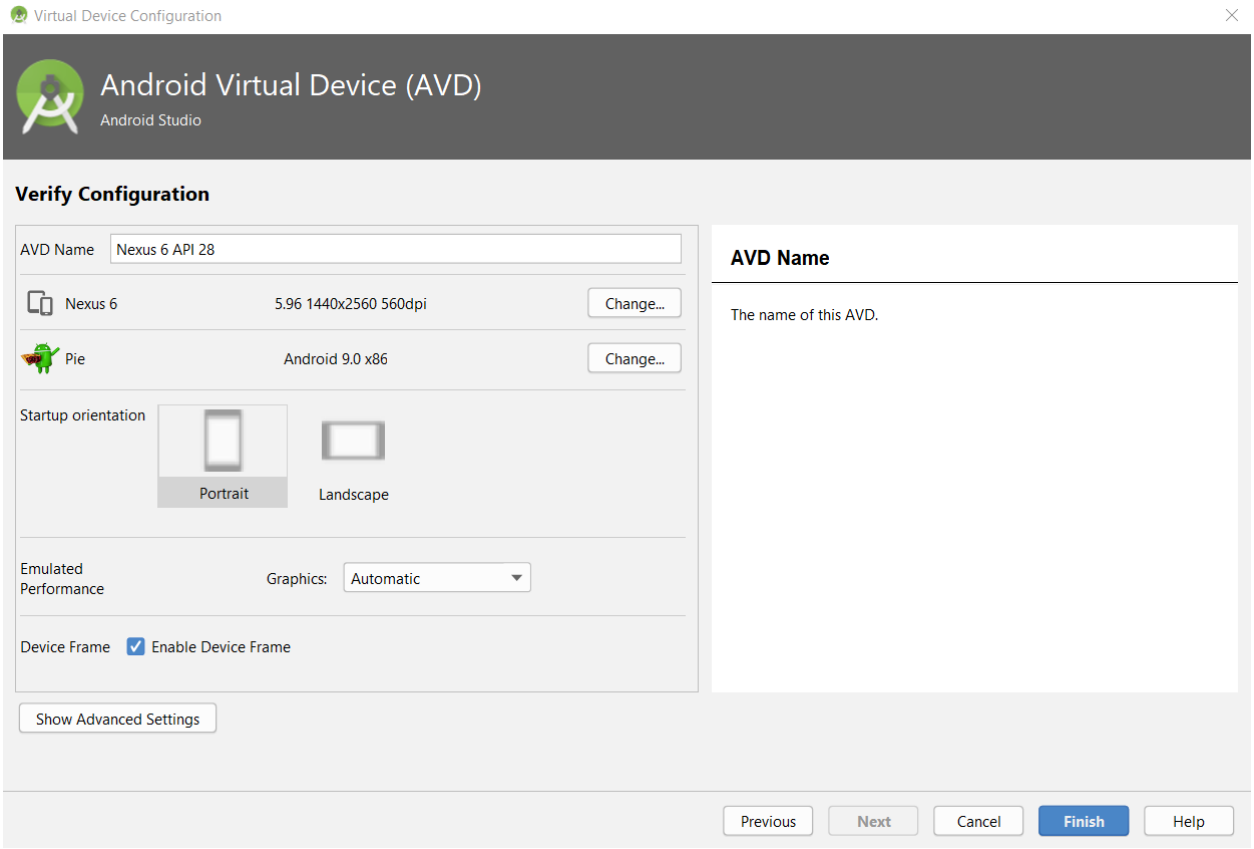


Figure 1-l: Creating a virtual device (step 4)

The last step contains the configuration details for the virtual device image, which you can normally leave to the default options, and then simply click **Finish**.

Awesome—you now have created a virtual device. You can create more than one if you wish, as it might help you test your application with multiple devices. In my case, I've also created another virtual device for a Pixel 2 XL phone API 28, which looks as follows.

www.dbooks.org

Figure 1-m: Virtual device—Android emulator

Testing our setup

With our virtual device in place, it's now time to run the application we have created and see what it does.

To do that, select the **Open Android Emulator** option from the **Android SDK built for x86** drop-down list, which is next to the run button. Once the Android emulator is opened, you'll be able to execute the application when you click run.

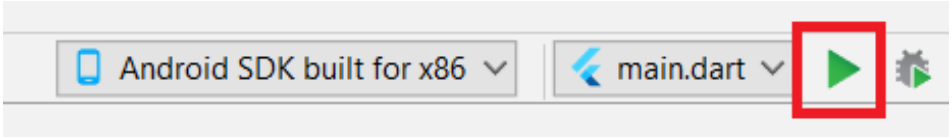


Figure 1-n: The emulator drop-down and run button

Try to run the application to see what happens—in my case, I get the following Android Studio console output.

Code Listing 1-b: Console output when running the application

```
Launching lib\main.dart on Android SDK built for x86 in debug mode...
Initializing gradle...
Resolving dependencies...

* Error running Gradle:
ProcessException: Process
"C:\Projects\test\flutter_app\android\gradlew.bat" exited abnormally:

> Configure project :app
Checking the license for package Android SDK Build-Tools 28.0.3 in
C:\Users\EdFreitas\AppData\Local\Android\sdk\licenses
Warning: License for package Android SDK Build-Tools 28.0.3 not accepted.



FAILURE: Build failed with an exception.

* Where:
Build file 'C:\Projects\test\flutter_app\android\build.gradle' line: 24

* What went wrong:
A problem occurred evaluating root project 'android'.
> A problem occurred configuring project ':app'.
   > Failed to install the following Android SDK packages as some
licences have not been accepted.
        build-tools;28.0.3 Android SDK Build-Tools 28.0.3
     To build this project, accept the SDK license agreements and install
the missing components using the Android Studio SDK Manager.
     Alternatively, to transfer the license agreements from one
workstation to another, see http://d.android.com/r/studio-ui/export-
licenses.html

     Using Android SDK: C:\Users\EdFreitas\AppData\Local\Android\sdk

* Try:
```


www.dbooks.org

```
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output. Run with --scan to get full insights.


* Get more help at https://help.gradle.org


BUILD FAILED in 1s
  Command: C:\Projects\test\flutter_app\android\gradlew.bat app:properties


Finished with error: Please review your Gradle project setup in the android/ folder.
```

If you didn't get this console output message after running your application, awesome—you may skip the rest of this section and go directly to the "Hot reloading" section.

If you did, then by carefully reviewing this output information, we find a reference to this **`http://d.android.com/r/studio-ui/export-licenses.html`** URL that redirects to here. This article explains how Gradle can automatically download packages that might be missing, and that are required when running an application.

We can also see that the last line of the message indicates to review the Gradle project setup in the Android project folder—which refers to the Build.gradle file found within the Android folder of our application. We can see this in the following screenshot.

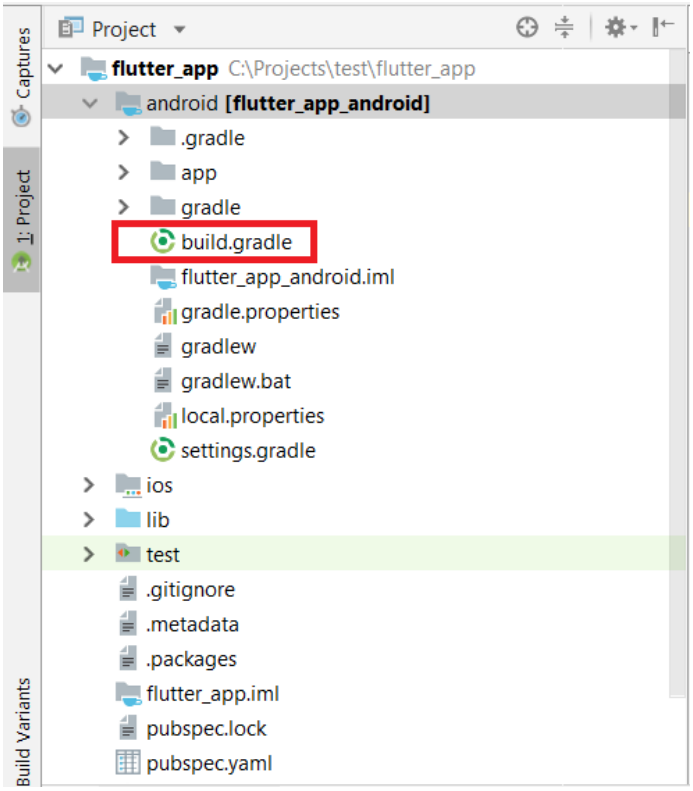


Figure 1-o: The project structure

Let's open the Build.gradle file and inspect its content. Notice in my case how it depends on **`com.android.tools.build:gradle:3.2.1`**—we can see this in the screenshot that follows.

main.dart × flutter_app_android ×

Flutter commands — Open for Editing in Android Studio

```
1  buildscript {
2      repositories {
3          google()
4          jcenter()
5      }
6
7      dependencies {
8          classpath 'com.android.tools.build:gradle:3.2.1'
9      }
10 }
11
12 allprojects {
13     repositories {
14         google()
15         jcenter()
16     }
17 }
```

Figure 1-p: The Build.gradle file contents

www.dbooks.org

The previous console output gives us a hint of what the problem might be: **`License for package Android SDK Build-Tools 28.0.3 not accepted`**.

What this is telling us is that it wasn't able to run the application with this **`com.android.tools.build:gradle:3.2.1`** dependency, because I have no license for it.

In other words, that specific **`com.android.tools.build:gradle`** version probably didn't get installed when I went through the Android Studio setup process.

In that case, the solution is to use a version of the **`com.android.tools.build:gradle`** dependency that was installed during the Android Studio setup process—which can be one version lower than the one mentioned on the Build.gradle file.

To resolve the problem, all I need to do is change that line on the Build.gradle file from **`com.android.tools.build:gradle:3.2.1`** to **`com.android.tools.build:gradle:3.2.0`**.

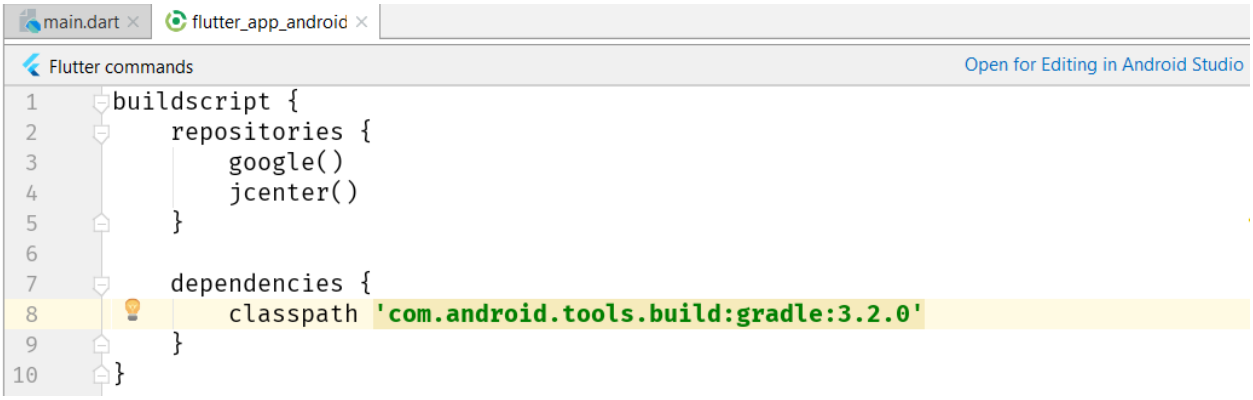


Figure 1-q: The Build.gradle file contents edited

After saving the change to the Build.gradle file, if I now click **Run**, I'll get the following build console output within Android Studio.

Code Listing 1-c: Console output when running the application (after updating Build.gradle)

```
Launching lib\main.dart on Android SDK built for x86 in debug mode...
Initializing gradle...
Resolving dependencies...
Gradle task 'assembleDebug'...
Built build\app\outputs\apk\debug\app-debug.apk.
```

Awesome—that's so much better! The application has been built, and it is running. We can see this on the Android emulator as follows.

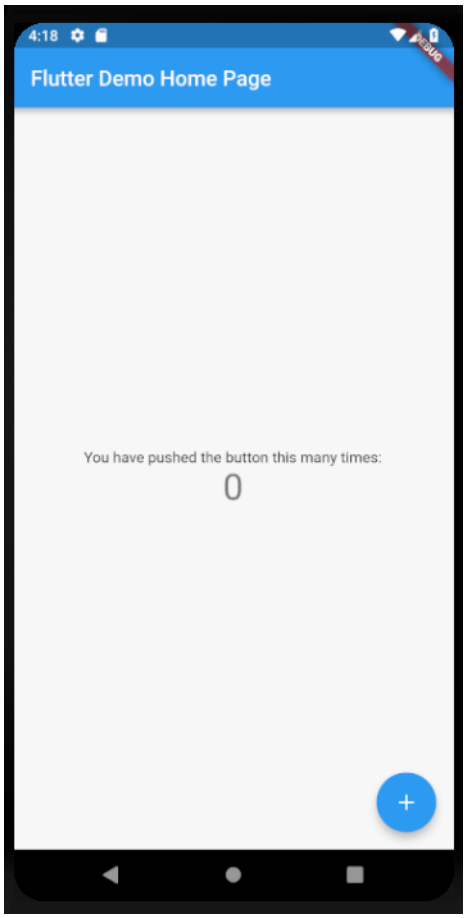


Figure 1-r: The demo app running

With the building issues sorted, let's now explore how Flutter's "hot reload" mechanism works, and what it does.

Hot reloading

Hot reloading is one of the coolest features of Flutter, and basically means that if a change to the code is made while the application is running, that change is almost immediately reflected within the running application.

Let's give hot reloading a try. With the application running, let's go to the main.dart file under the lib folder of our demo application, and locate the following code.

Code Listing 1-d: Snippet of code of main.dart

```
Text(
    'You have pushed the button this many times:',
)
```


www.dbooks.org

Let's replace the word **pushed** with the word **clicked**. The code should now look as follows.

Code Listing 1-e: Snippet of code of main.dart

```
Text(
    'You have clicked the button this many times:',
)
```

Figure 1-s shows the source code and the application running before the change.

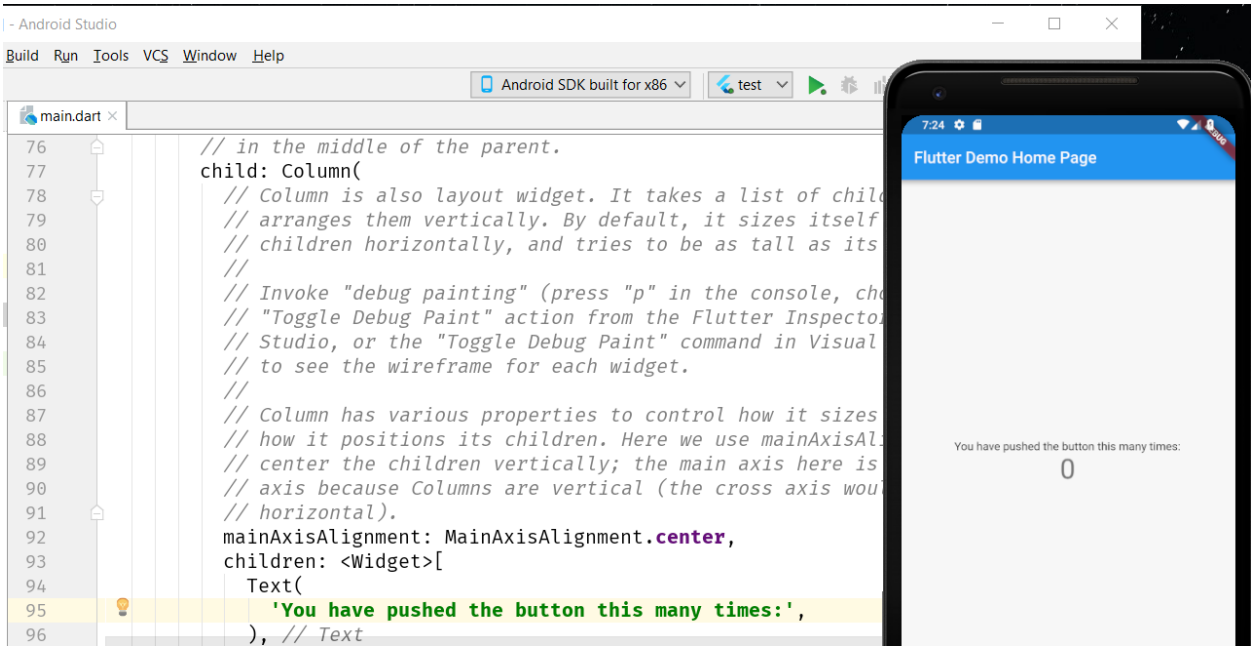


Figure 1-s: App running (before the change)

Figure 1-t shows the source code and the application running after the change.

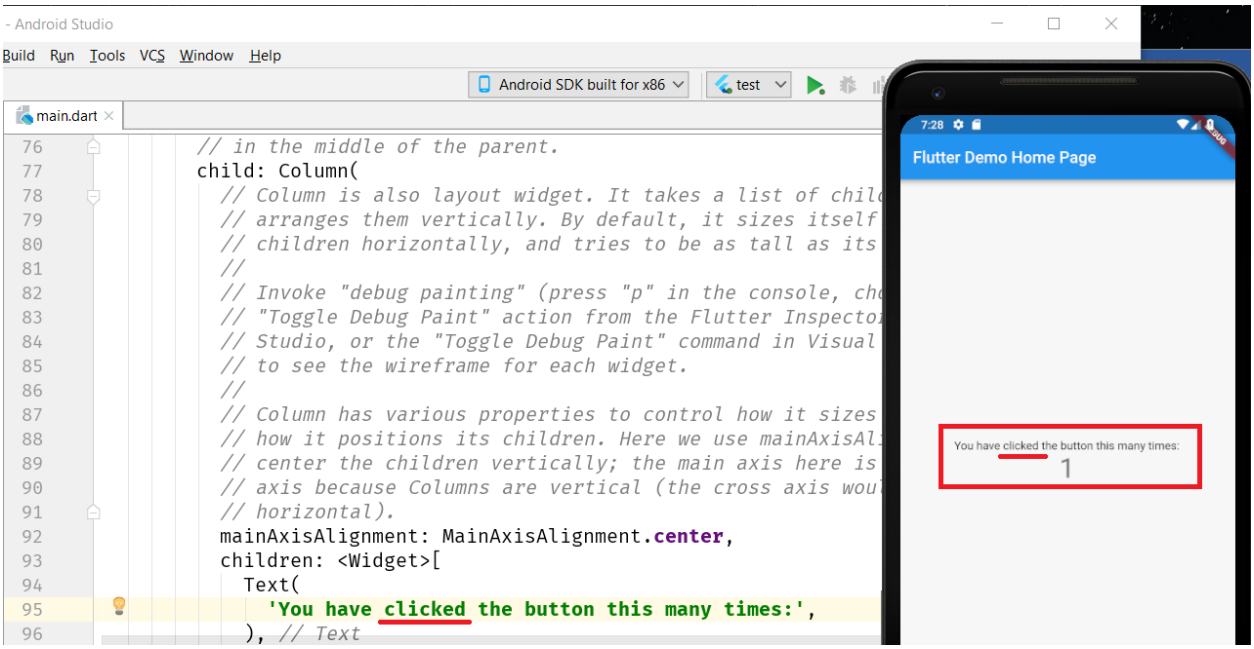


Figure 1-t: App running (after the change)

As you can clearly see, hot reloading worked. Something to notice about hot reloading is that the changes take a few seconds to propagate and become effective, as the application is basically redeployed to the emulator or device when changes take place.

Web developers will notice that hot reloading in Flutter is slower than when developing web applications. This is because during web development, hot reloading only applies to the HTML, CSS, or JavaScript being modified, whereas with Flutter, the actual application runtime needs to be synced to the device.

Nevertheless, Flutter's hot-reloading mechanism is impressive and very useful, as it helps us avoid having to stop the app and restart it.

Summary

The goal of this chapter was to set up Flutter and get started—that's exactly what we managed to achieve.

Next, we'll dive straight into the code and start writing the fundamental pillars of our app. It's going to be a lot of fun, as we'll look at how to design our app's UI and implement its essential logic.

www.dbooks.org

Chapter 2 App Fundamentals

Quick intro

To get a sense of what we will be building throughout this book, let's have a look at how the main screen of our finished application will look—we can see this as follows.

Figure 2-a: The finished app

We can see that the app contains a list of documents, each with the remaining days before they expire and their expiration date.

The app's UI is based on Google's Material Design library, which comes out of the box with Flutter.

Note: You can find all the finished Dart source code files and the Pubspec.yaml file for this app in the appendix at the end of the book.

Rewriting—main.dart

With the setup phase behind us, it's now time to start building the foundations of our application.

To do that, go to the main.dart file found under the lib folder, and remove all the existing code it contains. With all the out-of-the-box code removed from the main.dart file, let's add the following code.

Code Listing 2-a: The new, finished main.dart code

```
import 'package:flutter/material.dart';
import './ui/doclist.dart';

void main() => runApp(DocExpiryApp());

class DocExpiryApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'DocExpire',
      theme: new ThemeData(
        primarySwatch: Colors.indigo,
      ),
      home: DocList(),
    );
  }
}
```

What have we done here? Let's dissect this code into smaller pieces to understand it better.

Code Listing 2-b: The new, finished main.dart code (part 1)

```
import 'package:flutter/material.dart';
import './ui/doclist.dart';
```

The first line imports the Material Design library, which we will use to create our app's widgets (UI components).

The second line references a file that we have not created yet within our application, which will contain the logic for creating and displaying the list of documents, showing whether they have expired or not.

Next, we invoke the **`main`** method, which is the app's main entry point. This method invokes **`runApp`**, to which a new instance of the **`DocExpiryApp`** class is passed.

www.dbooks.org

Code Listing 2-c: The new, finished main.dart code (part 2)

```
void main() => runApp(DocExpiryApp());
```

Following that, we declare the **DocExpiryApp** class, which inherits from the **StatelessWidget** class. This is one of two types of widget classes that Flutter supports.

Code Listing 2-d: The finished DocExpiryApp code

```
class DocExpiryApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'DocExpire',
      theme: new ThemeData(
        primarySwatch: Colors.indigo,
      ),
      home: DocList(),
    );
  }
}
```

A **StatelessWidget** describes a widget that does not require mutable state—in other words, it is a widget without a state. You can find more details in the official documentation.

Basically, every UI component in Flutter is a widget, either with state, **StatefulWidget**, or without state, **StatelessWidget**.

Within the **DocExpiryApp** class, we are overriding the **Build** method inherited from the **StatelessWidget** class, which is responsible for returning a **Widget** object to its caller, and thus building the UI element.

So, in essence, the **Build** method creates a **MaterialApp** widget, the app's main component. The **home** property is assigned to the result that will be returned from the **DocList** method. The **DocList** method is imported from *doclist.dart* (yet to be created), which will return the list of documents to be displayed.

To better understand the relationship between what you see in the code and what is displayed on the device's screen when the app runs, let's have a look at the following diagram.

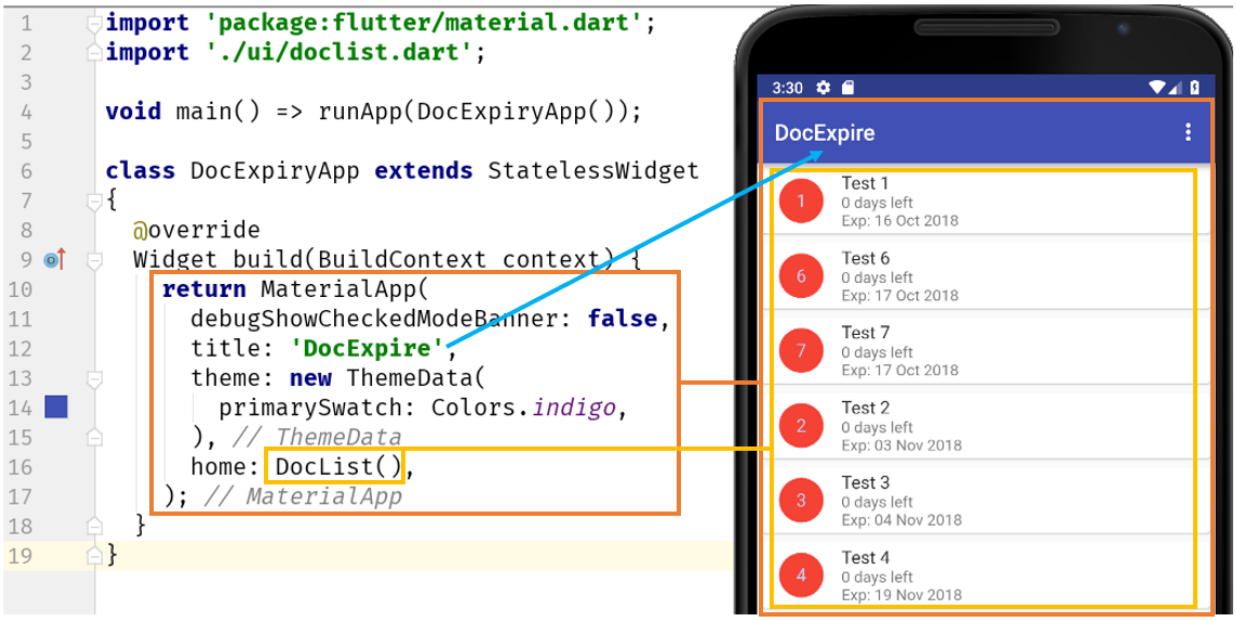


Figure 2-b: Relationship between the main.dart code and the app's main UI

The **`MaterialApp`** widget has properties that determine some of its functionality. The **`title`** property needs no explanation, but the **`debugShowCheckedModeBanner`** and **`theme`** properties do.

As its name indicates, the **`debugShowCheckedModeBanner`** property is used for displaying a smaller banner that indicates that the app is running in debug mode when the property is set to **`true`**.

This is how the app's main screen would look with the **`debugShowCheckedModeBanner`** property set to **`true`**. Notice the small Debug banner on the top, right-hand side of the screen.

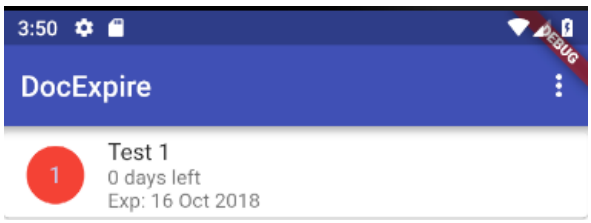


Figure 2-c: The app's main UI with debugShowCheckedModeBanner set to true

The **`theme`** property basically represents the main color used for the app's UI, which in this case is **`indigo`**. More information about theming in Flutter can be found in the official documentation.

If you're tempted to run the app after having made these changes to main.dart, you'll get some compilation errors. This is because we haven't yet created Doclist.dart—but we are referencing it in the code.

www.dbooks.org

Also, we have assigned to the **`home`** property the value returned by the **`DocList`** method—also part of Doclist.dart.

So, if you would like to run the app at this stage and check how it looks, you'll have to wait until we have written some more code, specifically within the doclist.dart file.

Project structure

With the main.dart code rewritten, let's explore how the current project structure looks, what other files and folders we would need to create, and what the finished application project structure will look like.

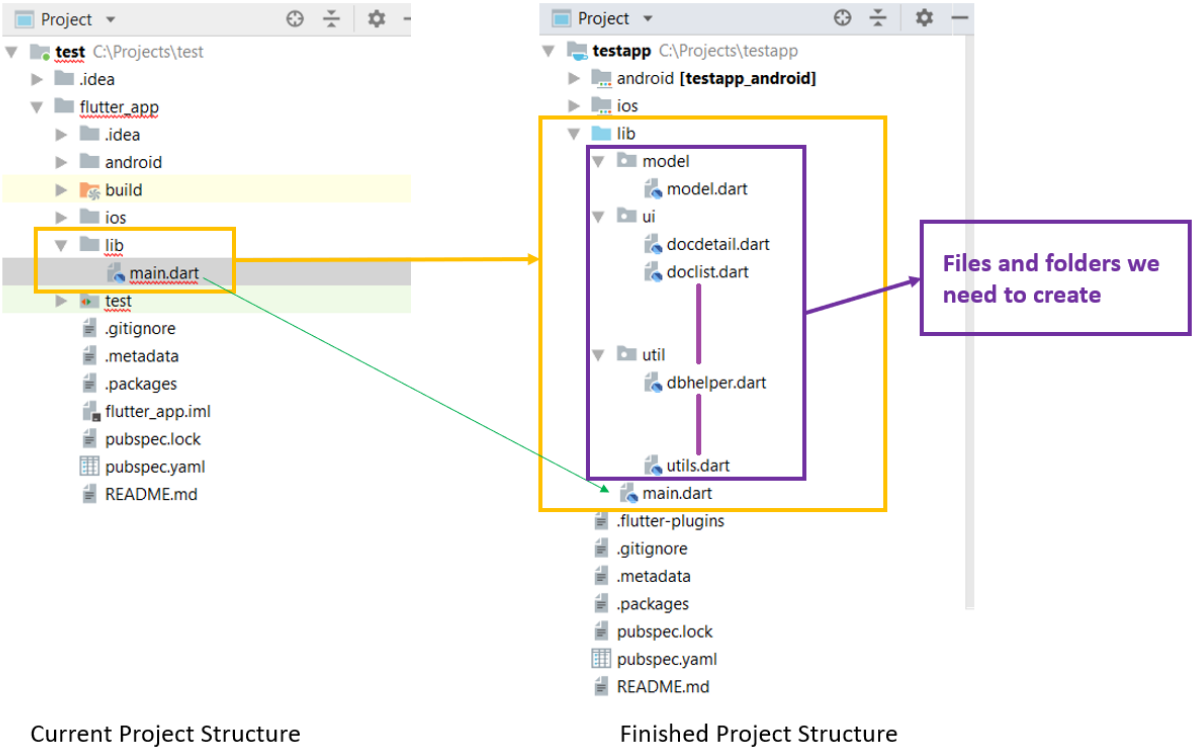


Figure 2-d: The app's project folder structure comparison

As we can see in the preceding diagram, the main differences between the current project folder structure and the finished one are the subfolders and files found under the project's **lib** folder, which we will create throughout this book.

The lib folder is where all our app's source code is going to be organized. To make the code organization easy, we'll have three subfolders:

- The **model** folder, which will be used to keep the object model that will be used for storing information on the SQLite database.

- The **ui** folder, which is where we will have all the logic related to the app's UI and navigation.
- The **util** folder, which will contain utility, general purpose code, and database helper classes and methods.

By using this structure, we can keep our source code organized. Notice that this is not the only possible project folder structure; you might want to name your folders differently or organize them in another way. Flutter doesn't impose a specific way of organizing source files—the way you organize your code is entirely up to you.

However, I suggest you keep this structure, as it will make it easier to follow along with the various stages of the development of the application.

Bottom-to-top coding approach

For our application to be fully functional, we need to be able to display the list of documents. To be able to do that, we need a database model, database helper functions, a detail page that will be used for each document, and some generic utility functions.

All this code needs to be written before we can attempt to display the list of documents. This means that doclist.dart will be the last module we will write for our application.

The first module we will write is going to be called utils.dart, which will contain generic and form validation functions that we will need throughout the application.

We'll be coding using a *bottom-to-top* approach, where we write the basic building blocks and then build up based on that. This approach will look as follows.

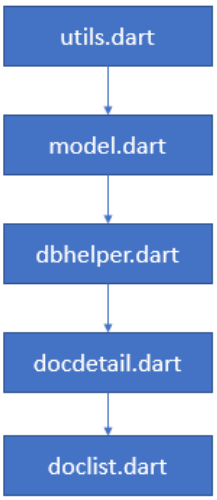


Figure 2-e: Bottom-to-top coding approach

Let's move on, starting with the utils.dart file.

www.dbooks.org

Writing utils.dart

To be able to properly validate new documents that we will enter using our application, we need to have some general validation routines. This is the main purpose of utils.dart.

To get a visual understanding of what this means, have a look at the following screenshot that represents the finished UI for adding a new document.

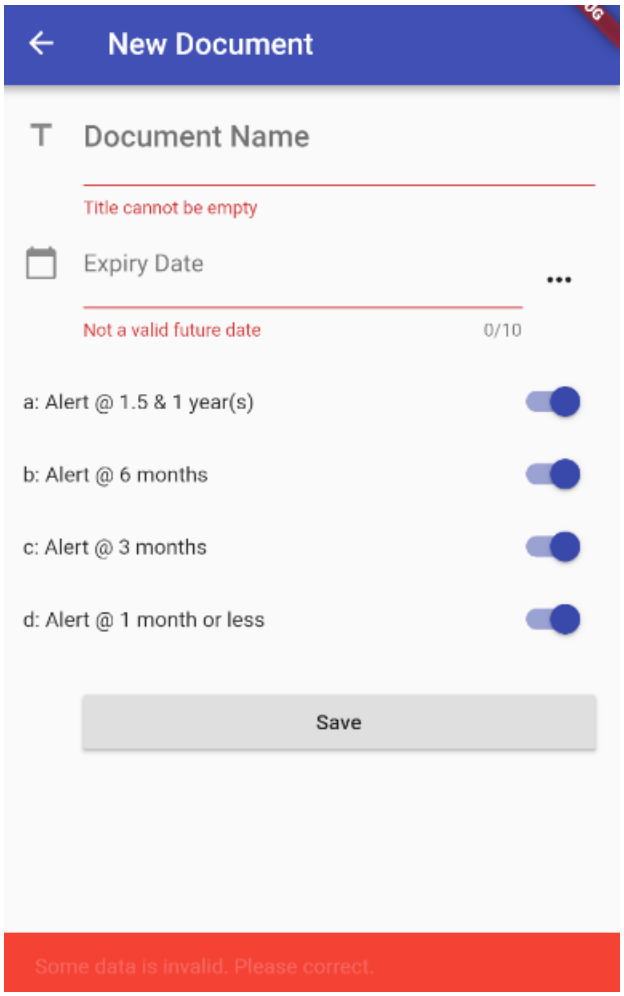


Figure 2-f: App screen to enter a new document

We can immediately see that there are two validations: one for checking that the Document Name field is not empty, and another that checks that the Expiry Date field is a valid future date. This is the logic we'll add to the utils.dart file—let's go ahead and do that.

To organize our code properly, create a subfolder under the lib folder called **util**—this is where we will create the utils.dart file.

You can do this by right-clicking on the **lib** folder within Android Studio, and then choosing **New** > **Directory**.

Next, select the **lib** folder, right-click, and choose **New** > **File**.

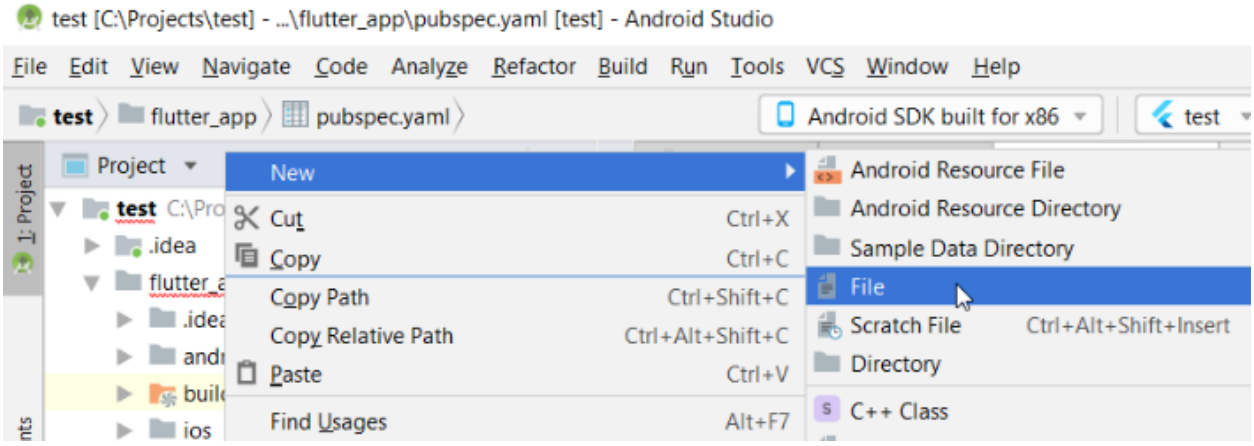


Figure 2-g: New file option

When prompted, enter the name of the file: **utils.dart**.

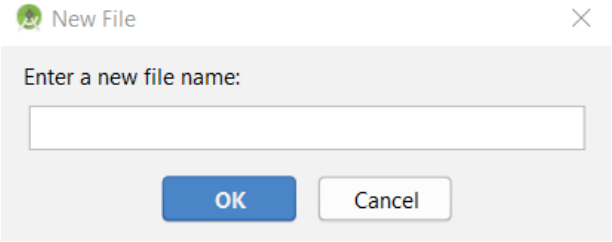


Figure 2-h: New file window

With the utils.dart file created under the lib\util folder path, let's add the following code to it.

Code Listing 2-e: Full utils.dart code

```
import 'package:intl/intl.dart';

class Val {
  // Validations
  static String ValidateTitle(String val) {
    return (val != null && val != "") ? null : "Title cannot be empty";
  }

  static String GetExpiryStr(String expires) {
    var e = DateUtils.convertToDate(expires);
    var td = new DateTime.now();

    Duration dif = e.difference(td);
    int dd = dif.inDays + 1;
    return (dd > 0) ? dd.toString() : "0";
  }
```


www.dbooks.org

```
  static bool StrToBool(String str) {
    return (int.parse(str) > 0) ? true : false;
  }

  static bool IntToBool(int val) {
    return (val > 0) ? true : false;
  }

  static String BoolToStr(bool val) {
    return (val == true) ? "1" : "0";
  }

  static int BoolToInt(bool val) {
    return (val == true) ? 1 : 0;
  }
}

class DateUtils {
  static DateTime convertToDate(String input) {
    try
    {
      var d = new DateFormat("yyyy-MM-dd").parseStrict(input);
      return d;
    } catch (e) {
      return null;
    }
  }

  static String convertToDateFull(String input) {
    try
    {
      var d = new DateFormat("yyyy-MM-dd").parseStrict(input);
      var formatter = new DateFormat('dd MMM yyyy');
      return formatter.format(d);
    } catch (e) {
      return null;
    }
  }

  static String convertToDateFullDt(DateTime input) {
    try
    {
      var formatter = new DateFormat('dd MMM yyyy');
      return formatter.format(input);
    } catch (e) {
      return null;
    }
  }
```

```
  static bool isDate(String dt) {
    try
    {
      var d = new DateFormat("yyyy-MM-dd").parseStrict(dt);
      return true;
    } catch (e) {
      return false;
    }
  }

  static bool isValidDate(String dt) {
    if (dt.isEmpty || !dt.contains("-") || dt.length < 10) return false;

    List<String> dtItems = dt.split("-");
    var d = DateTime(int.parse(dtItems[0]),
        int.parse(dtItems[1]), int.parse(dtItems[2]));

    return d != null && isDate(dt) &&
        d.isAfter(new DateTime.now());
  }

  // String functions
  static String daysAheadAsStr(int daysAhead) {
    var now = new DateTime.now();
    DateTime ft = now.add(new Duration(days: daysAhead));
    return ftDateAsStr(ft);
  }

  static String ftDateAsStr(DateTime ft) {
    return ft.year.toString() + "-" +
        ft.month.toString().padLeft(2, "0") + "-" +
        ft.day.toString().padLeft(2, "0");
  }

  static String TrimDate(String dt) {
    if (dt.contains(" ")) {
      List<String> p = dt.split(" ");
      return p[0];
    }
    else
      return dt;
  }
}
```

What is going on here? As you might have noticed, the code contains two classes: **`Val`** and **`DateUtils`**. There's quite a bit of code within the utils.dart file; it's not incredibly important right

now to understand each line of code within it, but rather, to understand how the main parts relate with each other.

Let's explore the relationship between both classes and the document entry application screen we looked at before.

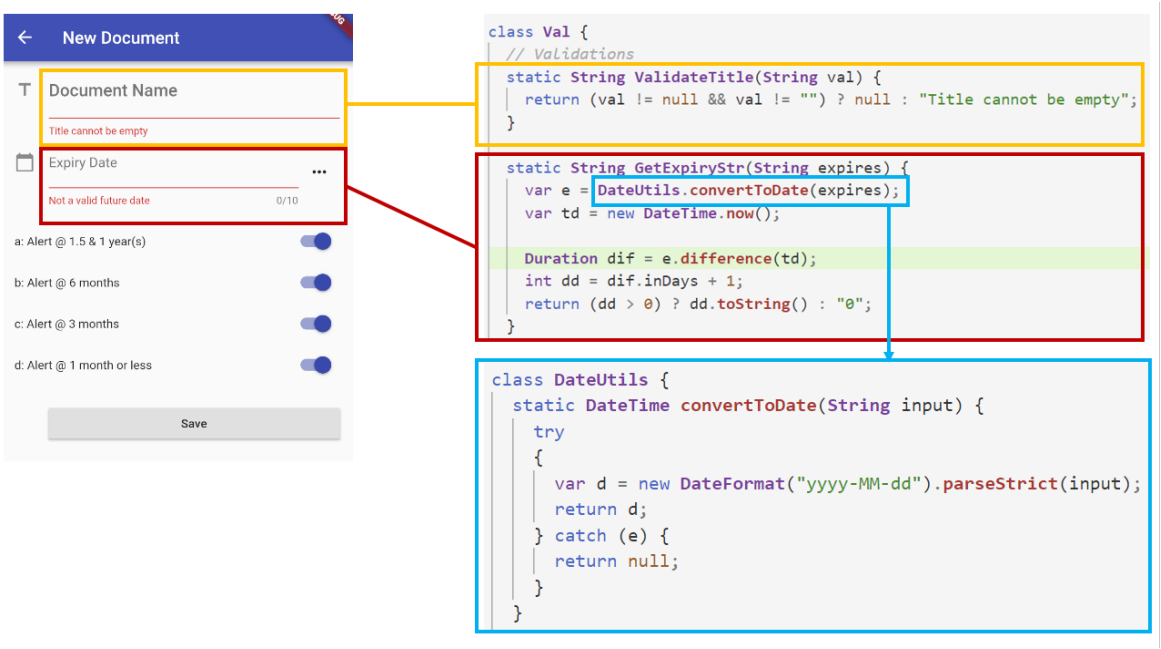


Figure 2-i: Interconnection between the document entry screen and utils.dart code

As you can see in Figure 2-i, the relationship among the parts and the application is very clear and easy to understand. Most of the methods within the **`Val`** and **`DateUtils`** classes are methods that we will need later as we progressively build our application.

Right now, let's quickly go over the ones that are essential and that directly relate to the document entry screen. Let's first have a look at the **`ValidateTitle`** method.

Code Listing 2-f: ValidateTitle method (utils.dart)

```
static String ValidateTitle(String val) {
  return (val != null && val != "") ? null : "Title cannot be empty";
}
```

As we can see, this method is very simple—it basically evaluates, through a ternary conditional expression, if the value of the parameter **`val`**—which represents the name of the document entered through the document entry screen—is an empty **`String`** object.

If it is not, then **`null`** is returned, which means that the validation has passed. If it is an empty **`String`**, then a message is returned, indicating that the string cannot be empty—this message is the one displayed in the document entry screen.

Now, let's explore the **GetExpiryStr** method and what it does. We can see this in the listing that follows.

Code Listing 2-g: GetExpiryStr method (utils.dart)

```
static String GetExpiryStr(String expires) {
  var e = DateUtils.convertToDate(expires);
  var td = new DateTime.now();

  Duration dif = e.difference(td);
  int dd = dif.inDays + 1;
  return (dd > 0) ? dd.toString() : "0";
}
```

This method is also quite simple. The first instruction converts the document's expiry date, represented by the **expires** variable, into a **DateTime** object by invoking the **convertToDate** method from the **DateUtils** class.

What follows is that the current **DateTime** is obtained by calling the **now** method. Then, the difference between the document's expiry date and the current date is calculated by calling the **difference** method.

If the result of **difference** is positive, then that value is converted to a **String** by calling the **toString** method, and then returned. If the value is negative or zero, then **"0"** is returned as a **String** object.

Now, let's explore the **convertToDate** method from the **DateUtils** class—we can see this in the listing that follows.

Code Listing 2-h: convertToDate method (utils.dart)

```
static DateTime convertToDate(String input) {
  try {
    var d = new DateFormat("yyyy-MM-dd").parseStrict(input);
    return d;
  } catch (e) {
    return null;
  }
}
```

This method is also very simple—it basically takes the date as an **input** variable and attempts to parse it using the format **"yyyy-MM-dd"** by calling the **parseStrict** method.

If successful, then the parsed value **d** is returned; otherwise, **null** is returned.

Awesome—we are almost done covering all we need to know about the utils.dart code. There's just one small piece missing: the first line of utils.dart.

www.dbooks.org

Code Listing 2-i: Import statement (utils.dart)

```
import 'package:intl/intl.dart';
```

This line basically tells Flutter that utils.dart needs to import a package called **`intl.dart`**—which is a Dart library used for supporting internationalization and localization capabilities.

The **`DateFormat`** method that is invoked from the **`convertToDate`** method is part of the **`intl.dart`** library.

However, there's still one thing missing—we need to import this package into our Flutter project. We can do this by opening the Pubspec.yaml file found within our main project folder, and then installing the package as follows.

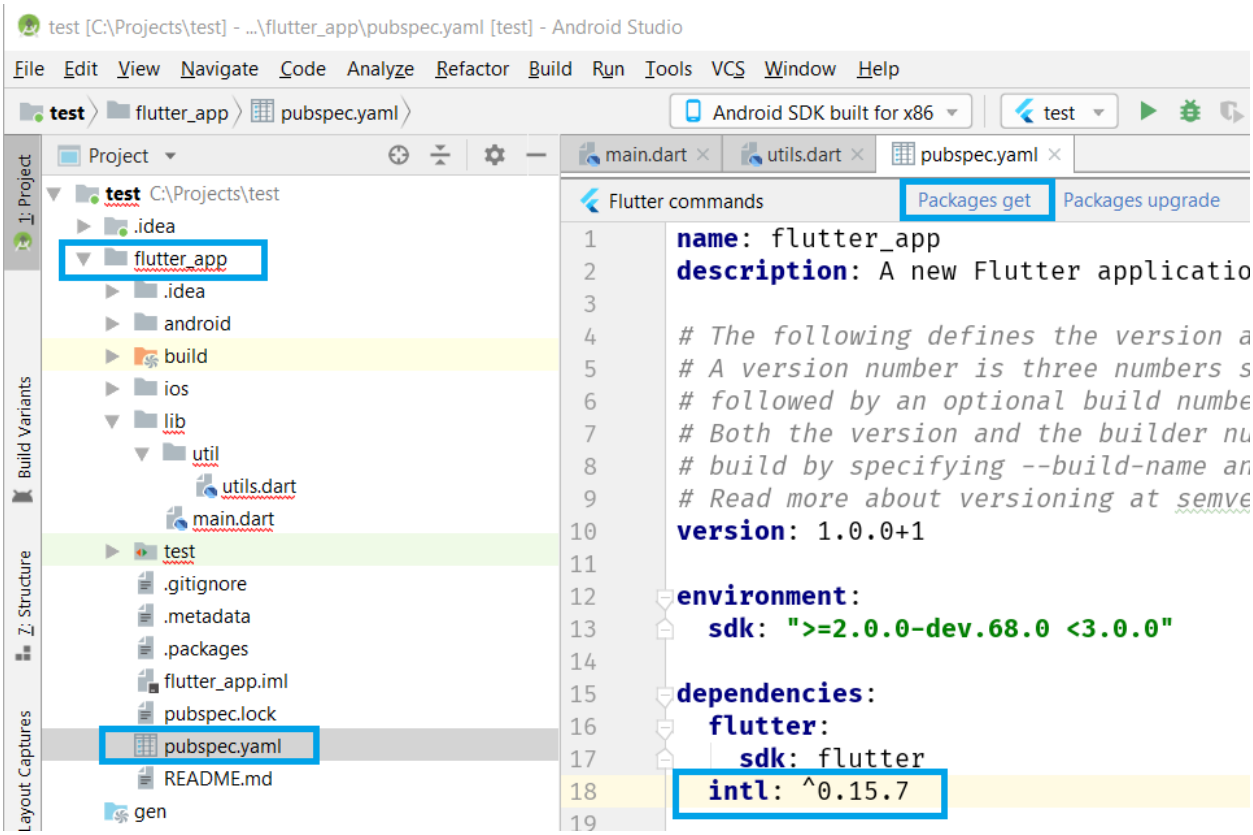


Figure 2-j: Adding the package to Pubspec.yaml

Once the package has been added to the Pubspec.yaml file, we'll need to run the **`flutter packages get`** command, which can be done within Android Studio directly with one click—this is highlighted in Figure 2-j.

Once you've clicked the **Packages get** option, you should see the following output in the Android Studio messages window.

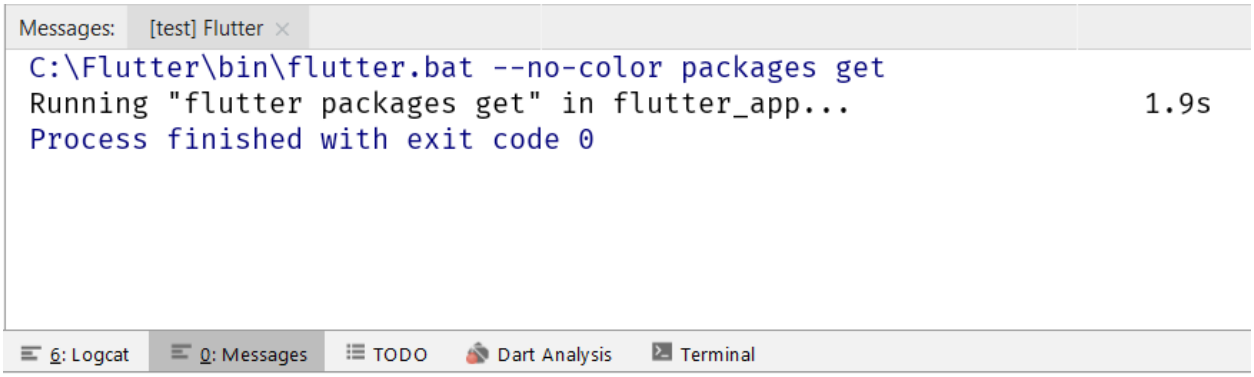


Figure 2-k: Message output in Android Studio after adding a package

Our utils.dart code can now use the **`DateFormat`** method, and we are done with utils.dart.

Writing model.dart

With utils.dart covered, let's move on and create the model.dart file, which represents the data model that will be used for storing and retrieving document data.

To keep things organized, let's create a *model* subfolder under the lib folder of our application. Inside this model subfolder, let's create the model.dart file—we can do this the same way we created the util subfolder and utils.dart with Android Studio.

With the model.dart file created, let's add the following code to it.

Code Listing 2-j: Full model.dart code

```
import '../util/utils.dart';

class Doc
{
  int id;
  String title;
  String expiration;

  int fqYear;
  int fqHalfYear;
  int fqQuarter;
  int fqMonth;

  Doc(this.title, this.expiration, this.fqYear,
      this.fqHalfYear, this.fqQuarter, this.fqMonth);

  Doc.withId(this.id, this.title, this.expiration, this.fqYear,
      this.fqHalfYear, this.fqQuarter, this.fqMonth);
```


www.dbooks.org

```
  Map<String, dynamic> toMap() {
    var map = Map<String, dynamic>();

    map["title"] = this.title;
    map["expiration"] = this.expiration;

    map["fqYear"] = this.fqYear;
    map["fqHalfYear"] = this.fqHalfYear;
    map["fqQuarter"] = this.fqQuarter;
    map["fqMonth"] = this.fqMonth;

    if (id != null) {
      map["id"] = id;
    }

    return map;
  }

  Doc.fromOject(dynamic o) {
    this.id = o["id"];
    this.title = o["title"];
    this.expiration = DateUtils.TrimDate(o["expiration"]);

    this.fqYear = o["fqYear"];
    this.fqHalfYear = o["fqHalfYear"];
    this.fqQuarter = o["fqQuarter"];
    this.fqMonth = o["fqMonth"];
  }
}
```

Let's break this into smaller parts so we can understand what this code does.

First, we import a reference to the utils.dart module we previously created—this is because the **`fromObject`** method invokes the **`TrimDate`** method from the **`DateUtils`** class.

Code Listing 2-k: Importing utils.dart in model.dart

```
import '../util/utils.dart';
```

Then, we have a **`Doc`** class, which in our data model represents the document that will be written and read to the SQLite embedded database we will be using. Let's first explore the properties of the **`Doc`** class.

Code Listing 2-l: Doc class properties

```
// Previous code…
```

```
class Doc
{
  int id;
  String title;
  String expiration;

  int fqYear;
  int fqHalfYear;
  int fqQuarter;
  int fqMonth;

  // The rest of the Doc class code
}
```

To better understand how these properties relate to a document database record, which is going to be stored within an SQLite table, let's look at the following diagram.

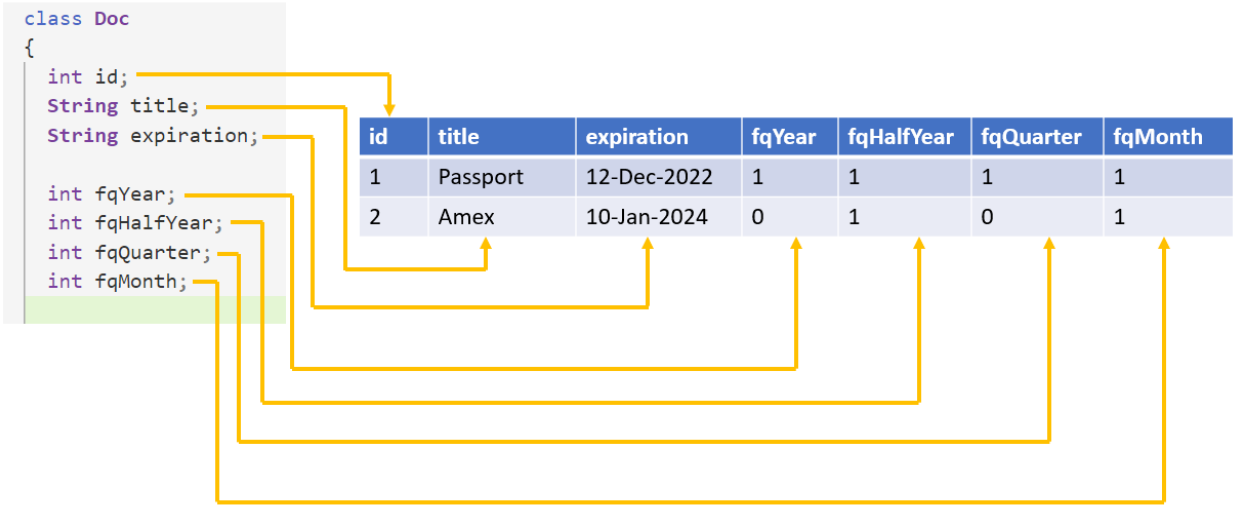


Figure 2-l: Data model fields

We can clearly establish the relationship between each of the **Doc** class properties and each of the columns as they will be stored in the SQLite database.

Our **Doc** class is the object representation of a document record stored in the database.

Moving on, we can see that our class has two constructors—one that will be invoked when creating a new document, and the other for existing documents in the database. This is shown in the following code.

Code Listing 2-m: Doc class constructors

```
// Previous code…
```


www.dbooks.org

```
class Doc
{
  // Doc class properties.

  // Constructor used if we don't want to assign an id immediately.
  Doc(this.title, this.expiration, this.fqYear,
      this.fqHalfYear, this.fqQuarter, this.fqMonth);
  // Constructor used if we want to assign an id immediately
  Doc.withId(this.id, this.title, this.expiration, this.fqYear,
      this.fqHalfYear, this.fqQuarter, this.fqMonth);

  // Rest of the Doc class code.
}
```

The first constructor (**Doc**) is used when we want to create an instance of the **Doc** class and we don't want to assign a value to the **id** property.

The second constructor (**Doc.withId**) is used when we want to create an instance of the **Doc** class and we want to assign a value to the **id** property right away.

When we create a new document or access an existing one, we'll need to invoke one of these constructors to create an instance of the **Doc** class—which is how we represent a document within our application.

Now, let's explore the rest of the **Doc** class code.

Code Listing 2-n: Rest of the Doc class code

```
// Previous code…

class Doc
{
  // All the previous code

  Map<String, dynamic> toMap() {
    var map = Map<String, dynamic>();

    map["title"] = this.title;
    map["expiration"] = this.expiration;

    map["fqYear"] = this.fqYear;
    map["fqHalfYear"] = this.fqHalfYear;
    map["fqQuarter"] = this.fqQuarter;
    map["fqMonth"] = this.fqMonth;

    if (id != null) {
      map["id"] = id;
```

```
    }

    return map;
  }

  Doc.fromOject(dynamic o) {
    this.id = o["id"];
    this.title = o["title"];
    this.expiration = DateUtils.TrimDate(o["expiration"]);

    this.fqYear = o["fqYear"];
    this.fqHalfYear = o["fqHalfYear"];
    this.fqQuarter = o["fqQuarter"];
    this.fqMonth = o["fqMonth"];
  }
}
```

The rest of the code consists of two methods—the **toMap** and **fromObject** methods. Let's see what each does. The **toMap** method is used when the document information needs to be written to the database.

Code Listing 2-o: The toMap method

```
Map<String, dynamic> toMap() {
  var map = Map<String, dynamic>();

  map["title"] = this.title;
  map["expiration"] = this.expiration;

  map["fqYear"] = this.fqYear;
  map["fqHalfYear"] = this.fqHalfYear;
  map["fqQuarter"] = this.fqQuarter;
  map["fqMonth"] = this.fqMonth;

  if (id != null) {
    map["id"] = id;
  }

  return map;
}
```

The first instruction of the **toMap** method is used to create an instance of **Map<String, dynamic>**. For further information regarding the usage of **Map** within the Dart programming language, please refer to the official documentation.

Within the **toMap** method, the existing **Doc** instance property values are assigned to their equivalent properties within a **Map** object—which is convenient for writing the data to the

www.dbooks.org

database. The **toMap** method returns a **Map** object back to its invoker—which, as we will see later, is going to be a method that writes to the database.

Let's now have a look at the **fromObject** method, which does the opposite of what the **toMap** method does.

Code Listing 2-p: The fromObject method

```
Doc.fromOject(dynamic o) {
  this.id = o["id"];
  this.title = o["title"];
  this.expiration = DateUtils.TrimDate(o["expiration"]);

  this.fqYear = o["fqYear"];
  this.fqHalfYear = o["fqHalfYear"];
  this.fqQuarter = o["fqQuarter"];
  this.fqMonth = o["fqMonth"];
}
```

Essentially, the **fromObject** method receives a **dynamic** object as a parameter—which is retrieved from the database—and the properties of this object are then assigned to their respective counterpart properties within the **Doc** instance.

To recap: the **toMap** method is used when writing to the database, and the **fromObject** method is used when reading from the database.

Creating the database—dbhelper.dart

With model.dart behind us, let's now focus on an essential part of the application, which is the database access layer and helper functions. Let's create the **dbhelper.dart** file within the util subfolder.

As the code for the dbhelper.dart file is quite extensive, I won't paste it all straight away, but instead we'll take a look at each individual part, one by one.

Code Listing 2-q: The import statements—dbhelper.dart

```
import 'package:flutter/material.dart';
import 'package:sqflite/sqflite.dart';
import 'package:path_provider/path_provider.dart';

import 'dart:async';
import 'dart:io';

import '../model/model.dart';
```

In the **`import`** section, we can see that the first three instructions reference three packages that we haven't seen before.

The **`material.dart`** package is built into Flutter, and was automatically added to the Pubspec.yaml file when the project was created. We can see this in the Pubspec.yaml file as follows.

```
dependencies:
  flutter:
    sdk: flutter
```

Figure 2-m: Built-in material and Flutter package

The following two lines refer to the **`sqflite.dart`** (a Flutter package used to access a SQLite database) and **`path_provider.dart`** packages, which have not been added to the Pubspec.yaml file yet. Let's do this now.

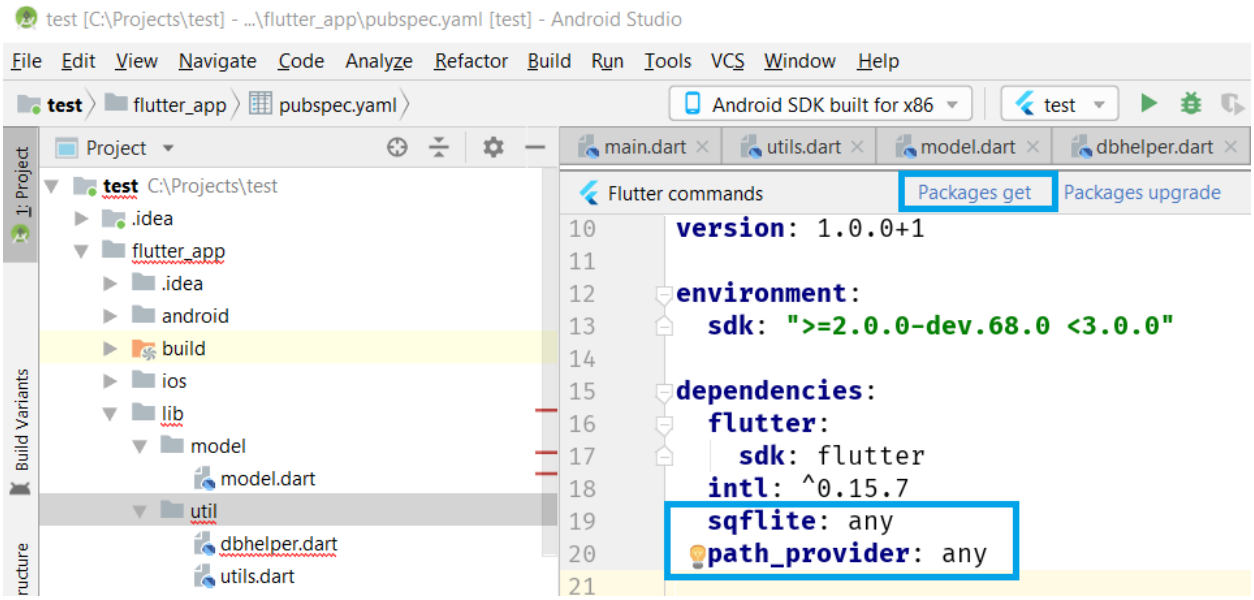


Figure 2-n: Adding the sqflite and path_provider packages

All we've done is add both package names below **intl**, within the Pubspec.yaml file. After doing that, click on the **Packages get** option, so both packages can get installed.

Following that are two **`import`** statements that reference the **`dart:async`** and **`dart:io`** libraries, which will be used for performing async and file operations.

Finally, we can see the **`import`** statement that references the model.dart file, which we will need to read and write documents to the database.

www.dbooks.org

Next, let's create the **`DbHelper`** class, which will contain all the helper functions we need to work correctly with the database.

For now, let's just add the names of the properties for the database table and fields—we can see this as follows.

Code Listing 2-r: DbHelper class—dbhelper.dart

```
// Import statements… previous code.

class DbHelper {
  // Tables
  static String tblDocs = "docs";

  // Fields of the 'docs' table.
  String docId = "id";
  String docTitle = "title";
  String docExpiration = "expiration";

  String fqYear = "fqYear";
  String fqHalfYear = "fqHalfYear";
  String fqQuarter = "fqQuarter";
  String fqMonth = "fqMonth";

  // More code will follow…
}
```

As you can see, the code is self-explanatory—we have **`tblDocs`**, which indicates the name of the table that will be used to store the documents, and following that, the names of each of the fields contained within that table.

Next, let's create the database entry point, which will be a singleton. This is because we want to limit the instantiation of the **`DbHelper`** class to one instance only. We can do this as follows.

Code Listing 2-s: DbHelper class—dbhelper.dart (continued)

```
// Import statements… previous code.

class DbHelper {
  // Previous code

  // Singleton
  static final DbHelper _dbHelper = DbHelper._internal();

  // Factory constructor
  DbHelper._internal();

  factory DbHelper() {
    return _dbHelper;
```

```
  }

  // Database entry point
  static Database _db;
}

// More code will follow…
```

The first thing we have done is declare the **_dbHelper** instance as a **final** variable, which means that it is a single-assignment variable. Once assigned, its value cannot change—this is what we want, as it needs to be a singleton.

That variable is assigned to the value returned by the class' **internal** constructor, also known as a Factory constructor, which is then declared.

This **Factory** constructor pattern is quite common in Dart, and it is primarily used for creating singletons—this Stack Overflow thread explains this Dart pattern very well, in case you would like to know more about it.

Next, we declare a **static** variable of type **Database** that will hold the reference to the database entry point. The **Database** class is part of the **sqflite** library that we added to the Pubspec.yaml file previously.

Following this, we'll need to get a runtime reference to the database and initialize it—we can do this as follows.

Code Listing 2-t: DbHelper class—dbhelper.dart (continued)

```
// Import statements… previous code.

class DbHelper {
  // Previous code

  Future<Database> get db async {
    if (_db == null) {
      _db = await initializeDb();
    }

    return _db;
  }

  // Initialize the database
  Future<Database> initializeDb() async {
    Directory d = await getApplicationDocumentsDirectory();
    String p = d.path + "/docexpire.db";
    var db = await openDatabase(p, version: 1, onCreate: _createDb);
    return db;
  }
```


www.dbooks.org

```
}

// More code will follow…
```

We can get the runtime reference to the database by using an **async** getter called **db**, which returns a **Future** object that will reference the database. This is done by checking that **_db** is not **null**, and then invoking the **initializeDb** method, which is responsible for opening the database.

The **initializeDb** method returns a **Future**, which is the runtime reference to the database. It does this by calling the **openDatabase async** method and passing the location of the database file on the device—Docexpire.db.

The location of the file is determined by invoking the **getApplicationDocumentsDirectory** method and concatenating it to the actual file name on the device—docexpire.db.

Notice how the **openDatabase** method has an **onCreate** parameter, which indicates the name of the method that will be invoked the first time the database is opened. This will create the actual database our app will need, the **_createDb** method, which we can see as follows.

Code Listing 2-u: DbHelper class—dbhelper.dart (continued)

```
// Import statements… previous code.

class DbHelper {
  // Previous code

  // Create database table
  void _createDb(Database db, int version) async {
    await db.execute(
    "CREATE TABLE $tblDocs($docId INTEGER PRIMARY KEY, $docTitle TEXT, "
    + "$docExpiration TEXT, " +
    "$fqYear INTEGER, $fqHalfYear INTEGER, $fqQuarter INTEGER, " +
    "$fqMonth INTEGER)"
    );
  }
}

// More code will follow…
```

As you can see, the **_createDb** method simply calls the **db.execute** method and passes a **CREATE TABLE** statement, which is responsible for creating the database table that will be used for storing the information our app will use.

This is all the code required for initializing and creating the database. Next, we'll explore how we can add extra functionality to be able to query the database, and insert and delete from the database.

Inserting a new document—dbhelper.dart

The next thing we need to do is to write a method that allows us to save a document to the database. We can do this as follows.

Code Listing 2-v: DbHelper class—dbhelper.dart (continued)

```
// Import statements… previous code.

class DbHelper {
  // Previous code

  // Insert a new doc
  Future<int> insertDoc(Doc doc) async {
    var r;

    Database db = await this.db;
    try {
      r = await db.insert(tblDocs, doc.toMap());
    }
    catch (e) {
      debugPrint("insertDoc: " + e.toString());
    }
    return r;
  }
}

// More code will follow…
```

Notice that this **insertDoc** method reads the reference to the database (**this.db**), and then it calls the **db.insert** method and passes the result of the **doc.toMap** method, which was previously defined in model.dart.

Since we are dealing with a database **insert** operation, we wrap this code in a **try-catch** block to prevent any unhandled exceptions from arising.

As you have seen, adding an item to the database is not so difficult. Notice, though, that all database operations so far return a **Future** object, and the calls are **async**. The reason for this is that you don't want to have the application blocked while waiting for a database operation to finish.

Getting the list of documents—dbhelper.dart

Now that we've seen how to insert a document, it's important to understand how to we can retrieve any document stored within the database. We can do this as follows.

Code Listing 2-w: DbHelper class—dbhelper.dart (continued)

```
// Import statements…previous code.

class DbHelper {
  // Previous code

  // Get the list of docs.
  Future<List> getDocs() async {
    Database db = await this.db;
    var r = await db.rawQuery(
      "SELECT * FROM $tblDocs ORDER BY $docExpiration ASC");
    return r;
  }
}

// More code will follow…
```

As you can see, the **getDocs** method that retrieves the list of documents from the database is very simple. We can see a common pattern again—get the reference to the database using **this.db**.

Then, the actual query to the database is executed by running the **db.rawQuery** method and passing it the SQL query as a string parameter.

We can see that we are returning the result **r** in ascending order by the document's expiration date—**docExpiration**.

Next, let's see how we can get a specific document from the database.

Getting a specific document—dbhelper.dart

Getting all the documents available in the database is useful when we want to build up the main list of documents in our UI. However, if we want to modify a specific document that we previously added, we'll need to be able to retrieve that document from the database—this is what we'll do next.

Code Listing 2-x: DbHelper class—dbhelper.dart (continued)

```
// Import statements… previous code.

class DbHelper {
  // Previous code

  // Gets a Doc based on the id.
  Future<List> getDoc(int id) async {
    Database db = await this.db;
    var r = await db.rawQuery(
```

```
      "SELECT * FROM $tblDocs WHERE $docId = " + id.toString() + "" );
    return r;
  }

  // Gets a Doc based on a String payload
  Future<List> getDocFromStr(String payload) async {
    List<String> p = payload.split("|");
    if (p.length == 2) {
      Database db = await this.db;
      var r = await db.rawQuery(
          "SELECT * FROM $tblDocs WHERE $docId = " + p[0] +
          " AND $docExpiration = '" + p[1] + "'" );
      return r;
    }
    else
      return null;
  }
}

// More code will follow…
```

We have two ways of retrieving a specific document. One way is to retrieve the document by its id (**docId**); this is what the **getDoc** method does. Another way is to retrieve it from the database by its **id** and by the document's expiration date (**docExpiration**)—this is what the **getDocFromString** method does.

The **getDoc** method simply calls the **db.rawQuery** method and runs a query on the **tblDocs** table. It retrieves the document where the **docId** has the same value as the **id** parameter—very straightforward.

The **getDocFromStr** method is slightly more complex, but not too much. The difference is that this method receives a **String** object payload, which contains two important bits of data, separated by a pipe (|) character. The first part of the payload is **docId**, and the second part is **docExpiration**.

The **payload** is **split** and each respective value concatenated onto the SQL query that gets passed to the **db.rawQuery** method. The SQL query retrieves the document by checking for the correct **id** and expiration date values.

By using these two methods, we can retrieve any specific document we need.

Counting documents—dbhelper.dart

Now that we know how to retrieve specific documents, we'll also need to count how many documents we have and retrieve the largest document ID from the database—this will be important for changing the UI state later. So, let's see how we can do this.

Code Listing 2-y: DbHelper class—dbhelper.dart (continued)

```
// Import statements… previous code.

class DbHelper {
  // Previous code

  // Get the number of docs.
  Future<int> getDocsCount() async {
    Database db = await this.db;
    var r = Sqflite.firstIntValue(
        await db.rawQuery("SELECT COUNT(*) FROM $tblDocs")
    );
    return r;
  }

  // Get the max document id available on the database.
  Future<int> getMaxId() async {
    Database db = await this.db;
    var r = Sqflite.firstIntValue(
        await db.rawQuery("SELECT MAX(id) FROM $tblDocs")
    );
    return r;
  }
}

// More code will follow…
```

The first method, **getDocsCount**, basically executes a SQL query that returns an **int** value, which counts how many documents there are on the **tblDocs** table.

The second method, **getMaxId**, executes a SQL query that returns an **int** value, which represents the maximum value existing on the **tblDocs** table for the **id** field of all documents. In other words, it returns the largest existing document **id** value within the table.

As you can see, the only difference between these methods is that one uses the SQL **COUNT** function and the other uses the SQL **MAX** function.

To be able to change the state of the UI, it's important we have these functions so we can get the number of documents in the database—this is main reason for having the **getDocsCount** function.

We use the **getMaxId** function because when a new document is being added, we need to make sure that this new document gets assigned an **id** value larger than the largest one available on the **tblDocs** table. This is done so that each document has an **id** that is consecutive to the previous one and is not repeated.

Updating and deleting documents—dbhelper.dart

We now almost have all the database functionality we require, but we are still missing an important part—this is the ability to delete and update documents. Let's add this code as follows.

Code Listing 2-z: DbHelper class—dbhelper.dart (continued)

```
// Import statements… previous code.

class DbHelper {
  // Previous code.

  // Update a doc.
  Future<int> updateDoc(Doc doc) async {
    var db = await this.db;
    var r = await db.update(tblDocs, doc.toMap(),
        where: "$docId = ?", whereArgs: [doc.id]);
    return r;
  }

  // Delete a doc.
  Future<int> deleteDoc(int id) async {
    var db = await this.db;
    int r = await db.rawDelete(
        "DELETE FROM $tblDocs WHERE $docId = $id");
    return r;
  }

  // Delete all docs.
  Future<int> deleteRows(String tbl) async {
    var db = await this.db;
    int r = await db.rawDelete("DELETE FROM $tbl");
    return r;
  }
}
```

The **updateDoc** method is responsible for updating a specific document on the database, based on the document's **id**. This is done by invoking the **db.update** method by passing the resultant object of the call to **doc.toMap**, which converts the document from a Dart object to its database model equivalent.

The **deleteDoc** method removes a document from the **tblDocs** table by running a scoped SQL **DELETE** statement, targeting a specific document **id** by invoking the **db.rawDelete** method.

Finally, the **deleteRows** method removes all the documents from the database. This is done by calling the **db.rawDelete** method and running a SQL **DELETE** statement that is not scoped to any specific document **id**.

Awesome—that concludes our dbhelper.dart file, which is responsible for all our app's database operations.

Summary

We've come a long way, and it's been a detailed, but certainly interesting chapter. So far, we've managed to lay out the foundations of our application and how it will be able to interact with the database by adding the model and all the utilities and helper functions required.

In the next chapter, we'll implement the UI by creating the Document Details screen, docdetail.dart, and from there, we'll move on to the main screen and finalize the application.

There are still a lot of things to discover and learn with the amazing Flutter framework.

Chapter 3 App UI—Document Details

Quick intro

Throughout this chapter, we'll look at how to create some of the required UI parts of our application, which is essentially the Document Details screen. Without further ado, let's dive right in.

Document Details

We've reached quite a milestone! We've pretty much written all the utility and underlying database code that our application will use. But we are not done yet—we still need to create the UI logic that our application will use.

We'll start writing that UI logic by creating the Document Details window, which will contain the details of each document that our application will store.

So, under the project's lib folder, create a subfolder called **ui**, and under it, create a new file called **docdetail.dart.** This is how your lib folder structure should look so far.

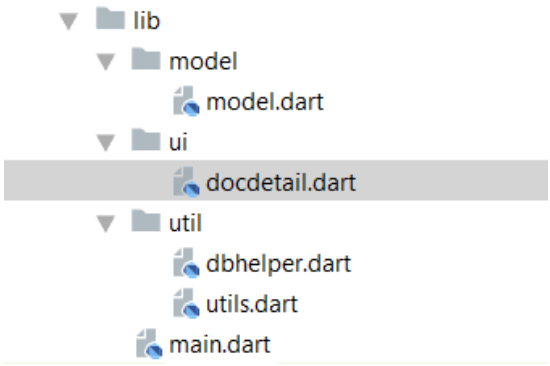


Figure 3-a: The project's lib folder structure so far

With the docdetail.dart file created, let's add some code—we'll start by importing the references to the libraries and packages we'll need.

Code Listing 3-a: Import statements—docdetail.dart

```
import 'dart:async';

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

import 'package:flutter_masked_text/flutter_masked_text.dart';
```


www.dbooks.org

```
import 'package:flutter_datetime_picker/flutter_datetime_picker.dart';

import '../model/model.dart';
import '../util/utils.dart';
import '../util/dbhelper.dart';
```

Okay—let's check what references we've imported. First, we are importing the **`dart:async`** library, which we'll need to make asynchronous calls to the database.

Next, we reference the Flutter Material Design (**`package:flutter/material.dart`**) and Services (**`package:flutter/services.dart`**) packages. Material Design is used for the app's UI layout, and the Services library exposes platform-specific services to Flutter apps, such as handling text input.

Following that, notice how we are referencing two packages we have not come across before. The first one is going to be used for a masked text input field, which we will use for entering dates manually: **`flutter_masked_text.dart`**.

The second is going to be used for selecting dates, as you would normally do using an iPhone. It is inspired by the Flutter Cupertino Date Picker component, **`flutter_datetime_picker.dart`**.

We can see an example of it as follows.

Figure 3-b: The Flutter Cupertino Date Picker

The final three **`import`** statements refer to the code files we previously wrote and will need to use in docdetail.dart, These are: model.dart, utils.dart, and dbhelper.dart.

Something very important we need to do is add a reference to the **`flutter_masked_text`** and **`flutter_datetime_picker`** packages to the Pubspec.yaml file, and then install the referenced packages using the **Packages get** option, which can be seen as follows.

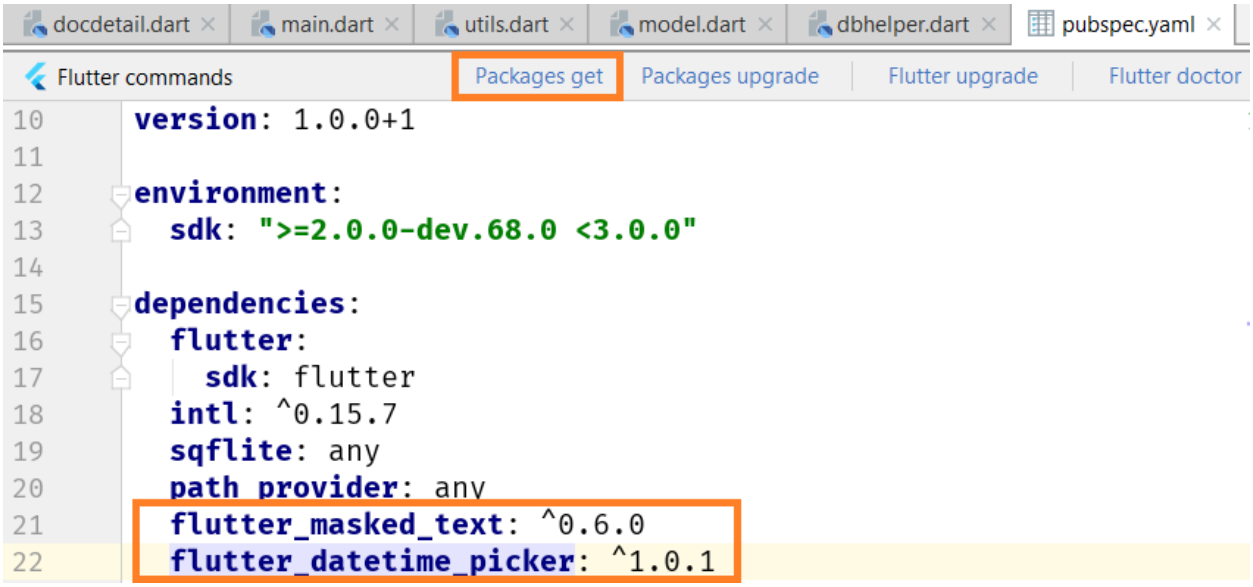


Figure 3-c: Adding the packages—Pubspec.yaml

That wraps up the **`import`** statements that the docdetail.dart file needs. Next, let's add a menu option that allows us to delete a document in case it was incorrectly entered.

Menu options

For every document that exists within the database that can be edited, we should also have the option to remove it. This is particularly useful if you've entered a document that is incorrect. We can do this as follows.

Code Listing 3-b: Delete document menu option—docdetail.dart

```
// Import statements… previous code.

// Menu item
const menuDelete = "Delete";
final List<String> menuOptions = const <String> [
  menuDelete
];

// More code will follow…
```

As you can see, the **`Delete`** menu option is simply an element that is part of a **`menuOptions`** array that we'll add to the Flutter UI shortly.

We could add more menu options to this array, but the only one we really need is the option to delete an existing document, for which we'll add the logic later.

www.dbooks.org

Stateful widget

We are now ready to create the base widget that will be used within docdetail.dart. This is going to be a stateful widget, which is a widget that describes part of the user interface by building a set of other widgets that describe the user interface more concretely.

The stateful widget has a state that can change. The state is information that can be read synchronously when the widget is built and might change during its lifetime.

Having a stateful widget is useful when part of the user interface that is being rendered can change dynamically. This is exactly our case, as docdetail.dart will contain document details that can vary.

Let's go ahead and define the stateful widget we will use for rendering the UI of docdetail.dart.

Code Listing 3-c: Stateful widget class—docdetail.dart

```
// Previous code…

class DocDetail extends StatefulWidget {
  Doc doc;
  final DbHelper dbh = DbHelper();

  DocDetail(this.doc);

  @override
  State<StatefulWidget> createState() => DocDetailState();
}

// More code will follow…
```

As we can see, a stateful widget is simply a Dart class—in this case called **DocDetail**—that inherits from the base **StatefulWidget** Flutter UI class.

This class has a constructor called **DocDetail** that is initialized with an instance of the **Doc** class from model.dart.

A very important part of the **DocDetail** class is the **createState** method inherited from the **StatefulWidget** class, which needs to be overridden—that's why the **@override** attribute is used. The overriding is done by invoking an instance of the **DocDetailState** class using the lambda or arrow (**=>**) syntax.

Notice that an instance of the **DbHelper** class is created and assigned to the **dbh** (database handler) variable, which will be used by the logic contained within docdetail.dart to read and write to the database.

As you can see in Code Listing 3-c, there is a pattern—for each **StatefulWidget** class, there is a corresponding **State** class. Let's go ahead and create that **State** class as follows.

Code Listing 3-d: State class—docdetail.dart

```
// Previous code…

class DocDetailState extends State<DocDetail> {
  final GlobalKey<FormState> _formKey = new GlobalKey<FormState>();
  final GlobalKey<ScaffoldState> _scaffoldKey =
      new GlobalKey<ScaffoldState>();

  final int daysAhead = 5475; // 15 years in the future.

  final TextEditingController titleCtrl = TextEditingController();
  final TextEditingController expirationCtrl = MaskedTextController(
      mask: '2000-00-00');

  bool fqYearCtrl = true;
  bool fqHalfYearCtrl = true;
  bool fqQuarterCtrl = true;
  bool fqMonthCtrl = true;
  bool fqLessMonthCtrl = true;

  // More code to follow…
}
```

Here we have our **DocDetailState** class declared. As you can see, it inherits from **State<DocDetail>**, which means that this class handles the state of the **DocDetail** stateful widget class we previously declared.

Within the **DocDetailState** class, we have declared a set of variables that we will need, and they represent the state of a document being created or edited. Let's explore these variables and what they are used for.

The first two variables we have declared, **_formKey** and **_scaffoldKey**, are going to be used for keeping the state of the form once it has been submitted—when the data is saved.

The **FormState** class in Flutter is associated with keeping the state of a Form widget, which we will add shortly to the **DocDetailState** class code.

The **ScaffoldState** class in Flutter handles the state for a Scaffold object. The Scaffold widget will contain the Form widget—we'll see this later.

Next, we have declared a variable called **daysAhead**, which indicates how far into the future we can assign an expiration date to a document.

This has been set to a maximum of 5,475 days (approximately 15 years) into the future, which is usually longer than the expiration date of standard types of documents such as passports, credit

www.dbooks.org

cards, and driver's licenses. So, the expiration date of a document cannot be greater than the value of **daysAhead**.

Following that, we have two **TextEditingController** variables: one for the document's title (or description), **titleCtrl**, and another for the document's expiration date, **expirationCtrl**.

A **TextEditingController** represents a handy controller for a text field. So, when the text field associated with the **TextEditingController** has been modified, the text field updates its value property and the controller notifies its listeners. You can find more details on the official Flutter documentation.

Given that both the document title and expiration date can be manually entered, it is logical to bind both to **TextEditingController** objects.

The main difference between them is that the expiration date field, which binds to **expirationCtrl**, is instantiated as a **MaskedTextController** and assigned a default mask, so when the date is manually entered, it follows the date format *YYYY-MM-DD* (for example, 2020-10-12).

Notice that, so far, all the variables declared within the **DocDetailState** class have been marked as **final**, which means that their value can only be set once.

Finally, we have five variables (which are technically objects) that represent the specific alerts of when we would like the application to remind us that a document is going to expire.

We won't implement the alert mechanism itself within the scope of this book. However, by adding these variables (which already have their matching columns in the database), we leave these building blocks ready. So in the future, you could add your own alert mechanism using these variables:

- The **fqYearCtrl** variable, when set to **true**, would indicate that we would like our app to remind us of the expiration date of a document when it's due to expire within a year's time.
- The **fqHalfYearCtrl** variable, when set to **true**, would indicate that we would like our app to remind us of the expiration date of a document when it's due to expire within the next six months.
- The **fqQuarterCtrl** variable, when set to **true**, would indicate that we would like our app to remind us of the expiration date of a document when it's due to expire within the next three months.
- The **fqMonthCtrl** variable, when set to **true**, would indicate that we would like our app to remind us of the expiration date of a document when it's due to expire within the next month.
- The **fqLessMonthCtrl** variable, when set to **true**, would indicate that we would like our app to remind us of the expiration date of a document when it's due to expire less than a month from now.

These are all the variables needed within the **`DocDetailState`** class. Let's move on with the rest of the code.

Initializing text controllers and variables

When the Document Details screen is shown, it is important that the right data is correctly displayed to the user—this could be data from an existing document on the database, or alternatively, blank data. Let's see an example of each.

Here's an example of how the Document Details screen looks when loading data from an existing document.

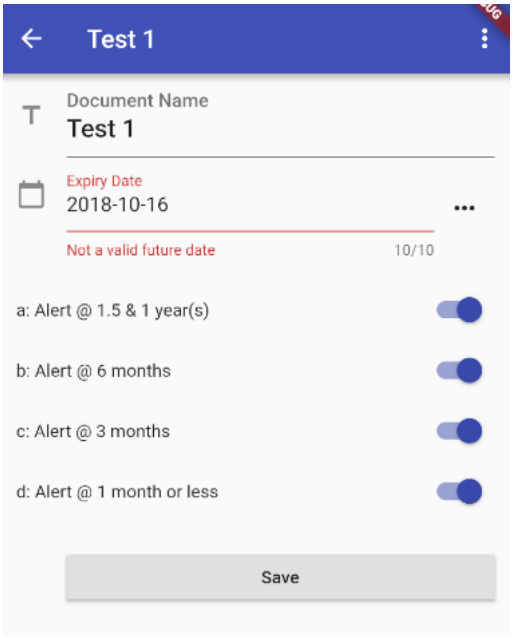


Figure 3-d: Document Details screen—Existing document

As you can see, all the respective variables we previously described—represented by those UI widgets—have corresponding values assigned, which represent the value stored in the database for that document.

Let's now have a look at how this same screen would look for an empty document.

www.dbooks.org

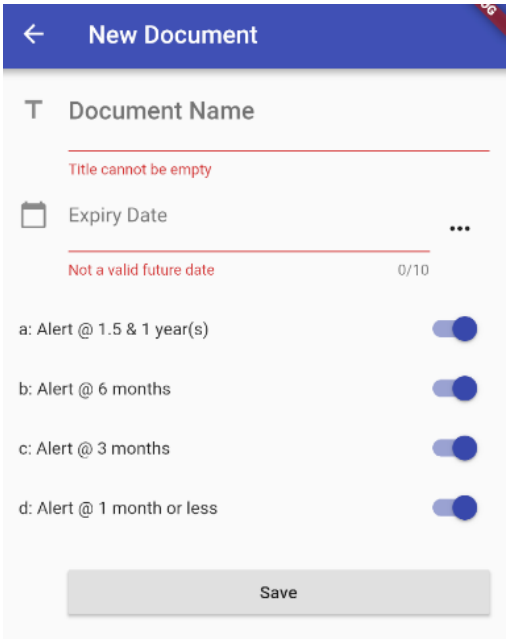


Figure 3-e: Document Details screen—New document

Notice that the Document Name and Expiry Date fields are empty; however, by default, the alert fields are all enabled, which is what the initialization code is responsible for.

Code Listing 3-e: Initialization code—docdetail.dart

```
// Previous code…

class DocDetailState extends State<DocDetail> {
  // Previous code

  // Initialization code
  void _initCtrls() {
    titleCtrl.text = widget.doc.title != null ? widget.doc.title : "";
    expirationCtrl.text =
        widget.doc.expiration != null ? widget.doc.expiration : "";

    fqYearCtrl = widget.doc.fqYear != null ?
        Val.IntToBool(widget.doc.fqYear) : false;
    fqHalfYearCtrl = widget.doc.fqHalfYear != null ?
        Val.IntToBool(widget.doc.fqHalfYear) : false;
    fqQuarterCtrl = widget.doc.fqQuarter != null ?
        Val.IntToBool(widget.doc.fqQuarter) : false;
    fqMonthCtrl = widget.doc.fqMonth != null ?
        Val.IntToBool(widget.doc.fqMonth) : false;
  }

  // More code will follow…
}
```

As you can see, all we are doing is assigning a value to each of the variables that are represented on the screen by using a ternary conditional expression.

This means that if there's a value assigned to its corresponding **doc** property, then that value is assigned to the variable if no default value is assigned.

In essence, if there's a **doc** object with values, those values will be used and assigned to their corresponding variables (field widgets seen on the screen). But where is this **doc** object coming from?

That **doc** object is the model representation of a document record stored in the database, if there is one. It's the **doc** property of the **DocDetail** class. To understand this better, let's have a look at the following diagram.

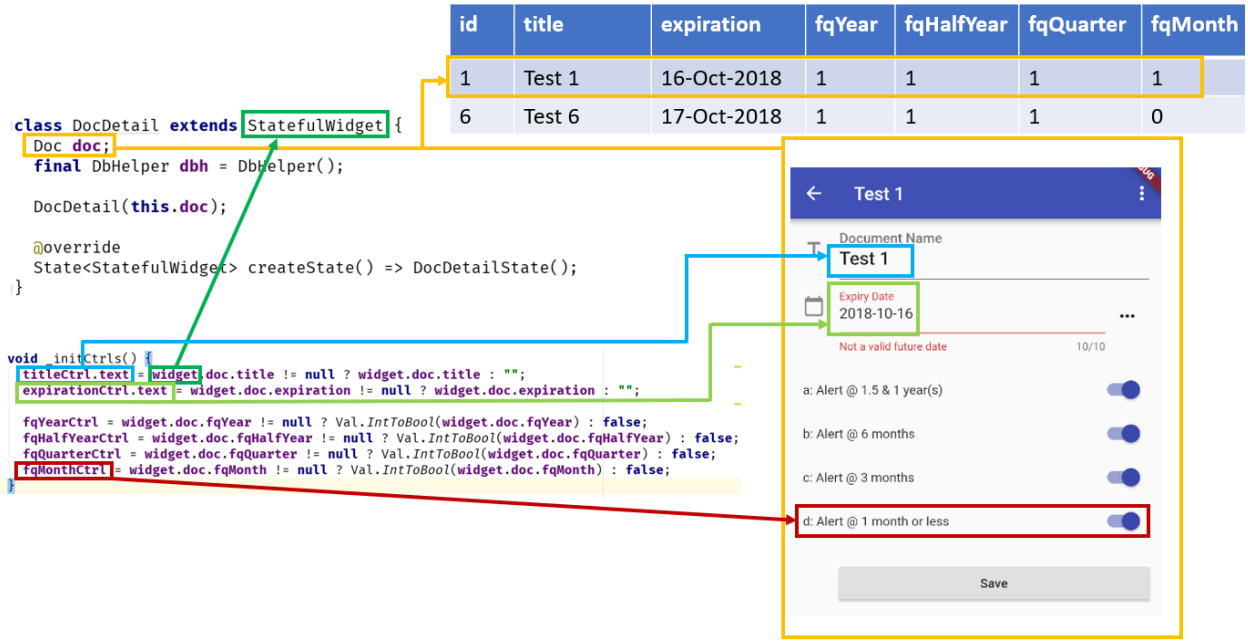


Figure 3-f: Relationship among the code, screen and database—Document Details

Now that we know how the initialization code relates to the Document Details screen, and how this relates to a record stored in the database, let's move on and focus on how to create the required UI widgets and their underlying logic.

Choosing a date

One of the nicest features about the Document Details screen is the possibility to be able to either manually enter the document's expiration date, or choose it from an iOS-styled date picker component, which looks like the following image.

www.dbooks.org

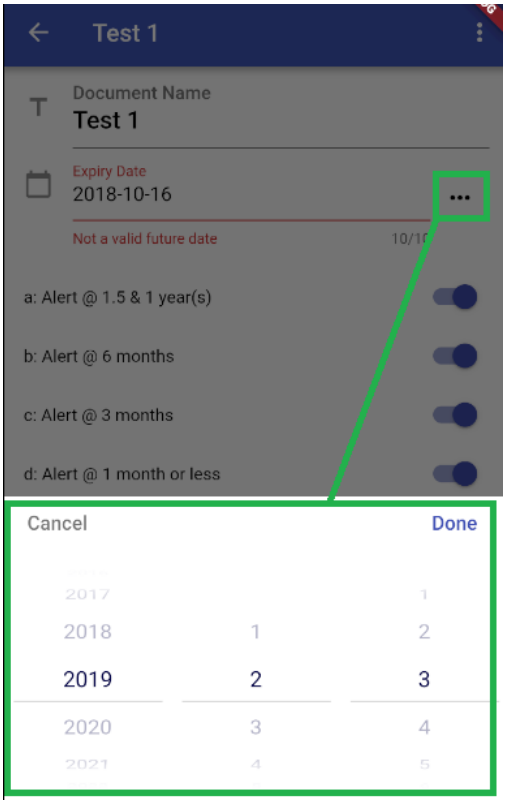


Figure 3-g: iOS-styled date-picker component

Let's write the code that displays this component and allows us to choose the date without having to write it manually.

Code Listing 3-f: Date-picker code—docdetail.dart

```
// Previous code…

class DocDetailState extends State<DocDetail> {
  // Previous code

  // Date Picker & Date function
  Future _chooseDate(BuildContext context, String initialDateString)
  async {
    var now = new DateTime.now();
    var initialDate = DateUtils.convertToDate(initialDateString) ?? now;

    initialDate = (initialDate.year >= now.year &&
        initialDate.isAfter(now) ? initialDate : now);

    DatePicker.showDatePicker(context, showTitleActions: true,
        onConfirm: (date) {
          setState(() {
            DateTime dt = date;
```

```
            String r = DateUtils.ftDateAsStr(dt);
            expirationCtrl.text = r;
          });
        },
        currentTime: initialDate);
  }

  // More code to follow…
}
```

Let's go over what this method does. The first thing we see is the **_chooseDate** method, which is marked as **async** and returns a **Future** object.

This is because the **_chooseDate** method is going to be triggered when the "**...**" button on the Expiry Date field is tapped—this corresponds to an **onPressed** event, so the operation needs to be asynchronous.

Notice the parameters being passed to the **_chooseDate** method. One is of type **BuildContext**, and the other of type **initialDateString**, which indicates an initialization date passed as a **String** object.

The **BuildContext** class handles the location of the widget in Flutter's internal widget tree. You can find additional information about this class here.

In the **_chooseDate** method, the first thing that happens within the first two lines of code is that the **initialDate** is assigned to either the **DateTime** value of **initialDateString**, or to the current **DateTime** value, when the **DateTime** value of **initialDateString** is **null**.

```
var now = new DateTime.now();

var initialDate = DateUtils.convertToDate(initialDateString) ?? now;
```

We need to ensure that the date-picker component doesn't give us the possibility to choose a date that is in the past. Therefore, the code checks if the value of **initialDate** is in the future, or at least equal to the current date. If so, the value of **initialDate** is used; otherwise, the current date value is used, represented by the variable **now**. This then becomes the final value of the **initialDate** variable.

```
initialDate = (initialDate.year >= now.year &&

    initialDate.isAfter(now) ? initialDate : now);
```

The final value of **initialDate** is passed on to the **showDatePicker** method and assigned to the **currentTime** property of the **DatePicker** instance.

The actual date assignment happens within the **onConfirm** event of the **DatePicker** instance.

www.dbooks.org

The **onConfirm** event gets triggered when a date is selected from the date-picker component and the Done button is tapped.

The **date** parameter passed to the **onConfirm** event corresponds to the date chosen using the date-picker component.

All the logic that follows is executed inside the **setState** method, which notifies the Flutter framework that the internal state of **DocDetailState** has changed.

```
setState(() {

  DateTime dt = date;

  String r = DateUtils.ftDateAsStr(dt);

  expirationCtrl.text = r;

});
```

That selected **date** is converted to a **DateTime** value, formatted accordingly using the **ftDateAsString** method from the **DateUtils** class, and assigned to the Expiry Date field, which is represented by **expirationCtrl.text**.

Deleting a document

Remember the menu options array we previously created? We are now going to put it to use.

The reason is that the application needs to give users the ability to be able to delete a document, in case it was entered incorrectly, or it is no longer valid (expired a long time ago).

This option is accessed by clicking the "**...**" button on the top, next to the title of the document.

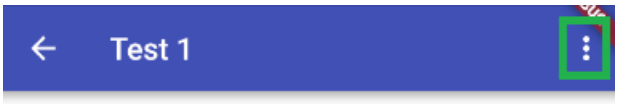


Figure 3-h: The document menu

Here is what the menu option to delete a document looks like.

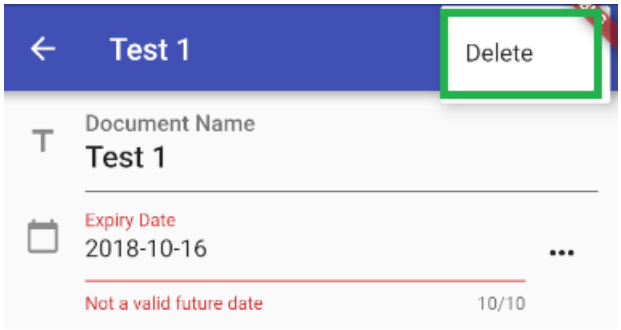


Figure 3-i: The delete document menu option

Let's explore the code-behind, to see what it does.

Code Listing 3-g: Delete document code—docdetail.dart

```
// Previous code…

class DocDetailState extends State<DocDetail> {
  // Previous code

  // Upper Menu
  void _selectMenu(String value) async {
    switch (value) {
      case menuDelete:
        if (widget.doc.id == -1) {
          return;
        }
        await _deleteDoc(widget.doc.id);
    }
  }

  // Delete doc
  void _deleteDoc(int id) async {
    int r = await widget.dbh.deleteDoc(widget.doc.id);
    Navigator.pop(context, true);
  }

  // More code to follow…
}
```

If we inspect the code more closely, we can see that we have two methods, each responsible for an action. The **`_selectMenu`** method is invoked when the user selects the **`menuDelete`** option—we'll look at that specific trigger later.

This method invokes the **`_deleteDoc`** method when the active document on the Document Details screen is not empty, which means that it exists in the database—it has a corresponding record.

www.dbooks.org

When the document is new (empty), the method returns to its caller. This is what the **`widget.doc.id == -1`** conditional check is for.

The **`_deleteDoc`** method is very simple—all it does is invoke the **`deleteDoc`** method from the **`dbh`** (database helper) instance through the parent **`DocDetail`** stateful widget. This will remove the corresponding document record from the database.

Then, the control is returned to the main screen by closing the current **`context`** (the **Document Details** screen) by calling the **`Navigator.pop`** method.

You can find additional information about how to navigate between Flutter screens in the official documentation.

Now that we know how to delete a document, let's explore how we can save the data entered through the Document Details screen.

Saving a document

A fundamental part of the Document Details screen is the ability to save data that has been entered or has changed—this what the following code does.

Code Listing 3-h: Save document code—docdetail.dart

```
// Previous code…

class DocDetailState extends State<DocDetail> {
  // Previous code

  // Save doc
  void _saveDoc() {
    widget.doc.title = titleCtrl.text;
    widget.doc.expiration = expirationCtrl.text;

    widget.doc.fqYear = Val.BoolToInt(fqYearCtrl);
    widget.doc.fqHalfYear = Val.BoolToInt(fqHalfYearCtrl);
    widget.doc.fqQuarter = Val.BoolToInt(fqQuarterCtrl);
    widget.doc.fqMonth = Val.BoolToInt(fqMonthCtrl);

    if (widget.doc.id > -1) {
      debugPrint("_update->Doc Id: " + widget.doc.id.toString());
      widget.dbh.updateDoc(widget.doc);
      Navigator.pop(context, true);
    }
    else {
      Future<int> idd = widget.dbh.getMaxId();
      idd.then((result) {
        debugPrint("_insert->Doc Id: " + widget.doc.id.toString());
```

```
        widget.doc.id = (result != null) ? result + 1 : 1;
        widget.dbh.insertDoc(widget.doc);
        Navigator.pop(context, true);
      });
    }
  }

  // More code to follow…
}
```

The first two lines of code read the values entered through **`titleCtrl.text`** and **`expirationCtrl.text`** fields, which correspond to the **Document Name** and **Expiry Date** fields seen on the screen, and assign them to their corresponding properties within the **`doc`** instance. This represents the object model that is stored in the database.

The following four lines of code do the same for the alert properties. The only difference is that those widgets on the screen have a value of **on** (**`true`**) or **off** (**`false`**), which need to be converted to their integer equivalent, so they can be saved in the database. This Boolean-to-integer conversion is done by invoking the **`BoolToInt`** method from the **`Val`** class (found in utils.dart).

To better understand this, let's have a look at the following figure.

www.dbooks.org

id	title	expiration	fqYear	fqHalfYear	fqQuarter	fqMonth
1	Test 1	16-Oct-2018	1	1	1	1
6	Test 6	17-Oct-2018	1	1	1	0

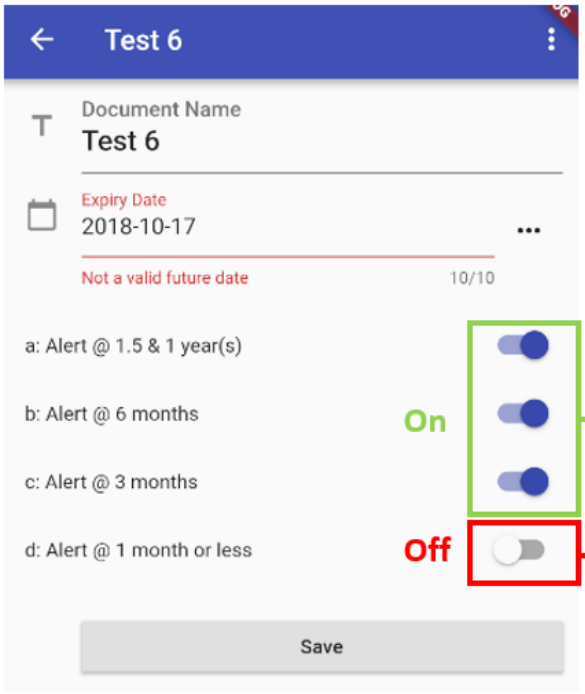


Figure 3-j: The on and off alerts database values

At this stage, we have assigned all the values to the **doc** instance that is going to be saved to the database. Next comes the most interesting part of the **_saveDoc** method.

To be able to save the data correctly, we need to check if the data being entered corresponds to a new document or an existing one.

If the condition (**widget.doc.id > -1**) evaluates to **true**, then we are modifying an existing document. Therefore, we invoke the **updateDoc** method from the **dbh** (database handler) instance and pass the document object model (**doc**) that is going to be updated.

If the condition (**widget.doc.id > -1**) evaluates to **false**, then we are saving a new document to the database. Therefore, we invoke the **insertDoc** method from the **dbh** (database handler) instance and pass the document object model (**doc**) that is going to be inserted.

Notice that when inserting a new document, we need to make sure that we assign a **doc.id** value that is larger than the largest **doc.id** value stored in the database table. Therefore, we invoke the **getMaxId** method.

Finally, to return the navigation control back to the main screen, we invoke **`Navigator.pop`**.

Submitting the form

We now know how we can save newly entered data, or data that has been modified from the Document Details screen—but how does that save action get triggered?

Just like with HTML, Flutter has the concept of a form—and as you might have guessed, forms in Flutter can also be submitted. Submitting a form is what triggers the save action.

Let's look at the code to understand better how this works.

Code Listing 3-i: Submitting a form code—docdetail.dart

```
// Previous code…

class DocDetailState extends State<DocDetail> {
  // Previous code

  // Submit form
  void _submitForm() {
    final FormState form = _formKey.currentState;

    if (!form.validate()) {
      showMessage('Some data is invalid. Please correct.');
    } else {
      _saveDoc();
    }
  }

  void showMessage(String message, [MaterialColor color = Colors.red]) {
    _scaffoldKey.currentState
        .showSnackBar(new SnackBar(backgroundColor: color,
            content: new Text(message)));
  }

  // More code to follow…
}
```

The **`_submitForm`** method is quite simple. First, we get the current state of the form. We do this by invoking **`_formKey.currentState`** and assigning that value to an instance of the **`FormState`** Flutter class.

The great thing about forms in Flutter is that they are almost self-managed and keep their state, so to know if something has changed, all we need to do is invoke the **`validate`** method from the **`FormState`** instance.

www.dbooks.org

If the **validate** method returns **true**, it means that the field values on the form are valid and the data is okay to be saved, so the **_saveDoc** method can be invoked.

If the **validate** method returns **false**, it means that one or more field values on the form are invalid and the data cannot be saved, so an alert message is displayed to the user. This is what the **showMessage** method does.

The **showMessage** method displays a message—this is achieved by using a **SnackBar** widget, which we can see as follows.

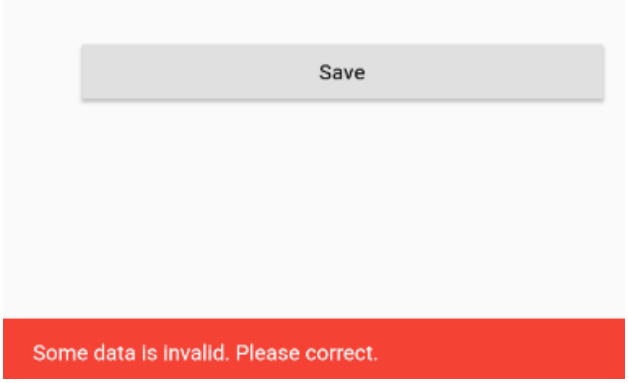


Figure 3-k: The SnackBar displayed—Invalid form data

Now that we know how the save action gets triggered, let's wrap this up and build the UI, which is what we'll do next.

Building the UI

We've reached the last part of docdetail.dart, which is both a major milestone for us and exciting at the same time.

Throughout this last part, our focus is going to be on how to build the UI and tie together all the previous docdetail.dart code parts we've written.

Let's start off by overriding a fundamental method of the inherited **State** class, which is the **initState** method.

Code Listing 3-j: The initState method—docdetail.dart

```
// Previous code…

class DocDetailState extends State<DocDetail> {
  // Previous code

  @override
  void initState() {
```

```
    super.initState();
    _initCtrls();
  }

  // More code to follow…
}
```

The **initState** method is responsible for initializing the state of its class—in this case, the **DocDetailState** class. This is where we can do all initializations needed before building the UI.

This method is described with the **@override** attribute, as it overrides the behavior of the **initState** method inherited from the **State<DocDetail>** class.

Within the method, the inherited **initState** method from the **State<DocDetail>** class is called. This is done by invoking it as **super.initState**.

Following that, the **_initCtrls** method is called, which, as you might remember, is responsible for initializing the values of the text controllers and alert reminder widgets.

It's important to note that the **initState** method gets triggered by the Flutter framework when the object is inserted into the widget tree. You can find more details about the **initState** method within the official Flutter documentation.

With the **initState** method described, we are now ready to build the UI—this is done with the **build** method. The complete code of the build method for the **DocDetailState** class looks as follows.

Code Listing 3-k: The build method—Docdetail.dart

```
// Previous code…

class DocDetailState extends State<DocDetail> {
  // Previous code

  @override
  Widget build(BuildContext context) {
    const String cStrDays = "Enter a number of days";
    TextStyle tStyle = Theme.of(context).textTheme.title;
    String ttl = widget.doc.title;

    return Scaffold(
        key: _scaffoldKey,
        resizeToAvoidBottomPadding: false,
        appBar: AppBar(
            title: Text(ttl != "" ? widget.doc.title : "New Document"),
            actions: (ttl == "") ? <Widget>[]: <Widget>[
              PopupMenuButton(
```


www.dbooks.org

```
            onSelected: _selectMenu,
            itemBuilder: (BuildContext context) {
              return menuOptions.map((String choice) {
                return PopupMenuItem<String>(
                  value: choice,
                  child: Text(choice),
                );
              }).toList();
            },
          ),
        ]
      ),
      body: Form(
          key: _formKey,
          autovalidate: true,
          child: SafeArea(
            top: false,
            bottom: false,
            child: ListView(
              padding: const EdgeInsets.symmetric(horizontal: 16.0),
              children: <Widget>[
                TextFormField (
                  inputFormatters: [
                    WhitelistingTextInputFormatter(
                      RegExp("[a-zA-Z0-9 ]"))
                  ],
                  controller: titleCtrl,
                  style: tStyle,
                  validator: (val) => Val.ValidateTitle(val),
                  decoration: InputDecoration(
                    icon: const Icon(Icons.title),
                    hintText: 'Enter the document name',
                    labelText: 'Document Name',
                  ),
                ),
                Row(children: <Widget>[
                  Expanded(
                      child: TextFormField(
                        controller: expirationCtrl,
                        maxLength: 10,
                        decoration: InputDecoration(
                            icon: const Icon(Icons.calendar_today),
                            hintText: 'Expiry date (i.e. ' +
                            DateUtils.daysAheadAsStr(daysAhead) + ')',
                            labelText: 'Expiry Date'
                        ),
                        keyboardType: TextInputType.number,
                        validator: (val) => DateUtils.isValidDate(val)
                          ? null : 'Not a valid future date',
```

```
            )),
          IconButton(
            icon: new Icon(Icons.more_horiz),
            tooltip: 'Choose date',
            onPressed: (() {
              _chooseDate(context, expirationCtrl.text);
            }),
          )
        ]),
        Row(children: <Widget>[
          Expanded(child: Text(' ')),
        ]),
        Row(children: <Widget>[
          Expanded(child: Text('a: Alert @ 1.5 & 1 year(s)')),
          Switch(
              value: fqYearCtrl, onChanged: (bool value) {
            setState(() {
              fqYearCtrl = value;
            });
          }),
        ]),
        Row(children: <Widget>[
          Expanded(child: Text('b: Alert @ 6 months')),
          Switch(
              value: fqHalfYearCtrl, onChanged: (bool value) {
            setState(() {
              fqHalfYearCtrl = value;
            });
          }),
        ]),
        Row(children: <Widget>[
          Expanded(child: Text('c: Alert @ 3 months')),
          Switch(
              value: fqQuarterCtrl, onChanged: (bool value) {
            setState(() {
              fqQuarterCtrl = value;
            });
          }),
        ]),
        Row(children: <Widget>[
          Expanded(child: Text('d: Alert @ 1 month or less')),
          Switch(
              value: fqMonthCtrl, onChanged: (bool value) {
            setState(() {
              fqMonthCtrl = value;
            });
          }),
        ]),
        Container(
```


www.dbooks.org

```
                    padding: const EdgeInsets.only(
                      left: 40.0, top: 20.0),
                    child: RaisedButton(
                      child: Text("Save"),
                      onPressed: _submitForm,
                    )
                  ),
                ],
              ),
            ));
  }
}
```

The **build** method is what builds the UI—in this case, the UI of the Document Details window.

That's quite a bit of code, so to understand how the UI has been built using this method, it's best to break the code into smaller chunks to describe how each individual part is composed. Let's start from the top.

The first thing to notice is that the **build** method has the **@override** attribute, which means that its logic will override any logic from the **build** method inherited from **State<DocDetail>**.

Next, notice how the **build** method returns a type of **Widget**, which describes the configuration for an Element. You can find more details about the **Widget** class in the official documentation.

Because the **build** method is going to return a **Widget**, it's necessary to handle the location of the widget returned within the widget tree. Therefore, the **BuildContext** class is passed as a parameter.

Next, let's explore the following three lines, which correspond to internal initializations.

```
const String cStrDays = "Enter a number of days";

TextStyle tStyle = Theme.of(context).textTheme.title;

String ttl = widget.doc.title;
```

The first instruction declares and initializes a **String** object constant that is totally self-descriptive.

The second instruction basically initializes the **TextStyle** instance that will be used for this UI. You can find more details about text styling in Flutter in the official documentation.

The third instruction, also self-descriptive, simply initializes the **ttl** variable with the value of the document title, accessible through **widget.doc.title**.

Scaffold

With all the initializations ready, now comes the interesting part: the **build** method returns a Scaffold object, which implements a basic Material Design visual layout structure.

You can think of the **Scaffold** object as a layout container that will contain the widgets displayed on the screen. The **Scaffold** object has two essential properties: **appBar** and **body**.

The **Scaffold** object also has two other properties that are quite important: **key** and **resizeToAvoidBottomPadding**.

The **key** property is assigned to the **_scaffoldKey** property, which is used for tracking the state of the **Scaffold**.

The resizeToAvoidBottomPadding property is used to indicate whether the **body** or floating widgets should size themselves to avoid the window's bottom padding.

AppBar

The **appBar** represents the uppermost area of the layout that includes the **title** and any menu options. The **body** represents the rest of the useable screen area, which contains all the other fields, such as the Document Name, Expiry Date, and the various alert reminders.

The **appBar** section of the code is quite easy to understand without looking at a diagram, but the **body** section is more elaborate. To understand how the different parts of the screen are composed, it's easier to associate the code with a diagram—which we'll look at shortly.

The **appBar** section is made of an AppBar Material Design-based Flutter object, which has **title** and **actions** properties.

The **actions** property, which is a **Widget** array, represents possible menu options that the widget can have—in our case, the Delete option.

In our case, we are doing something unconventional with the **actions** property—we are using a ternary conditional expression (**?**) and assigning an empty **Widget** array (**Widget[]**) if the **title** variable (**ttl**) contains an empty **String** value.

An empty **ttl** indicates that we are working on a new document and are not editing an existing document. For a new document, there is no need to have a Delete option, as obviously there's nothing to delete—the document hasn't been created yet.

For an existing document that is already present in the database, it is then logical to have the Delete menu option available, as we can then opt to remove the document from the database if we don't need the document any longer, if its data is incorrect, or if it has expired.

www.dbooks.org

So, when **`ttl`** is not empty (the document exists in the database), we assign to the **`appBar`** property a **`Widget`** array (**`Widget[]`**), which contains a PopupMenuButton object that is responsible for building the Delete menu-option widget.

The **`PopupMenuButton`** has two properties that are being used: the **`onSelected`** property, which is assigned to the **`_selectMenu`** method that we previously wrote, and **`itemBuilder`**, which is responsible for building the menu options and adding them to the **`appBar`**.

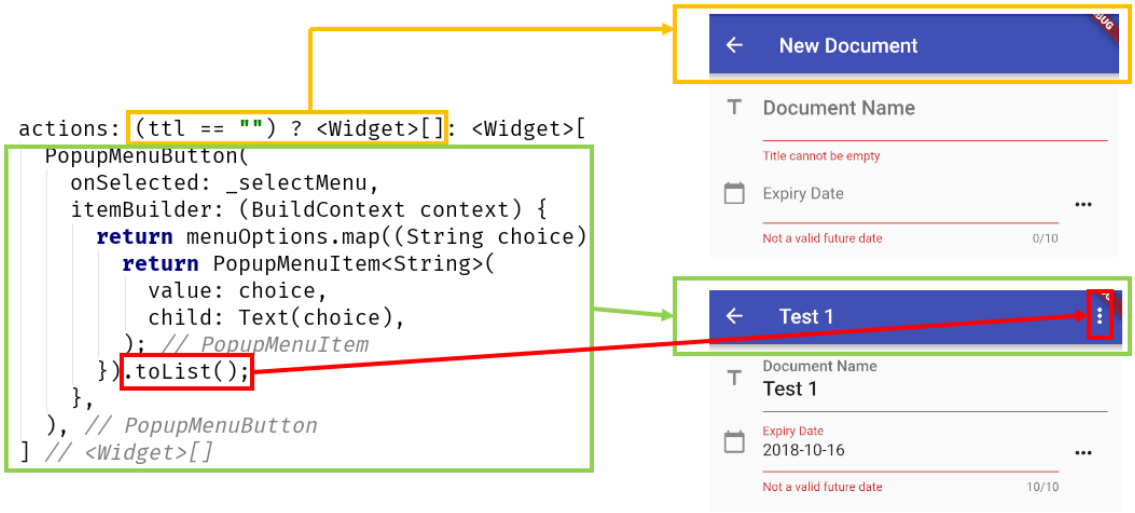


Figure 3-I: The appBar actions property

From the preceding diagram, we can see that the actual menu is built by converting the result of executing the **`menuOptions.map`** method to a list using the **`toList`** method.

We can also see how each section of the **`actions`** property relates to what is seen on the screen when a new document is added, or an existing document is being edited.

So, **`itemBuilder`** is assigned to an anonymous function, which receives a **`BuildContext`** parameter and returns a mapped list of items contained within the **`menuOptions`** array we previously declared. Each individual menu option is an instance of PopupMenuItem.

Body

Let's now focus on the most extensive part of the **`Scaffold`** object, which is the **`body`** property. Given that the **`body`** code is quite long, we'll have to break it down into chunks. To understand this better, let's start off by looking at the following diagram.

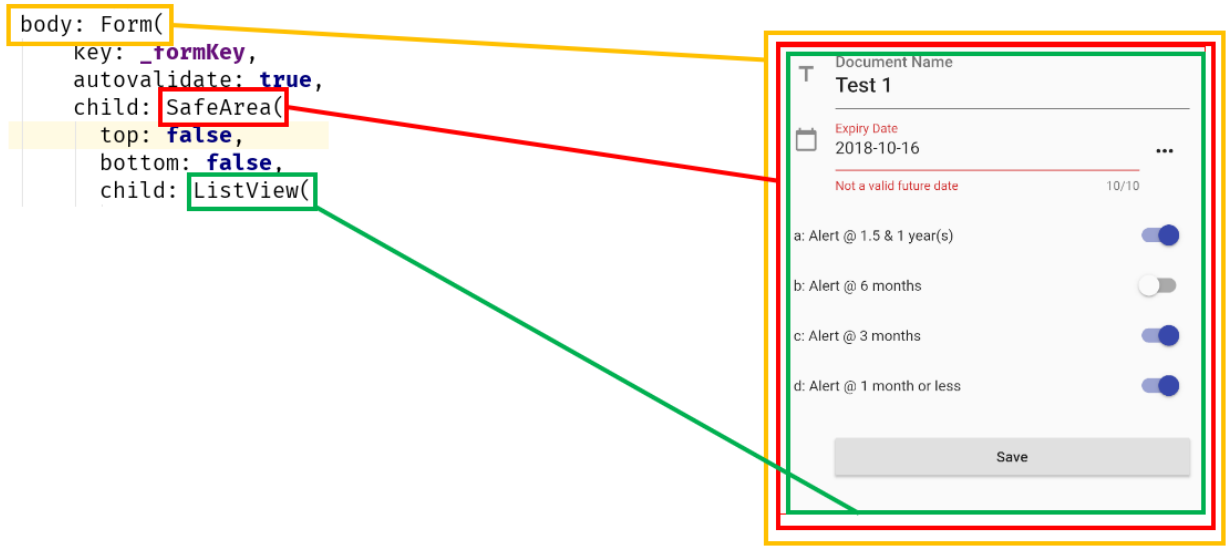


Figure 3-m: The main body section parts

We can see that the **body** is nothing more than a Flutter Form, which includes a SafeArea object as a **child**, which includes a ListView.

The reason for this nested layout is that we want to be able to have a list of widgets arranged one below the other. This is why we are using a **ListView** and have enough padding to avoid any visual intrusions from the device's operating system—which is why we are using a **SafeArea** as the **child** of the **Form**.

Notice that the key property of the **Form** object is assigned to the **_formKey** method that we previously wrote. The **autovalidate** property is set to **true**, which means that validation takes place when the form is submitted.

With this well-organized, top-level layout, we have a good foundation on which to render and organize the rest of the widgets that are seen on the screen.

Document Name and Expiry Date

Given that the **ListView** object will contain all the widgets seen on the screen, let's now explore how the **ListView** code is organized in more detail, by looking at the following diagram—this explains how the first two fields are composed.

www.dbooks.org

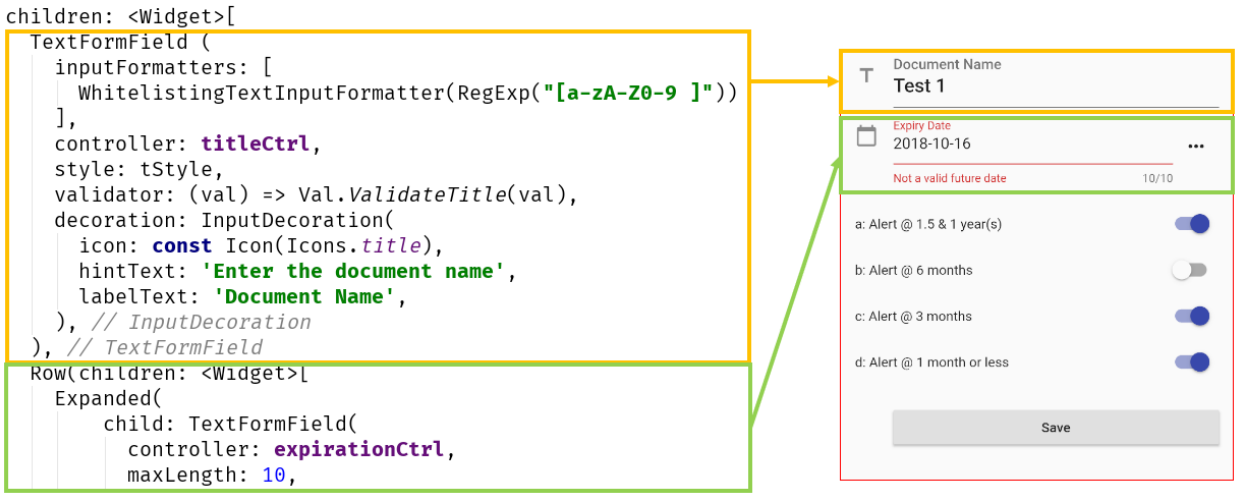


Figure 3-n: The first two data-entry widgets—Document Details screen

We can see that the first widget within the **`ListView`** is the text input for the **Document Name** field.

Document Name field

This field is of type TextFormField, and it is basically a FormField that contains a TextField object.

The **`TextFormField`** contains an **`inputFormatters`** array that includes a regular expression that specifies which characters can be typed within the field—these are assigned to the **`WhiteListingTextInputFormatter`** property.

The **`controller`** property is used for indicating that this **`TextFormField`** binds to the **`titleCtrl`** variable (which is a **`TextEditingController`**).

The **`validator`** property is used for running field validations; here the **`Val.ValidateTitle`** method is executed when the value of the field has changed.

The **`decoration`** property, as its name implies, basically sets the **`icon`**, hint (**`hintText`**), and label text (**`labelText`**) messages that are seen on the screen.

That wraps up the Document Name field—let's now talk about the field that follows, the Expiry Date.

Expiry Date field

The Expiry Date field is slightly more complex, as it renders as a **`TextFormField`** contained within an **`Expanded`** object, which is part of a **`Row`** object.

The reason for this composition is that the **`Row`** object also contains an **`IconButton`** object, which is used for displaying the Cupertino (iOS-styled) Date Picker widget that was previously explained.

To get a better understanding of how the Expiry Date field is composed, let's look at the following diagram.

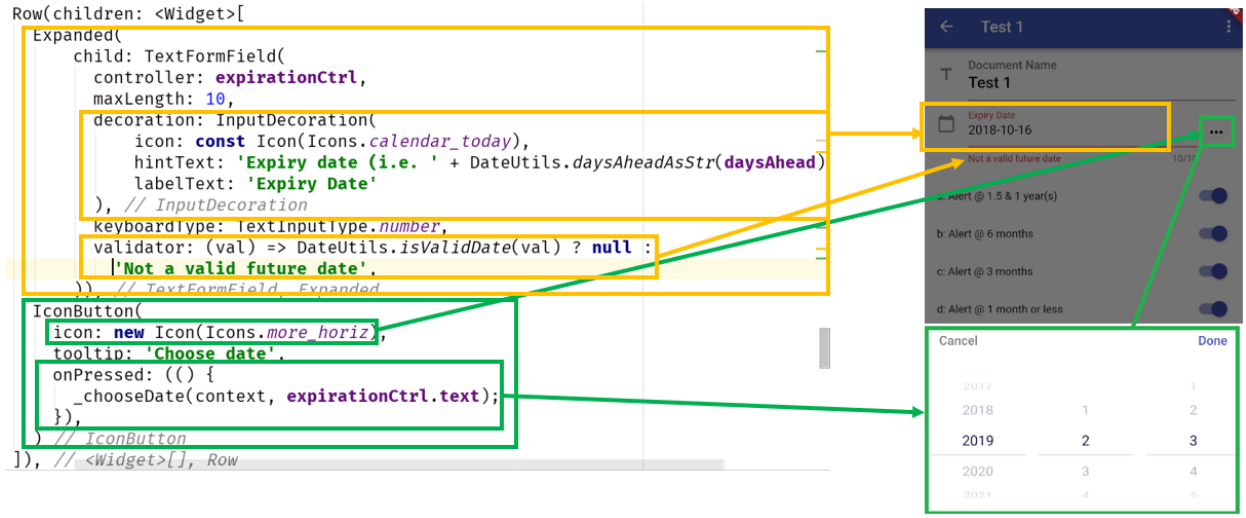


Figure 3-o: The Expiry Date field—Document Details screen

You can clearly see that the **`Expanded`** widget corresponds to the area where the text can be manually entered.

The **`decoration`** and **`validator`** properties of **`TextFormField`** work the same way as the Document Name field.

Notice though, how **`TextFormField`** binds to the **`expirationCtrl`** variable, and the maximum number of characters that can be entered is set to **`10`**—the **`maxLength`** property.

For the Expiry Date, the **`DateUtils.isValidDate`** method gets triggered when the field value changes and checks if the expiration date is a future date or not.

As for the **`IconButton`** object, you can clearly see in the diagram that its **`icon`** property renders the "**`...`**" button, and the **`onPressed`** event triggers the execution of the **`_chooseDate`** method, which we previously explored, and runs the iOS-styled Date Picker widget.

With the main two fields explored, let's move on to the alert fields of the form.

Alert fields

The way the alert fields are composed is almost the same for each one of them—the only difference is the text displayed on the screen and the variables to which they bind.

www.dbooks.org

Let's explore each one. Each alert field is composed within a **`Row`** object.

I have intentionally left an empty **`Row`** object before the first alert field—this is to give enough space on the screen between the Expiry Date field and the first alert field. We can see this as follows.

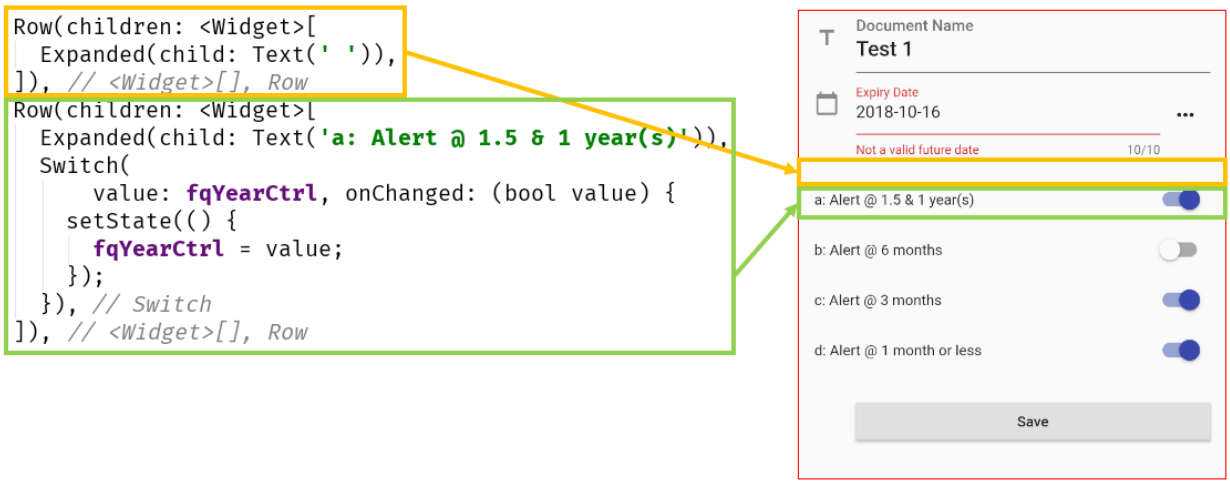


Figure 3-p: The empty row and first alert field—Document Details screen

Let's explore the details of how a **`Row`** object is composed. We can see that the empty **`Row`** object contains a **`children`** property, assigned to an **`Expanded`** object. The **`Expanded`** object has a **`child`** property that is assigned to a **`Text`** object containing an empty **`String`** value.

As for the non-empty **`Row`** object, we can clearly see that the **`Expanded`** object contains a **`Text child`** object with the caption that is displayed on the screen.

The `Switch` widget is used to toggle the on/off state of a single setting. The **`Switch`** widget is bound to the **`fqYearCtrl`** variable, which indicates that an alert would get triggered when there's one year remaining for the document to expire.

The **`Switch`** widget includes an **`onChanged`** event that gets triggered when its value changes (goes from on to off, or vice versa). When its **`value`** changes, the setState function is executed and notifies Flutter that the internal state of the object has changed; the changed **`value`** is assigned to **`fqYearCtrl`**.

As for the other alerts seen on the screen (**6 months**, **3 months**, and **1 month or less**), the code is the same as this one. The only difference is that each binds to a different variable—to **`fqHalfYearCtrl`**, **`fqQuarterCtrl`**, and **`fqMonthCtrl`**, respectively.

```
Row(children: <Widget>[
  Expanded(child: Text('d: Alert @ 1 month or less')),
  Switch(
      value: fqMonthCtrl, onChanged: (bool value) {
    setState(() {
      fqMonthCtrl = value;
    });
  }), // Switch
]), // <Widget>[], Row
```

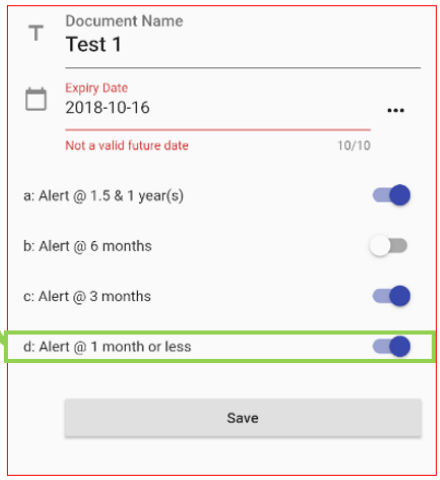


Figure 3-q: Last alert field—Document Details screen

That wraps up the alert fields—let's now finalize the form by exploring the Save button.

Save button

The Save button is the last part of the form that makes the Document Details screen. Let's explore how it is composed—to wrap up the Docdetail.dart code.

```
Container(
    padding: const EdgeInsets.only(left: 40.0, top: 20.0),
    child: RaisedButton(
      child: Text("Save"),
      onPressed: _submitForm,
    ) // RaisedButton
), // Container
```

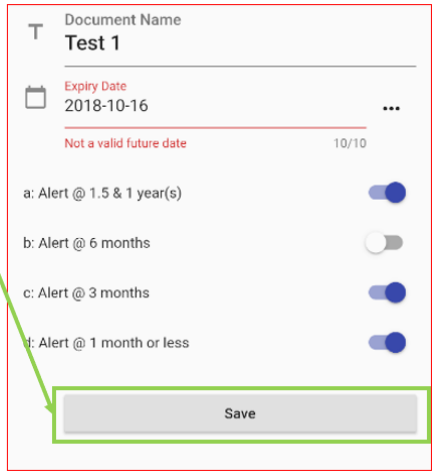


Figure 3-r: Save button—Document Details screen

As we can see, the Save button itself is wrapped around a Container widget. It helps combine common painting, positioning, and decoration of child widgets with **`padding`**.

The button is assigned to the **`child`** property of the **`Container`** widget, and it is a RaisedButton widget. The **`_submitForm`** event gets executed when the **`onPressed`** event occurs.

www.dbooks.org

Summary

This has been a long and challenging chapter, so if you've read this far—congrats! It's quite interesting what we've been able to achieve so far with Flutter, with a relatively little amount of code.

When I say it's not that much code, we necessarily must praise the awesome framework that the engineers at Google have created, which is Flutter.

As we've been able to see, the syntax feels familiar for any developer coming from a C or Java family-based language, and the constructs are easy to grasp and follow.

Something I really enjoy about Flutter is that the UI can be written within Dart without the need to use a markup language.

Normally, this is contrary to what most frameworks do, but with Flutter, it feels natural and enjoyable to create the UI using Dart code—which to me, was a very pleasant surprise.

We're almost at the end of our application, but we've got one final hurdle to overcome: creating the main screen and the logic behind it, which we will do in the next chapter.

Chapter 4 App UI—Main Screen

Quick intro

We're just one step away from finishing our Flutter application, which I'm excited about and looking forward to accomplishing. In this final step, we need to create the main screen of our application. This screen will display the list of documents and allow the user to add new ones. This is how the finished screen will look.

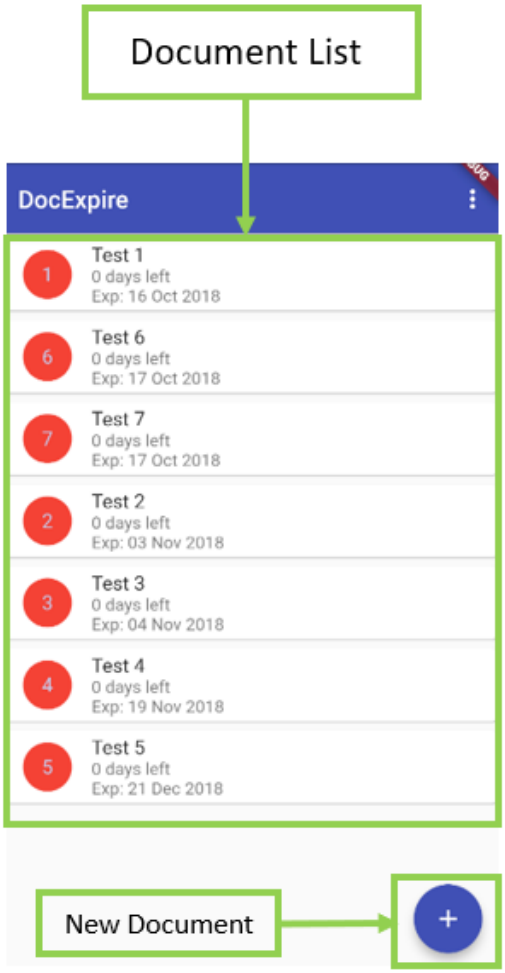


Figure 4-a: The main application screen

The code for the app's main screen will reside within the doclist.dart file, which you can create under the **lib\ui** subfolder of your app's main project folder.

www.dbooks.org

Getting started: main menu option

Just like we have a menu option on the Document Details screen—which allows us to delete an existing document—we need to have the option to remove all documents from the list and delete them from the database. This is known as Reset Local Data.

To add this option, we need to define a **`menuOptions`** array, just like we did with the details screen.

Before doing that, let's go ahead and reference all the modules and packages we'll need.

Code Listing 4-a: Creating the main menu—Doclist.dart

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

import '../model/model.dart';
import '../util/dbhelper.dart';
import '../util/utils.dart';
import './docdetail.dart';

// Menu item
const menuReset = "Reset Local Data";
List<String> menuOptions = const <String> [
  menuReset
];
```

In the first two lines of code, we **`import`** the Dart **`async`** package and the Material Design package that comes with the Flutter framework, which contains the UI widgets our application will use.

Next, we import the other project modules we created previously—this is because we'll need to reference various classes declared within those modules.

Finally, we create **`menuOptions`** as a generic **`List`** collection (created from an array), which includes only the **`menuReset`** option.

Main stateful widget

Just like we did within Docdetail.dart, we are ready to create the base widget that will be used within the app's main screen. This is going to be a stateful widget, which will describe the main screen's user interface.

As mentioned previously, a stateful widget has a state that can change, which is useful when parts of the user interface that are being rendered can change dynamically, given that the items on the list within the main screen can vary.

Let's define the stateful widget that we will use for rendering the UI of the main screen.

Code Listing 4-b: Stateful widget—main screen

```
// Previous code…

class DocList extends StatefulWidget {
  @override
  State<StatefulWidget> createState() => DocListState();
}
```

As we can see, it's very simple—all we do is **override** the **createState** method, which will create an instance of the **DocListState** class.

The **DocListState** class is where we are going to have the logic that creates and manipulates the main screen's UI. Let's explore that now.

Code Listing 4-c: Stateful widget—main screen

```
// Previous code…

class DocListState extends State<DocList> {
  DbHelper dbh = DbHelper();
  List<Doc> docs;
  int count = 0;
  DateTime cDate;

  // More code to follow...
}
```

The **DocListState** class will not only be responsible for creating the main screen's UI, but also for handling its state.

To be able to do that, it's important to keep track of a few things, such as the list of documents, represented by **docs**; the number of documents, represented by **count**; a reference to the database, represented by **dbh**; and the current datetime, represented by **cDate**.

Now that we know which variables are going to keep track of the main screen's state, let's now initialize it. We can do this as follows.

Code Listing 4-d: State initialization—main screen

```
// Previous code…

class DocListState extends State<DocList> {
  // Previous code

  @override
```


www.dbooks.org

```
  void initState() {
    super.initState();
  }
}
```

All we are doing is overriding the implementation of the **initState** method, and within it, invoking the **initState** method from the inherited **State<DocList>** class.

Awesome—we now have the foundation of our **DocListState** class laid out. Let's move on to a more interesting aspect, which is retrieving the data needed to populate the list of documents.

Getting a list of documents

Possibly the most important aspect of the application—at least from a usage point of view—is retrieving the data needed to display the list of documents to expire or that have expired.

Let's explore the full code of the method that makes this happen.

Code Listing 4-e: Retrieving the data—main screen

```
// Previous code…

class DocListState extends State<DocList> {
  // Previous code

  Future getData() async {
    final dbFuture = dbh.initializeDb();
    dbFuture.then(
      // result here is the actual reference to the database object.
      (result) {
          final docsFuture = dbh.getDocs();
          docsFuture.then(
            // result here is the list of docs in the database.
            (result) {
                if (result.length >= 0) {
                  List<Doc> docList = List<Doc>();
                  var count = result.length;
                  for (int i = 0; i <= count - 1; i++) {
                    docList.add(Doc.fromOject(result[i]));
                  }
                  setState(() {
                    if (this.docs.length > 0) {
                      this.docs.clear();
                    }

                    this.docs = docList;
```

```
                    this.count = count;
                  });
                }
              });
        });
  }
}
```

What is going on here? The first important aspect to consider is that the **getData** method is **async** (runs asynchronously), and it returns a Future object, which is the preferred way of handling data requests. This is done to avoid blocking the application when performing a computation that might be delayed.

For those familiar with JavaScript web development, think of a **Future** as a construct very similar to a Promise.

The first instruction within **getData** is to call the **initializeDb** method from the database helper class. This ensures that the connection with the database is established.

```
final dbFuture = dbh.initializeDb();
```

When the connection to the database is established, the **then** method of the **Future** object returned is executed.

```
dbFuture.then(…)
```

The result of that database connection is passed as a parameter—**(result)**— to an anonymous function that is executed within the scope of the **then** method of the **Future** object returned—**dbFuture**.

```
(result) {…}
```

Within this anonymous method scope, the next thing we do is retrieve the list of documents from the database—this is done by invoking the **getDocs** method, which also returns a **Future** object.

```
final docsFuture = dbh.getDocs();
```

When the retrieval of the list of documents has been finalized, the **then** method of the **Future** object returned is executed.

```
docsFuture.then(…)
```

The result of that action—**(result)**—is the list of documents, which is passed as a parameter to an anonymous function that is executed within the scope of the **then** method of the **Future** object returned—**docsFuture**.

```
(result) {…}
```

Within that anonymous function scope, we first check the **length** of the **result** returned, which indicates the number of documents in the database.

```
if (result.length >= 0) {…}
```

We then create a list of documents using the **Doc** class, which we will use to populate with the document data retrieved from the database.

```
List<Doc> docList = List<Doc>();
```

We also initialize the **count** variable to the value of **result.length**, which indicates how many documents were retrieved from the database.

```
var count = result.length;
```

Then, we loop through the **result** object obtained from the database—where each iteration represents a database row—and then convert each row into a **Doc** object, which we add to **docList**.

```
for (int i = 0; i <= count - 1; i++) {

  docList.add(Doc.fromOject(result[i]));

}
```

With the database rows converted to a **List<Doc>** object, we need to change the state of our **DocListState** class so that the UI can be rendered. We do this by calling the **setState** method.

```
setState(() {…} )
```

Within **setState**, the first thing we do is empty the contents of **this.docs**, which is one of the variables we declared at the beginning of the **DocListState** class.

```
if (this.docs.length > 0) {

  this.docs.clear();

}
```

Then, to **this.count**—which was also declared at the beginning of the **DocListState** class—we assign the value of count, obtained from **result.length**.

```
this.docs = docList;

this.count = count;
```

As you can see, the logic behind **getData** is quite simple and easy to follow once you understand the concept of **Future** objects in Dart, and how results are returned.

Checking dates

Another intrinsic aspect of our application is checking dates, which is important because the goal of the app is to help us keep track of important documents before they expire.

Let's have a look at the following function, which periodically checks for date and time discrepancies, and if there are any (for example, the phone's date-time is different the current date-time), executes the **getDate** method.

This way, we can have the latest document data and expiration dates. Here is the function's code.

Code Listing 4-f: Check date—main screen

```
// Previous code…

class DocListState extends State<DocList> {
  // Previous code

  void _checkDate() {
    const secs = const Duration(seconds:10);

    new Timer.periodic(secs, (Timer t) {
      DateTime nw = DateTime.now();

      if (cDate.day != nw.day || cDate.month != nw.month ||
        cDate.year != nw.year) {
        getData();
        cDate = DateTime.now();
      }
    });
  }

  // More code will follow…
}
```

As we can see, the code is quite straightforward. It creates a **Timer** object that executes every 10 seconds. The timer's execution is performed by invoking the **periodic** method. Within this method, the **cDate** object represents the date-time from the moment the main screen widget was created (first rendered), and the current date-time is represented by the **nw** object.

If there's a difference between the **day**, **month**, or **year** of both **DateTime** objects, then the **getData** method is invoked.

www.dbooks.org

Normally there shouldn't be a difference between the **day**, **month**, or **year** of when the main screen widget was rendered (**cDate**) and the date-time of the **nw** object. This is because shortly after the screen is rendered, the current date-time should be calculated, and its value assigned to **nw**.

However, there might be cases where the app runs for the first time on a phone that has the wrong date-time settings, and this would result in a wrong computation of the expiry dates for each document. So, the execution of the **_checkDate** function—after the phone date-time settings have been adjusted—would force a correct computation of the expiry date for each document.

Navigating to the document details

Given that the app's main screen contains a list of the documents that are in the database, if we would like to edit or delete a specific document, we would need to navigate to it. To do this, we need to define a **navigateToDetail** method. Let's see how it looks.

Code Listing 4-g: Navigating to a document—main screen

```
// Previous code…

class DocListState extends State<DocList> {
  // Previous code

  void navigateToDetail(Doc doc) async {
    bool r = await Navigator.push(context,
      MaterialPageRoute(builder: (context) => DocDetail(doc))
    );

    if (r == true) {
      getData();
    }
  }

  // More code will follow…
}
```

The navigation works by displaying the details of the document tapped from the document list. This is achieved by using the Navigator.push method and passing a **DocDetail** instance of the document selected.

If the result (**r**) of the **push** method is **true**—which means that the document has been modified—then the **getData** method is called to retrieve the latest document information from the database.

Resetting the local data

Given that we want to have the option to delete all the data stored by the app, we need to add some code that will remove any data stored locally.

Before any removal operations take place, we need to be able ask the user for confirmation, and if it is confirmed, then we can remove the data stored in the database. Let's have a look at the code.

Code Listing 4-h: Resetting the local data—main screen

```
class DocListState extends State<DocList> {

  // Previous code

  void _showResetDialog() {

    showDialog(

      context: context,

      builder: (BuildContext context) {

        return AlertDialog(

          title: new Text("Reset"),

          content: new Text("Do you want to delete all local data?"),

          actions: <Widget>[

            FlatButton(

              child: new Text("Cancel"),

              onPressed: () {

                Navigator.of(context).pop();

              },

            ),

            FlatButton(

              child: new Text("OK"),
```


www.dbooks.org

```
              onPressed: () {
                Future f = _resetLocalData();
                f.then(
                  (result) {
                    Navigator.of(context).pop();
                  }
                );
              },
            ),
          ],
        );
      },
    );
  }


  // More code will follow…
}
```

Let's have a look at what we are doing here. The **_showResetDialog** simply invokes the Flutter **showDialog** method.

The **builder** property of the **showDialog** method is assigned to an anonymous method, which receives a **BuildContext** parameter and returns an AlertDialog instance.

The **AlertDialog** object contains **title**, **content**, and **actions** properties. The **title** and **content** properties are self-explanatory, but the really interesting part is what happens within the **actions** property.

The **actions** property is assigned to an array of type Widget, which contains two FlatButton objects. The first button represents the Cancel option, and the second one (OK) represents the reset data option.

Both buttons have a **child** property that contains the **Text** object displayed on each. They also contain an **onPressed** event that triggers specific functionality.

In the case of the Cancel button, the **onPressed** event executes code that returns the focus to the app's main screen. This is done by invoking the **pop** method.

As for the OK button, the **onPressed** event makes a call to the **_resetLocalData** method—which is responsible for removing the data from the database—and then returns the control to the app's main screen, which is also done by invoking the **pop** method.

Let's have a look at the logic behind the **_resetLocalData** method.

Code Listing 4-i: Resetting the local data (2)—main screen

```
// Previous code…

class DocListState extends State<DocList> {
  // Previous code

  Future _resetLocalData() async {
    final dbFuture = dbh.initializeDb();
    dbFuture.then(
      (result) {
        final dDocs = dbh.deleteRows(DbHelper.tblDocs);
        dDocs.then(
          (result) {
            setState(() {
              this.docs.clear();
              this.count = 0;
            });
          }
        );
      }
    );
  }

  // More code will follow…
}
```

A call is made to the **initializeDb** method, which establishes the connection to the database.

When the connection is established, and within the anonymous function invoked within the **then** method of **dbFuture**, the **deleteRows** method is called, which is responsible for deleting the rows on the database.

The **setState** function is called to reset the list of documents—**this.docs**—and set the number of documents to zero—**this.count**.

Selecting the menu option

Now that we've seen how we can reset the information stored in the database, let's check how we can trigger this functionality manually by selecting the menu option associated with it. Here's the code.

Code Listing 4-j: Selecting the menu option—main screen

```
// Previous code…

class DocListState extends State<DocList> {
  // Previous code

  void _selectMenu(String value) async {
    switch (value) {
      case menuReset:
        _showResetDialog();
    }
  }

  // More code will follow…
}
```

We have a **_selectMenu** method that includes a **switch** statement with a condition when the **menuReset** option has been selected. When that happens, the **_showResetDialog** method is invoked—super simple.

I decided to use a **switch** statement because if later there's a need to add additional menu options, all that would be required would be to add additional **case** expressions to it.

Building the list of documents

We are now reaching one of the most interesting, useful, and important parts of our application: building the list of documents that are presented on the app's main screen. Let's look at the complete code for this and dissect it, piece by piece.

Code Listing 4-k: Creating the list of documents—main screen

```
// Previous code…

class DocListState extends State<DocList> {
  // Previous code
```

```
  ListView docListItems() {
    return ListView.builder(
      itemCount: count,
      itemBuilder: (BuildContext context, int position) {
        String dd = Val.GetExpiryStr(this.docs[position].expiration);
        String dl = (dd != "1") ? " days left" : " day left";
        return Card(
          color: Colors.white,
          elevation: 1.0,
          child: ListTile(
            leading: CircleAvatar(
              backgroundColor:
              (Val.GetExpiryStr(this.docs[position].expiration) != "0") ?
              Colors.blue : Colors.red,
              child: Text(
                this.docs[position].id.toString(),
              ),
            ),
            title: Text(this.docs[position].title),
            subtitle: Text(
                Val.GetExpiryStr(this.docs[position].expiration) + dl +
                    "\nExp: " + DateUtils.convertToDateFull(
                    this.docs[position].expiration)),
            onTap: () {
              navigateToDetail(this.docs[position]);
            },
          ),
        );
      },
    );
  }

  // More code will follow…
}
```

To understand this better, let's go over each part of the code. The first thing that is being done within the **`docListItems`** method is a returning of a ListView object that will contain the list of documents that the app is going to display.

To create that list, we need to call the **`ListView.builder`** method, which has two important properties: **`itemCount`** and **`itemBuilder`**.

The **`itemCount`** property indicates how many documents are going to be added to the list—notice how we are assigning the value of the **`count`** variable that was retrieved from the database.

The **`itemBuilder`** property is the one used to build the document list; this is achieved with an anonymous function that receives a BuildContext object as one of its parameters.

www.dbooks.org

Let's explore the content of the anonymous function assigned to the **itemBuilder** property—this is where things get interesting.

On the first two lines of the anonymous function, all we are doing is getting the expiry date of each document as a **String**—this is done by invoking the **GetExpiryStr** function—and then determining the remaining days left (that will be displayed on the screen).

```
String dd = Val.GetExpiryStr(this.docs[position].expiration);

String dl = (dd != "1") ? " days left" : " day left";
```

Notice that the parameter **position** indicates the current document being added to the list.

With the remaining days determined, the next thing we do is return a Card object, which will contain the document details. Each document is displayed within its own **Card** object.

To understand better the composition of the **Card** object, let's look at the following diagram.

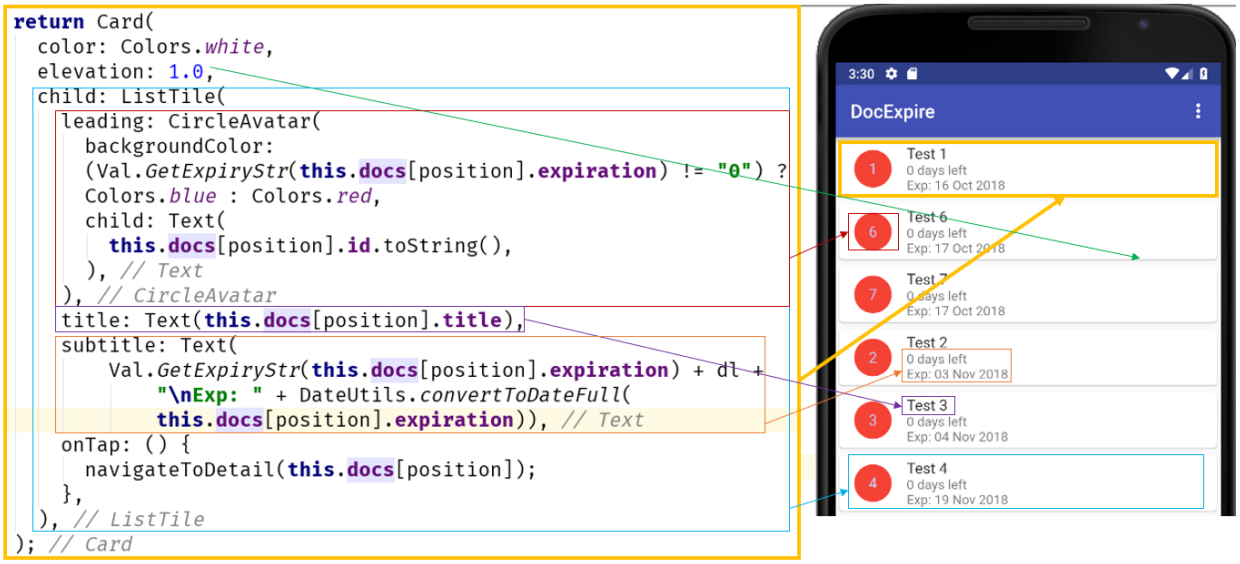


Figure 4-b: Card objects within the document list

The first two properties of the **Card** object are self-descriptive. The first indicates the **color** used for the background of the **Card**, in this case **white**.

The second indicates whether the **Card** object has a slight visual elevation with respect to the document list, which is seen as a thin, gray line below each **Card** object.

The third property of the **Card** object is its **child** property, to which we assign a ListTile object. So basically, the actual content of the **Card** object is determined by a **ListTile** object.

The **ListTile** object contains a CircleAvatar object that is red when the document has expired, and blue when the document has not expired. The **backgroundColor** of the **CircleAvatar** object is determined by the following ternary conditional expression.

```
(Val.GetExpiryStr(this.docs[position].expiration) != "0") ?

Colors.blue : Colors.red
```

The **Text** (number) contained within the **CircleAvatar** object indicates the **position** (order) of the document within the database.

Next, within the **ListTile** property we have the **title** property, which indicates the name of the document.

Following that, there's the **subtitle** property, which displays the remaining days a document before it expires.

```
Text(Val.GetExpiryStr(this.docs[position].expiration) + dl +

"\nExp: " + DateUtils.convertToDateFull(

this.docs[position].expiration))
```

Finally, the **ListTile** object has an event that gets triggered when a user taps on the document, the **onTap** event, which basically invokes the **navigateToDetail** method. This method opens the details of the document selected, which can then be edited.

```
onTap: () {

  navigateToDetail(this.docs[position]);

}
```

Finalizing the main screen

The biggest chunk of our main screen, the document list, is finished. However, we still need to wrap that document list around the main **Scaffold** object that will hold it, and then **Stack** it properly so that it displays correctly during runtime (both in portrait and landscape modes).

We'll also need to add an **AppBar** object and link our menu option to it. Let's go ahead and do all this.

Code Listing 4-l: Finishing the main screen

```
// Previous code…
```


www.dbooks.org

```
class DocListState extends State<DocList> {
  // Previous code

  @override
  Widget build(BuildContext context) {
    this.cDate = DateTime.now();

    if (this.docs == null) {
      this.docs = List<Doc>();
      getData();
    }

    _checkDate();

    return Scaffold(
      resizeToAvoidBottomPadding: false,
      appBar: AppBar(
          title: Text("DocExpire"),
          actions: <Widget>[
            PopupMenuButton(
              onSelected: _selectMenu,
              itemBuilder: (BuildContext context) {
                return menuOptions.map((String choice) {
                  return PopupMenuItem<String>(
                    value: choice,
                    child: Text(choice),
                  );
                }).toList();
              },
            ),
          ]
      ),
      body: Center(
        child: Scaffold(
          body: Stack(
              children: <Widget>[
                docListItems(),
              ]),
          floatingActionButton: FloatingActionButton(
            onPressed: () {
              navigateToDetail(Doc.withId(-1, "", "", 1, 1, 1, 1));
            },
            tooltip: "Add new doc",
            child: Icon(Icons.add),
          ),
        ),
      ),
    );
  }
```

```
}
```

Just like we did with the Document Details screen, to render its content we need to **override** the **build** method inherited from the **DocListState** class. Let's explore each part of the **build** method so we can understand exactly what it does.

The first thing that happens within the **build** method is that we get the current date-time, which will represent the date-time when the main screen is rendered. This value is stored within the **cDate** variable we previously explained.

Next, if the document list doesn't exist yet—which means that no documents have ever been retrieved—then the list of documents is initialized, and a call to the **getData** method is made to check whether there are any documents within the database.

```
if (this.docs == null) {

  this.docs = List<Doc>();

  getData();

}
```

Following that, we invoke the **_checkDate** method. This is to check that the current date-time is aligned with the date-time the screen was rendered, checking that the amount of days left for each document is accurate.

Then, the **build** method returns a Scaffold object, which will render the main screen. The **Scaffold** object has two main properties that take most of the logic of the **build** method code: the **appBar** and the **body**.

The **appBar** property is assigned to an AppBar object, which basically constitutes the application's top navigation bar, including the app's name and the menu option. To understand it better, let's look at the following diagram.

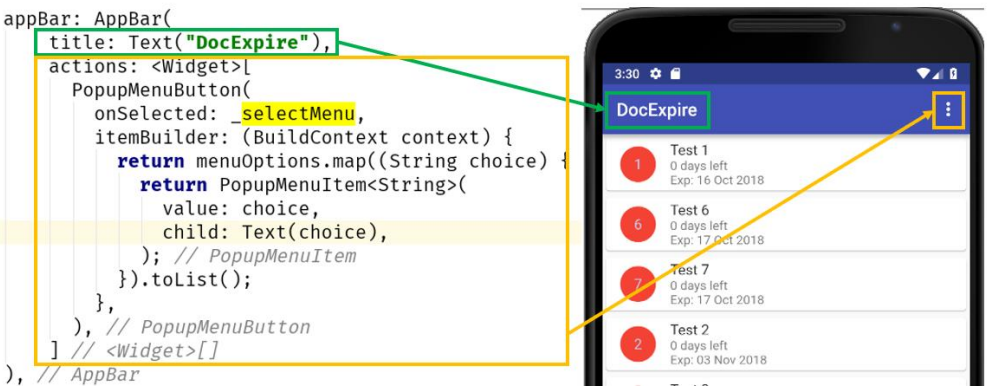


Figure 4-c: The main screen's AppBar

www.dbooks.org

As we can see, the **actions** property of the **AppBar** object is nothing more than an array of type **Widget**, which includes a PopupMenuButton object.

The **PopupMenuButton** object contains an **onSelected** property that is assigned to the **_selectMenu** method, which opens the **Reset Local Data** option.

The menu itself is built by an anonymous function that is assigned to the **itemBuilder** property and returns a **menuOptions** array object as a list, with each menu item being a **PopMenuItem** object (in our case, there's only one menu option).

By doing it this way, we could expand the application later and add extra menu options to the **menuOptions** array without having to modify any of the rendering functionality.

To better understand its composition of the **body** property, let's have a look at the following diagram.

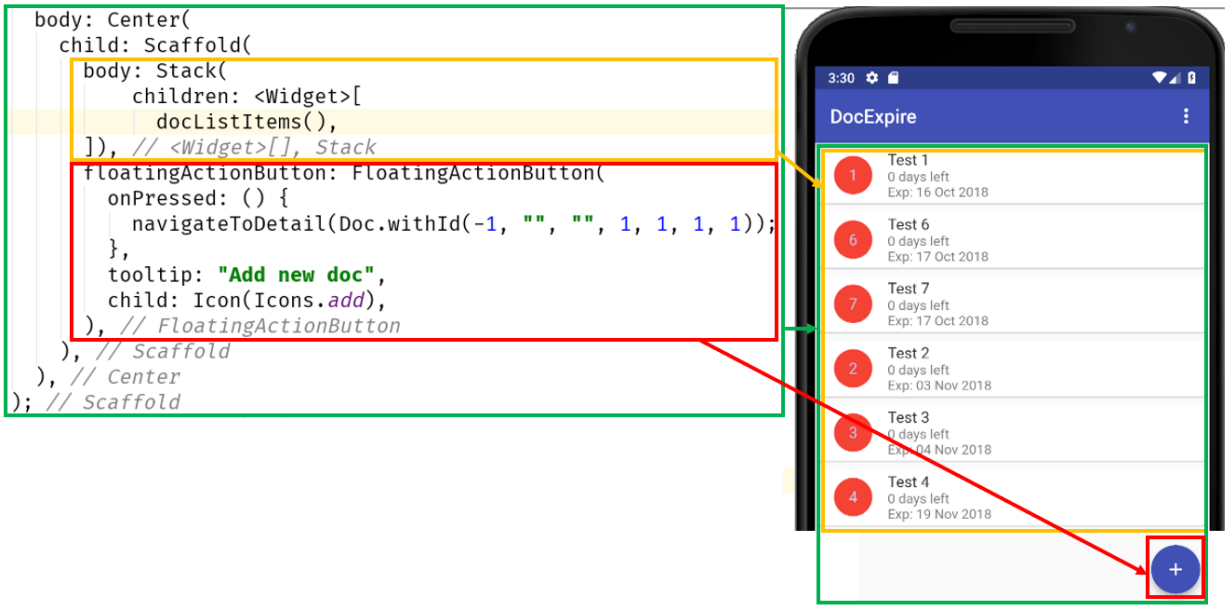


Figure 4-d: The main screen's body

We can see that the **Scaffold** object contains the document list and a floating button. The **Scaffold** object is wrapped within a Center object, so that all the content is properly centered in any rotation that the device is used.

The document list generated by the **docListItems** method is wrapped around a **Stack** widget, which is done to ensure that the list of documents is positioned correctly relative to the edges of its surrounding box.

The floating button used to create a new document is created by the FloatingActionButton object—assigned to the **floatingActionButton** property—and it contains an Icon object, a **tooltip** property, and an **onPressed** event. This event calls the **navigateToDetail** method that displays an empty Document Details screen when creating a new document.

That wraps up the app's main screen. If you recall, the main screen (represented by the **`DocList`** class) is invoked within main.dart.

If you have followed all the steps described, you should now be able to run the application from Android Studio (don't forget to select an emulator device).

A minute later (sometimes Android Studio takes a minute or so to resolve and update all the required dependencies), you'll be able to see the app running with no documents in it.

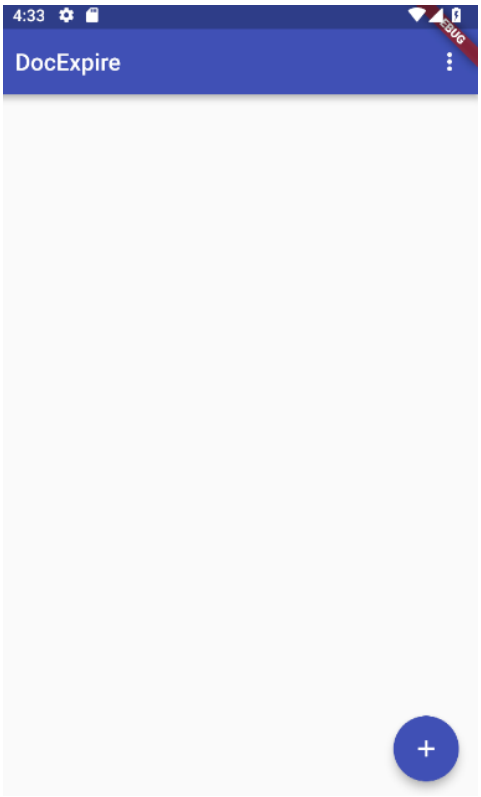


Figure 4-e: The finished app running (clean database)

The first time it runs, the app shows no documents because the database is empty, and just newly created. You should now be able to add your own documents.

I'm going to add a new test document with the following data. Let's have a look.

www.dbooks.org

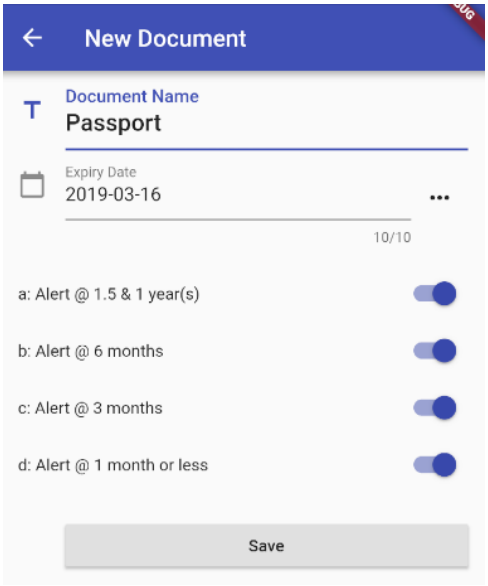


Figure 4-f: A new document

If we click **Save**, the document will appear on the app's main screen—we can see this in the following figure.

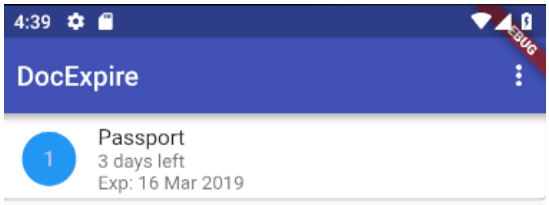


Figure 4-g: The new cocument zdded

Awesome—it's working as expected. Now, let's try the Reset Local Data menu option to see if the database is cleared.

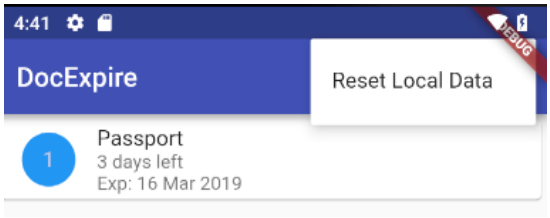


Figure 4-h: The reset cata option

If we tap on the **Reset Local Data** menu option, we should see the following dialog box.

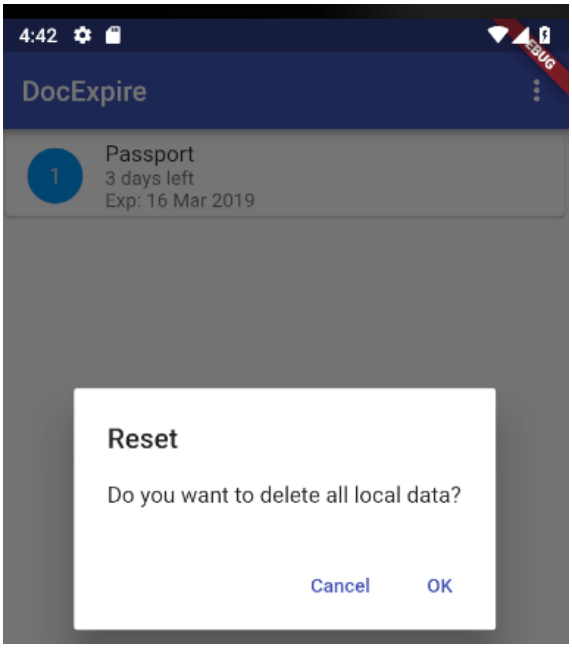


Figure 4-i: Confirming the reset data option

If we now tap **OK**, the contents of the database should be deleted and the document list should appear empty. Let's see if that's the case.

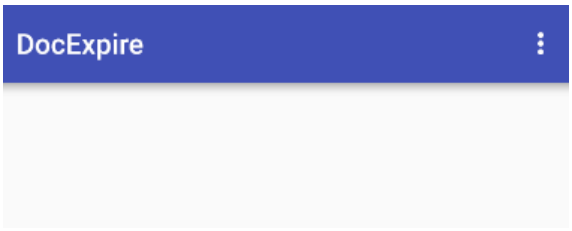


Figure 4-j: An empty document list

Awesome—the Reset Local Data menu option worked, and we have a fully working application.

You can find the full source code and Android Studio project for this application on the Syncfusion GitHub repository that comes along with this book. Alternatively, you may also get the full source code files here.

Summary

We've now explored the Flutter framework and built this cool application together—and although we've reached the end of the book, this is hopefully just the beginning of your journey with Flutter.

Going forward, I encourage you to expand the capabilities of this application and, if I may, I'd like to suggest you add a cool feature to this application that I couldn't cover in this book. That's

www.dbooks.org

the ability to have a mechanism in place that allows the app to keep multiple document lists, each associated with an email address, which can be synced to the cloud (i.e. using Google's Cloud Firestore database).

Say, for instance, that I'd like to keep a list of documents for myself, but I'd also like to keep a list of documents for my wife—with documents that are relevant to her.

If the app would give me the option to store two lists, each one associated with a different user (email address), I could keep track of multiple document lists and have them synced to the cloud in case my phone gets lost, damaged, or stolen.

This would also give me the capability of installing the app on a new phone, and being able to retrieve those document lists by using the username (email address) each was saved with—wouldn't that be cool?

Also, why not add the alerting mechanism for which we built the UI—this would also be something very useful.

So, there you go—a interesting challenge and problem to solve, which can be used to expand the code that has already been written for this application.

Before you go, don't forget to check the Appendix, where you can find the finished source code of each Dart file of the application.

I'm keen to see what you will build. Until next time, have fun with Flutter. Thanks so much for reading!

Appendix—Full Code

Full main.dart code

Here's the complete finished source code for our app's main.dart file.

Code Listing Appendix-a: Finished main.dart

```
import 'package:flutter/material.dart';
import './ui/doclist.dart';

void main() => runApp(DocExpiryApp());

class DocExpiryApp extends StatelessWidget
{
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'DocExpire',
      theme: new ThemeData(
        primarySwatch: Colors.indigo,
      ),
      home: DocList(),
    );
  }
}
```

Full utils.dart code

Here's the complete finished source code for our app's utils.dart file.

Code Listing Appendix-b: Finished utils.dart

```
import 'package:intl/intl.dart';

class Val {
  // Validations
  static String ValidateTitle(String val) {
    return (val != null && val != "") ? null : "Title cannot be empty";
  }

  static String GetExpiryStr(String expires) {
    var e = DateUtils.convertToDate(expires);
    var td = new DateTime.now();
```


www.dbooks.org

```
    Duration dif = e.difference(td);
    int dd = dif.inDays + 1;
    return (dd > 0) ? dd.toString() : "0";
  }

  static bool StrToBool(String str) {
    return (int.parse(str) > 0) ? true : false;
  }

  static bool IntToBool(int val) {
    return (val > 0) ? true : false;
  }

  static String BoolToStr(bool val) {
    return (val == true) ? "1" : "0";
  }

  static int BoolToInt(bool val) {
    return (val == true) ? 1 : 0;
  }
}

class DateUtils {
  static DateTime convertToDate(String input) {
    try
    {
      var d = new DateFormat("yyyy-MM-dd").parseStrict(input);
      return d;
    } catch (e) {
      return null;
    }
  }

  static String convertToDateFull(String input) {
    try
    {
      var d = new DateFormat("yyyy-MM-dd").parseStrict(input);
      var formatter = new DateFormat('dd MMM yyyy');
      return formatter.format(d);
    } catch (e) {
      return null;
    }
  }

  static String convertToDateFullDt(DateTime input) {
    try
    {
      var formatter = new DateFormat('dd MMM yyyy');
```

```
      return formatter.format(input);
    } catch (e) {
      return null;
    }
  }

  static bool isDate(String dt) {
    try
    {
      var d = new DateFormat("yyyy-MM-dd").parseStrict(dt);
      return true;
    } catch (e) {
      return false;
    }
  }

  static bool isValidDate(String dt) {
    if (dt.isEmpty || !dt.contains("-") || dt.length < 10) return false;

    List<String> dtItems = dt.split("-");
    var d = DateTime(int.parse(dtItems[0]),
        int.parse(dtItems[1]), int.parse(dtItems[2]));

    return d != null && isDate(dt) &&
        d.isAfter(new DateTime.now());
  }

  // String functions
  static String daysAheadAsStr(int daysAhead) {
    var now = new DateTime.now();
    DateTime ft = now.add(new Duration(days: daysAhead));
    return ftDateAsStr(ft);
  }

  static String ftDateAsStr(DateTime ft) {
    return ft.year.toString() + "-" +
        ft.month.toString().padLeft(2, "0") + "-" +
        ft.day.toString().padLeft(2, "0");
  }

  static String TrimDate(String dt) {
    if (dt.contains(" ")) {
      List<String> p = dt.split(" ");
      return p[0];
    }
    else
      return dt;
  }
}
```


www.dbooks.org

Full dbhelper.dart code

Here's the complete finished source code for our app's dbhelper.dart file.

Code Listing Appendix-c: Finished dbhelper.dart

```
import 'package:flutter/material.dart';
import 'package:sqflite/sqflite.dart';
import 'package:path_provider/path_provider.dart';

import 'dart:async';
import 'dart:io';

import '../model/model.dart';

class DbHelper {
  // Tables
  static String tblDocs = "docs";

  // Fields of the 'docs' table.
  String docId = "id";
  String docTitle = "title";
  String docExpiration = "expiration";

  String fqYear = "fqYear";
  String fqHalfYear = "fqHalfYear";
  String fqQuarter = "fqQuarter";
  String fqMonth = "fqMonth";

  // Singleton
  static final DbHelper _dbHelper = DbHelper._internal();

  DbHelper._internal();

  factory DbHelper() {
    return _dbHelper;
  }

  // Database entry point.
  static Database _db;

  Future<Database> get db async {
    if (_db == null) {
      _db = await initializeDb();
    }

    return _db;
  }
```

```
// Initialize the database.
Future<Database> initializeDb() async {
  Directory d = await getApplicationDocumentsDirectory();
  String p = d.path + "/docexpire.db";
  var db = await openDatabase(p, version: 1, onCreate: _createDb);
  return db;
}

// Create database tables.
void _createDb(Database db, int version) async {
  await db.execute(
  "CREATE TABLE $tblDocs($docId INTEGER PRIMARY KEY, $docTitle TEXT, "
  + "$docExpiration TEXT, " +
  "$fqYear INTEGER, $fqHalfYear INTEGER, $fqQuarter INTEGER, " +
  "$fqMonth INTEGER)"
  );
}

// Insert a new doc.
Future<int> insertDoc(Doc doc) async {
  var r;

  Database db = await this.db;
  try {
    r = await db.insert(tblDocs, doc.toMap());
  }
  catch (e) {
    debugPrint("insertDoc: " + e.toString());
  }
  return r;
}

// Get the list of docs.
Future<List> getDocs() async {
  Database db = await this.db;
  var r = await db.rawQuery(
    "SELECT * FROM $tblDocs ORDER BY $docExpiration ASC");
  return r;
}

// Gets a Doc based on the id.
Future<List> getDoc(int id) async {
  Database db = await this.db;
  var r = await db.rawQuery(
    "SELECT * FROM $tblDocs WHERE $docId = " + id.toString() + "" );
  return r;
}

// Gets a Doc based on a String payload.
```


www.dbooks.org

```
Future<List> getDocFromStr(String payload) async {
  List<String> p = payload.split("|");
  if (p.length == 2) {
    Database db = await this.db;
    var r = await db.rawQuery(
        "SELECT * FROM $tblDocs WHERE $docId = " + p[0] +
        " AND $docExpiration = '" + p[1] + "'" );
    return r;
  }
  else
    return null;
}

// Get the number of docs.
Future<int> getDocsCount() async {
  Database db = await this.db;
  var r = Sqflite.firstIntValue(
      await db.rawQuery("SELECT COUNT(*) FROM $tblDocs")
  );
  return r;
}

// Get the max document id available on the database.
Future<int> getMaxId() async {
  Database db = await this.db;
  var r = Sqflite.firstIntValue(
      await db.rawQuery("SELECT MAX(id) FROM $tblDocs")
  );
  return r;
}

// Update a doc.
Future<int> updateDoc(Doc doc) async {
  var db = await this.db;
  var r = await db.update(tblDocs, doc.toMap(),
      where: "$docId = ?", whereArgs: [doc.id]);
  return r;
}

// Delete a doc.
Future<int> deleteDoc(int id) async {
  var db = await this.db;
  int r = await db.rawDelete(
    "DELETE FROM $tblDocs WHERE $docId = $id");
  return r;
}

// Delete all rows.
Future<int> deleteRows(String tbl) async {
```

```
    var db = await this.db;
    int r = await db.rawDelete("DELETE FROM $tbl");
    return r;
  }
}
```

Full model.dart code

Here's the complete finished source code for our app's model.dart file.

Code Listing Appendix-d: Finished model.dart

```
import '../util/utils.dart';

class Doc
{
  int id;
  String title;
  String expiration;

  int fqYear;

  int fqHalfYear;
  int fqQuarter;
  int fqMonth;

  Doc(this.title, this.expiration, this.fqYear,
      this.fqHalfYear, this.fqQuarter, this.fqMonth);

  Doc.withId(this.id, this.title, this.expiration, this.fqYear,
      this.fqHalfYear, this.fqQuarter, this.fqMonth);

  Map<String, dynamic> toMap() {
    var map = Map<String, dynamic>();

    map["title"] = this.title;
    map["expiration"] = this.expiration;

    map["fqYear"] = this.fqYear;
    map["fqHalfYear"] = this.fqHalfYear;
    map["fqQuarter"] = this.fqQuarter;
    map["fqMonth"] = this.fqMonth;

    if (id != null) {
      map["id"] = id;
    }
```


www.dbooks.org

```
    return map;
  }

  Doc.fromOject(dynamic o) {
    this.id = o["id"];
    this.title = o["title"];
    this.expiration = DateUtils.TrimDate(o["expiration"]);

    this.fqYear = o["fqYear"];
    this.fqHalfYear = o["fqHalfYear"];
    this.fqQuarter = o["fqQuarter"];
    this.fqMonth = o["fqMonth"];
  }
}
```

Full docdetail.dart code

Here's the complete finished source code for our app's docdetail.dart file.

Code Listing Appendix-e: Finished docdetail.dart

```
import 'dart:async';

import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:flutter_masked_text/flutter_masked_text.dart';
import 'package:flutter_datetime_picker/flutter_datetime_picker.dart';

import '../model/model.dart';
import '../util/utils.dart';
import '../util/dbhelper.dart';

// Menu item
const menuDelete = "Delete";
final List<String> menuOptions = const <String> [
  menuDelete
];

class DocDetail extends StatefulWidget {
  Doc doc;
  final DbHelper dbh = DbHelper();

  DocDetail(this.doc);

  @override
  State<StatefulWidget> createState() => DocDetailState();
}
```

```
class DocDetailState extends State<DocDetail> {
  final GlobalKey<FormState> _formKey = new GlobalKey<FormState>();
  final GlobalKey<ScaffoldState> _scaffoldKey =
    new GlobalKey<ScaffoldState>();

  final int daysAhead = 5475; // 15 years in the future

  final TextEditingController titleCtrl = TextEditingController();
  final TextEditingController expirationCtrl =
    MaskedTextController(mask: '2000-00-00');

  bool fqYearCtrl = true;
  bool fqHalfYearCtrl = true;
  bool fqQuarterCtrl = true;
  bool fqMonthCtrl = true;
  bool fqLessMonthCtrl = true;

  // Initialization code
  void _initCtrls() {
    titleCtrl.text = widget.doc.title != null ? widget.doc.title : "";
    expirationCtrl.text =
      widget.doc.expiration != null ? widget.doc.expiration : "";

    fqYearCtrl = widget.doc.fqYear != null ?
      Val.IntToBool(widget.doc.fqYear) : false;
    fqHalfYearCtrl = widget.doc.fqHalfYear != null ?
      Val.IntToBool(widget.doc.fqHalfYear) : false;
    fqQuarterCtrl = widget.doc.fqQuarter != null ?
      Val.IntToBool(widget.doc.fqQuarter) : false;
    fqMonthCtrl = widget.doc.fqMonth != null ?
      Val.IntToBool(widget.doc.fqMonth) : false;
  }

  // Date Picker & Date functions
  Future _chooseDate(BuildContext context, String initialDateString)
  async {
    var now = new DateTime.now();
    var initialDate = DateUtils.convertToDate(initialDateString) ?? now;

    initialDate = (initialDate.year >= now.year &&
    initialDate.isAfter(now) ? initialDate : now);

    DatePicker.showDatePicker(context, showTitleActions: true,
        onConfirm: (date) {
          setState(() {
            DateTime dt = date;
            String r = DateUtils.ftDateAsStr(dt);
            expirationCtrl.text = r;
```


www.dbooks.org

```
        });
      },
      currentTime: initialDate);
}

// Upper Menu
void _selectMenu(String value) async {
  switch (value) {
    case menuDelete:
      if (widget.doc.id == -1) {
        return;
      }
      await _deleteDoc(widget.doc.id);
  }
}

// Delete doc
void _deleteDoc(int id) async {
  int r = await widget.dbh.deleteDoc(widget.doc.id);
  Navigator.pop(context, true);
}

// Save doc
void _saveDoc () {
  widget.doc.title = titleCtrl.text;
  widget.doc.expiration = expirationCtrl.text;

  widget.doc.fqYear = Val.BoolToInt(fqYearCtrl);
  widget.doc.fqHalfYear = Val.BoolToInt(fqHalfYearCtrl);
  widget.doc.fqQuarter = Val.BoolToInt(fqQuarterCtrl);
  widget.doc.fqMonth = Val.BoolToInt(fqMonthCtrl);

  if (widget.doc.id > -1) {
    debugPrint("_update->Doc Id: " + widget.doc.id.toString());
    widget.dbh.updateDoc(widget.doc);
    Navigator.pop(context, true);
  }
  else {
    Future<int> idd = widget.dbh.getMaxId();
    idd.then((result) {
      debugPrint("_insert->Doc Id: " + widget.doc.id.toString());
      widget.doc.id = (result != null) ? result + 1 : 1;
      widget.dbh.insertDoc(widget.doc);
      Navigator.pop(context, true);
    });
  }
}

// Submit form
```

```
void _submitForm() {
  final FormState form = _formKey.currentState;

  if (!form.validate()) {
    showMessage('Some data is invalid. Please correct.');
  } else {
    _saveDoc();
  }
}

void showMessage(String message, [MaterialColor color = Colors.red]) {
  _scaffoldKey.currentState.showSnackBar(
  new SnackBar(backgroundColor: color, content: new Text(message)));
}

@override
void initState() {
  super.initState();
  _initCtrls();
}

@override
Widget build(BuildContext context) {
  const String cStrDays = "Enter a number of days";
  TextStyle tStyle = Theme.of(context).textTheme.title;
  String ttl = widget.doc.title;

  return Scaffold(
      key: _scaffoldKey,
      resizeToAvoidBottomPadding: false,
      appBar: AppBar(
          title: Text(ttl != "" ? widget.doc.title : "New Document"),
          actions: (ttl == "") ? <Widget>[]: <Widget>[
            PopupMenuButton(
              onSelected: _selectMenu,
              itemBuilder: (BuildContext context) {
                return menuOptions.map((String choice) {
                  return PopupMenuItem<String>(
                    value: choice,
                    child: Text(choice),
                  );
                }).toList();
              },
            ),
          ]
      ),
      body: Form(
          key: _formKey,
          autovalidate: true,
```


www.dbooks.org

```
          child: SafeArea(
            top: false,
            bottom: false,
            child: ListView(
              padding: const EdgeInsets.symmetric(horizontal: 16.0),
              children: <Widget>[
                TextFormField (
                  inputFormatters: [
                    WhitelistingTextInputFormatter(
                      RegExp("[a-zA-Z0-9 ]"))
                  ],
                  controller: titleCtrl,
                  style: tStyle,
                  validator: (val) => Val.ValidateTitle(val),
                  decoration: InputDecoration(
                    icon: const Icon(Icons.title),
                    hintText: 'Enter the document name',
                    labelText: 'Document Name',
                  ),
                ),
                Row(children: <Widget>[
                  Expanded(
                      child: TextFormField(
                        controller: expirationCtrl,
                        maxLength: 10,
                        decoration: InputDecoration(
                            icon: const Icon(Icons.calendar_today),
                            hintText: 'Expiry date (i.e. ' +
                             DateUtils.daysAheadAsStr(daysAhead) + ')',
                            labelText: 'Expiry Date'
                        ),
                        keyboardType: TextInputType.number,
                        validator: (val) => DateUtils.isValidDate(val)
                           ? null :
                          'Not a valid future date',
                      )),
                  IconButton(
                    icon: new Icon(Icons.more_horiz),
                    tooltip: 'Choose date',
                    onPressed: (() {
                      _chooseDate(context, expirationCtrl.text);
                    }),
                  )
                ]),
                Row(children: <Widget>[
                  Expanded(child: Text(' ')),
                ]),
                Row(children: <Widget>[
                  Expanded(child: Text('a: Alert @ 1.5 & 1 year(s)')),
```

```
                  Switch(
                      value: fqYearCtrl, onChanged: (bool value) {
                    setState(() {
                      fqYearCtrl = value;
                    });
                  }),
                ]),
                Row(children: <Widget>[
                  Expanded(child: Text('b: Alert @ 6 months')),
                  Switch(
                      value: fqHalfYearCtrl, onChanged: (bool value) {
                    setState(() {
                      fqHalfYearCtrl = value;
                    });
                  }),
                ]),
                Row(children: <Widget>[
                  Expanded(child: Text('c: Alert @ 3 months')),
                  Switch(
                      value: fqQuarterCtrl, onChanged: (bool value) {
                    setState(() {
                      fqQuarterCtrl = value;
                    });
                  }),
                ]),
                Row(children: <Widget>[
                  Expanded(child: Text('d: Alert @ 1 month or less')),
                  Switch(
                      value: fqMonthCtrl, onChanged: (bool value) {
                    setState(() {
                      fqMonthCtrl = value;
                    });
                  }),
                ]),
                Container(
                    padding:
                      const EdgeInsets.only(left: 40.0, top: 20.0),
                    child: RaisedButton(
                      child: Text("Save"),
                      onPressed: _submitForm,
                    )
                ),
              ],
            ),
          )));
  }
}
```


www.dbooks.org

Full doclist.dart code

Here's the complete finished source code for our app's doclist.dart file.

Code Listing Appendix-f: Finished doclist.dart

```
import 'dart:async';

import 'package:flutter/material.dart';

import '../model/model.dart';
import '../util/dbhelper.dart';
import '../util/utils.dart';
import './docdetail.dart';

// Menu item
const menuReset = "Reset Local Data";
List<String> menuOptions = const <String> [
  menuReset
];

class DocList extends StatefulWidget {
  @override
  State<StatefulWidget> createState() => DocListState();
}

class DocListState extends State<DocList> {
  DbHelper dbh = DbHelper();
  List<Doc> docs;
  int count = 0;
  DateTime cDate;

  @override
  void initState() {
    super.initState();
  }

  Future getData() async {
    final dbFuture = dbh.initializeDb();
    dbFuture.then(
      // result here is the actual reference to the database object
      (result) {
          final docsFuture = dbh.getDocs();
          docsFuture.then(
            // result here is the list of docs in the database
            (result) {
                if (result.length >= 0) {
                  List<Doc> docList = List<Doc>();
                  var count = result.length;
```

```
                for (int i = 0; i <= count - 1; i++) {
                  docList.add(Doc.fromOject(result[i]));
                }
                setState(() {
                  if (this.docs.length > 0) {
                    this.docs.clear();
                  }

                  this.docs = docList;
                  this.count = count;
                });
              }
            });
      });
}

void _checkDate() {
  const secs = const Duration(seconds:10);

  new Timer.periodic(secs, (Timer t) {
    DateTime nw = DateTime.now();

    if (cDate.day != nw.day || cDate.month != nw.month ||
        cDate.year != nw.year) {
      getData();
      cDate = DateTime.now();
    }
  });
}

void navigateToDetail(Doc doc) async {
  bool r = await Navigator.push(context,
      MaterialPageRoute(builder: (context) => DocDetail(doc))
  );

  if (r == true) {
    getData();
  }
}

void _showResetDialog() {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: new Text("Reset"),
        content: new Text("Do you want to delete all local data?"),
        actions: <Widget>[
          FlatButton(
```


www.dbooks.org

```
            child: new Text("Cancel"),
            onPressed: () {
              Navigator.of(context).pop();
            },
          ),
          FlatButton(
            child: new Text("OK"),
            onPressed: () {
              Future f = _resetLocalData();
              f.then(
                      (result) {
                    Navigator.of(context).pop();
                  }
              );
            },
          ),
        ],
      );
    },
  );
}

Future _resetLocalData() async {
  final dbFuture = dbh.initializeDb();
  dbFuture.then(
    (result) {
      final dDocs = dbh.deleteRows(DbHelper.tblDocs);
      dDocs.then(
        (result) {
          setState(() {
            this.docs.clear();
            this.count = 0;
          });
        }
      );
    }
  );
}

void _selectMenu(String value) async {
  switch (value) {
    case menuReset:
      _showResetDialog();
  }
}

ListView docListItems() {
  return ListView.builder(
    itemCount: count,
```

```
      itemBuilder: (BuildContext context, int position) {
        String dd = Val.GetExpiryStr(this.docs[position].expiration);
        String dl = (dd != "1") ? " days left" : " day left";
        return Card(
          color: Colors.white,
          elevation: 1.0,
          child: ListTile(
            leading: CircleAvatar(
              backgroundColor:
              (Val.GetExpiryStr(this.docs[position].expiration) != "0") ?
              Colors.blue : Colors.red,
              child: Text(
                this.docs[position].id.toString(),
              ),
            ),
            title: Text(this.docs[position].title),
            subtitle: Text(
                Val.GetExpiryStr(this.docs[position].expiration) + dl +
                    "\nExp: " + DateUtils.convertToDateFull(
                    this.docs[position].expiration)),
            onTap: () {
              navigateToDetail(this.docs[position]);
            },
          ),
        );
      },
    );
  }

  @override
  Widget build(BuildContext context) {
    this.cDate = DateTime.now();

    if (this.docs == null) {
      this.docs = List<Doc>();
      getData();
    }

    _checkDate();

    return Scaffold(
      resizeToAvoidBottomPadding: false,
      appBar: AppBar(
          title: Text("DocExpire"),
          actions: <Widget>[
            PopupMenuButton(
              onSelected: _selectMenu,
              itemBuilder: (BuildContext context) {
                return menuOptions.map((String choice) {
```


www.dbooks.org

```
                return PopupMenuItem<String>(
                  value: choice,
                  child: Text(choice),
                );
              }).toList();
            },
          ),
        ]
      ),
      body: Center(
        child: Scaffold(
          body: Stack(
              children: <Widget>[
                docListItems(),
              ]),
          floatingActionButton: FloatingActionButton(
            onPressed: () {
              navigateToDetail(Doc.withId(-1, "", "", 1, 1, 1, 1));
            },
            tooltip: "Add new doc",
            child: Icon(Icons.add),
          ),
        ),
      ),
    );
  }
}
```

Full Pubspec.yaml code

Here's the complete finished code for our app's Pubspec.yaml file.

Code Listing Appendix-g: Finished Pubspec.yaml

```
name: flutter_app
description: A new Flutter application.

# The following defines the version and build number for your
application.
# A version number is three numbers separated by dots, like 1.2.43
# followed by an optional build number separated by a +.
# Both the version and the builder number may be overridden in Flutter
# build by specifying --build-name and --build-number, respectively.
# Read more about versioning at semver.org.
version: 1.0.0+1

environment:
```

```
  sdk: ">=2.0.0-dev.68.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  intl: ^0.15.7
  sqflite: any
  path_provider: any
  flutter_masked_text: ^0.6.0
  flutter_datetime_picker: ^1.0.1
  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^0.1.2

dev_dependencies:
  flutter_test:
    sdk: flutter

# For information on the generic Dart part of this file, see the
# following page: https://www.dartlang.org/tools/pub/pubspec

# The following section is specific to Flutter.
flutter:

  # The following line ensures that the Material Icons font is
  # included with your application, so that you can use the icons in
  # the material Icons class.
  uses-material-design: true
```

Full Android Studio project

Here's the full Android Studio Project, which contains source files, plus all the build files for Android.

www.dbooks.org