Chapter-1
Overview of Dart Packages

Introduction to Dart Packages

- **What are Dart Packages?**

  Dart packages are collections of reusable code that can be shared across different Dart projects. They help to modularize and reuse code efficiently.

- **Importance and Benefits of Using Packages**
  - **Reusability: Packages allow developers to reuse existing code, saving time and effort.**
  - **Modularity: Breaking down code into smaller, manageable pieces.**
  - **Community Support: Many packages are developed and maintained by the community, providing a wide range of functionalities.**
- **Types of Packages**
  - **Pub Packages: Available on the pub.dev repository. They can be easily added to any Dart project.**
  - **User-defined Packages: Custom packages created by developers to share specific functionalities across their own projects.**

Finding and Adding Packages from pub.dev

- **Finding Packages**
  - **Visit pub.dev to search for packages.**
  - **Use keywords or browse categories to find the desired package.**
- **Adding Dependencies to pubspec.yaml**

```
Dependencies:
yaml:
  http: ^0.14.0
```

- **Installing Packages**
  - **Run flutter pub get in the terminal to install the packages specified in pubspec.yaml.**

Using and Managing Packages

Adding Dependencies to `pubspec.yaml`

- **Understanding `pubspec.yaml`**

- The `pubspec.yaml` file is the configuration file for Dart and Flutter projects, specifying project dependencies and other settings.

- **Example of Adding Dependencies**

```
name: my_app
description: A new Flutter project.
dependencies:
  flutter:
    sdk: flutter
  provider: ^6.0.0
  http: ^0.14.0
```

Installing Packages

- After adding dependencies to `pubspec.yaml`, run `flutter pub get` to fetch and install the packages.

Creating and Importing User-Defined Packages

Structure of a Dart Package
Basic Structure

```
my_package/
├── lib/
│   ├── src/
│   │   └── my_package_base.dart
│   └── my_package.dart
├── test/
│   └── my_package_test.dart
├── pubspec.yaml
└── README.md
```

- **Explanation**
  - `lib/`: Contains the main code of the package.
  - `src/`: Subdirectory for implementation files.
  - `my_package.dart`: Entry point for the package.
  - `test/`: Directory for unit tests.
  - `pubspec.yaml`: Configuration file for the package.

Creating a Simple Package

1. **Create Package Directory Structure**
   - Use the following command to create a new package:

Command : dart create -t package-simple my_package

2. Implementing Package Functionality

- **Example: Create a simple utility function in lib/my_package.dart:**
  dart

```
library my_package;
int add(int a, int b) {
return a + b;
}
```

1. **Publishing a Package**
   - **Update pubspec.yaml with necessary information.**
   - **Run dart pub publish to publish the package to pub.dev.**

Importing and Using Custom Packages in a Flutter Project

1. **Add Package as Dependency**
   - **In the Flutter project, add the custom package to pubspec.yaml**

```
dependencies:
 flutter:
   sdk: flutter
 my_package:
   path: ../my_package
```

Import and Use the Package

Example:

```
import 'package:my_package/my_package.dart';
void main() {
 int result = add(3, 4);
 print('The result is $result');
}
```

Chapter-2
Asynchronous Programming with Future and Async/Await

Understanding Future

- **What is a Future?**
  - A `Future` represents a potential value or error that will be available at some time in the future.
  - It is used for asynchronous programming in Dart.
- **Creating a Future**

```
Future<String> fetchData() async {
  return 'Hello, Future!';
}
```

- **Using then() Method `then()` method is used to register callbacks when the future completes.**

```
fetchData().then((value) {
  print(value); // Output: Hello, Future!
});
```

Using async and await for Asynchronous Operations

- **async and await Keywords**
  - **`async` keyword is used to declare an asynchronous function.**
  - **`await` keyword is used to wait for the completion of a future.**

Example of async and await

```
Future<void> main() async {
  String data = await fetchData();
  print(data); // Output: Hello, Future!
}
```

Error Handling in Asynchronous Code

- **Use `try-catch` blocks to handle errors in asynchronous code.**

```
Future<void> main() async {
  try {
    String data = await fetchData();
    print(data);
  } catch (e) {
    print('Error: $e');
  }
}
Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 2)); // Simulate network delay
  return 'Data from server';
}
Future<void> main() async {
  print('Fetching data...');
  String data = await fetchData();
  print(data); // Output: Data from server
}
```

Chapter-3
REST API Basics

Web Server

A web server is a software application that handles HTTP requests from clients (typically web browsers) and serves HTTP responses. The primary functions of a web server are to store, process, and deliver web pages to clients. Popular web servers include Apache, Nginx, and Microsoft IIS.

Host

In the context of web services, a host refers to the domain name or IP address of the server where the web application is deployed. For example, in the URL http://www.example.com, www.example.com is the host.

HTTP Protocol

HTTP (Hypertext Transfer Protocol) is the foundation of any data exchange on the web. It is an application layer protocol for transmitting hypermedia documents, such as HTML. It is designed for communication between web browsers and web servers, but it can also be used for other purposes.

HTTP Request Methods

HTTP defines several request methods to perform different actions:

- **GET:** Requests data from a specified resource.
- **POST:** Submits data to be processed to a specified resource.
- **PUT:** Updates a current resource with new data.
- **DELETE:** Deletes a specified resource.
- **PATCH:** Applies partial modifications to a resource.

HTTP Response Object

An HTTP response is the data a server sends back to a client after receiving and processing an HTTP request. Key components of an HTTP response are:

- **Status Line:** Includes the HTTP version, status code, and status message (e.g., HTTP/1.1 200 OK).
- **Headers:** Provide metadata about the response (e.g., Content-Type, Content-Length).
- **Body:** Contains the data requested or the result of the POST operation.

Response Types

- **text/html:** Indicates the response content is HTML.
- **application/json:** Indicates the response content is JSON (JavaScript Object Notation), a lightweight data-interchange format.

JSON (JavaScript Object Notation)

JSON is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is commonly used for transmitting data in web applications, particularly between a server and a client.

Structure of JSON

JSON is built on two structures:

1. **Objects:** An unordered collection of key/value pairs. Each key is a string, and the value can be a string, number, object, array, true, false, or null.
2. **Arrays:** An ordered list of values. Each value can be a string, number, object, array, true, false, or null.

JSON Syntax

- **Objects** are enclosed in curly braces {} and contain key/value pairs.
- **Arrays** are enclosed in square brackets [] and contain values.
- **Key/Value Pairs** are written as "key": value.

Example of a JSON Object

```
{
 "name": "John Doe",
 "age": 30,
 "isStudent": false,
 "address": {
  "street": "123 Main St",
  "city": "Anytown",
  "zipcode": "12345"
 },
 "courses": ["Math", "Science", "History"]
}
```

This JSON object represents a person with the following details:

- **name:** A string "John Doe"
- **age:** A number 30
- **isStudent:** A boolean false

- **address:** An object with nested key/value pairs
- **courses:** An array of strings

Example of a JSON Array

```
[
 {
  "name": "Alice",
  "age": 25,
  "email": "alice@example.com"
 },
 {
  "name": "Bob",
  "age": 28,
  "email": "bob@example.com"
 }
]
```

This JSON array contains two objects, each representing a person with name, age, and email fields.

Web APIs

Web APIs are the most common type of APIs used in web development. They allow communication between client applications (like web browsers or mobile apps) and server-side applications.

RESTful APIs

REST (Representational State Transfer) is a popular architectural style for designing networked applications. RESTful APIs use standard HTTP methods and are stateless, meaning each request from a client to the server must contain all the information needed to understand and process the request.

Key Characteristics of RESTful APIs

- **Stateless:** Each request from the client to the server must contain all the information needed to understand and process the request.
- **Client-Server Architecture:** Separation of client and server responsibilities.
- **Uniform Interface:** Standardized way of accessing resources.
- **Resource Identification:** Resources are identified using URIs (Uniform Resource Identifiers).
- **Representation:** Resources are represented using standard formats such as JSON or XML.
- **Self-Descriptive Messages:** Each message contains enough information to describe how to process the message.

HTTP Library in Dart

The http package in Dart allows you to make HTTP requests to interact with REST APIs. Here's a brief overview of the methods and their return types:

- **GET:** Retrieves data from a server. Returns Future<http.Response>.
- **POST:** Sends data to a server to create a resource. Returns Future<http.Response>.
- **PUT:** Updates an existing resource on the server. Returns Future<http.Response>.
- **DELETE:** Deletes a resource on the server. Returns Future<http.Response>.

Example of GET Method

```
import 'package:http/http.dart' as http;
import 'dart:convert';
void main() async {
  await fetchData();
}
Future<void> fetchData() async {
  final response = await
http.get(Uri.parse('https:  jsonplaceholder.typicode.com/posts'));

  if (response.statusCode  = 200) {
       If the server returns a 200 OK response, parse the JSON
    List<dynamic> data = json.decode(response.body);
    print(data);
  } else {
       If the server does not return a 200 OK response, throw an exception
    throw Exception('Failed to load data');
  }
}
```

Explanation

This Dart program demonstrates how to make a simple HTTP GET request to retrieve data from a REST API using the http package. The main function, marked as async, initiates the process by calling the fetchData function and waiting for it to complete. Inside fetchData, an HTTP GET request is made to https://jsonplaceholder.typicode.com/posts to fetch a list of posts. The response is awaited, and upon success (status code 200), the JSON data in the response body is decoded into a Dart list and printed.

If the response status code is not 200, the program throws an exception indicating that the data could not be loaded. This example illustrates basic error handling and JSON parsing, showcasing how to fetch and handle data from a web service in Dart.

Chapter-4
Creating Product Service API

Introduction to Flask

Flask is a lightweight and flexible web framework for Python that allows you to build web applications quickly with minimal code. It provides essential tools and libraries to handle routing, request handling, and templating, making it ideal for small to medium-sized projects and microservices.

Installing Flask in a Virtual Environment

1. **Create a Virtual Environment**: **python -m venv myenv**
2. **Activate the Virtual Environment**:
   For Windows: myenv\Scripts\activate
   For Linux:source myenv/bin/activate

3. **Install Flask: command is pip install Flask**

4. Creating a JSON File with Product List

1. **Create a JSON File (e.g., `products.json`):**

```
[
 {
  "id": 1,
  "name": "Product A",
  "price": 29.99
 },
 {
  "id": 2,
  "name": "Product B",
  "price": 39.99
 }
]
```

Python Code for CRUD Operations with Flask

1. **Create a Flask Application (`app.py`);**

```
from flask import *
import json
app = Flask(__name__)
# Load initial product data from JSON file
with open('products.json') as f:
    products = json.load(f)
@app.route('/products', methods=['GET'])
def get_products():
    return jsonify(products)
```

```
@app.route('/products/<int:id>', methods=['GET'])
def get_product(id):
    product = next((prod for prod in products if prod['id']   = id),
None)
    if product is None:
        return jsonify({'error': 'Product not found'}), 404
    return jsonify(product)
@app.route('/products', methods=['POST'])
def add_product():
    new_product = request.get_json()
    products.append(new_product)
    return jsonify(new_product), 201
@app.route('/products/<int:id>', methods=['PUT'])
def update_product(id):
    updated_product = request.get_json()
    for index, product in enumerate(products):
        if product['id']   = id:
            products[index] = updated_product
            return jsonify(updated_product)
    return jsonify({'error': 'Product not found'}), 404
@app.route('/products/<int:id>', methods=['DELETE'])
def delete_product(id):
    global products
    products = [prod for prod in products if prod['id']   ÷ id]
    return jsonify({'message': 'Product deleted'})
```

How to run this code:

In the Terminal Type the

flask run

Installing an API Client Extension in VS Code

1. **Open Visual Studio Code:**
   - **Launch VS Code on your machine.**
2. **Open the Extensions View:**
   - **Click on the Extensions icon in the Activity Bar on the side of the window (it looks like four squares).**
3. **Search for an API Client Extension:**
   - **In the Extensions view, type "API Client" in the search bar.**
4. **Install the REST Client Extension:**
   - **Find "REST Client" by Huachao Mao in the list of results.**
   - **Click the "Install" button next to the extension.**
5. **Use the REST Client Extension:**
   - **After installation, you can start using the REST Client to make API requests directly from VS Code.**

Basic Usage of the REST Client Extension

1. **Create a New File:**
    - **Open a new file in VS Code (e.g., `api.http`).**
2. **Write Your API Requests:**
    - **In the new file, write your HTTP requests. For example:**

Explanation

1. **GET /products:**

- **Returns the entire list of products as a JSON array.**

Output:

```
[
 {
  "id": 1,
  "name": "Product A",
  "price": 29.99
 },
 {
  "id": 2,
  "name": "Product B",
  "price": 39.99
 }
]
```

2. **GET /products/2:**

- **Retrieves a single product by its ID and returns it as JSON; returns a 404 error if the product is not found**

Output:

```
 {
   "id": 2,
   "name": "Product B",
   "price": 39.99
 }
```

3. **POST /products:**

Request:

```
{
  “Id”:3,
  “name”:”Product C”,
 “price”:68.47
}
```

Output:

```
 Response:
 Status:201
 {
  “Id”:3,
```

"name":"Product C",
"price":68.47
}

- **Adds a new product to the list and returns the created product with a 201 status code.**

4. PUT /products/2:

Request
{
Id:2,
name:"Product B",
price:85.23
}
Output
{
Id:2,
name:"Product B",
price:85.23
}

- **Updates an existing product by its ID and returns the updated product; returns a 404 error if the product is not found.**

5. DELETE /products/2:

Output:
{
"message":"Product deleted"}
}

- **Deletes a product by its ID and returns a confirmation message.**

Chapter-5

Accessing REST API

Accessing a REST API involves interacting with a web-based application interface to retrieve or manipulate data.

This interaction follows a standardized set of rules (RESTful architecture) to ensure efficient communication and data exchange. Clients (such as web applications, mobile apps, or other systems) send requests to the API server, specifying the desired action (e.g., retrieving data, creating new records, updating existing data, deleting data) and the format of the expected response. The API server processes the request, performs the requested action, and returns the appropriate data in a structured format (often JSON or XML). Essentially, it's a method for applications to communicate and share data over the internet.

FakeStoreAPI: A Playground for Testing and Learning
FakeStoreAPI, hosted at https://fakestoreapi.com/, is a free online service that provides a **RESTful API** specifically designed for e-commerce applications.

Here's what makes it special:

- **Fake Data:** Unlike real APIs, FakeStoreAPI provides simulated (fake) data related to products, categories, carts, and users. This eliminates the need for a real backend server and allows developers to test their applications without worrying about actual data manipulation.
- **Easy to Use:** The API is straightforward and readily accessible from any programming language.
- **Learning Tool:** FakeStoreAPI is fantastic for learning about REST APIs, e-commerce data structures, and practicing code for fetching and manipulating data.

Understanding the REST API with JSON Format
The API uses JSON (JavaScript Object Notation) to represent data. JSON is a human-readable format similar to key-value pairs, making it easy to understand and work with.
Here's a breakdown of the FakeStoreAPI's product API and an explanation of a sample JSON response:

API Endpoint:

- `GET https://fakestoreapi.com/products` (This retrieves a list of all products)

Sample JSON Response:

```
[
  {
    "id": 1,
```

```
    "title": "Fjallraven - Foldsack No. 2 Backpack",
    "price": 109.95,
      "description": "Your perfect everyday bag for errands, school, or travel. This spacious backpack is made from durable, water-resistant fabric.",
    "image": "https:  i.imgur.com/uGkQlpv.png",
    "category": "fashion"
  },
  {
    "id": 2,
    "title": "Leather Jacket",
    "price": 299.99,
      "description": "This rugged leather jacket is perfect for colder weather. It's made from high-quality leather and features a classic design.",
    "image": "https:  i.imgur.com/zYcA1ln.png",
    "category": "men's clothing"
  },
  ]
```


Explanation of the JSON Data:

- Each product is represented by an object within square brackets ([]).
- Each object has key-value pairs separated by colons (:).
    - id: Unique identifier for the product (number)
    - title: Product name (string)
    - price: Product price (number)
    - description: Description of the product (string)
    - image: URL of the product image (string)
    - category: Category the product belongs to (string)

Additional Points:

- The API offers various endpoints for different functionalities. You can retrieve a specific product by its ID, get products within a specific category, and even perform actions like adding or updating products (these functionalities are for demonstration purposes only; modifying data on FakeStoreAPI isn't permanent).
- Refer to the official documentation at https://fakestoreapi.com/ for a complete list of endpoints and examples.

By using FakeStoreAPI, you can practice fetching and manipulating data in JSON format, simulate interactions with a real e-commerce API, and gain valuable experience with RESTful APIs.

Using the http package in Dart to interact with FakeStoreAPI

The http package in Dart offers a clean and efficient way to make HTTP requests. Let's explore its methods, use it with FakeStoreAPI, and understand the code step-by-step.

1. Adding the dependency:

First, you need to add the http package to your project. You can achieve this by modifying your pubspec.yaml file and adding the following line under the dependencies section:

```
dependencies:
  http: ^0.13.4
```

Then, run flutter pub get to download the package.

2. Importing the library and making a GET request:

Here's the code for making a GET request to retrieve all products from FakeStoreAPI:

```
import 'package:http/http.dart' as http;
Future<void> main() async {
     Define the API endpoint URL
  final url = Uri.parse('https:  fakestoreapi.com/products');
     Make the GET request
  final response = await http.get(url);
     Check for successful response
  if (response.statusCode  = 200) {
       Parse the JSON response
    final data = jsonDecode(response.body);
       Print the product information
    for (var product in data) {
         print('Product ID: ${product['id']}, Title: ${product['title']},
Price: ${product['price']}');
    }
  } else {
    print('Failed to fetch products. Status code: ${response.statusCode}');
  }
}
```

Explanation:

1. We import the http package with an alias (as http).
2. Inside the main function, we define the API endpoint URL as a Uri object.
3. We use http.get(url) to make the GET request asynchronously and store the response in a variable.
4. We check if the response status code is 200 (indicating success).
5. If successful, we use jsonDecode(response.body) to convert the response body (containing JSON data) into a Dart object.
6. We iterate through the parsed data, which is a list of product objects.
7. For each product, we access its properties (ID, title, price) using their keys (['id'], ['title'], ['price']) and print them.
8. If the request fails, we print an error message along with the status code.

3. Making a POST request:

While FakeStoreAPI doesn't support creating real products, let's see a simplified example of a POST request:

```
import 'package:http/http.dart' as http;
Future<void> main() async {
```

```
  final url = Uri.parse('https:  your-api-endpoint.com/products');
     Sample product data for the POST request
  final productData = {
    'title': 'My New Product',
    'price': 99.99,
    'description': 'This is a new and exciting product!',
    'image': 'https:  example.com/product-image.jpg',
    'category': 'electronics'
  };
     Make the POST request with JSON data
  final response = await http.post(
    url,
    headers: {'Content-Type': 'application/json'},
    body: jsonEncode(productData),
  );
     Check for successful response
  if (response.statusCode  = 201) {
    print('Product created successfully!');
  } else {
    print('Failed to create product. Status code: ${response.statusCode}');
  }
}
```

Explanation:

1. Similar to the GET request, we define the API endpoint URL.
2. We define a map containing sample product data to be sent in the request body.
3. We use `http.post` with additional arguments:
    - `headers`: This sets the content type to JSON.
    - `body`: This contains the JSON-encoded product data using `jsonEncode(productData)`.
4. We check if the response status code is 201 (indicating successful creation).
5. We print appropriate messages based on the response status code.

Chapter-6
Accessing Product Service API with Flutter

FutureBuilder: A Reliable Partner for Asynchronous Data Management in Flutter

In Flutter app development, when dealing with operations that take an unknown amount of time to complete, like fetching data from remote servers or performing lengthy calculations, asynchronous programming becomes essential. The FutureBuilder widget acts as a cornerstone in managing these asynchronous operations, offering a structured approach to displaying appropriate UI elements as the operation progresses, succeeds, or encounters errors.

Understanding Snapshots: Timely Updates on Operation Status

FutureBuilder leverages the concept of snapshots, which provide real-time updates on the asynchronous operation's state. Imagine snapshots as progress reports, similar to how a delivery service might send you updates like "Order confirmed" or "Out for delivery." In FutureBuilder, snapshots offer three key pieces of information:

- ConnectionState**:** This property reflects the current stage of the operation, indicating whether it's:
  - waiting: The operation hasn't started or is still in progress.
  - done: The operation has successfully completed.
  - active (for some futures): The operation is ongoing and might provide updates occasionally.
  - error: The operation encountered an unexpected issue.
- data**:** This property stores the retrieved data upon successful completion (the product list in our example).
- hasError**:** This signals if the operation ran into an error, allowing for appropriate error handling.

Properties of FutureBuilder: Tailoring UI Dynamics

- future **(required):** This is the core of FutureBuilder. It represents the asynchronous operation itself. In our example, it will be a function responsible for making the HTTP GET request using the http package.
- builder **(required):** This is your UI architect. It's a function that takes a BuildContext and an AsyncSnapshot as arguments and constructs the UI based on the snapshot's information. You can use conditional statements to display different UI elements based on the ConnectionState:
  - A loading indicator while waiting (ConnectionState.waiting).
  - The retrieved data (product list) upon successful completion (ConnectionState.done).
  - An error message if an error occurred (ConnectionState.error).

- initialData **(optional):** This property allows you to specify an initial value to display while the asynchronous operation is in progress. This is particularly useful for preventing your UI from appearing blank during the waiting phase.

HTTP GET Request Example with FutureBuilder: A Comprehensive Illustration

1. Dependencies:

```
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;     for making HTTP requests
```

2. Data Model (Product) (Optional):

(Create a file Product.dart)

```
class Product {
  final int id;
  final String title;
  final String price;
  Product({required this.id, required this.title, required this.price});
  factory Product.fromJson(Map<String, dynamic> json) {
    return Product(
      id: json['id'],
      title: json['title'],
      price: json['price'],
    );
  }
}
```

3. Fetching Products with Error Handling:

(Create a function to encapsulate the HTTP GET request, handle the response, and gracefully manage errors)

```
Future<List<Product » fetchProducts() async {
  final response = await
http.get(Uri.parse('https:  fakestoreapi.com/products'));
  if (response.statusCode  = 200) {
   final List<dynamic>
jsonProductList = jsonDecode(response.body);
    return jsonProductList.map((productJson) =€
Product.fromJson(productJson)).toList();
  } else {
    throw Exception('Failed to load products: Status code
${response.statusCode}');
  }
}
```

4. Create an API service class: Create a new file `api_service.dart` to define the `ApiService` class that will fetch the products from the API.

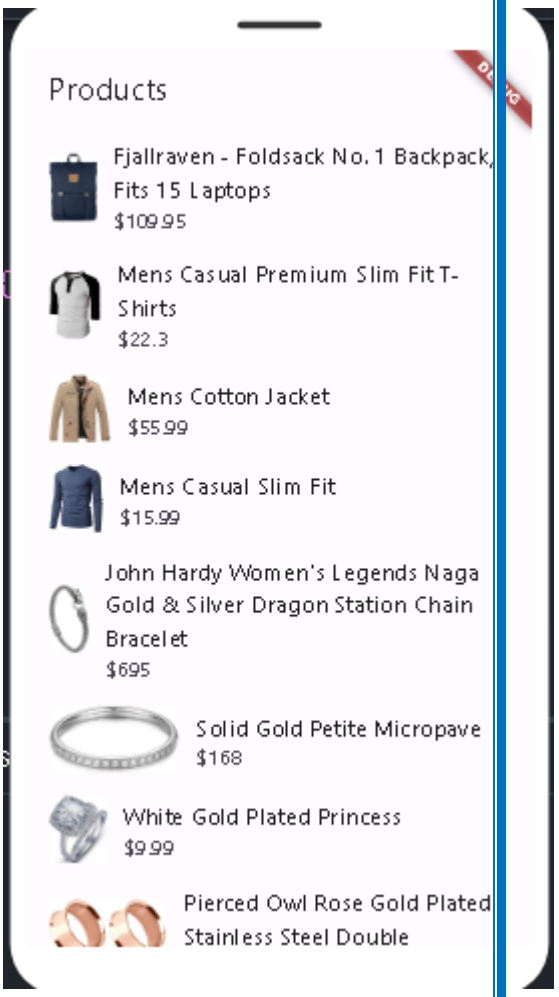
```
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'product.dart';
class ApiService {
  static const String url = 'https:  fakestoreapi/products';
  Future<List<Product » fetchProducts() async {
    final response = await http.get(Uri.parse(url));
```

```
    if (response.statusCode  = 200) {
      List jsonResponse = json.decode(response.body);
      return jsonResponse.map((product) =€
Product.fromJson(product)).toList();
    } else {
      throw Exception('Failed to load products');
    }
  }
}
```

5. Building the UI with FutureBuilder:

Here's how you can construct the UI using FutureBuilder, incorporating professional UI design conventions and error handling:

```
import 'package:flutter/material.dart';
import 'api_service.dart';
import 'product.dart';
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Product App',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: ProductListScreen(),
    );
  }
}
class ProductListScreen extends StatelessWidget {
  final ApiService apiService = ApiService();
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Products'),
      ),
      body: FutureBuilder<List<Product »(
        future: apiService.fetchProducts(),
        builder: (context, snapshot) {
          if (snapshot.connectionState  = ConnectionState.waiting) {
            return Center(child: CircularProgressIndicator());
          } else if (snapshot.hasError) {
            return Center(child: Text('Error: ${snapshot.error}'));
          } else if (snapshot.hasData) {
            return ListView.builder(
              itemCount: snapshot.data!.length,
              itemBuilder: (context, index) {
                final product = snapshot.data![index];
                return ListTile(
                  leading: Image.network(product.image),
```



Output

```
                    title: Text(product.title),
                    subtitle: Text('\$${product.price.toString()}'),
                  );
                },
              );
            } else {
              return Center(child: Text('No products found'));
            }
          },
        ),
      );
    }
}
```

Explanation

1. **Model Class (product.dart):**
   - Product class represents the product data structure.
   - Product.fromJson is a factory constructor that creates a Product instance from a JSON object.
2. **API Service (api_service.dart):**
   - ApiService class contains the method fetchProducts which performs the HTTP GET request.
   - fetchProducts method returns a list of Product objects by decoding the JSON response.
3. **Main Application (main.dart):**
   - MyApp is the root widget of the application.
   - ProductListScreen is the main screen which displays the list of products.
   - FutureBuilder widget is used to handle the asynchronous fetching of data.
     - future property is assigned to the fetchProducts method.
     - builder property defines the UI based on the state of the Future.
     - When the Future is waiting, a loading indicator is shown.
     - If there is an error, the error message is displayed.
     - If data is successfully fetched, a ListView is built to display the products.

HTTP POST Request to fakestoreapi.com

Introduction

The HTTP POST method is used to send data to the server to create or update a resource. For this example, we'll be using `fakestoreapi.com`, a free online REST API for testing and prototyping.

Endpoint

To create a new product, the endpoint is:

https://fakestoreapi.com/products

Request Body

The data sent in the request body must be in JSON format. Here is an example of the data we will send:

```
JSON
{
  "title": "Product Title",
  "price": 29.99,
  "description": "Product Description",
  "image": "https:  example.com/product-image.jpg",
  "category": "Category"
}
```

Using Dart's HTTP Package

We will use the `http` package to make HTTP requests in Flutter. Add `http` to your `pubspec.yaml` file:

dependencies:
http: ^0.13.4

Flutter Example: Creating a Product

Step 1: Setting Up the Project

Create a new Flutter project and add the `http` package as mentioned above.

Step 2: Creating the Form

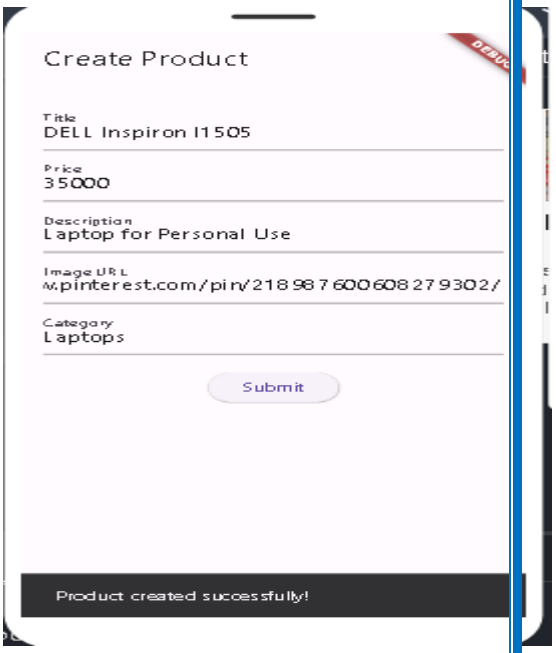
Create a form with text fields for title, price, description, image URL, and category, and a button to submit the form using TextField and TextEditingController.

```
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:convert';
void main() =€ runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: CreateProductPage(),
    );
  }
}
class CreateProductPage extends StatefulWidget {
  @override
```

```
  _CreateProductPageState createState() =€ _CreateProductPageState();
}
class _CreateProductPageState extends State<CreateProductPage> {
     Controllers to capture input from text fields
  final TextEditingController _titleController = TextEditingController();
  final TextEditingController _priceController = TextEditingController();
  final TextEditingController _descriptionController =
TextEditingController();
  final TextEditingController _imageUrlController =
TextEditingController();
  final TextEditingController _categoryController =
TextEditingController();
     Function to submit product data
  Future<void> _submitProduct() async {
       API URL
    final url = Uri.parse('https:  fakestoreapi.com/products');
       Making the POST request
    final response = await http.post(
      url,
      headers: {'Content-Type': 'application/json'},
      body: json.encode({
        'title': _titleController.text,
        'price': double.parse(_priceController.text),
        'description': _descriptionController.text,
        'image': _imageUrlController.text,
        'category': _categoryController.text,
      }),
    );
       Displaying snackbar based on response
    if (response.statusCode  = 200) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Product created successfully!')),
      );
    } else {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Failed to create product.')),
      );
    }
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Create Product')),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: [
            TextField(
              controller: _titleController,
              decoration: InputDecoration(labelText: 'Title'),
            ),
            TextField(
              controller: _priceController,
```

```
              decoration: InputDecoration(labelText: 'Price'),
              keyboardType: TextInputType.number,
            ),
            TextField(
              controller: _descriptionController,
              decoration: InputDecoration(labelText: 'Description'),
            ),
            TextField(
              controller: _imageUrlController,
              decoration: InputDecoration(labelText: 'Image URL'),
            ),
            TextField(
              controller: _categoryController,
              decoration: InputDecoration(labelText: 'Category'),
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: _submitProduct,
              child: Text('Submit'),
            ),
          ],
        ),
      ),
    );
  }
}
```

Output:



Explanation

1. **Importing Packages**:
   - flutter/material.dart: Provides Flutter's material design components.
   - http/http.dart as http: Allows making HTTP requests.
   - dart:convert: Provides utilities to convert Dart objects to JSON and vice versa.
2. **Main Function**:
   - void main() => runApp(MyApp());: The entry point of the application which runs MyApp.
3. **MyApp StatelessWidget**:
   - Returns a MaterialApp with CreateProductPage as the home widget.
4. **CreateProductPage StatefulWidget**:
   - **State Management**: Uses _CreateProductPageState to manage the state of the CreateProductPage.
5. **Controllers**:
   - TextEditingController: Captures user input from each TextField.
6. **_submitProduct Function**:
   - Constructs the URL for the API endpoint.

- Makes an HTTP POST request to the API with headers specifying JSON content.
- Converts the input data to JSON format using json.encode.
- Sends the data in the request body.
- Displays a Snackbar with a success or failure message based on the HTTP response status code.

7. **Build Method**:
   - Returns a Scaffold with an AppBar and a Padding widget containing a Column of TextField and ElevatedButton widgets.
   - Each TextField is linked to a corresponding TextEditingController.
   - The ElevatedButton triggers the _submitProduct function when pressed.

Chapter-7

Database Concepts: SQLite and SQLiteDbProvider

What We Will Do

1. **Introduction to SQLite:** Learn what SQLite is and its key features.
2. **Basic SQLite Operations:** Understand how to perform CRUD operations.
3. **Using SQLite in Flutter:** Learn how to use SQLite in a Flutter app with the sqflite package.
4. **SQLiteDbProvider Object:** Create a custom class to manage database operations.
5. **SQLiteDbProvider Methods:** Learn about methods for opening the database, inserting, querying, updating, and deleting data.

1. Introduction to SQLite

SQLite is a small, fast, self-contained database engine widely used across various platforms. It doesn't need a separate server to operate and is easy to set up.

Key Features:

- **Serverless:** Works without a separate server.
- **Zero Configuration:** No setup needed.
- **Transactional:** Ensures data consistency.
- **Cross-Platform:** Runs on different operating systems.
- **Lightweight:** Small in size and easy to deploy.

2. Basic SQLite Operations

SQLite supports the following basic operations:

- **Create:** Add new data.
- **Read:** Retrieve data.
- **Update:** Modify existing data.
- **Delete:** Remove data.

3. Using SQLite in Flutter

To use SQLite in a Flutter app, you can use the sqflite package. This package makes it easy to perform database operations.

Add sqflite to your project: Open the pubspec.yaml file and add the sqflite and path dependencies.
dependencies:

sqflite: ^2.0.0+4
path: ^1.8.0

4. SQLiteDbProvider Object

The SQLiteDbProvider is a custom class that manages database operations like opening the database, creating tables, and performing CRUD operations. It uses the singleton pattern to ensure only one instance of the database is used.

Basic Structure of SQLiteDbProvider:

1. **Singleton Pattern:**

```
class SQLiteDbProvider {
  SQLiteDbProvider._();    Private constructor
  static final SQLiteDbProvider db = SQLiteDbProvider._();    Single
instance
}
```

   - **This code ensures only one instance of `SQLiteDbProvider` is created.**

2. **Database Initialization:**

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';
import 'package:path_provider/path_provider.dart';
import 'dart:io';
class SQLiteDbProvider {
  SQLiteDbProvider._();
  static final SQLiteDbProvider db = SQLiteDbProvider._();
  Database _database;
  Future<Database> get database async {
    if (_database  ÷ null) return _database;    Return existing
database if it exists
    _database = await _initDB();    Initialize the database if it
doesn't exist
    return _database;
  }
  Future<Database> _initDB() async {
    Directory documentsDirectory = await
getApplicationDocumentsDirectory();
    String path = join(documentsDirectory.path, "TestDB.db");
Database file path
    return await openDatabase(path, version: 1, onOpen: (db) {},
        onCreate: (Database db, int version) async {
      await db.execute("CREATE TABLE Test ("
          "id INTEGER PRIMARY KEY,"
          "name TEXT,"
          "value INTEGER,"
          "num REAL"
          ")");
    });
```

```
  }
}
```

- This code initializes the database and creates a table named "Test."
- The `get database` method ensures only one instance of the database is created.
- The `_initDB` method sets up the database file path and creates a table if it doesn't already exist.

5. SQLiteDbProvider Methods

1. Open Database:

```
Future<Database> openDB() async {
  Directory documentsDirectory = await
getApplicationDocumentsDirectory();
  String path = join(documentsDirectory.path, "example.db");
Define the path to the database
  return await openDatabase(path, version: 1, onOpen: (db) {});
Open the database
}
```

- Purpose: Opens the database located at the specified path.

Explanation:

- `getApplicationDocumentsDirectory()` gets the directory where the app can store its data.
- `join(documentsDirectory.path, "example.db")` creates the full path to the database file.
- `openDatabase(path, version: 1, onOpen: (db) {})` opens the database file, creating it if it doesn't exist.

2. Insert:

```
Future<int> insert(String table, Map<String, dynamic> values) async {
  final db = await database;    Get the database instance
  return await db.insert(table, values);    Insert values into the
specified table }
```

- Purpose: Inserts new data into the specified table.

Explanation:

- `database` gets the database instance.
- `db.insert(table, values)` inserts the `values` into the specified `table`.
- This method returns the ID of the inserted row.

3. Query:

```
Future<List<Map<String, dynamic  € queryAllRows(String table) async {
  final db = await database;     Get the database instance
  return await db.query(table);     Retrieve all rows from the
specified table
}
```

- **Purpose: Retrieves all rows from the specified table.**

Explanation:

- `database` **gets the database instance.**
- `db.query(table)` **retrieves all rows from the** `table` **and returns them as a list of maps (key-value pairs).**

4. Update:

```
Future<int> update(String table, Map<String, dynamic> values, String
whereClause, List<dynamic> whereArgs) async {
  final db = await database;     Get the database instance
  return await db.update(table, values, where: whereClause,
whereArgs: whereArgs);     Update rows in the specified table
}
```

- **Purpose: Updates existing data in the specified table.**

Explanation:

- `database` **gets the database instance.**
- `db.update(table, values, where: whereClause, whereArgs: whereArgs)` **updates the rows in the** `table` **that match the** `whereClause` **and** `whereArgs`**.**
- **This method returns the number of rows affected.**

5. Delete:

```
Future<int> delete(String table, String whereClause, List<dynamic>
whereArgs) async {
  final db = await database;     Get the database instance
  return await db.delete(table, where: whereClause, whereArgs:
whereArgs);     Delete rows from the specified table
}
```

- **Purpose: Deletes data from the specified table.**

Explanation:

- `database` **gets the database instance.**
- `db.delete(table, where: whereClause, whereArgs: whereArgs)` **deletes the rows in the** `table` **that match the** `whereClause` **and** `whereArgs`**.**

- This method returns the number of rows deleted.

6. Close Database:

```
Future<void> closeDB() async {
  final db = await database;    Get the database instance
  db.close();    Close the database connection
}
```

- Purpose: Closes the database connection.

Explanation:

- `database` gets the database instance.
- `db.close()` closes the database connection.

Example Usage:

```
void main() async {
  SQLiteDbProvider databaseProvider = SQLiteDbProvider.db;
     Insert a row
  await databaseProvider.insert('Test', {'name': 'Item 1', 'value': 123,
'num': 45.67});
     Query all rows
  List<Map<String, dynamic » rows = await
databaseProvider.queryAllRows('Test');
  print(rows);    Print all rows
     Update a row
  await databaseProvider.update('Test', {'value': 456}, 'id = ?', [1]);
     Delete a row
  await databaseProvider.delete('Test', 'id = ?', [1]);
}
```

Unit-5

Question Bank

2 Marks Questions

1. What is a Dart package?
2. Mention one benefit of using Dart packages.
3. Where can you find pub packages for Dart?
4. Which file in a Dart project specifies the project dependencies?
5. What command is used to install packages specified in pubspec.yaml?
6. What does a Future represent in Dart?
7. What does the then() method do with a Future?
8. How do you declare an asynchronous function in Dart?
9. How do you handle errors in asynchronous code using Dart?
10. What is the primary function of a web server?
11. Define the term "host" in the context of web services.
12. What does the GET method do in HTTP request methods?
13. What does JSON stand for and what is its primary use?
14. How are JSON objects represented syntactically?
15. What is Flask used for in Python development?
16. How do you install Flask in a virtual environment on Windows?
17. What is the primary purpose of accessing a REST API?
18. Describe the type of data FakeStoreAPI provides and its use case.
19. What is the purpose of the FutureBuilder widget in Flutter?
20. What HTTP method is used to send data to the server to create or update a resource?
21. How do you handle errors in FutureBuilder when fetching data?
22. What is the role of the initialData property in FutureBuilder?
23. What are the key features of SQLite?
24. What is the purpose of the database getter in the SQLiteDbProvider class?
25. How does the insert method in SQLiteDbProvider work?
26. What does the queryAllRows method in SQLiteDbProvider do?
27. How do you open a database using SQLiteDbProvider?

5 Marks Questions

1. Explain the importance and benefits of using Dart packages.
2. Describe the steps to add a dependency from pub.dev to a Dart project.
3. What is the structure of a Dart package? Explain each component briefly.
4. Outline the process to create and publish a simple Dart package.
5. How can you add and use a custom package in a Flutter project?
6. Explain the purpose of the `async` and `await` keywords in Dart.
7. Describe the process of handling errors in asynchronous Dart code.

8. How do you use the `then()` method with a `Future`? Provide an example.
9. Outline the steps to create and use a `Future` in Dart. Include code examples.
10. Demonstrate how to simulate a network delay using `Future.delayed` and handle its result.
11. Describe the key characteristics of RESTful APIs.
12. Explain the structure of JSON and provide an example of a JSON object.
13. Outline the role and functionality of the `http` package in Dart for making HTTP requests.
14. How does the `fetchData` function work in the provided Dart example? Include details about handling HTTP responses and errors.
15. Discuss the HTTP response components and their significance in web communication.
16. Describe the steps to set up and run a Flask application, including virtual environment setup and installation commands.
17. Outline the structure and contents of the `products.json` file used in the Flask application. How does it interact with the CRUD operations?
18. Detail the structure of a sample JSON response from the FakeStoreAPI. What information does each key in the JSON object represent?
19. How does the POST request example handle the creation of a new product? Describe the process of sending data and checking for a successful response.
20. Explain how FutureBuilder manages asynchronous operations in Flutter.
21. Describe the process of creating a Product class in Flutter and how it is used to handle JSON data from an API response.
22. Outline the steps to fetch product data from FakeStoreAPI using Flutter's FutureBuilder. Describe how to handle different states such as loading, error, and data available.
23. Explain the Singleton pattern used in the SQLiteDbProvider class. How does it ensure only one instance of the database is used?
24. Describe the process of initializing the database and creating a table using SQLiteDbProvider. Include details about the `_initDB` method.
25. Discuss the basic CRUD operations supported by SQLite and provide an example of how each operation is performed using SQLiteDbProvider methods.
26. Outline the steps to perform a database update using SQLiteDbProvider. Include details on how to specify the rows to be updated and the values to be changed.