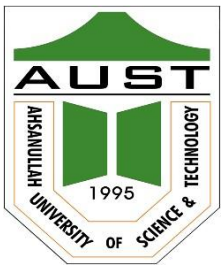

Ahsanullah University of Science and Technology (AUST)
Department of Computer Science and Engineering

LABORATORY MANUAL

Course No.: CSE 2100
Course Title: Software Development - II Lab

For the students of 2nd Year, 1st semester of
B.Sc. in Computer Science and Engineering program

TABLE OF CONTENTS

COURSE OUTCOMES

After successful completion of this course, the students should be able to

- Understand the basics of Flutter development, including widgets, layouts, and state management.
- Build simple Flutter apps, such as a to-do list or a calculator.
- Use Flutter packages to extend the functionality of their apps.

PREFFERED TOOL(S)

1. Android Studio

REFERENCE Website(s)

1. https://dart.dev/language
2. https://docs.flutter.dev/

ADMINISTRATIVE POLICY OF THE LABORATORY

1. Students must perform class assessment tasks individually.
2. Viva will be taken for each assignment and marks on assignment will substantially depend on viva.
3. Plagiarism is strictly forbidden and will be dealt with punishment.

Session 01

Introduction to Mobile Application development

"Mobile application development with Flutter" is a versatile and efficient way to create mobile apps for various platforms, including iOS, Android, and even the web. Flutter, an open-source UI framework developed by Google, allows developers to write code once and run it on multiple platforms, making it a popular choice for mobile app development.
Here's a brief introduction to mobile application development with Flutter:

What is Flutter?

- Flutter is an open-source UI software development kit (SDK) that is used to build natively compiled applications for mobile, web, and desktop from a single codebase. It is written in the Dart programming language and is maintained by Google.

Key Features of Flutter:

- **Single Codebase:** Flutter allows developers to write one set of code for multiple platforms, reducing development time and costs.
- **Hot Reload:** Flutter's hot reload feature lets you instantly see the results of your code changes, making development and testing more efficient.
- **Rich Set of Widgets:** Flutter provides a wide range of customizable widgets that help in creating beautiful and responsive user interfaces.
- **Native Performance:** Flutter compiles to native ARM code, providing excellent performance and a native look and feel.
- **Strong Community:** Flutter has a thriving and supportive developer community, with a wealth of resources, packages, and plugins available.

Setting up the Development Environment and creating first Flutter app:

Setting up a development environment for Flutter involves several steps. Here's a guide to help you get started:

System Requirements:

- Make sure your development machine meets the system requirements for Flutter. You'll need a macOS, Windows, or Linux computer.

Install Flutter:

- Visit the Flutter website (https://flutter.dev) and download the Flutter SDK for your operating system.
- Extract the downloaded archive to a location on your system. For example, on macOS and Linux, you can unzip it to your home directory, and on Windows, you can extract it to a location like `C:\`.
- Add the Flutter `bin` directory to your system's `PATH` environment variable. This step allows you to run Flutter commands from any directory in your terminal. For example, on macOS or Linux, you can add the following line to your `~/.bashrc` or `~/.zshrc` file:

```
export PATH="$PATH:`pwd`/flutter/bin"
```

On Windows, you can add the path to the `flutter/bin` directory to the system's Environment Variables.

Install Visual Studio Code (Optional):

- While you can use any code editor, Visual Studio Code is a popular choice for Flutter development due to its excellent Flutter and Dart extensions. You can download it from the Visual Studio Code website.

Install Dart:

- Flutter uses the Dart programming language. Flutter SDK includes a compatible version of Dart, so you don't need to install Dart separately.

Verify Your Installation:

- Open a terminal and run the following command to verify that Flutter is correctly installed:

```
flutter --version
```

Set Up an Emulator or Physical Device:

- To run and test Flutter applications, you can use an emulator or a physical device. If you want to use an emulator, you can set up Android Emulator (for Android development) or an iOS Simulator (for iOS development) as per your needs.

Install Android Studio (Optional):

- If you plan to develop Android applications with Flutter, you can install Android Studio, which includes Android SDK and tools. This is not necessary if you're only targeting iOS or web.

Initialize Flutter:

- In your terminal, navigate to the directory where you want to create your Flutter projects and run the following command to initialize Flutter:

```
flutter doctor
```

- The `flutter doctor` command will check for any missing dependencies and provide instructions on how to install them.

Install Plugins (Optional):

- If you're using Visual Studio Code, you can install the Flutter and Dart extensions to enhance your development experience.

Create Your First Flutter Project:

- After completing the above steps, you can create your first Flutter project using the following command:

```
flutter create my_flutter_app
```

Run Your Flutter Application:

- Navigate to your project's directory and run your app on an emulator or physical device using the `flutter run` command:

```
cd my_flutter_app

flutter run
```

Congratulations, you've set up your Flutter development environment! Now you can start building Flutter applications.

Introduction to Dart Programming Language:

Dart is a client-optimized programming language for developing fast apps on any platform. It has a modern syntax that is easy to learn and use, and it is supported by a wide range of tools and libraries.

Here are some of the basics of the Dart language, with examples:

Variables and Data Types

Dart is a strongly typed language, which means that all variables must have a declared type. There are several primitive data types in Dart, including int, double, bool, and String. You can also create your own custom data types using classes and interfaces.

To declare a variable, you use the `var` keyword, followed by the variable name and its type. For example:

```
var name = "John Doe";

var age = 30;
```

Functions

Functions are reusable blocks of code that can be called with parameters and return values. To define a function in Dart, you use the `function` keyword, followed by the function name and its parameters. The function body is enclosed in curly braces.

For example, here is a simple function that prints a message to the console:

```
void printMessage(String message) {
```

```
  print(message);}
```

To call the function, you simply use its name, followed by parentheses and any arguments that the function requires. For example:

```
printMessage("Hello, world!");
```

Classes and Objects

Classes are used to create reusable templates for objects. Objects are instances of classes, and they have their own unique state and behavior.

To define a class in Dart, you use the `class` keyword, followed by the class name and its body. The class body contains the fields and methods of the class.

For example, here is a simple class called `Person`:

```
class Person {

  String name;

  int age;

  Person(this.name, this.age);

  void printInfo() {

    print("$name is $age years old.");

  }

}
```

To create an instance of the `Person` class, you use the `new` keyword. For example:

```
var person = new Person("John Doe", 30);
```

You can then access the fields and methods of the object using the dot notation. For example:

```
person.name; // "John Doe"

person.age; // 30

person.printInfo(); // Prints "John Doe is 30 years old."
```

Control Flow

Dart supports all of the common control flow statements, such as `if`, `else`, `for`, `while`, and `do-while`.

For example, here is a simple `if` statement:

```
int number = 10;

if (number > 5) {

  print("The number is greater than 5.");

} else {

  print("The number is less than or equal to 5.");

}
```

Asynchronous Programming

Dart supports asynchronous programming, which allows you to perform tasks without blocking the main thread. This is useful for things like fetching data from the network or performing long-running calculations.

To use asynchronous programming in Dart, you use the `async` and `await` keywords. For example, here is a simple function that fetches a message from a server:

```
Future<String> fetchMessage() async {

  // Make a network request to the server.

  var response = await
http.get('https://example.com/message');



  // Decode the response and return the message.

  return utf8.decode(response.bodyBytes);

}
```

To call the `fetchMessage()` function, you use the `await` keyword. This will cause the main thread to wait until the function has finished executing and returned a value. For example:

```
var message = await fetchMessage();

print(message); // Prints the message from the server.
```

Conclusion

These are just a few of the basics of the Dart language. For more information, please see the official Dart documentation (https://dart.dev).

Discussion about app life-cycle In Flutter

The Flutter app lifecycle is the process that a Flutter app goes through from the time it is launched until it is terminated. This process is managed by the Flutter framework, and it is important to understand how it works in order to develop efficient and responsive apps.

The Flutter app lifecycle has four main states:

- Inactive: The app is in the background and is not receiving user input.
- Paused: The app is not currently visible to the user, but it is still running in the background.
- Resumed: The app is visible and responding to user input.
- Suspending: The app is about to be suspended momentarily.

The Flutter framework provides a number of methods that can be used to handle these lifecycle events. For example, you can use the `didChangeAppLifecycleState()` method to perform actions when the app transitions between states. You can also use the `dispose()` method to clean up any resources when the app is about to be terminated.

In addition to the four main states, the Flutter app lifecycle also has a number of other events that can occur. For example, the app may be restarted when the user's device is rebooted, or it may be paused when the user receives a phone call.

Here are some examples of how to handle Flutter app lifecycle events:

- To save data when the app is paused or suspended: You can use the `didChangeAppLifecycleState()` method to save data when the app transitions to the `paused` or `suspending` states. For example, you could save the state of the app's current screen.

- To release resources when the app is terminated: You can use the `dispose()` method to release any resources that your app is using, such as open files or network connections.
- To handle app restarts: You can use the `initState()` method to initialize your app's state when it is restarted. For example, you could load any saved data from the previous session.

It is important to handle Flutter app lifecycle events carefully in order to ensure that your app behaves as expected. For example, if you do not save your app's state when it is paused or suspended, users may lose their progress when they return to the app.

Here are some additional tips for handling the Flutter app lifecycle:

- Use the `didChangeAppLifecycleState()` method to listen for changes to the app's lifecycle state.
- Use the `dispose()` method to release any resources that your app is using when it is terminated.
- Be careful not to perform any long-running or blocking tasks when the app is in the `paused` or `suspending` states.
- Use the `initState()` method to initialize your app's state when it is restarted.

By following these tips, you can develop Flutter apps that are efficient and responsive.

Session 02

Handling user input in Flutter

There are a number of ways to handle user input in Flutter. The most common way is to use widgets such as TextFields, Buttons, Radio Buttons, Checkboxes, and Sliders. These widgets provide built-in event handlers that can be used to respond to user input.

For example, the TextField widget has an `onChanged` event handler that can be used to respond to changes in the text field's value. The Button widget has an `onPressed` event handler that can be used to respond to button taps.

Here is an example of how to use the TextField widget to handle user input:

```
class MyApp extends StatelessWidget {

  TextEditingController _nameController =
TextEditingController();



  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Handling User Input'),

      ),

      body: Center(

        child: Column(

          mainAxisAlignment: MainAxisAlignment.center,

          children: [
```

```
            TextField(
              controller: _nameController,
              decoration: InputDecoration(
                hintText: 'Enter your name',
              ),
              onChanged: (value) {
                print('Name: $value');
              },
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {
                print('Hello, ${_nameController.text}!');
              },
              child: Text('Submit'),
            ),
          ],
        ),
      ),
    );
  }
}
```

In this example, the `nameController` variable is used to track the text that the user enters into the TextField. The `onChanged` event handler is used to print the current value of the text field to the console.

When the user taps the "Submit" button, the `onPressed` event handler is triggered, which prints a greeting to the console.

In addition to using built-in event handlers, you can also use the `GestureDetector` widget to handle user input. The `GestureDetector` widget allows you to listen for a variety of gestures, such as taps, double taps, long presses, and drags.

Here is an example of how to use the `GestureDetector` widget to handle user input:

```
class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Handling User Input'),

      ),

      body: Center(

        child: GestureDetector(

          onTap: () {

            print('Tapped!');

          },

          child: Text('Tap me!'),

        ),

      ),

    );

  }

}
```

In this example, the `GestureDetector` widget is used to listen for taps on the screen. When the user taps the screen, the `onTap` event handler is triggered, which prints "Tapped!" to the console.

Whichever method you choose to use, handling user input is a fundamental part of developing Flutter apps. By understanding how to handle user input, you can create apps that are responsive and engaging.

Managing State in flutter

State management in Flutter is the process of managing the data that is used to render the app's UI. This data can include things like the text of a button, the selected item in a list, or the current position of a slider.

There are a number of different ways to manage state in Flutter. The most common way is to use stateful widgets. Stateful widgets have a `state` object that stores the widget's state. When the state changes, the widget is rebuilt.

Here is an example of a stateful widget that displays a counter:

```
class CounterWidget extends StatefulWidget {

  @override

  _CounterWidgetState createState() =>
_CounterWidgetState();

}



class _CounterWidgetState extends State<CounterWidget> {

  int _counter = 0;



  void increment() {

    setState(() {

      _counter++;

    });

  }
```

```
  @override
  Widget build(BuildContext context) {
    return Text('Counter: $_counter');
  }
}
```

In this example, the `_counter` variable stores the current value of the counter. The `increment()` method increments the counter and then calls the `setState()` method to rebuild the widget.

When the widget is rebuilt, the `build()` method is called, which returns a Text widget that displays the current value of the counter.

In addition to using stateful widgets, there are a number of other ways to manage state in Flutter. Some popular state management libraries include:

- Provider: Provider is a simple and effective state management library. It uses a change notification system to notify widgets when their state changes.
- Riverpod: Riverpod is a newer state management library that is built on top of Provider. It offers a number of features that make it easier to manage state in complex apps.
- MobX: MobX is a state management library that is based on the functional reactive programming (FRP) paradigm. It offers a number of features that make it easy to manage state in concurrent and distributed apps.

Which state management solution you choose will depend on your specific needs and preferences. If you are new to Flutter, I recommend starting with Provider. It is a simple and easy-to-learn library that is well-suited for most apps.

Here are some additional tips for managing state in Flutter:

- Use stateful widgets to manage the state of individual widgets.
- Use state management libraries to manage the state of multiple widgets.
- Avoid storing state in global variables. This can make your code difficult to understand and maintain.
- Use change notification systems to notify widgets when their state changes. This will ensure that your UI is always up-to-date.

Styling the app in flutter

There are a number of ways to style an app in Flutter. The most common way is to use themes. Themes allow you to define the overall look and feel of your app, such as the colors, fonts, and text styles.

To create a theme, you use the `ThemeData` class. The `ThemeData` class has a number of properties that can be used to configure the theme, such as `primaryColor`, `accentColor`, and `textTheme`.

Once you have created a theme, you can apply it to your app using the `Theme` widget. The `Theme` widget wraps your app's widgets and applies the specified theme to them.

Here is an example of how to create and apply a theme:

```
class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Theme(

      data: ThemeData(

        primaryColor: Colors.blue,

        accentColor: Colors.red,

        textTheme: TextTheme(

          bodyText1: TextStyle(

            fontSize: 16,

            color: Colors.black,

          ),

        ),
```

```
      ),
      child: MaterialApp(
        home: Scaffold(
          appBar: AppBar(
            title: Text('My App'),
          ),
          body: Center(
            child: Text('This is my app!'),
          ),
        ),
      ),
    );
  }
}
```

In this example, the `ThemeData` class is used to create a theme with a blue primary color, a red accent color, and a 16px black body text style. The `Theme` widget is then used to apply the theme to the `MyApp` widget.

As a result, all of the widgets in the `MyApp` widget will use the blue primary color, the red accent color, and the 16px black body text style.

In addition to using themes, there are a number of other ways to style an app in Flutter. You can use the `TextStyle` class to style text, the `ShapeDecoration` class to style shapes, and the `BoxDecoration` class to style backgrounds.

You can also use custom widgets to style your app. For example, you could create a custom button widget that has a specific style.

Whichever method you choose to use, styling your app is an important part of the Flutter development process. By carefully styling your app, you can create a beautiful and engaging user experience.

Here are some additional tips for styling your Flutter app:

- Use themes to define the overall look and feel of your app.
- Use the `TextStyle` class to style text.
- Use the `ShapeDecoration` class to style shapes.
- Use the `BoxDecoration` class to style backgrounds.
- Use custom widgets to style your app in a unique way.
- Be consistent with your styling throughout your app.
- Use colors and fonts that are easy to read and accessible.
- Test your styling on different devices and screen sizes.

Session 03

Working with different components in Flutter

Layouts in Flutter are used to arrange widgets on the screen. There are a number of different layout widgets available, such as `Row`, `Column`, `Stack`, and `GridView`.

Colors in Flutter are represented by the `Color` class. The `Color` class has a number of properties that can be used to define a color, such as `red`, `green`, `blue`, and `alpha`.

Fonts in Flutter are represented by the `TextStyle` class. The `TextStyle` class has a number of properties that can be used to define a font, such as `fontFamily`, `fontSize`, and `fontWeight`.

Images in Flutter are represented by the `Image` widget. The `Image` widget can be used to display images from a variety of sources, such as the local filesystem, the network, or a memory buffer.

Assets in Flutter are resources that are bundled with your app. Assets can include things like images, fonts, and sounds.

Buttons in Flutter are represented by the `Button` widget. The `Button` widget can be used to create a variety of different types of buttons, such as text buttons, elevated buttons, and floating action buttons.

Icons in Flutter are represented by the `Icon` widget. The `Icon` widget can be used to display icons from a variety of icon fonts, such as Material Icons and Cupertino Icons.

Containers in Flutter are used to group widgets together. Containers can be used to add things like padding, margins, borders, and backgrounds to widgets.

Rows in Flutter are used to arrange widgets in a horizontal line. The `Row` widget can be used to create a variety of different types of layouts, such as toolbars, bottom navigation bars, and tab bars.

Columns in Flutter are used to arrange widgets in a vertical line. The `Column` widget can be used to create a variety of different layouts, such as lists, grids, and cards.

Cards in Flutter are used to display content in a visually appealing way. Cards can be used to display things like images, text, and buttons.

Here is an example of a simple Flutter layout that uses all of the widgets mentioned above:

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

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      home: Scaffold(

        appBar: AppBar(

          title: Text('My App'),

        ),

        body: Center(

          child: Card(

            child: Column(

              children: [


Image.asset('assets/images/my_image.png'),

                Text('This is my card'),

                Row(

                  mainAxisAlignment:
MainAxisAlignment.center,

                  children: [

                    TextButton(

                      onPressed: () {},
```

```
                    child: Text('Like'),
                  ),
                  TextButton(
                    onPressed: () {},
                    child: Text('Comment'),
                  ),
                ],
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

This layout uses a `Card` widget to display an image, some text, and two buttons. The `Card` widget is centered on the screen using a `Center` widget.

The `Column` widget is used to arrange the widgets inside the `Card` widget in a vertical line. The `Row` widget is used to arrange the two buttons in a horizontal line.

The `TextButton` widget is used to create the two buttons. The `TextButton` widget is a simple type of button that has a text label.

The `onPressed` callback is used to handle button taps. In this example, the `onPressed` callback is empty, but it could be used to perform some action, such as navigating to a new screen or liking a post.

This is just a simple example of how to use layouts, colors, fonts, images, assets, buttons, icons, containers, rows, columns, and cards in Flutter. There are many other ways to use these widgets to create complex and visually appealing layouts.

Page Navigation in Flutter

Page navigation in Flutter is the process of moving between different screens in your app. There are a number of different ways to navigate between pages, but the most common way is to use the `Navigator` widget.

The `Navigator` widget is responsible for managing the stack of pages in your app. It can be used to push new pages onto the stack, pop pages off the stack, and replace the current page with a new page.

To push a new page onto the stack, you use the `Navigator.push()` method. The `Navigator.push()` method takes a `Route` object as its argument. A `Route` object represents a single page in your app.

To pop a page off the stack, you use the `Navigator.pop()` method. The `Navigator.pop()` method removes the current page from the stack and returns to the previous page.

To replace the current page with a new page, you use the `Navigator.replace()` method. The `Navigator.replace()` method takes a `Route` object as its argument and replaces the current page with the new page.

Here is an example of how to navigate between pages using the `Navigator` widget:

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

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      home: Scaffold(

        appBar: AppBar(

          title: Text('My App'),
```

```
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              TextButton(
                onPressed: () {
                  Navigator.push(
                    context,
                    MaterialPageRoute(builder: (context)
=> MySecondPage()),
                  );
                },
                child: Text('Go to MySecondPage'),
              ),
            ],
          ),
        ),
      ),
    );
  }
}

class MySecondPage extends StatelessWidget {
  @override
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('MySecondPage'),
      ),
      body: Center(
        child: Text('This is MySecondPage'),
      ),
    );
  }
}
```

In this example, the `MyApp` widget has a button that navigates to the `MySecondPage` widget when it is tapped.

When the button is tapped, the `Navigator.push()` method is used to push the `MySecondPage` route onto the stack. This displays the `MySecondPage` widget on top of the `MyApp` widget.

To return to the `MyApp` widget, the user can tap the back button on the device. This will call the `Navigator.pop()` method, which will pop the `MySecondPage` route off the stack and return to the `MyApp` widget.

In addition to using the `Navigator` widget, there are a number of other ways to navigate between pages in Flutter. For example, you can use the `NamedRoute` widget to define named routes. Named routes are routes that have a unique name. You can then use the `Navigator.pushNamed()` method to navigate to a named route.

You can also use the `Router` widget to implement custom routing logic. The `Router` widget is a more powerful way to navigate between pages, but it is also more complex to use.

Whichever method you choose to use, page navigation is an important part of developing Flutter apps. By understanding how to navigate between pages, you can create apps that are easy to use and navigate.

Here are some additional tips for page navigation in Flutter:

- Use the `Navigator` widget to manage the stack of pages in your app.
- Use the `Navigator.push()` method to push new pages onto the stack.
- Use the `Navigator.pop()` method to pop pages off the stack.
- Use the `Navigator.replace()` method to replace the current page with a new page.
- Use the `NamedRoute` widget to define named routes.
- Use the `Navigator.pushNamed()` method to navigate to a named route.
- Use the `Router` widget to implement custom routing logic.
- Be consistent with your navigation patterns throughout your app.
- Use clear and concise labels for your navigation elements.
- Make it easy for users to navigate back to previous pages.

Session 04

Networking and HTTP in flutter

Networking and HTTP in Flutter are used to fetch data from and send data to servers over the internet. This is essential for many Flutter apps, such as social media apps, e-commerce apps, and weather apps.

To use networking and HTTP in Flutter, you can use the `http` package. The `http` package provides a simple and easy-to-use way to make HTTP requests.

To use the `http` package, you first need to add it to your project's `pubspec.yaml` file. Once you have added the package, you can import it into your code:

```
import 'package:http/http.dart' as http;
```

To make an HTTP request, you can use the `http.get()`, `http.post()`, `http.put()`, or `http.delete()` methods. These methods take a URL as their argument and return a `Future` object that resolves to an `http.Response` object.

The `http.Response` object contains the response from the server. You can use the `Response.body` property to get the body of the response.

Here is an example of how to make an HTTP request using the `http` package:

```
Future<http.Response> fetchPosts() async {
  return http.get('https://example.com/posts');
}


void main() async {
  final response = await fetchPosts();
  if (response.statusCode == 200) {
    // Success!
  } else {
```

```
    // Something went wrong.
  }
}
```

In this example, the `fetchPosts()` function makes an HTTP GET request to the `/posts` endpoint on the `example.com` server. The function returns a `Future` object that resolves to an `http.Response` object.

The `main()` function calls the `fetchPosts()` function and awaits the result. If the request is successful, the `response.statusCode` property will be equal to 200. Otherwise, the `response.statusCode` property will be a different value.

In addition to the `http` package, there are a number of other packages that can be used for networking and HTTP in Flutter. Some popular packages include:

- `dio`: A more powerful and feature-rich HTTP library.
- `retrofit`: A library that generates code to make HTTP requests based on a REST API definition.
- `graphql_flutter`: A library for making GraphQL requests.

Whichever package you choose to use, networking and HTTP are essential for many Flutter apps. By understanding how to use networking and HTTP in Flutter, you can create apps that can communicate with servers and fetch and send data.

Working with APIs in Flutter

To work with APIs in Flutter, you can use the `http` package. The `http` package provides a simple and easy-to-use way to make HTTP requests.

To use the `http` package, you first need to add it to your project's `pubspec.yaml` file. Once you have added the package, you can import it into your code:

```
import 'package:http/http.dart' as http;
```

To make an HTTP request to an API, you can use the `http.get()`, `http.post()`, `http.put()`, or `http.delete()` methods. These methods take a URL as their argument and return a `Future` object that resolves to an `http.Response` object.

The `http.Response` object contains the response from the API. You can use the `Response.body` property to get the body of the response.

Here is an example of how to make an HTTP GET request to an API using the `http` package:

```
Future<http.Response> fetchTodos() async {

  return http.get('https://api.example.com/todos');

}



void main() async {

  final response = await fetchTodos();

  if (response.statusCode == 200) {

    // Success!

    final todos = json.decode(response.body) as
List<dynamic>;

    // Do something with the todos.

  } else {

    // Something went wrong.

  }

}
```

In this example, the `fetchTodos()` function makes an HTTP GET request to the `/todos` endpoint on the `api.example.com` server. The function returns a `Future` object that resolves to an `http.Response` object.

The `main()` function calls the `fetchTodos()` function and awaits the result. If the request is successful, the `response.statusCode` property will be equal to 200. Otherwise, the `response.statusCode` property will be a different value.

If the request is successful, you can use the `json.decode()` function to decode the JSON response into a Dart object. Once you have decoded the response, you can use the data in your app.

In addition to the `http` package, there are a number of other packages that can be used for working with APIs in Flutter. Some popular packages include:

- `dio`: A more powerful and feature-rich HTTP library.
- `retrofit`: A library that generates code to make HTTP requests based on a REST API definition.
- `graphql_flutter`: A library for making GraphQL requests.

Whichever package you choose to use, working with APIs is essential for many Flutter apps. By understanding how to work with APIs in Flutter, you can create apps that can communicate with servers and fetch and send data.

Here are some additional tips for working with APIs in Flutter:

- Use the `http` package to make HTTP requests to APIs.
- Use the `json.decode()` function to decode JSON responses into Dart objects.
- Use the `dio`, `retrofit`, or `graphql_flutter` packages for more complex API requirements.
- Use documentation from the API provider to understand how to make requests and interpret the responses.
- Handle errors gracefully in your app.
- Use caching to improve the performance of your app.

Session 05

JSON Serialization in Flutter

JSON serialization in Flutter is the process of converting JSON data to Dart objects and vice versa. This can be done manually or using a library such as json_serializable.

To serialize JSON data manually, you can use the `dart:convert` library. The `dart:convert` library provides a `jsonEncode()` function to encode Dart objects to JSON and a `jsonDecode()` function to decode JSON data to Dart objects.

Here is an example of how to serialize JSON data manually:

```
import 'dart:convert';

class User {

  final String name;

  final int age;



  User({required this.name, required this.age});



  Map<String, dynamic> toJson() => {

    'name': name,

    'age': age,

  };



  factory User.fromJson(Map<String, dynamic> json) =>
User(

    name: json['name'],
```

```
    age: json['age'],
  );
}

void main() {
  final user = User(name: 'John Doe', age: 30);

  // Encode the user object to JSON.
  final jsonString = jsonEncode(user);

  // Decode the JSON string back to a user object.
  final userFromJson =
User.fromJson(jsonDecode(jsonString));

  // Print the user object.
  print(userFromJson);
}
```

Output:

```
User(name: John Doe, age: 30)
```

To serialize JSON data using the json_serializable library, you first need to add it to your project's `pubspec.yaml` file. Once you have added the library, you can import it into your code:

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

Next, you need to annotate your Dart classes with the `@JsonSerializable` annotation. This will generate code to serialize and deserialize your classes to and from JSON.

Here is an example of a Dart class with the `@JsonSerializable` annotation:

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

part 'user.g.dart';

@JsonSerializable()

class User {

  final String name;

  final int age;

  User({required this.name, required this.age});

}
```

Once you have annotated your classes, you need to run the `build_runner` command to generate the code to serialize and deserialize your classes.

Once the code has been generated, you can serialize and deserialize your classes to and from JSON using the `toJson()` and `fromJson()` methods.

Here is an example of how to serialize a `User` object to JSON using the json_serializable library:

```
import 'user.dart';

void main() {

  final user = User(name: 'John Doe', age: 30);

  // Serialize the user object to JSON.

  final jsonString = user.toJson();

  // Print the JSON string.

  print(jsonString);

}
```

Output:

```
{"name":"John Doe","age":30}
```

Here is an example of how to deserialize a JSON string to a `User` object using the json_serializable library:

```
import 'user.dart';

void main() {

  // The JSON string.

  final jsonString = '{"name":"John Doe","age":30}';

  // Deserialize the JSON string to a user object.

  final userFromJson = User.fromJson(jsonString);

  // Print the user object.

  print(userFromJson);

}
```

Output:

```
User(name: John Doe, age: 30)
```

The json_serializable library is a convenient way to serialize and deserialize JSON data in Flutter. It is especially useful for large and complex projects.

Reading and writing files in flutter

To read and write files in Flutter, you can use the `dart:io` library. The `dart:io` library provides a number of classes and methods for interacting with the filesystem.

To read a file, you can use the `File.readAsString()` method. This method takes the path to the file as its argument and returns the contents of the file as a string.

For example, to read the contents of the file `my_file.txt`, you would use the following code:

```
import 'dart:io';

void main() async {

  final file = File('my_file.txt');

  final contents = await file.readAsString();

  print(contents);

}
```

To write a file, you can use the `File.writeAsString()` method. This method takes the path to the file and the contents of the file as its arguments.

For example, to write the string "Hello, world!" to the file `my_file.txt`, you would use the following code:

```
import 'dart:io';

void main() async {

  final file = File('my_file.txt');

  await file.writeAsString('Hello, world!');

}
```

You can also use the `File.writeAsBytes()` method to write binary data to a file.

To read and write files in a more convenient way, you can use the `path_provider` plugin. The `path_provider` plugin provides a number of functions for getting the paths to common directories, such as the documents directory and the temporary directory.

For example, to read the contents of the file `my_file.txt` from the documents directory, you would use the following code:

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

void main() async {

  final documentsDirectory = await
getApplicationDocumentsDirectory();

  final filePath = join(documentsDirectory.path,
'my_file.txt');



  final file = File(filePath);

  final contents = await file.readAsString();



  print(contents);

}
```

To write the string "Hello, world!" to the file `my_file.txt` in the documents directory, you would use the following code:

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

void main() async {

  final documentsDirectory = await
getApplicationDocumentsDirectory();

  final filePath = join(documentsDirectory.path,
'my_file.txt');



  final file = File(filePath);

  await file.writeAsString('Hello, world!');

}
```

Reading and writing files is an important part of developing Flutter apps. By understanding how to read and write files, you can create apps that can store and retrieve data from the filesystem.

Here are some additional tips for reading and writing files in Flutter:

- Use the `dart:io` library for basic file operations.
- Use the `path_provider` plugin for a more convenient way to read and write files from common directories.
- Consider using encryption to protect your data.
- Back up your data regularly to avoid data loss.

Persisting of data with SQLite in Flutter:

To persist data with SQLite in Flutter, you can use the sqflite package. The sqflite package provides a simple and easy-to-use way to interact with SQLite databases.

To use the sqflite package, you first need to add it to your project's `pubspec.yaml` file. Once you have added the package, you can import it into your code:

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

Next, you need to create a database object. You can do this using the `openDatabase()` function. The `openDatabase()` function takes the path to the database file as its argument and returns a `Database` object.

Here is an example of how to create a database object:

```
Future<Database> openDatabase() async {

  final databasePath = await getDatabasesPath();

  final database = await openDatabase(

    join(databasePath, 'my_database.db'),

    onCreate: (database, version) async {

      // Create the database tables.

    },

  );
```

```
  return database;

}
```

Once you have created a database object, you can use it to insert, update, and delete data from the database.

To insert data into the database, you can use the `insert()` function. The `insert()` function takes the table name and a map of column names and values as its arguments.

Here is an example of how to insert data into the database:

```
Future<void> insertUser(String name, int age) async {

  final database = await openDatabase();

  await database.insert(

    'users',

    {

      'name': name,

      'age': age,

    },

  );

}
```

To update data in the database, you can use the `update()` function. The `update()` function takes the table name, a map of column names and values, and a where clause as its arguments.

Here is an example of how to update data in the database:

```
Future<void> updateUser(int id, String name) async {

  final database = await openDatabase();

  await database.update(
```

```
    'users',
    {
      'name': name,
    },
    where: 'id = ?',
    whereArgs: [id],
  );
}
```

To delete data from the database, you can use the `delete()` function. The `delete()` function takes the table name and a where clause as its arguments.

Here is an example of how to delete data from the database:

```
Future<void> deleteUser(int id) async {
  final database = await openDatabase();
  await database.delete(
    'users',
    where: 'id = ?',
    whereArgs: [id],
  );
}
```

The sqflite package is a powerful tool for persisting data with SQLite in Flutter. It is easy to use and provides a number of features that make it suitable for a variety of applications.

Here are some additional tips for persisting data with SQLite in Flutter:

- Use the sqflite package to interact with SQLite databases.
- Create a database object using the `openDatabase()` function.
- Insert, update, and delete data from the database using the `insert()`, `update()`, and `delete()` functions, respectively.
- Use the `where` clause to filter the results of your database queries.
- Use transactions to ensure that your database operations are atomic.
- Back up your database regularly to avoid data loss.

Session 06

Flutter with Firebase

Flutter with Firebase is a powerful combination that allows you to build high-quality, native mobile apps for iOS and Android. Flutter is a cross-platform development framework that provides a unified way to build native apps for multiple platforms from a single codebase. Firebase is a backend-as-a-service (BaaS) platform that provides a wide range of services, including authentication, database, storage, analytics, and more.

To use Flutter with Firebase, you first need to add the Firebase SDK to your Flutter project. You can do this using the Firebase CLI or the Flutter plugin system.

Once you have added the Firebase SDK to your project, you need to initialize Firebase in your app. You can do this by adding the following code to your `main()` function:

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

Future<void> main() async {

  WidgetsFlutterBinding.ensureInitialized();

  await Firebase.initializeApp();

  // Run your app.

}
```

Once Firebase has been initialized, you can start using the various Firebase services in your app. For example, you can use the Firebase Authentication service to authenticate users and the Firebase Realtime Database to store and sync data in real time.

Here is an example of a simple Flutter app that uses Firebase Authentication to authenticate users:

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

import 'package:firebase_core/firebase_core.dart';

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

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('MyApp'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Text('Welcome to MyApp!'),
              TextButton(
                onPressed: () async {
                  // Authenticate the user.
                  final user = await
FirebaseAuth.instance.signInWithEmailAndPassword(
                    email: 'john.doe@example.com',
                    password: 'password123',
                  );
```

```
                  // If the user is authenticated
successfully, navigate to the next screen.
                  if (user != null) {
                    Navigator.pushNamed(context,
'/next_screen');
                  }
                },
                child: Text('Login'),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

This app uses the Firebase Authentication service to authenticate users. When the user clicks the "Login" button, the app attempts to authenticate the user using the `signInWithEmailAndPassword()` method. If the user is authenticated successfully, the app navigates to the next screen.

Flutter with Firebase is a powerful combination that can be used to build a wide variety of high-quality mobile apps. By using Flutter and Firebase, you can develop apps that are fast, reliable, and scalable.

Here are some additional tips for using Flutter with Firebase:

- Use the Firebase CLI to manage your Firebase projects and services.
- Use the Flutter plugin system to add Firebase services to your Flutter project.

- Initialize Firebase in your app before you start using any of the Firebase services.
- Use the Firebase documentation to learn more about the various Firebase services and how to use them in your Flutter app.
- Test your Flutter app with Firebase thoroughly before releasing it to production.

Session 07

Things to keep in mind for final project submission

Here are some things to keep in mind for final project submission in Flutter:

- **Follow the submission guidelines** provided by your instructor or mentor. This may include specific requirements for code formatting, documentation, and testing.
- **Make sure your project is complete and functional.** This means that all of the features that you were assigned to implement should be working as expected.
- **Test your project** thoroughly on a variety of devices and screen sizes. This will help to ensure that your project is compatible with a wide range of users.
- **Write clear and concise documentation** for your project. This should include explanations of how to use your project, as well as any technical details that are relevant to your instructor or mentor.
- **Submit your project on time.** This shows that you are reliable and that you respect the deadlines that have been set for you.

Here are some additional tips for submitting a high-quality Flutter project:

- **Use a consistent coding style.** This will make your code easier to read and maintain.
- **Use descriptive variable names and function names.** This will help to make your code self-documenting.
- **Add comments to your code to explain what it is doing.** This will make your code easier to understand for yourself and others.
- **Use a version control system such as Git to track changes to your code.** This will allow you to roll back to previous versions of your code if necessary.
- **Deploy your project** to a production environment such as the Google Play Store or the Apple App Store if possible. This will allow your instructor or mentor to test your project in a real-world setting.

By following these tips, you can submit a high-quality Flutter project that is complete, functional, well-documented, and easy to use.