LEARNING MODULE

IT 331

Application Development and Emerging Technologies

INTRODUCTION TO APPLICATION DEVELOPMENT

Topic Outcomes

In this lesson, students will be able to have a foundational understanding of application development processes, a grasp of mobile application development complexities, and the ability to distinguish between different types of mobile applications.

Application Development

Application development is a process of creating a computer program or set of programs to perform the different tasks a business requires. In application development, the processes involved are as follows:

1. **Planning** – The initial phase where project goals, requirements, scope, resources, timeline, and risks are defined to create a strategic blueprint for the application development.
2. **Creating** – The development phase where the actual coding and building of the application take place, based on the planned design and specifications.
3. **Testing** – The quality assurance phase where the application is rigorously evaluated for bugs, performance issues, and compliance with requirements to ensure it functions correctly.
4. **Deploying** – The final phase where the completed application is released to the production environment, making it available for end-users to access and use.

Mobile Application Development

Mobile application development is the process of making software for smartphones and digital assistants for Android and iOS. The software can be:

- Pre-installed on the device
- Downloaded from the AppStore
- Accessed through Mobile Web Browser

Types of Mobile Applications

NATIVE MOBILE APPLICATIONS

The native mobile applications are specifically built for a mobile device operating system. That means, you can have native Android mobile applications or native iOS

applications, not to mention all the other devices and platforms. However, native mobile apps are built for one platform, you cannot use it on another device. For instance, use an Android application on an iOS device, or use an iOS application on a Windows phone. Meanwhile, it can use the device's camera, GPS, compass, contacts, et cetera.

Advantages

In terms of its performance, native apps are faster and more reliable, due to its singular focus. Generally, they are more efficient with the device resources, as compared to other types of mobile applications. To provide the users a more optimized customer experience, the native apps utilize the native device UI. Native apps can directly connect with the device's hardware, it means they have access to a broad choice of device features like Bluetooth, phonebook contacts, camera roll, etc.

Disadvantages

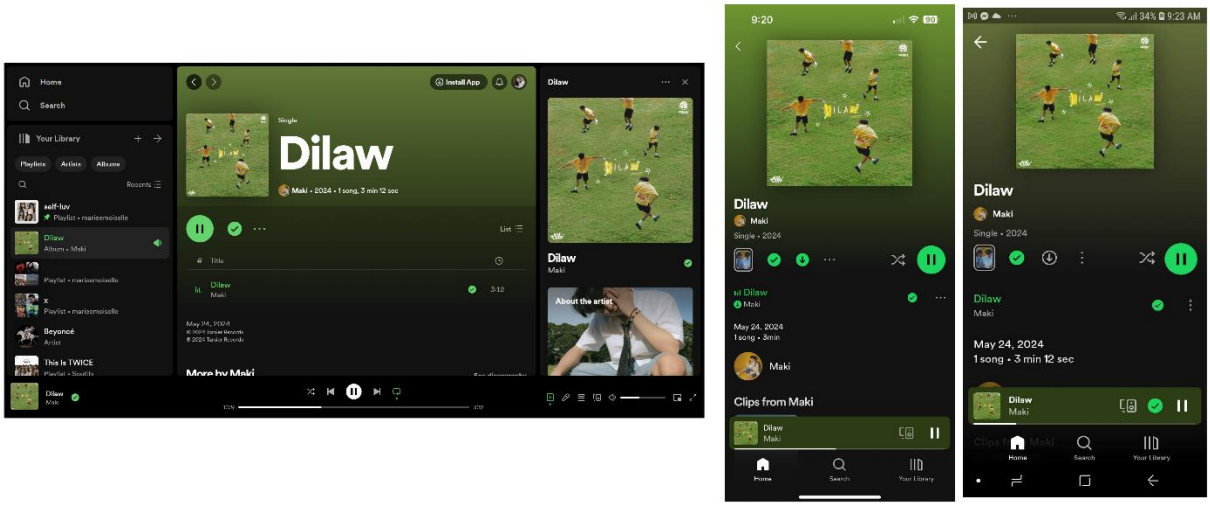
When you develop native apps, it has to have duplicate efforts for various platforms. Meaning, the code that was created for one platform cannot be used on another platform, which will cause the costing to rise, and not to mention, even the effort needed for the maintenance and updating the codebase for each version.

Another con of the native web is for updates. The user has to download the updated file and re-install it in their device, meaning, it would take up space in the device storage.

Examples

By definition, mobile applications are specifically designed and developed for a particular mobile operating system, using the native tools and programming languages of that operating system. Some examples of a native mobile application are:

- **Google Maps** – uses Swift/Objective-C for iOS and Kotlin/Java for Android.
- **Instagram** – developed for iOS using Swift, and Kotlin/Java for Android.
- **Spotify** – developed using Swift for iOS and Kotlin/Java for Android.
- **Uber** – built for iOS with Swift/Objective-C and for Android with Kotlin/Java.
- **WhatsApp** – built with Swift/Objective-C for iOS, and Kotlin/Java for Android.



MOBILE WEB APPS

In terms of behavior, web apps and native apps are similar, however, **web applications are accessed using the web browser on your mobile device**. They are not considered as standalone applications, in a sense of having to download and install code in the user's device. They are **responsive websites with an adapting user interface to the device of the user**. If ever the user wants to install a web app, it will be bookmarked to the website URL of the device. An example of a web app is the progressive web app (PWA), that is basically a native application running in a browser.

Advantages

Web apps are advantageous in terms of development cost, for it is web-based, and customization to a platform or OS is not needed. Alongside that, there is nothing to download, and won't take up space in the device storage, as compared to a native app, making maintenance easier. The updates are no longer to be downloaded at the AppStore.

Disadvantages

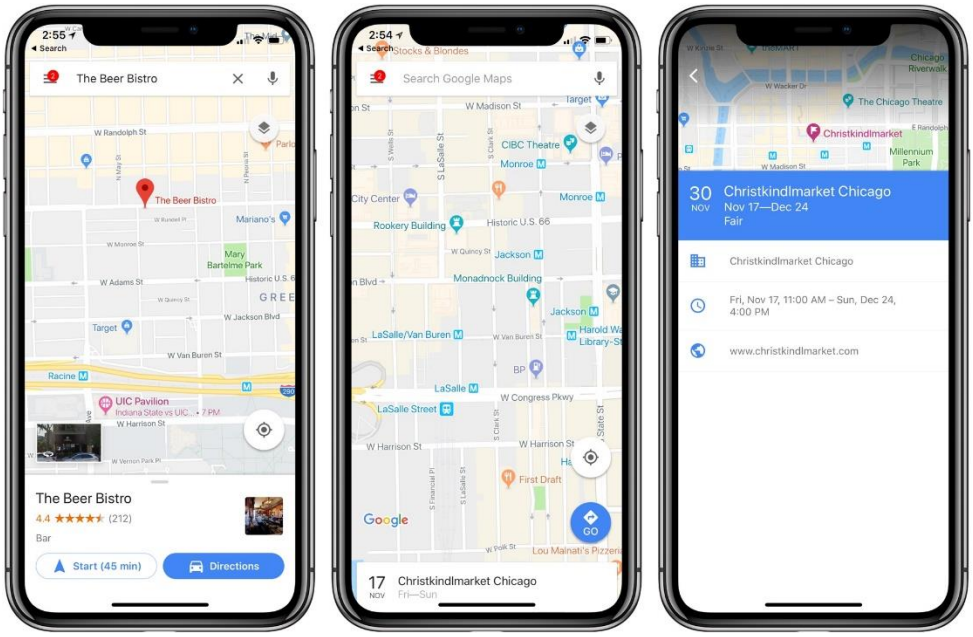
And because they're shells for websites, they won't completely work offline. Even if they have an offline mode, the device will still need an internet connection to back up the data on your device, offer up any new data, or refresh what's on screen.

However, web applications are dependent on the browser on the user's device, and some functionalities may be limited, possibly giving the users varying experience. Despite having an offline mode, the device will still need an internet connection for web apps to be used.

Examples

Mobile web applications are optimized for mobile devices and are accessible through a mobile web browser, without installing them from the app store. They are optimized for mobile devices. Some examples are:

- **Facebook** – has a mobile web version offered, and is accessible via m.facebook.com.
- **Google Maps** – is accessible via maps.google.com as a mobile web app that includes location and navigation features.
- **Spotify** – has a web player working on mobile browsers via open.spotify.com.
- **X (formerly Twitter)** – has a mobile-friendly web interface accessible using mobile.twitter.com
- **YouTube** – has a web app for video streaming, optimized for mobile available using m.youtube.com.



HYBRID APPLICATIONS

Hybrid applications are web applications that look and feel like native. They can contain a home screen app icon, responsive design, fast performance, even be able to function offline, but in short, **hybrid applications are web applications that were made to look native**.

Advantages

A hybrid application can be a minimum viable product (MVP), or a way to prove the viability of building a native app. It is much faster and economical to build, as

compared to native app. Hybrid applications are also ideal for usage in countries with slower internet connections, for it provides a consistent user experience. In terms of its maintenance, since they use a single code base, lesser code is to be maintained.

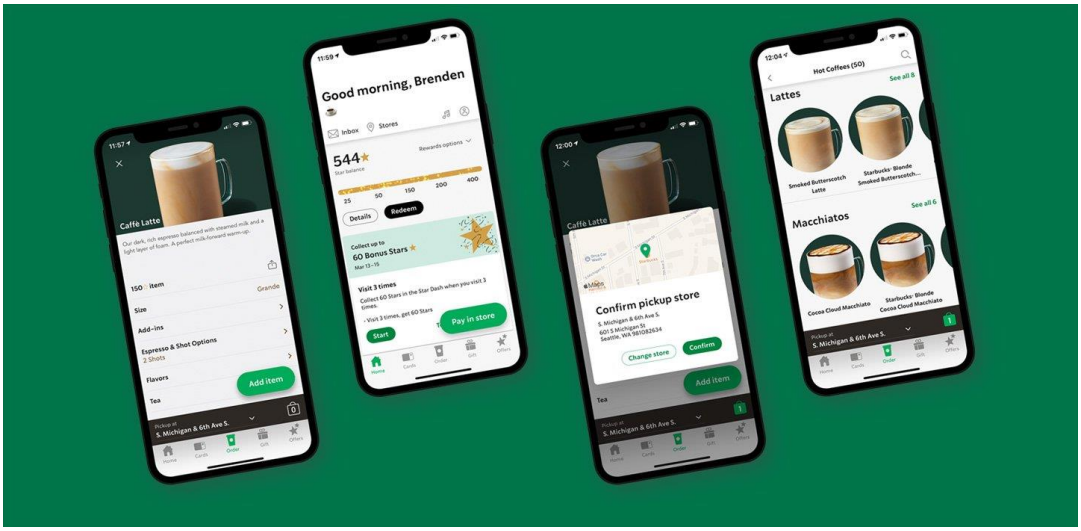
Disadvantages

Hybrid apps might lack in power and speed, which are trademarks of native apps.

Examples

Hybrid mobile applications are built using web technologies like HTML, CSS, and JavaScript, and wrapped in a native container, allowing them to run on different platforms, like iOS and Android, using a single codebase. Examples include:

- **Evernote** – is a note taking-app combining web technologies to provide a consistent interface on various platforms.
- **Gmail** – its mobile app version is incorporating hybrid techniques to provide a seamless experience.
- **Instagram** – it is considered to be predominantly native, but initially supports hybrid approaches to speed up the development of some components.
- **Starbucks** – it uses a combination of web technologies and native functionalities in order to provide a consistent user experience across different mobile platforms.
- **X (formerly Twitter)** – this application used hybrid technologies initially in order to provide consistency in terms of user experience across various platforms.



How to Choose Just One?

Consideration: Need for an App ASAP

Building a **web app** is recommended if an **application is needed in the shortest amount of time possible.** The users already have a mobile browser, which is what they need to use the app. Also, having one codebase drastically speeds up the development time.

Consideration: Limited Resources

In this case, where **resources are limited**, time and money are not on your side, a **web app or a hybrid app are the possible options**. A hybrid app enables you to test the market with a minimum viable product that can be in the hands of the users within a few months' time, which can be built into a full-fledged native version later on, if successful.

Consideration: Fast and Stable Application

If **performance is highly considered**, then the only choice is to develop a **native application**, for it provides you the stability, speed, and customization feature that is crucial for its success.

Note:

Choosing the type of mobile app to build is not a one-and-done decision. It will always depend on the needs of your user.

INTRODUCTION TO FLUTTER

Topic Outcomes

In this topic, students will be able to recognize the fundamentals of the Flutter open-source SDK, its architecture, and how to set up the Flutter development environment.

Meet Flutter

Flutter (developed December 4, 2018) is a Google Mobile SDK used for building native iOS and Android applications **using a single codebase**. In building applications with Flutter, everything is towards **Widgets**, or the blocks with which Flutter applications are built.

The User Interface of the application is comprised of many simple widgets, and each of them are handling one particular job, making Flutter developers think of their Flutter application as a tree of devices.

Flutter is much known for having a single codebase for Android and iOS, compared to its contemporaries like React Native, Kotlin, and Java. Also, Flutter is known for having a reusable UI and its business logic, high compatibility, performance, and productivity.

KEY FEATURES OF FLUTTER

- **Cross-platform Development.** Developers are allowed to write a single codebase for both Android and iOS, reducing the time and cost for development.
- **Fast Development.** The "hot reload" feature allows the developers to see the changes to the code immediately, making the development process faster and more efficient.
- **Attractive UI.** Developers can create visually appealing and responsive user interfaces, through the use of the rich set of customizable widgets provided by Flutter.
- **Performance.** The Dart programming language is used by Flutter. Meanwhile, its efficient rendering engine, Skia ensures high performance, fast app startup times, and smooth animations.
- **Large Community.** Development in Flutter has been made easy because of its growing and supportive community that provides Flutter developers with vast documentation, resources, and third-party packages.
- **Open-Source.** It is an open-source and free framework for mobile application development.

ARCHITECTURE APPLICATION

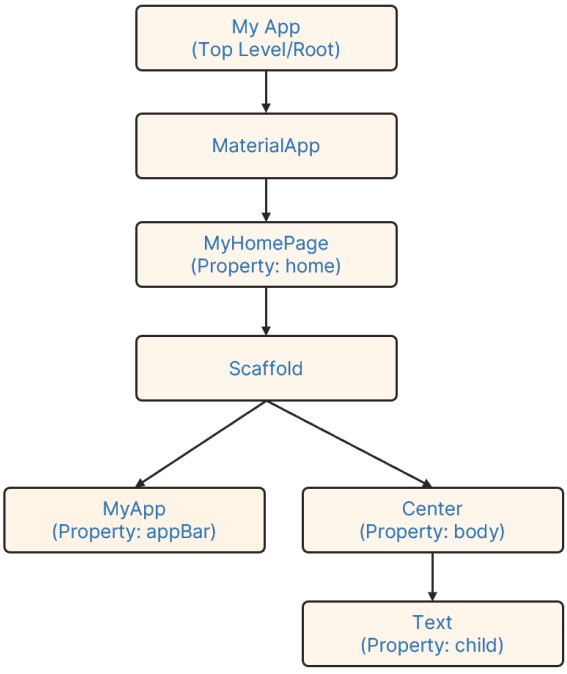
The Flutter architecture application consists of the following:

- **Widgets.** The primary component of any flutter application acting as the UI for the user to interact with the application.

- **Layers.** The hierarchy of categories based on the decreasing level of complexity.
- **Gestures.** An invisible widget used for processing the physical interaction with the flutter application.
- **Concept of State.** The data objects.

Widgets

Widgets are the **primary component of any flutter application**. It acts as a UI for the user to interact with the application. Any flutter application is itself a widget that is made up of a combination of widgets. In a standard application, the **root** defines the structure of the application followed by a **Material App** widget which basically **holds its internal components in place**. This is where the properties of the UI and the application itself is set. The Material App has a **Scaffold** widget that **consists of the visible components** (widgets) of the application. The Scaffold has two primary properties, **body** and **appbar**. It holds all the child widgets and this is where all its properties are defined. The below diagram shows the hierarchy of a flutter application:



The root defines the structure of the application, followed by a **Material App** widget that basically holds the components in place. A **Scaffold** widget, which is part of the Material App consists of the visible components (widgets) of the application.

There are two primary properties of a Scaffold, **body** and **appbar**. It holds all the child widgets and it is where all the properties are defined.

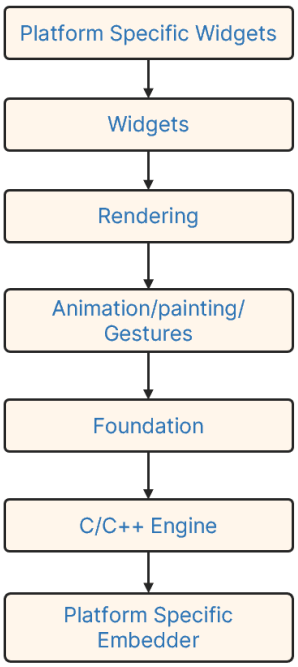
Usually, there is an **appbar** widget in Scaffold, that defines the appbar of the application. Meanwhile, **body** is used to place all the component widgets, and where the widget properties are set. Combining these widgets form the Homepage of the application itself.

The **Center** widget includes a property named **Child**, that refers to the actual content and is build using the **Text** widget.

Layers

The Flutter framework is categorized by its complexity, establishing a hierarchy based on the decreasing level of these complexities. These categories are often called layers. They are built on top of one another.

A widget specific to the operating system of the device, for instance, Android or iOS is the **topmost layer**. Meanwhile, the **second layer** contains the native Flutter widgets, comprising the structural UI components, gesture detectors, state management components, etc. The **third layer** is where all the UI and state rendering occur. This layer includes all the visible components of the Flutter application. Following that, it consists of animations used in transitions, image flow, and gestures. It further goes on to the very high level of system design.



Gestures

Pre-defined gestures are **used for all the physical form of interaction with a Flutter application**. Gesture-Detectors are used for the same function. It is an invisible widget used for processing physical interaction with the Flutter application.

Some of the interactions are gestures like **tapping**, **dragging**, **swiping**, etc. Such features can be implemented for the creative enhancement of the user experiences of the application, making it perform the desired actions that were based on simple gestures.

Concept of State

Stateful-Widget is used for managing the state of a Flutter application. Similar to the concept of state in React-JS, the re-rendering of widgets specific to the state occurs whenever there is a state change, avoiding the re-rendering of the entire application in every state change of a widget.

Generally, the architecture of a Flutter app or Flutter framework consists of a combination of small and larger widgets that interact in conjuncture to build the application. **All of its layers are integral to its design and functionality**. Building an application in Flutter may be simple, but it is built with equally complex components as its core.

SETTING UP THE FLUTTER SDK

Proceed to the Flutter Documentation for the full guide in installing Flutter.

REFERENCES

- What Are the Different Types of Mobile Apps? And How Do You Choose?
- Flutter Tutorial - GeeksForGeeks
- Flutter Documentation

12. **Scrolling.** This provides the scrollability of to a set of other widgets that are not scrollable by default.
13. **Styling.** This deals with the theme, responsiveness, and sizing of the app.
14. **Text.** These displays text.

Types of Widgets

Stateless Widget

A stateless widget is an **immutable widget**, that when **once created, it cannot be modified**. With that said, its properties and state cannot be changed. They are used for static content or UI content that does not need any changes after time. Stateless widgets are characterized as immutable, no state, and lightweight.

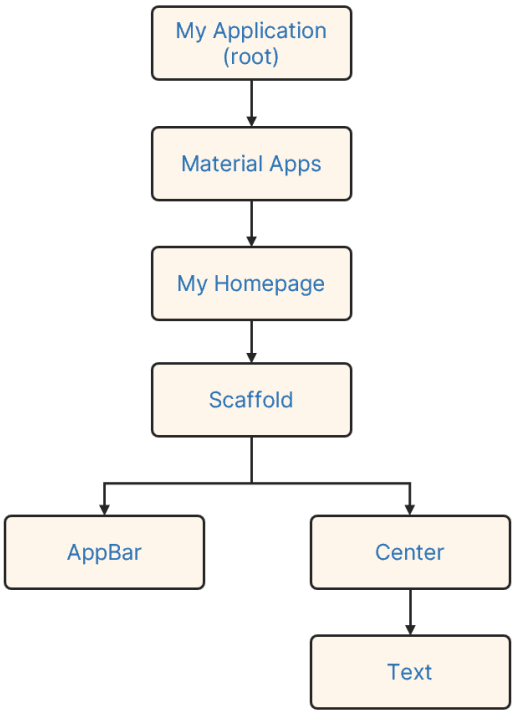
Some examples of stateless widgets are display text, icons, images, etc.

Stateful Widget

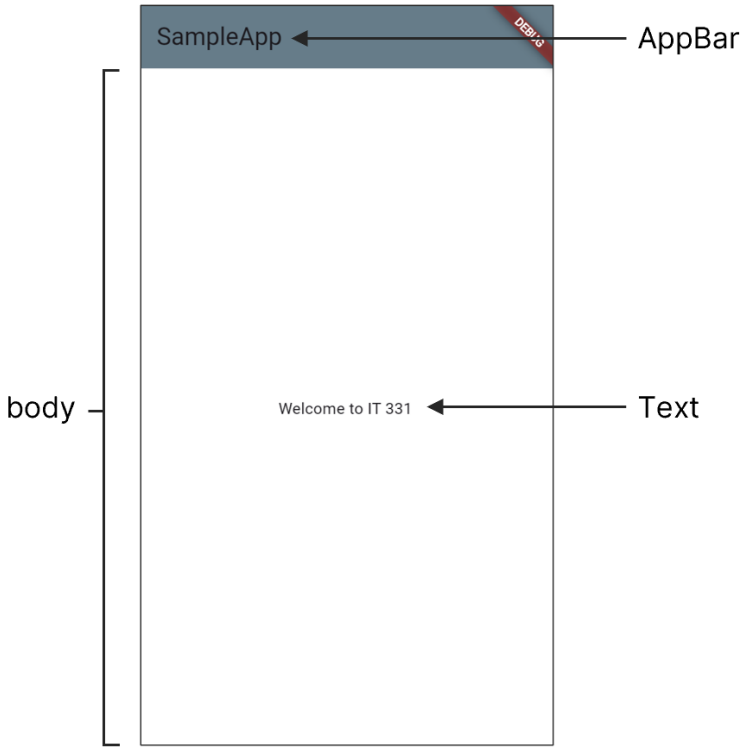
A stateful widget is a type of widget that **can change its state**, meaning, it is **able to maintain and update the appearance in the response to change state**. They are used for dynamic changes in properties and appearance over time. Stateful widgets are characterized as mutable state, state lifecycle, and dynamic updates.

Some examples of stateful widgets are buttons, sliders, text fields, etc.

Implementation of Stateless and Stateful Widget



Output



To sum up, the **StatelessWidget** is used when the UI does not depend on any changeable data. Meanwhile, the **StatefulWidget** is used when the UI needs dynamic updates, like responding to user input or changing over time. The second example, **StatefulWidget** is more flexible for future changes and dynamic content.

Container Class

In Flutter, a Container class is a convenience widget combining common painting, positioning, and sizing of widgets. It can be used to store one or more widgets and position them on the screen according to your convenience.

In other words, a container is like to a box for storing contents. A basic container element storing a widget has the following:

- **Margin.** Separates the present container from other contents.
- **Border.** Can be given, of different shapes, for example, rounded rectangle, etc.
- **Padding.** Adds padding inside the container, around the child widget. Additional constraints are applied to the padded extent, incorporating the width and height as constraints, if either is non-null.

```
    this.floatingActionButtonLocation,
    this.floatingActionButtonAnimator,
    this.persistentFooterButtons,
    this.drawer,
    this.endDrawer,
    this.bottomNavigationBar,
    this.bottomSheet,
    this.backgroundColor,
    this.resizeToAvoidBottomPadding,
    this.resizeToAvoidBottomInset,
    this.primary = true,
    this.drawerDragStartBehavior = DragStartBehavior.start,
    this.extendBody = false,
    this.drawerScrimColor,
})
```

Properties of the Scaffold Class

The Scaffold class in Flutter provides a framework for implementing the basic material design visual layout structure of an application. Here are some key properties of the Scaffold class.

App-Bar

Displays a horizontal bar which mainly placed at the top of the Scaffold. appBar uses the widget AppBar which has its own properties like elevation, title, brightness, etc.

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

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Colors.amber,
          title: const Text('Bonjour!'),
        ),
        body: const Center(
```

```
          child: Text(
            'IT 331: Application Development and Emerging Technologies',
            style: TextStyle(fontSize: 20),
          ),
        ),
      ),
    );
  }
}
```

This code results to an output:

Body

displays the main or primary content in the Scaffold. It is below the appBar and under the floatingActionButton. The widgets inside the body are at the left-corner by default.

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Colors.amber,
          title: const Text('Bonjour!'),
        ),
        body: const Center(
          child: Text(
            'IT 331: Application Development and Emerging Technologies',
            style: TextStyle(fontSize: 20),
          ),
        ),
      ),
    );
  }
}
```

Output:

In the example, the text **IT 331: Application Development and Emerging Technologies** was displayed in the body. It was placed at the center of the page by using the Center widget. The **TextStyle** widget was used for styling the text.

floatingActionButton

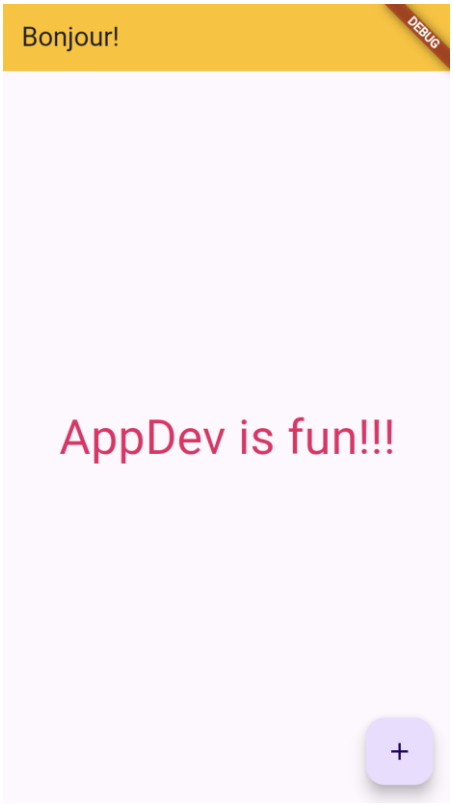
The floatingActionButton is a button placed at the right bottom corner by default. It is an icon button that floats over the content of the screen at a fixed place. When scrolling the page, it's position remains the same, hence it is fixed.

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

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Colors.amber,
          title: const Text('Bonjour!'),
        ),
        body: const Center(
                child: Text( "AppDev is fun!!!",
        style: TextStyle(
          color: Colors.pink,
          fontSize: 40.0,
        ),
      ),
    ),
    floatingActionButton: FloatingActionButton(
      elevation: 10.0,
      child: const Icon(Icons.add),
      onPressed: () {
        // action on button press
      },
    ),
        ),
      );
  }
}
```



The elevation property was used for the shadow effect to the button. The icon is used to put the button's icon, using preloaded icons in Flutter SDK. Meanwhile, the onPressed() is a method to be called after pressing the button, and the statements inside the function is executed.

Drawer

A drawer is a slider menu or panel that is displayed on the Scaffold's side. To access the drawer menu, the user has to swipe left to right, or right to left, depending on the action defined to access the Drawer menu.

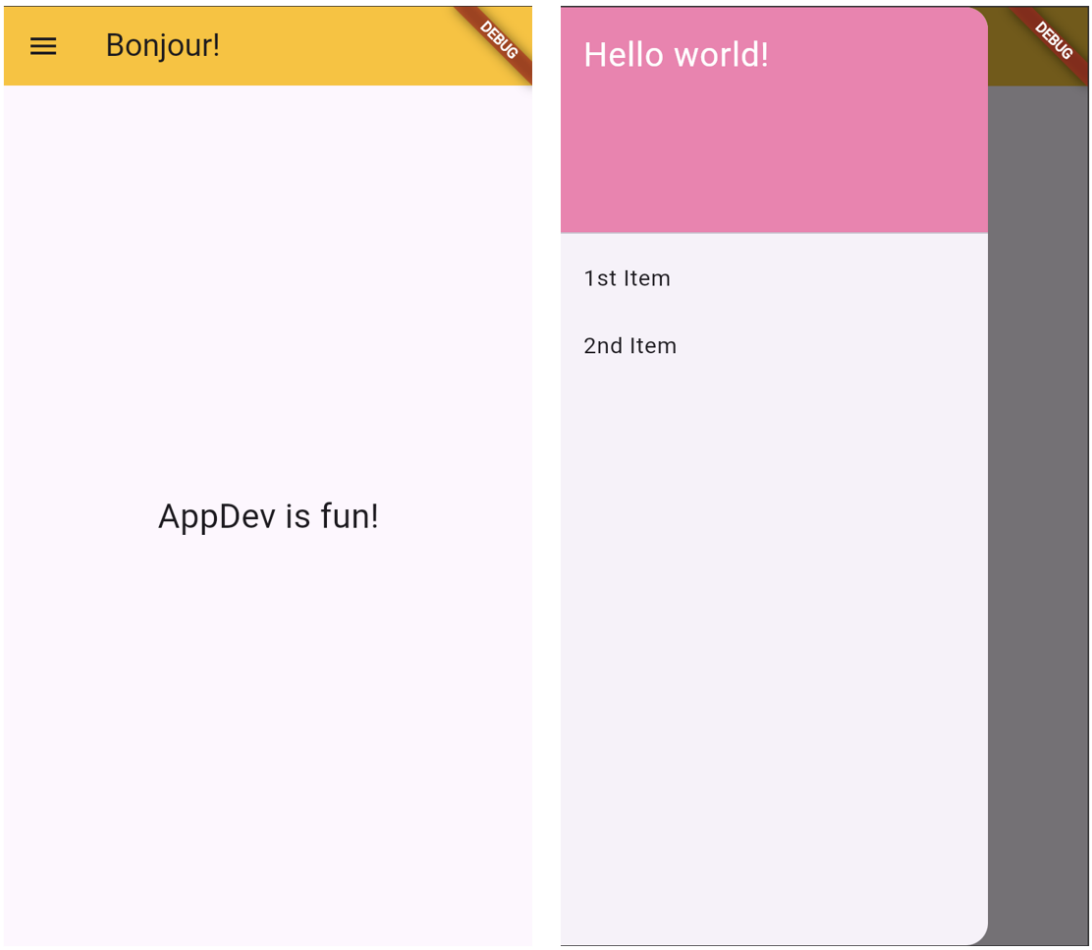
In the AppBar, there is already an appropriate icon for the drawer set automatically, and at a particular position. Even the gesture to open the drawer is also set automatically, and is being handled by the Scaffold.

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

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
```

```dart
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Colors.amber,
          title: const Text('Bonjour!'),
        ),
        body: const Center(
          child: Text(
            'AppDev is fun!',
            style: TextStyle(fontSize: 24),
          ),
        ),
        drawer: Drawer(
          child: ListView(
            padding: EdgeInsets.zero,
            children: const <Widget>[
              DrawerHeader(
                decoration: BoxDecoration(
                  color: Color.fromARGB(255, 248, 126, 177),
                ),
                child: Text(
                  'Hello world!',
                  style: TextStyle(
                    color: Colors.white,
                    fontSize: 24,
                  ),
                ),
              ),
              ListTile(
                title: Text('1st Item'),
              ),
              ListTile(
                title: Text('2nd Item'),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```



The ListView is the parent widget, and in it, there are two panels divided, namely Header and Menu.DrawerHeader that is used to modify the header of the panel.

In the Header, the details of the user according to the application or the icon can be displayed. ListTile was used to add items to the menu. Icons can also be added before the items, using the ListTile leading property, and inside it, the Icon widget should be used.

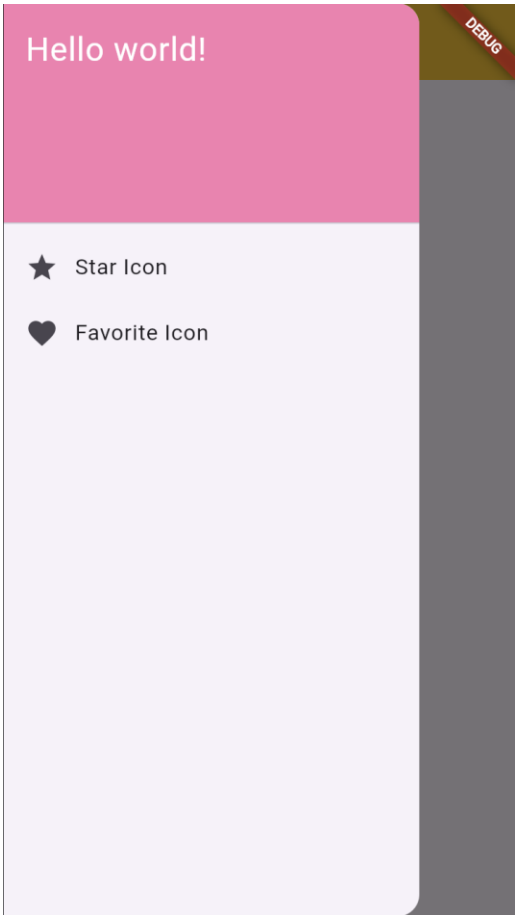
```
import 'package:flutter/material.dart';

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
```

```
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Colors.amber,
          title: const Text('Bonjour!'),
        ),
        body: const Center(
          child: Text(
            'AppDev is fun!',
            style: TextStyle(fontSize: 24),
          ),
        ),
        drawer: Drawer(
          child: ListView(
            padding: EdgeInsets.zero,
            children: const <Widget>[
              DrawerHeader(
                decoration: BoxDecoration(
                  color: Color.fromARGB(255, 248, 126, 177),
                ),
                child: Text(
                  'Hello world!',
                  style: TextStyle(
                    color: Colors.white,
                    fontSize: 24,
                  ),
                ),
              ),
              ListTile(
              title: Text('Star Icon'),
                leading : Icon(Icons.star), ),
              ListTile(
                  title: Text('Favorite Icon'),
                    leading: Icon(Icons.favorite), ),
            ],
          ),
        ),
      ),
    );
  }
}
```



bottomNavigationBar

bottomNavigationBar is like a menu at the bottom of the Scaffold. We have seen this navigationbar in most of the applications. We can add multiple icons or texts or both in the bar as items.

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

void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyHomePage(),
    );
```

FLUTTER UI COMPONENTS

Topic Outcomes

In this lesson, students will be able to implement various UI components in Flutter, such as tabs, horizontal lists, icons, dialogs, progress indicators, and staggered grid views, to create rich and interactive user interfaces.

Tabs

In Flutter, Tabs are referred to a part of the UI allowing the users to navigate in various routes, or pages, when clicked upon. Using them in applications is a standard practice. Using the **material library**, tab layouts can be created in Flutter.

For better understanding of the concept of tabs and its functionality in a Flutter application, try building a simple app with 5 tabs, following these steps.

1. Design a TabController
2. Add tabs to the app.
3. Add content in each tab.

Designing a TabController

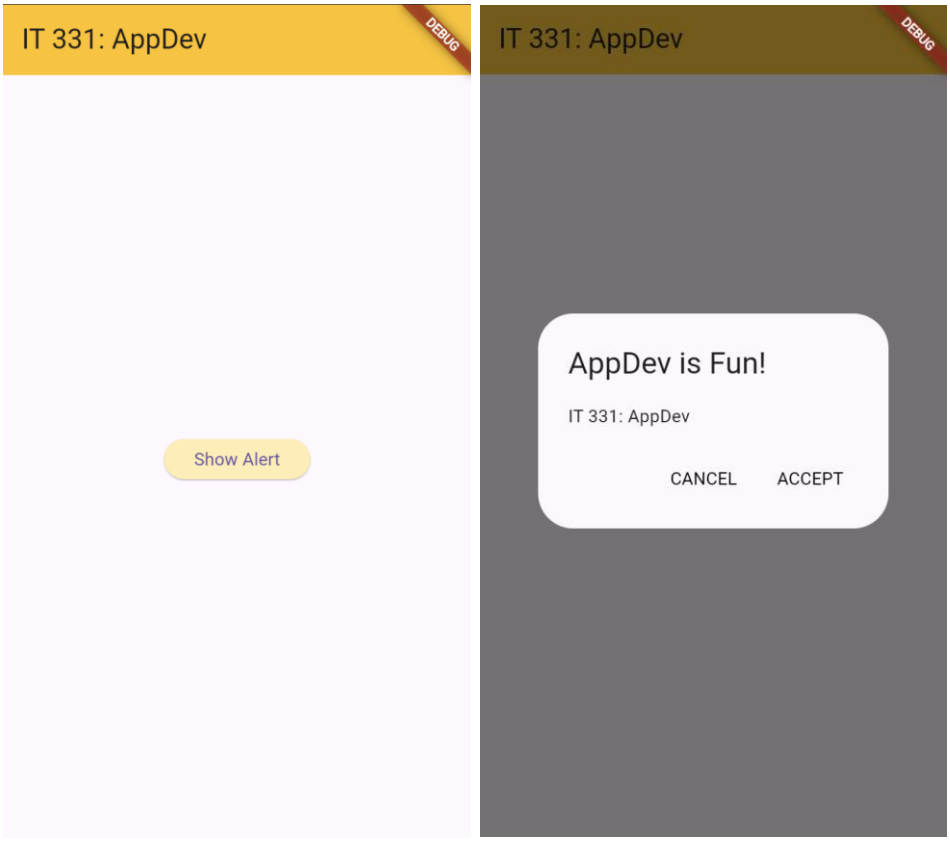
The **TabController** is responsible for controlling the functionality of each tab by syncing them and its contents with each other. The **DefaultTabController** widget is one of the simplest ways to use for tab creation in Flutter. Here's how it's done:

```
DefaultTabController(
// total 5 tabs
length: 5,
child:
);
```

Adding Tabs to the App

The **TabBar widget** is used for creating tabs in Flutter.

```
home: DefaultTabController(
length: 5,
child: Scaffold(
      appBar: AppBar(
      bottom: const TabBar(
            tabs: [
```



Comparisons

AlertDialog

- This is used for showing any type of alert notification.
- An option to react to the alert may be provided, for instance, **Accept** or **Reject** buttons.

SimpleDialog

- This is used for showing simple options in a form of a dialog box.
- Meaning, there are various options to choose from, and perform actions according to them.
- For instance, choosing between different accounts/emails.

ShowDialog

- This is used for creating an option that pops up a dialog box.
- When used, alert dialog and simple dialog can be pop-up as well, as a sub-widget.