

**VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY**



**Nguyen Cong Thuan
ID: 18021250**

Midterm report

Flutter beginner course - Udemy

Course: Mobile Application Development

Supervisor: Ms. Nguyen Duc Anh

HA NOI - 2021

Table of Contents

Table of Contents	1
Introduction	2
About the report	2
Flutter beginner course - Udemy	3
Flutter introduction	4
Course notes & exercises	7
Progress: 100%	7
Source code	8
Section 1: First Flutter Application	8
Section 2: States	9
Section 3: Layouts	11
Section 4: AppBar & TabBar Widget	15
Section 5: Custom Widgets	17
Section 6: Input & Selections Widgets	18
Section 7: Drawer Widget & Routes	20
Section 8: Notification Widgets	23
Personal project	25
Source code	25
Project description	25
Project showcase	25
Project structure	29
Conclusion	30
Reference	30

1. Introduction

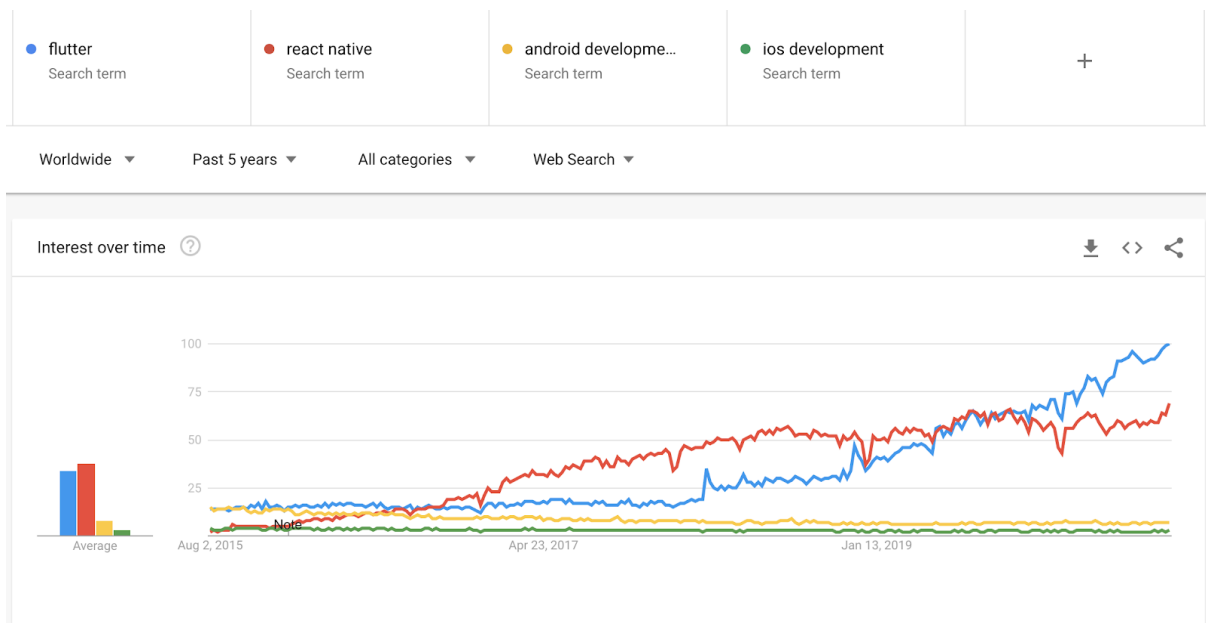
1.1. About the report

This report is a demonstration of the work that I did for my midterm assignment in the Mobile Application Development course. My efforts are spent on learning Flutter framework, Dart on a course on Udemy and practicing them in building a simple application, which is demonstrated in the following 2 sections:

- [Course notes & exercises](#)
- [Personal project](#)

I wrote this report in **English** as an effort to improve my technical writing skills, which, I believe, is very useful for my future career.

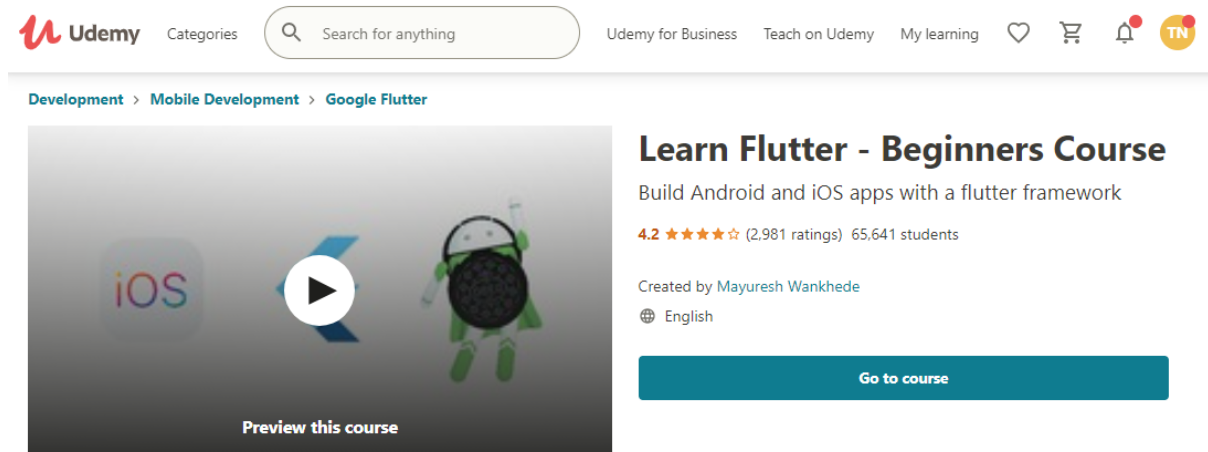
I chose Flutter because right now, it is really trending and has many advantages. According to Google Trends search results from Aug 2015 to 2021, comparing search results on Google between Flutter and React Native shows Flutter to be the most searched term.



Another reason is that Flutter is the technology on which my teammates agreed to use in our final project for its benefits. So I thought it is sensible to learn Flutter.

1.2. Flutter beginner course - Udemy

<https://www.udemy.com/course/learn-flutter-beginners-course/>



This course's objectives are:

- Provides fundamental understanding about flutter framework and Dart
- Gain better understanding of flutter's components and it's basic widgets
- Develop basic flutter application for android and iOS

This course taught me how to create a Flutter application in a very practical and simple manner, as every lecture comes with full coding screencast and corresponding code exercise in the notebook.

The content of the course consists of 8 sections:

- Section 1: First Flutter Application
- Section 2: States
- Section 3: Layouts
- Section 4: AppBar & TabBar Widget
- Section 5: Custom Widgets
- Section 6: Input & Selections Widgets
- Section 7: Drawer Widget & Routes
- Section 8: Notification Widgets

The content and course note is described in more detail in the **Course notes & exercises** section.

1.3. Flutter introduction

1.3.1. Intro

Flutter [1] is a framework that supports building cross-platform mobile applications. Currently, Flutter officially supports the two most popular mobile operating systems, iOS and Android, along with Web development.

Flutter is very convenient for building applications that focus on the interface and user experience, helping the development team to quickly build the software. software and testing features in a short time.

Flutter is programmed in **Dart** [2], a programming language developed by Google that supports source code translation for many different platforms, as well as to JavaScript for web programming.

1.3.2. Flutter advantages

After spending some time learning the basics and having written a simple app, I can conclude some of the benefit when using Flutter as follows:

- Nice UI: with Material Design

Flutter uses [Material Design](#) and Cupertino for its widget, which provides an expressive and flexible UI.

- High performance

For both Android and iOS, because Flutter is compiled into the native ARM code, performance problems are further mitigated.

- Cross-platform

Flutter uses a single codebase for both web and mobile development. This saves a lot of time and effort for developers.

- Hot reload

With Stateful Hot Reload, Flutter provides a quick development cycle, without recompiling each time the code is changed. In the development

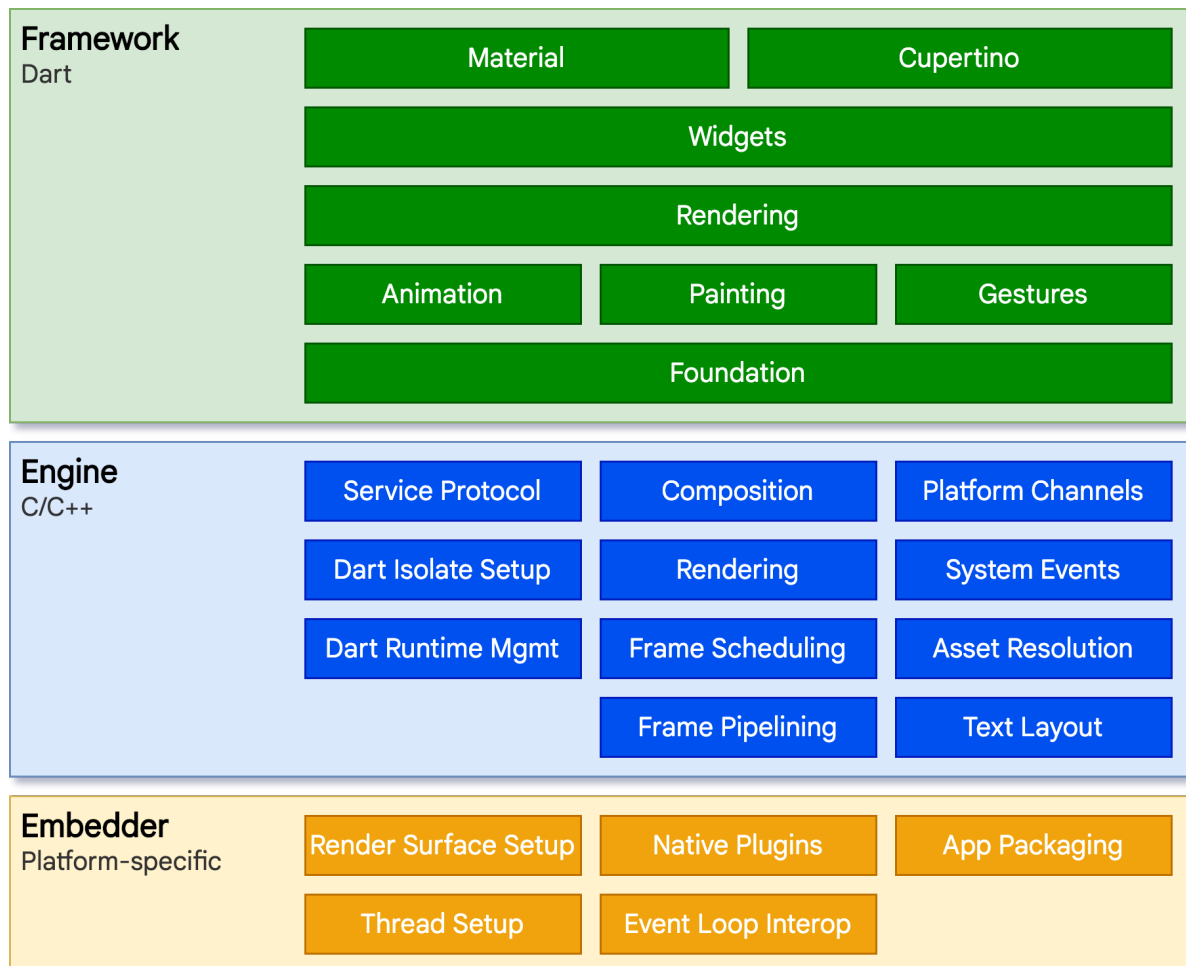
process, changes in the code are reflected immediately on the running device.

- Large community support

1.3.3. Flutter architecture

Layers architecture comprise of:

- Foundation library, written in Dart
- Flutter engine, written in C++
- Embedder, written in platform-specific/native code



1.3.4. Widgets

In Flutter, **everything is a widget**.

Flutter emphasizes widgets as a unit of composition. Each widget is a building block, an immutable declaration of part of a Flutter app's UI.

Widgets form a hierarchical structure. Each widget nests inside its parent and can receive context from the parent.

To respond to events (such as a user interaction), the UI is updated by replacing a widget in the hierarchy with another widget. The framework compares the new and old widgets, and then updates the UI efficiently.

These are the key concepts about Flutter Widgets, which will be discussed further in the next section as noted from the course.

- Composition
- Widget state
- State management

2. Course notes & exercises

2.1. Progress: 100%

Course content ✕

Section 1: Introduction To Flutter
2 / 2 | 4min ^

☒ 1. How to Install Flutter on Android Studio
2min

☒ 2. First Flutter Application
2min

Resources ▾

Section 2: Flutter States
2 / 2 | 9min ^

☒ 3. StatelessWidget Class
3min

☒ 4. StatefullWidget Class
6min

Resources ▾

Resources ▾

Section 3: layouts
6 / 6 | 22min ▾

Section 4: AppBar & TabBar Widget
3 / 3 | 16min ▾

Section 5: Custom Widgets
2 / 2 | 5min ▾

Section 6: Input & Selections Widgets
6 / 6 | 23min ▾

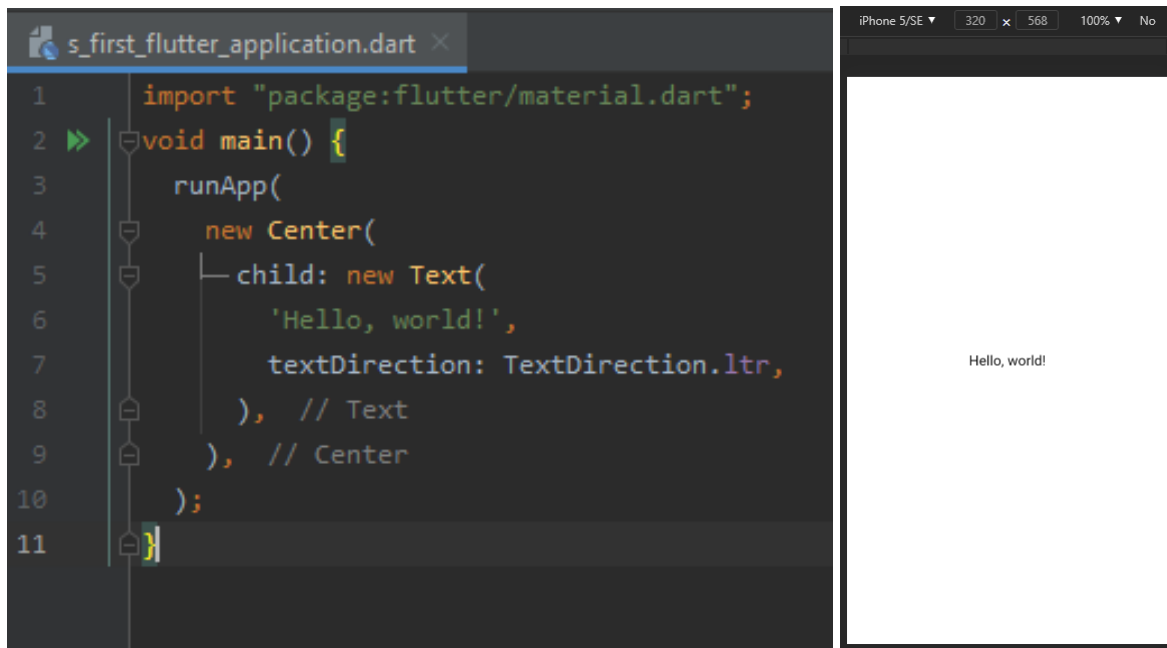
Section 7: Drawer Widget & Routes
2 / 2 | 10min ▾

Section 8: Notification Widgets
3 / 3 | 12min ▾

2.2. Source code

<https://github.com/ncthuan/learning-flutter/tree/main/flutter-course-udemy>

2.3. Section 1: First Flutter Application



This minimal hello world app is really straightforward, the code is concise and very easy to understand.

Components:

- **Text:** The Text widget lets you create a run of styled text within your application.

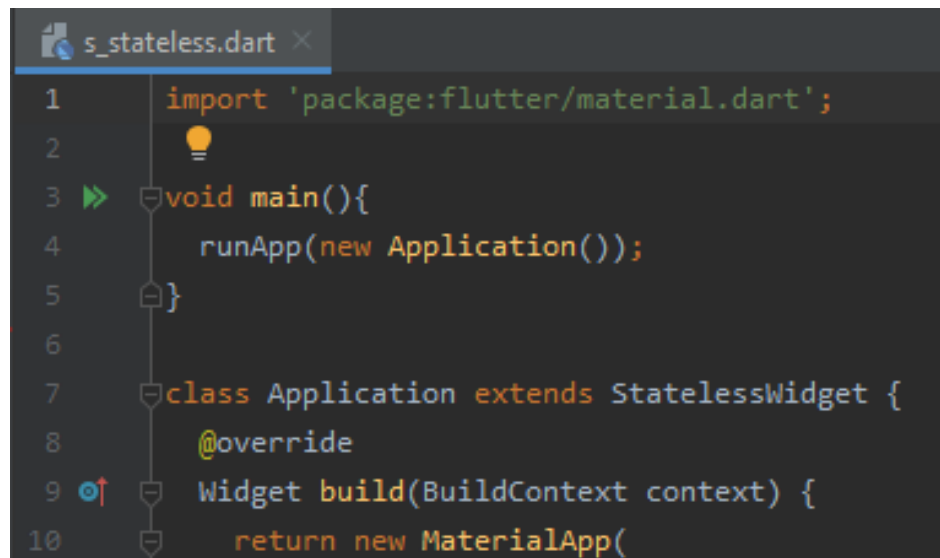
The Text widget displays a string of text with single style. The string might break across multiple lines or might all be displayed on the same line depending on the layout constraints.

- **Center:** A widget that centers its child within itself.

This widget will be as big as possible if its dimensions are constrained and **widthFactor** and **heightFactor** are null. If a dimension is unconstrained and the corresponding size factor is null then the widget will match its child's size in that dimension.

2.4. Section 2: States

The framework introduces two major classes of widget: **stateful** and **stateless** widgets.



```
1 import 'package:flutter/material.dart';
2
3 void main(){
4   runApp(new Application());
5 }
6
7 class Application extends StatelessWidget {
8   @override
9   Widget build(BuildContext context) {
10    return new MaterialApp(
```

StatelessWidget with build() method

- **StatelessWidget** is useful when the part of the user interface you are describing does not depend on anything other than the configuration information in the object itself. Many widgets have no mutable state: they don't have any properties that change over time (for example, an icon or a label). These widgets subclass **StatelessWidget**.
 - StatelessWidget class are static
 - Does not require mutable State
 - Cannot access the State operations

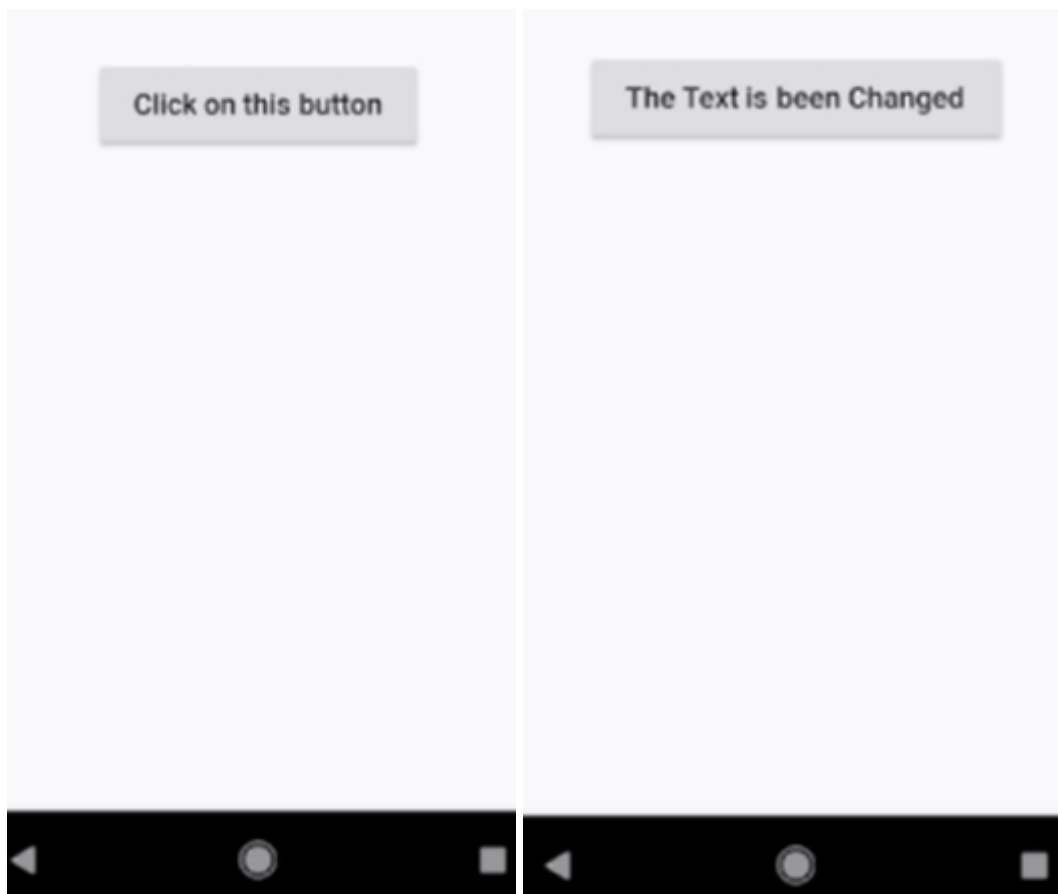
However, if the unique characteristics of a widget need to change based on user interaction or other factors, that widget is stateful. For example, if a widget has a counter that increments whenever the user taps a button, then the value of the counter is the state for that widget. When that value changes, the widget needs to be rebuilt to update its part of the UI. These widgets subclass **StatefulWidget**, and (because the widget itself is immutable) they store mutable state in a separate class that subclasses **State**. **StatefulWidget**s don't have a build method; instead, their user interface is built through their **State** object.

- **StatefulWidget** class are dynamic. Its state is stored in a State object

- State might change during lifetime of the **StatefulWidget** as the user interact
- `initState()` - call's this method exactly once for each state object it creates
- `setState()` - called when developer wants to change the internal state of the widget

Whenever you mutate a **State** object (for example, by incrementing the counter, or clicking the button), you must call **setState()** to signal the framework to update the user interface by calling the State's build method again.

In the exercise, I practiced managing the state of the text in a Button in the application.



```
void method1(){  
    setState(){  
        ttext = 'The text is been Changed';  
    });  
}
```

Here **ttext** is a property in the state object of the StatefulWidget.

```
return new MaterialApp(  
  home: new Scaffold(  
    body: new Center(  
      child: new RaisedButton(onPressed:(){method1();},child: new Text(ttext)),  
    ), // Center  
  ), // Scaffold  
); // MaterialApp
```

Components:

- **Scaffold**
 - Implements the basic material design visual layout structure.
 - This class provides APIs for showing drawers and bottom sheets.
 - To display a persistent bottom sheet, obtain the **ScaffoldState** for the current **BuildContext** via **Scaffold.of** and use the **ScaffoldState.showBottomSheet** function.
- **RaisedButton**: Button, think of it as similar to <button> in html, with typical event handler **onPressed**

2.5. Section 3: Layouts

Components:

- Container

Similar to bootstrap container in web development, the **Container** widget lets you create a rectangular visual element. A container can be decorated with a **BoxDecoration**, such as a background, a border, or a shadow. A **Container** can also have margins, padding, and constraints applied to its size. In addition, a **Container** can be transformed in three dimensional space using a matrix.

- Row, Column

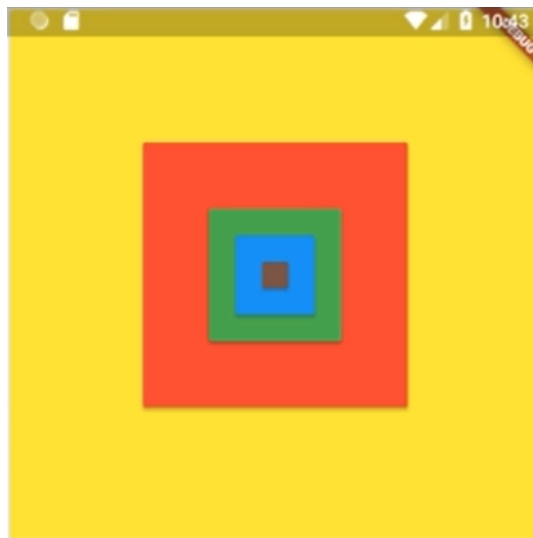
These flex widgets let you create flexible layouts in both the horizontal (Row) and vertical (Column) directions. The design of these objects is based on the **web's flexbox layout model**.

- Stack

Instead of being linearly oriented (either horizontally or vertically), a Stack widget lets you place widgets on top of each other in paint order. You can then use the Positioned widget on children of a Stack to position them relative to the top, right, bottom, or left edge of the stack. Stacks are based on the **web's absolute positioning layout model**.

```
14 Widget build(BuildContext context) {  
15   return new Scaffold(  
16     body: new Stack(  
17       alignment: Alignment.center,  
18       children: <Widget>[  
19         new Card(color: Colors.pink,child:new Padding(padding: const EdgeInsets.all(200.0))),  
20         new Card(color: Colors.green,child:new Padding(padding: const EdgeInsets.all(100.0))),  
21         new Card(color: Colors.blue,child:new Padding(padding: const EdgeInsets.all(50.0))),  
22         new Card(color: Colors.yellow,child:new Padding(padding: const EdgeInsets.all(10.0))),  
23       ] // <Widget>[]  
24     ), // Stack  
25   ); // Scaffold
```

Result:



- GridView

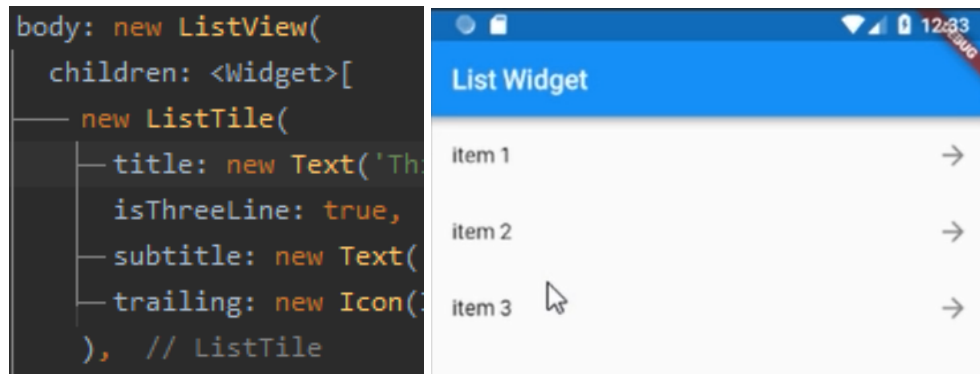
```
return new Scaffold(
  body: new GridView.builder(
    gridDelegate: new SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 4,),
    itemCount: _items.length,
    itemBuilder: (BuildContext context,int index){
      return cards(index);
    }
  ) // GridView.builder
); // Scaffold
```



- **gridDelegate**: To create a grid with a large (or infinite) number of children, use the GridView.builder constructor with **gridDelegate** of either a
 - SliverGridDelegateWithFixedCrossAxisCount or
 - SliverGridDelegateWithMaxCrossAxisExtent
 - **SliverGridDelegateWithFixedCrossAxisCount**: Creates grid layouts with a fixed number of tiles in the cross axis.
 - **itemBuilder**: function returns a grid item Widget given its index.
- ListView

There are four options for constructing a ListView:

1. The default constructor takes an explicit List<Widget> of **children**. This constructor is appropriate for list views with a small number of children because constructing the List requires doing work for every child that could possibly be displayed in the list view instead of just those children that are actually visible.



ListTile: a single list item in the list.

2. The **ListView.builder** constructor takes an **IndexedWidgetBuilder**, which builds the children on demand. This constructor is appropriate for list views with a large (or infinite) number of children because the builder is called only for those children that are actually visible.

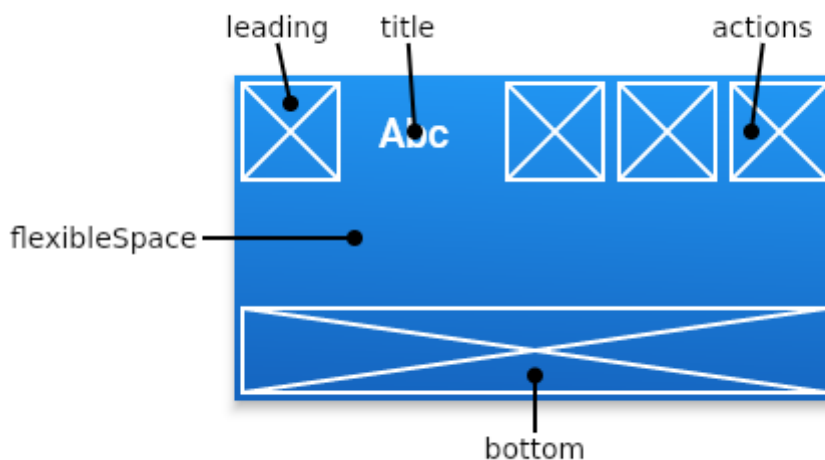
```
body: new ListView.builder(
  itemCount: _listitems.length,
  itemBuilder: (BuildContext context, int index){
    return new ListTile(
      title: new Text('This is Title'),
      isThreeLine: true,
      subtitle: new Text('This is our Subtitle'),
      trailing: new Icon(Icons.close),
    ); // ListTile
  }
) // ListView.builder
```

3. The **ListView.separated** constructor takes two **IndexedWidgetBuilders**: **itemBuilder** builds child items on demand, and **separatorBuilder** similarly builds separator children which appear in between the child items. This constructor is appropriate for list views with a fixed number of children.
4. The **ListView.custom** constructor takes a **SliverChildDelegate**, which provides the ability to customize additional aspects of the child model. For example, a **SliverChildDelegate** can control the algorithm used to estimate the size of children that are not actually visible.

2.6. Section 4: AppBar & TabBar Widget

App bar: A material design app bar.

An app bar consists of a toolbar and potentially other widgets, such as a **TabBar** and a **FlexibleSpaceBar**. App bars typically expose one or more common **actions** with **IconButton**s which are optionally followed by a **PopupMenuButton** for less common operations (sometimes called the "overflow menu").



```
return new Scaffold(  
  appBar: new AppBar(  
    title: new Text('AppBar'),  
    leading: new Icon(Icons.menu),  
    actions: <Widget>[  
      new IconButton(icon: new Icon(Icons.arrow_forward), onPressed: (){}),  
      new IconButton(icon: new Icon(Icons.add), onPressed: (){print('you')}),  
      new IconButton(icon: new Icon(Icons.close), onPressed: (){print('')}),  
    ], // <Widget>[]  
  ), // AppBar  
); // Scaffold
```

- **leading:** A widget to display before the toolbar's title. Typically the leading widget is an **Icon** or an **IconButton**.
- **title:** the title in Text.
- **actions:** A list of Widgets to display in a row after the title widget. Typically these widgets are **IconButton**s representing common

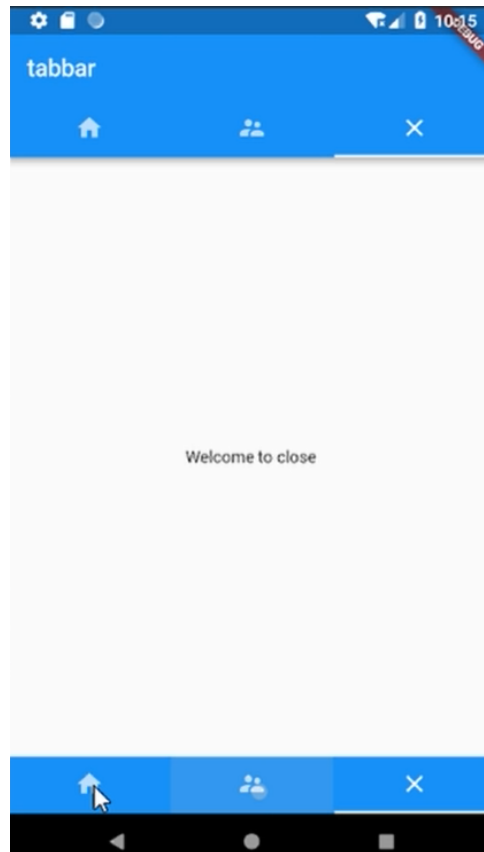
operations. For less common operations, consider using a **PopupMenuButton** as the last action.

- **flexibleSpace**: This widget is stacked behind the toolbar and the tab bar. Its height will be the same as the app bar's overall height.



Tab bar: A material design widget that displays a horizontal row of tabs. Typically created as the **AppBar.bottom** part of an **AppBar** and in conjunction with a **TabBarView**.

```
home: new Scaffold(  
  appBar: new AppBar(  
    title: new Text("Application001"),  
    bottom: new TabBar(  
      controller: controller,  
      tabs:[  
        new Tab(text: "TAB 1" ,),  
        new Tab(text: "TAB 2",),  
        new Tab(text: "TAB 3",),  
      ]), // TabBar  
    backgroundColor: Colors.deepOrange,  
  ), // AppBar
```



2.7. Section 5: Custom Widgets

Custom widget method: a method that return a **Widget** instance

For example:

```
Widget cards(val){
    return new Card(
        color: Colors.pink,
        child: new Padding(
            padding: const EdgeInsets.all(10.0),
            child: new Text('$val'),
        ),); // Padding, Card
}
```

Custom widget class: a class that extends a **Widget** class, and overrides **build** method.

```
class customwidgets extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
```

2.8. Section 6: Input & Selections Widgets

Components:

- **TextField**

A text field lets the user enter text, either with a hardware keyboard or with an onscreen keyboard.

The text field calls the **onChanged** callback whenever the user changes the text in the field. If the user indicates that they are done typing in the field (e.g., by pressing a button on the soft keyboard), the text field calls the **onSubmitted** callback.

controller: To control the text that is displayed in the text field

- **Button**

ElevatedButton: A Material Design "elevated button".

Use elevated buttons to add dimension to otherwise mostly flat layouts, e.g. in long busy lists of content, or in wide spaces. Avoid using elevated buttons on already-elevated content such as dialogs or cards.

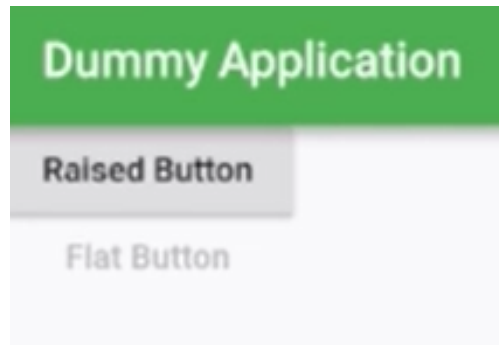
TextButton: A Material Design "Text Button".

Use text buttons on toolbars, in dialogs, or inline with other content but offset from that content with padding so that the button's presence is obvious. Text buttons do not have visible borders and must therefore rely on their position relative to other content for context. In dialogs and cards, they should be grouped together in one of the bottom corners. Avoid using text buttons where they would blend in with other content, for example in the middle of lists.

onPressed callback: function gets executed when user press the button.

RaisedButton: deprecated, replaced by **ElevatedButton**

FlatButton: deprecated, replaced by **TextButton**



- **CheckBox:** A material design checkbox.

The checkbox itself does not maintain any state. Instead, when the state of the checkbox changes, the widget calls the **onChanged** callback. Most widgets that use a checkbox will listen for the **onChanged** callback and rebuild the checkbox with a new value to update the visual appearance of the checkbox.

value: Whether the checkbox is checked. Usually, you pass a Widget state property here. In the example, **checkvalue** is a state property.

```
body: new Center(  
  child: new Checkbox(  
    value: checkvalue,  
    onChanged: (bool checkbool) {  
      method3(checkbool);  
    },  
  ),  
) // Checkbox
```

- **Radio:** A material design radio button.

Used to select between a number of mutually exclusive values. When one radio button in a group is selected, the other radio buttons in the group cease to be selected. The values are of type T, the type parameter of the Radio class. Enums are commonly used for this purpose.

- **Slider:** A Material Design slider.

Used to select from a range of values.

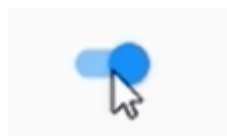
value: similar to **CheckBox**'s value, it represents the value that user is currently choosing. A Widget state property.

min, max: the minimum and maximum value that the user can select.

```
return new Scaffold(  
  body: new Center(  
    child: new Slider(  
      value: svalue, |  
      min: 1.0,  
      max: 10.0,  
      onChanged: (double value) {  
        method1(value);  
      }  
    ), // Slider  
  ) // Center
```



- **Switch:** A material design switch.



Used to toggle the on/off state of a single setting.

The rest is pretty much similar to checkbox.

2.9. Section 7: Drawer Widget & Routes

Drawer: A material design panel that slides in horizontally from the edge of a **Scaffold** to show navigation links in an application.

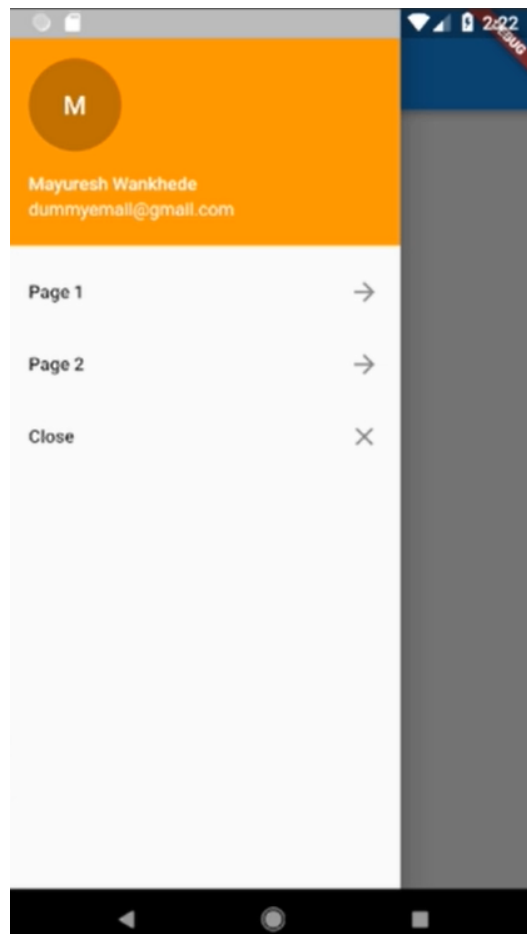
Drawers are typically used with the **Scaffold.drawer** property. The child of the drawer is usually a **ListView** whose first child is a **DrawerHeader** that displays status information about the current user. The remaining drawer children are often constructed with **ListTiles**, often concluding with an **AboutListTile**.

The **AppBar** automatically displays an appropriate **IconButton** to show the Drawer when a **Drawer** is available in the **Scaffold**. The **Scaffold** automatically handles the edge-swipe gesture to show the drawer.

```

return new Scaffold(
  appBar: new AppBar(title: new Text('Drawer')),
  drawer: new Drawer(
    child: new ListView(
      children: <Widget>[
        new UserAccountsDrawerHeader(
          accountName: new Text('Mayuresh Wankhede'),
          accountEmail: new Text('dummy@email.com'),
          currentAccountPicture: new CircleAvatar(
            backgroundColor: Colors.black26,
            child: new Text('M'),
          ), // CircleAvatar
          decoration: new BoxDecoration(color: Colors.orange),
        ), // UserAccountsDrawerHeader
        new ListTile(

```



Next, I used **Navigator** APIs to navigate to page 1 and page 2 when the user tap into “Page 1” and “Page 2”.

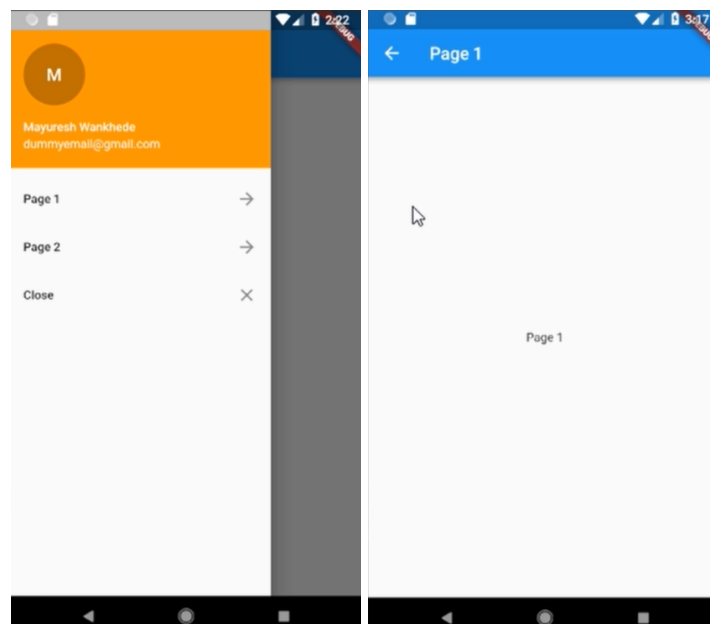
Navigator: A widget that manages a set of child widgets with a stack discipline.

Mobile apps typically reveal their contents via full-screen elements called "screens" or "pages". In Flutter these elements are called routes and they're managed by a **Navigator** widget. The navigator manages a stack of **Route** objects and provides two ways for managing the stack, the declarative API **Navigator.pages** or imperative API **Navigator.push** and **Navigator.pop**.

```
new ListTile(  
  title: new Text('Page 1'),  
  trailing: new Icon(Icons.arrow_forward),  
  onTap: () => Navigator.of(context).push(  
    new MaterialPageRoute(builder: (BuildContext context)=>new op('page1'))  
  ),  
), // ListTile
```

- push: Push the given route onto the navigator that most tightly encloses the given context.

This changes the current view to the view returned by the builder.



- pop: Pop the top-most route off the navigator that most tightly encloses the given context. This returns the most previous view.

Route: An abstraction for an entry managed by a **Navigator**.

- **MaterialPageRoute:** A modal route that replaces the entire screen with a platform-adaptive transition.

2.10. Section 8: Notification Widgets

SnackBar: A lightweight message with an optional action which briefly displays at the bottom of the screen.

To display a snack bar, call **ScaffoldMessenger.of(context).showSnackBar()**, passing an instance of **SnackBar** that describes the message.

```
void _method1(){
  _scaffoldkey.currentState.showSnackBar(new SnackBar(content: new Text('Activated Snackbar')));
}

Widget build(BuildContext context) {
  return new Scaffold(
    key: _scaffoldkey,
    body: new Center(
      child: new RaisedButton(onPressed: (){_method1();}, child: new Text('Activate Snackbar'),),
    ), // Center
  ); // Scaffold
}
```

With this, when I press the button, a message (“Activate Snackbar”) will pop up on the bottom like this:



showDialog: a function displays a Material dialog above the current contents of the app.

AlertDialog: A material design alert dialog.

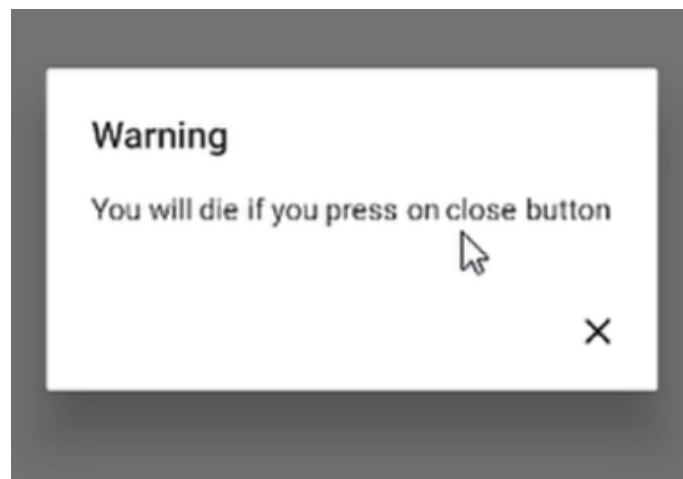
An alert dialog informs the user about situations that require acknowledgement. An alert dialog has an optional title and an optional list of actions. The **title** is displayed above the **content** and the **actions** are displayed below the content. Belows are the common properties:

- **title:** The title of the dialog is displayed in a large font at the top of the dialog

- **content:** The content of the dialog is displayed in the center of the dialog in a lighter font
- **actions:** can be some buttons that trigger some actions.

```
showDialog(  
  context: context,  
  child: new AlertDialog(  
    title: new Text('Warning'),  
    content: new Text('You will die if you press on close button'),  
    actions: <Widget>[  
      new IconButton(icon: new Icon(Icons.close), onPressed: (){Navigator.pop(context);})  
    ], // <Widget>[]  
  ) // AlertDialog  
);
```

Result when I click on the button that calls this showDialog:



3. Personal project

3.1. Source code

Github: <https://github.com/ncthuan/learning-flutter>

3.2. Project description

Objectives: In this project, I tried to apply as much knowledge that I acquired from the lectures and exercises to practice my skills developing Flutter application.

Application name: Product management application

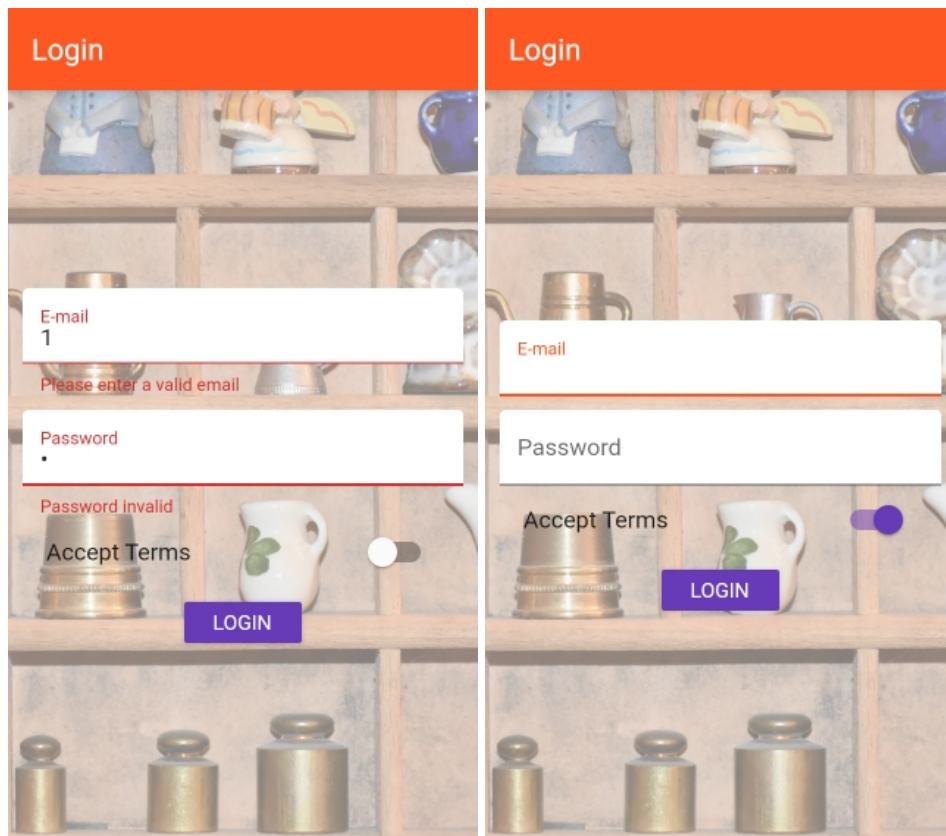
Features & functions:

- Login
- Shows a list view of products
 - Show a detail view of a product
- Manage products
 - Create products
 - Update products
- Mark product as loved, filter loved products

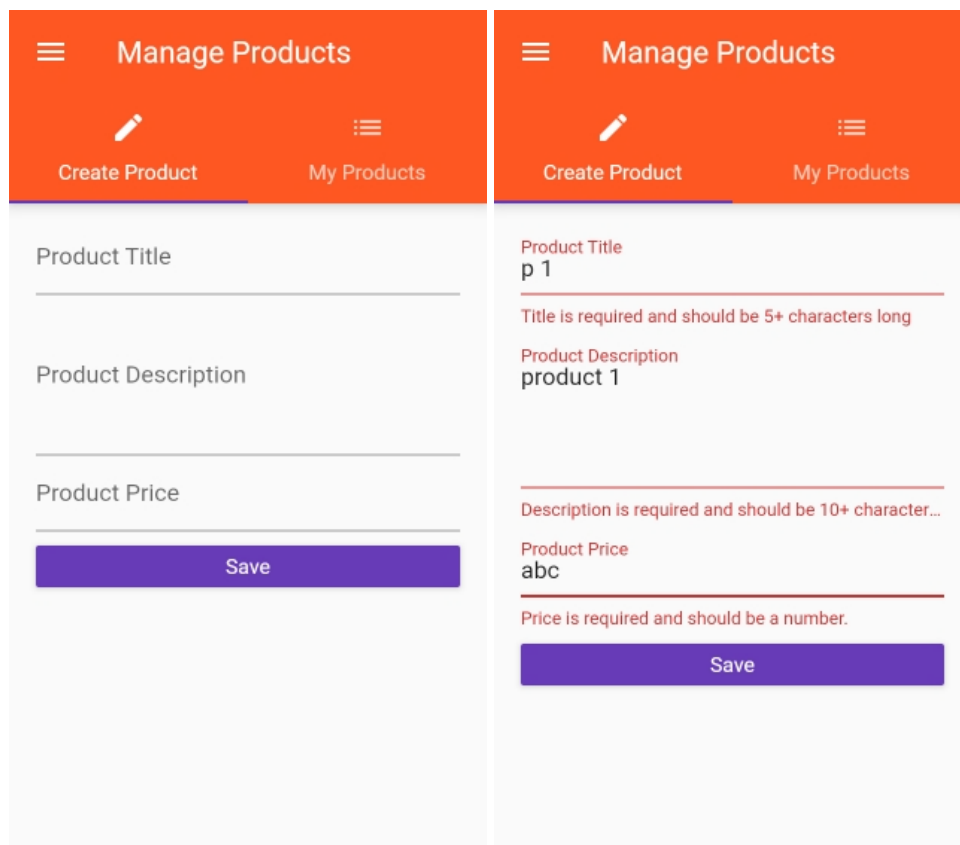
3.3. Project showcase

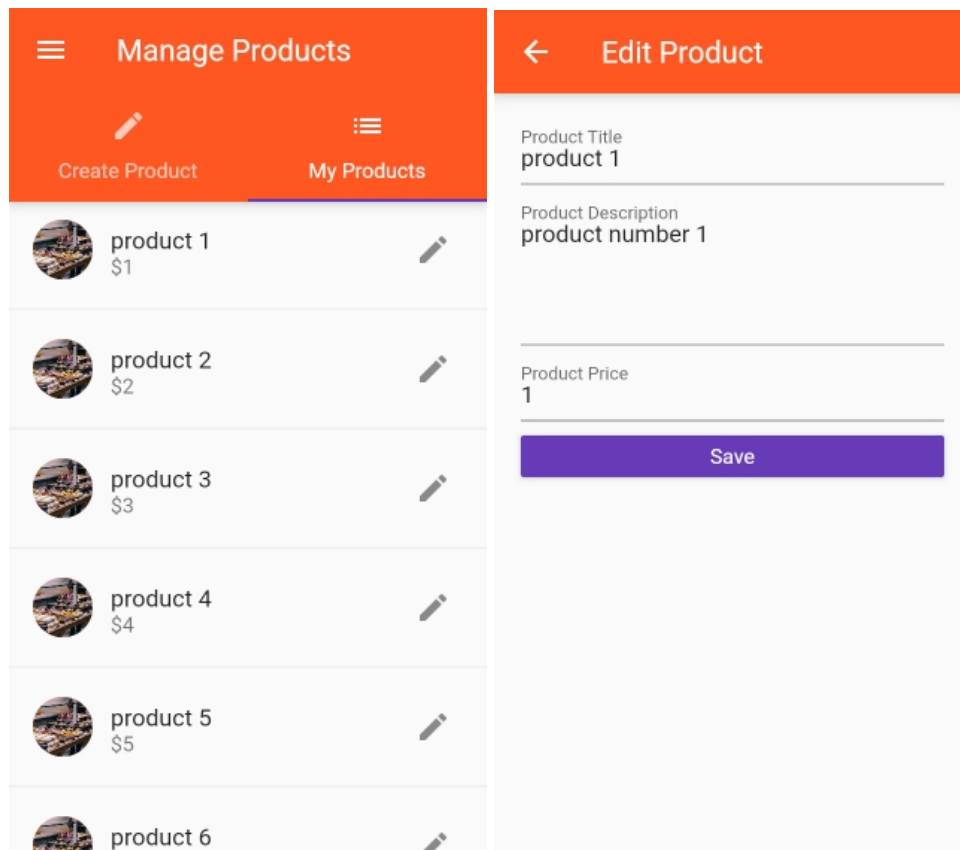
3.3.1. Login

The app does simple email, password validation and the accept term toggle for logging in.

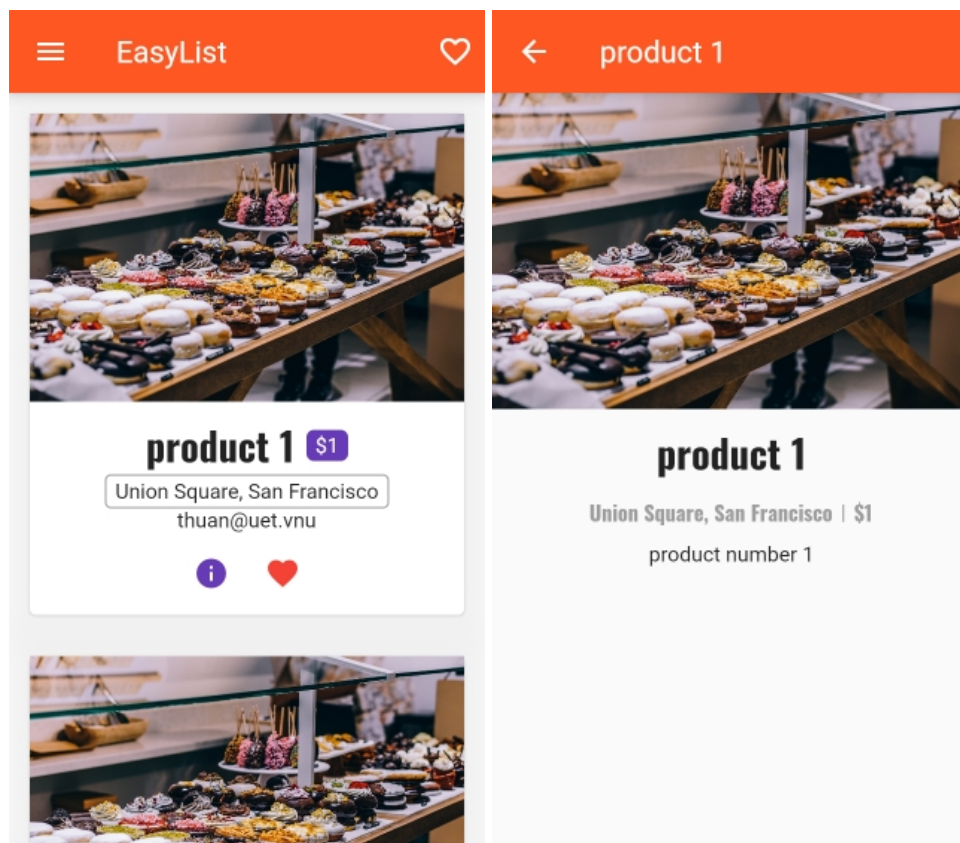


3.3.2. Manage products





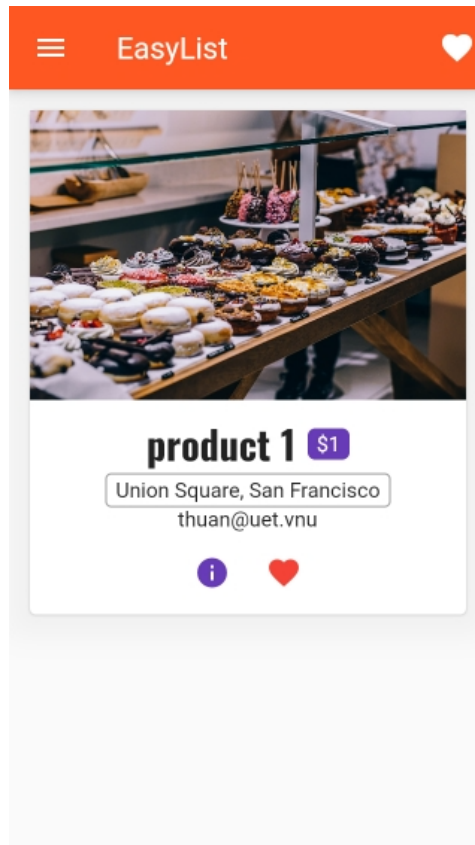
3.3.3. Show a list view of products



3.3.4. Filter loved products

When I press on the heart icon on the top right corner, the app will filter the loved products.

Here I only marked product 1 as loved, so only this product is shown



3.4. Project structure

cmd: tree /f lib/

```
lib
├── main.dart
├── models
│   ├── product.dart
│   └── user.dart
├── pages
│   ├── auth.dart
│   ├── product.dart
│   ├── products.dart
│   ├── products_admin.dart
│   ├── product_edit.dart
│   └── product_list.dart
├── scoped-models
│   ├── connected_products.dart
│   ├── main.dart
│   ├── products.dart
│   └── user.dart
└── widgets
    ├── helpers
    │   └── ensure-visible.dart
    ├── products
    │   ├── address_tag.dart
    │   ├── price_tag.dart
    │   ├── products.dart
    │   └── product_card.dart
    └── ui_elements
        └── title_default.dart
```

- `main.dart`: the main file with **main()** method that builds the app.
- `models`: classes that model the data structures of entities
- `pages`: widgets that are the whole page. They are utilized in `main.dart`, in routes.
- `scoped-models`: utilities that allow you to easily pass a data Model from a parent Widget down to its descendants.
- `widgets`: some helper widgets and smaller components

4. Conclusion

In this report, I demonstrate the work that I did to accomplish this midterm assignment.

Below lists all the **efforts** that I have spent:

- 2 hours: lectures on Udemy course
- 2 hours: lecture exercises, comprehending the code
- 16 hours: personal project
- 3 hours: setting up LaTeX template, cover, but as the deadline was near and I'm still unfamiliar with LaTeX, I had to switch to Google docs for faster writing.
- 8 hours: writing course notes and reports
- days on debugging, stackoverflowing, additional learning and research on Flutter and Dart

What I achieved after all those efforts:

- Completed an Udemy course
- Acquired basic knowledge and understanding about Flutter framework and Dart.
- Gained some hands-on experience with developing a simple mobile application with Flutter
- Gained LaTeX insights, found a good [reference](#) and developed technical writing skill.

5. Reference

[1] Google, "Flutter." [Online]. Available: <https://flutter.dev/docs> under [CC 4.0 license](#).

[2] Google, "Dart." [Online]. Available: <https://dart.dev/guides>