ChatGPT

1. What are shared preferences? Create a flutter app using pref.getBool('darkTheme') and explain your solution.
2. When to use shared preferences? Write Dart code and justify your answer with appropriate example.
3. Write Dart code for adding required dependency in pubspec.yaml if you want to use shared preferences. Develop flutter app which stores username in shared preferences. Discuss your solution.
4. Mr. Mahesh is learning flutter. His teacher assigned him a task to store app usage counter in shared preferences. He needs to increment counter value whenever app is launched. Write Dart code and help Mr. Mahesh in developing this demo flutter app. Add necessary comments in your code so that Mr. Mahesh can understand Dart code easily.
5. Analyze and explain the following code snippet:

class MainAppState extends State<MainApp> { SharedPreferences? prefs;
TextEditingController usernameController = TextEditingController(); String _username = "Guest";
@override
void initState() {
_loadData();
super.initState();
}
void _loadData() async {
}
try {
prefs= await SharedPreferences.getInstance();
} catch (e) {
}
print(e);
setState(() {
});
_username prefs?.getString("username") ?? "Guest";
Define serialization. Analyze and explain following code snippet:

ANSWER DESCRIPTIVELY WITH FULL CODE

1. What are Shared Preferences? Create a Flutter app using `pref.getBool('darkTheme')` and explain your solution.

SharedPreferences is a simple key-value storage mechanism for storing small amounts of data persistently. It is primarily used for saving simple types of data such as booleans, integers, strings, and lists of strings. Shared Preferences is often used for storing user settings (like dark theme preferences, authentication states, etc.) locally on the device.

To demonstrate how you can use `SharedPreferences` to store and retrieve a boolean value (e.g., whether the user prefers dark mode or not), let's build a simple Flutter app.

Code Implementation

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

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData.light(),
      darkTheme: ThemeData.dark(),
      themeMode: ThemeMode.system, // Default theme mode, we will override with dark mode pr
eference.
      home: DarkModeToggle(),
    );
  }
}

class DarkModeToggle extends StatefulWidget {
  @override
  _DarkModeToggleState createState() => _DarkModeToggleState();
}

class _DarkModeToggleState extends State<DarkModeToggle> {
  bool _isDarkMode = false;

  @override
  void initState() {
    super.initState();
    _loadThemePreference();
  }

  // Load the theme preference from SharedPreferences
  void _loadThemePreference() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    setState(() {
      _isDarkMode = prefs.getBool('darkTheme') ?? false; // Defaults to false if not set.
    });
  }

  // Save the theme preference to SharedPreferences
  void _saveThemePreference(bool value) async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    prefs.setBool('darkTheme', value);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Dark Mode Preference")),
      body: Center(
        child: SwitchListTile(
          title: Text("Enable Dark Mode"),
          value: _isDarkMode,
          onChanged: (bool value) {
            setState(() {
              _isDarkMode = value;
              _saveThemePreference(value); // Save the new preference.
            });
          },
        ),
      ),
    );
  }
}
```

Explanation:

- **SharedPreferences** is used to store the boolean value `darkTheme`, which indicates if the user wants dark mode enabled or not.
- The app uses a `SwitchListTile` widget to toggle between light and dark themes.
- In the `_loadThemePreference()` method, the value of the `darkTheme` preference is loaded when the app starts (via `SharedPreferences.getInstance()`), and the UI is updated to reflect the user's choice.
- The `_saveThemePreference()` method stores the new value of dark mode to `SharedPreferences` when the user toggles the switch.

---

2. When to use SharedPreferences? Write Dart code and justify your answer with appropriate example.

When to use SharedPreferences: SharedPreferences is suitable for storing small amounts of simple data that need to persist across app restarts. This could include:

- User preferences (e.g., theme, language, etc.)
- App settings (e.g., notification preferences)
- Simple authentication states (e.g., logged-in status)
- Data that doesn't require complex querying or relations (e.g., storing user data like a name or email)

Example:

Consider an app that stores a simple boolean indicating whether the user is logged in or not. This information can be stored in SharedPreferences for quick retrieval and persistence.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: LoginPage(),
    );
  }
}

class LoginPage extends StatefulWidget {
  @override
  _LoginPageState createState() => _LoginPageState();
}

class _LoginPageState extends State<LoginPage> {
  bool _isLoggedIn = false;
```

```
    @override
    void initState() {
      super.initState();
      _loadLoginStatus();
    }

    // Load the login status from SharedPreferences
    void _loadLoginStatus() async {
      SharedPreferences prefs = await SharedPreferences.getInstance();
      setState(() {
        _isLoggedIn = prefs.getBool('isLoggedIn') ?? false;
      });
    }

    // Save the login status to SharedPreferences
    void _saveLoginStatus(bool status) async {
      SharedPreferences prefs = await SharedPreferences.getInstance();
      prefs.setBool('isLoggedIn', status);
    }

    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(title: Text('Login Status')),
        body: Center(
          child: ElevatedButton(
            onPressed: () {
              setState(() {
                _isLoggedIn = !_isLoggedIn;
                _saveLoginStatus(_isLoggedIn); // Save the status to SharedPreferences
              });
            },
            child: Text(_isLoggedIn ? 'Log Out' : 'Log In'),
          ),
        ),
      );
    }
  }
```

Justification:

In this example, SharedPreferences is used to persist the user's login status. Every time the app starts, the `_loadLoginStatus()` method checks if the user is logged in and updates the UI accordingly. If the user logs in or logs out, the status is saved in SharedPreferences. This is an ideal use case for SharedPreferences because login status is simple and doesn't require complex data management or security features (though for more sensitive data, you'd use secure storage).

---

3. Write Dart code for adding the required dependency in `pubspec.yaml` if you want to use SharedPreferences. Develop a Flutter app that stores the username in SharedPreferences. Discuss your solution.

1. Add Dependency in `pubspec.yaml`:

```yaml
dependencies:
  flutter:
    sdk: flutter
  shared_preferences: ^2.0.15 # Add the latest version of shared_preferences
```

2. Store Username in SharedPreferences:

```
dart



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

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

  class MyApp extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
      return MaterialApp(
        home: UsernamePage(),
      );
    }
  }

  class UsernamePage extends StatefulWidget {
    @override
    _UsernamePageState createState() => _UsernamePageState();
  }

  class _UsernamePageState extends State<UsernamePage> {
    final TextEditingController _usernameController = TextEditingController();
    String _savedUsername = 'Guest';

    @override
    void initState() {
      super.initState();
      _loadUsername();
    }

    // Load the username from SharedPreferences
    void _loadUsername() async {
      SharedPreferences prefs = await SharedPreferences.getInstance();
      setState(() {
        _savedUsername = prefs.getString('username') ?? 'Guest'; // Default to "Guest" if no v
  alue
      });
    }

    // Save the username to SharedPreferences
    void _saveUsername(String username) async {
      SharedPreferences prefs = await SharedPreferences.getInstance();
      prefs.setString('username', username);
    }

    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(title: Text('Username Storage')),
        body: Padding(
          padding: const EdgeInsets.all(16.0),
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: <Widget>[
              Text('Enter Username:'),
              TextField(
                controller: _usernameController,
                decoration: InputDecoration(hintText: 'Enter your username'),
              ),
              ElevatedButton(
                onPressed: () {
                  String username = _usernameController.text;
                  _saveUsername(username); // Save the entered username
                  setState(() {
                    _savedUsername = username; // Update UI
                  });
                },
                child: Text('Save Username'),
              ),
            SizedBox(height: 20),
            Text('Saved Username: $_savedUsername'),
          ],
        ),
      ),
    );
  }
}
```

Discussion:

- **Dependency**: The `shared_preferences` package is added in `pubspec.yaml` to allow us to use the SharedPreferences API.
- **UI Components**: The app has a `TextField` where the user can input their username and a button to save it.
- **SharedPreferences Operations**:
  - The `_loadUsername` method loads the stored username from SharedPreferences when the app starts.
  - The `_saveUsername` method stores the username in SharedPreferences whenever the user presses the button.

---

4. Mr. Mahesh's task: Store app usage counter in SharedPreferences.

Here's how you can help Mr. Mahesh increment the usage counter:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: UsageCounterPage(),
    );
  }
}

class UsageCounterPage extends StatefulWidget {
  @override
  _UsageCounterPageState createState() => _UsageCounterPageState();
}

class _UsageCounterPageState extends State<UsageCounterPage> {
  int _counter = 0;

  @override
  void initState() {
    super.initState();
    _loadCounter();
  }

  // Load the usage counter from SharedPreferences
  void _loadCounter() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    setState(() {
      _counter = prefs.getInt('usage_counter') ?? 0; // Defaults to 0 if not found
    });
  }
```

```
    // Increment and save the counter in SharedPreferences
    void _incrementCounter() async {
      SharedPreferences prefs = await SharedPreferences.getInstance();
      setState(() {
        _counter++;
        prefs.setInt('usage_counter', _counter); // Save updated counter
      });
    }

    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(title: Text('App Usage Counter')),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Text('App launched $_counter times'),
              ElevatedButton(
                onPressed: _incrementCounter,
                child: Text('Increment Counter'),
              ),
            ],
          ),
        ),
      );
    }
  }
```

Explanation:

- **Counter Management**: This app tracks how many times it has been launched or how many times the user presses the "Increment Counter" button.
- **SharedPreferences**: The counter is saved using `` `prefs.setInt('usage_counter', _counter)` `` and loaded using `` `prefs.getInt('usage_counter')` ``.

---

5. Analyze and explain the following code snippet:

```dart
  class MainAppState extends State<MainApp> {
    SharedPreferences? prefs;
    TextEditingController usernameController = TextEditingController();
    String _username = "Guest";

    @override
    void initState() {
      _loadData();
      super.initState();
    }

    void _loadData() async {
      try {
        prefs = await SharedPreferences.getInstance();
      } catch (e) {
        print(e);
      }
      setState(() {
        _username = prefs?.getString("username") ?? "Guest";
      });
    }
  }
```

Analysis:

- **SharedPreferences Usage**: The `SharedPreferences` instance (`prefs`) is used to retrieve the stored username.
- **Error Handling**: The `try-catch` block handles potential errors when initializing SharedPreferences.
- **Default Value**: If no username is found, "Guest" is used as the default value.

6. Define serialization. Analyze and explain following code snippet:

import 'package: json_annotation/json_annotation.dart';
part 'address.g.dart';
@JsonSerializable()
class Address {
}
String street, landmark, city, state;
Address(
{required this.street,
required this. landmark,
required this.city,
required this.state});
factory Address.fromJson (Map<String, dynamic> data) { return _$Address FromJson(data);
}
Map<String, dynamic> toJson() => _$AddressToJson(this);

7.
What is JSON serialization? Write and explain Dart code demonstrating
use of “json_annotation” and “json_serializable”.
8.
Analyze the following code snippet and discuss the errors underlined.
Write and explain the command to resolve these errors.
import 'package:json_annotation/json_annotation.dart';

part user.g.dart';
Target of URI hasn't been generated: 'user.g.dart'.
class User {
User(this.name, this.email);
String name; String email;

factory User.fromJson (Map<String, dynamic> json) => $UserFromJson(json);
Map<String, dynamic> toJson() => $UserToJson(this);
9.
How to add json_serializable dependency in a flutter project? Explain the
use of following:
• dart run build_runner build
• dart run build_runner build –delete-conflicting-outputs
• dart run build_runner watch
• dart run build_runner watch –delete-conflicting-outputs

10. Write Dart code and discuss use of automated JSON serialization using code generation.

6. Define Serialization. Analyze and explain the following code snippet:

Serialization refers to the process of converting an object into a format (usually a JSON string) that can be easily transmitted or stored. It is often used when sending data over a network or saving it locally (e.g., to a file or database). The reverse process, where a serialized object is converted back to its original form, is called **deserialization**.

Analyzing the Code Snippet:

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

part 'address.g.dart';

@JsonSerializable()
class Address {
  String street, landmark, city, state;

  Address({
    required this.street,
    required this.landmark,
    required this.city,
    required this.state,
  });

  factory Address.fromJson(Map<String, dynamic> data) => _$AddressFromJson(data);
  Map<String, dynamic> toJson() => _$AddressToJson(this);
}
```

Explanation:

- `@JsonSerializable()`: This annotation marks the class to be serializable using the `json_serializable` package. It tells the code generator to create serialization code for this class.
- **Class Properties**: The `Address` class has four string properties (`street`, `landmark`, `city`, and `state`), which will be serialized and deserialized.
- **Constructor**: The constructor uses named parameters with the `required` keyword to ensure all fields are provided.
- `fromJson` **Method**: This factory constructor is used to create an instance of `Address` from a JSON map. It uses the generated `_$AddressFromJson` function.
- `toJson` **Method**: This method converts an `Address` object into a JSON map using the generated `_$AddressToJson` function.

To make this work, the `build_runner` tool generates the necessary code (`address.g.dart`), which contains the implementation of `_$AddressFromJson` and `_$AddressToJson`.

---

7. What is JSON serialization? Write and explain Dart code demonstrating the use of `json_annotation` and `json_serializable`.

JSON Serialization is the process of converting an object into a JSON string and vice versa. In Dart, it can be done with the help of the `json_serializable` package, which automates the creation of serialization logic (for converting to/from JSON).

Steps to Use `json_annotation` and `json_serializable`:

1. Add the `json_annotation` and `json_serializable` dependencies in `pubspec.yaml`.
2. Use `@JsonSerializable()` to mark the class.
3. Run code generation using `build_runner` to generate the necessary `.g.dart` file.

Example Code:

1. `pubspec.yaml` dependencies:

```yaml
dependencies:
  flutter:
    sdk: flutter
  json_annotation: ^4.8.0

dev_dependencies:
  build_runner: ^2.3.0
  json_serializable: ^6.0.0
```

2. **Dart Model Code (User.dart)**:

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

part 'user.g.dart';

@JsonSerializable()
class User {
  String name;
  String email;

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

  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);
  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

3. **Running Code Generation**: After adding the model and annotation, run the following command to generate the `.g.dart` file:

```bash
dart run build_runner build
```

This will generate the file `user.g.dart` with the `_$UserFromJson` and `_$UserToJson` functions.

4. **Using the Model**:

```dart
void main() {
  // Create a User object
  var user = User(name: 'John Doe', email: 'john@example.com');

  // Convert to JSON
  var json = user.toJson();
  print(json);

  // Convert from JSON
  var newUser = User.fromJson(json);
  print(newUser.name);  // Outputs: John Doe
}
```

Explanation:

- `` `@JsonSerializable()` `` tells Dart to automatically generate the serialization code.
- `` `fromJson()` `` and `` `toJson()` `` are used to serialize and deserialize the object.
- `` `build_runner` `` generates the functions `` `_$UserFromJson()` `` and `` `_$UserToJson()` `` in the `` `user.g.dart` `` file.

---

8. Analyze the following code snippet and discuss the errors underlined. Write and explain the command to resolve these errors.

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

part 'user.g.dart';

class User {
  User(this.name, this.email);
  String name;
  String email;

  factory User.fromJson(Map<String, dynamic> json) => $UserFromJson(json);
  Map<String, dynamic> toJson() => $UserToJson(this);
}
```

Errors:

1. **Typo in part statement**:
   - The `` `part` `` directive should refer to the generated `` `.g.dart` `` file. The correct syntax is:

     ```dart
     part 'user.g.dart';
     ```

   - The error happens because the generated file (`` `user.g.dart` ``) hasn't been created yet.
2. **Incorrect function names**:

- In the `fromJson` and `toJson` methods, the functions should be prefixed with `_`. The correct code should be:

  ```dart
  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);
  Map<String, dynamic> toJson() => _$UserToJson(this);
  ```

Command to Resolve:

After correcting the code, run the following command to generate the missing `user.g.dart` file:

```bash
dart run build_runner build
```

This will generate the required `user.g.dart` file that contains the necessary serialization code.

---

9. How to add `json_serializable` dependency in a Flutter project? Explain the use of the following commands:

1. **Add the Dependencies**: In `pubspec.yaml`, add:

   ```yaml
   dependencies:
     json_annotation: ^4.8.0

   dev_dependencies:
     build_runner: ^2.3.0
     json_serializable: ^6.0.0
   ```

2. **Explanation of Commands**:
   - `dart run build_runner build`: This command generates the necessary `.g.dart` files for your classes marked with `@JsonSerializable()`. It creates serialization and deserialization code automatically.
   - `dart run build_runner build --delete-conflicting-outputs`: This command generates the `.g.dart` files, and if there are any conflicting outputs (for example, an older generated file that doesn't match the current code), it deletes them and regenerates the correct files.
   - `dart run build_runner watch`: This command watches your code files for changes. Whenever you make changes to your model classes, it automatically regenerates the `.g.dart` files.
   - `dart run build_runner watch --delete-conflicting-outputs`: Similar to the above, but it also deletes any conflicting generated files when changes occur.

---

10. Write Dart code and discuss the use of automated JSON serialization using code generation.

Automated JSON serialization allows developers to focus on business logic without worrying about manually writing the serialization and deserialization code. The `json_serializable` package in Dart uses code generation to automatically generate these methods.

Example Code:

1. `pubspec.yaml`:

```yaml
dependencies:
  flutter:
    sdk: flutter
  json_annotation: ^4.8.0

dev_dependencies:
  build_runner: ^2.3.0
  json_serializable: ^6.0.0
```

2. **Model Code (Product.dart)**:

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

part 'product.g.dart';

@JsonSerializable()
class Product {
  final String id;
  final String name;
  final double price;

  Product({
    required this.id,
    required this.name,
    required this.price,
  });

  factory Product.fromJson(Map<String, dynamic> json) => _$ProductFromJson(json);
  Map<String, dynamic> toJson() => _$ProductToJson(this);
}
```

3. **Usage**:

```dart
void main() {
  // Create a product
  var product = Product(id: '001', name: 'Laptop', price: 1299.99);

  // Convert product to JSON
  var json = product.toJson();
  print(json);

  // Convert JSON back to Product
  var newProduct = Product.fromJson(json);
```

```
    print(newProduct.name);  // Outputs: Laptop
  }
```

4. **Command to Generate Code**:

After defining the `Product` class, run:

```bash
dart run build_runner build
```

This generates the necessary `product.g.dart` file with the `_$ProductFromJson` and `_$ProductToJson` methods.

Discussion:

- **Automation**: The `json_serializable` package automates the serialization process, generating the `fromJson` and `toJson` methods.
- **Code Generation**: The `build_runner` tool automatically generates code based on annotations, ensuring you don't have to write boilerplate code for serializing and deserializing objects manually.

11. Analyze and explain the following code snippet:

Future<Person> fetchPerson() async {
}
final response = await http
.get(Uri.parse('http://192.168.0.103/getStudentService.php/'));
if (response.statusCode == 200) {
// If the server did return a 200 OK response,
// then parse the JSON.
return Person. fromJson(jsonDecode(response.body)); } else {
}
// If the server did not return a 200 OK response, // then throw an exception.
throw Exception('Failed to load person');
12. Write and explain Dart code for a flutter application to download data
from web API using http package and display on screen.
13. What is the use of chopper package? Explain at least two advantages of
Chopper compare to http package.
14. Analyze and explain the following code snippet:

import 'package: chopper/chopper.dart'; part 'student_service. chopper.dart';
@ChopperApi()
abstract class Student Service extends ChopperService {
static StudentService create() {
final client = ChopperClient(
baseUrl:
Uri.parse('http://192.168.0.103/getStudentService.php/'),

interceptors: [_addQuery],
services: [_$StudentService()],
converter: const JsonConverter(),
); // ChopperClient
return _$StudentService(client);
}
}
static Request _addQuery (Request req) {
}
final params = Map<String, dynamic>. from (req.parameters); return req.copyWith (parameters: params);
@Get()
Future<Response> getStudent (@Query() String rank,);
15. Write and explain Dart code to demonstrate use of chopper interceptor.

11. Analyze and explain the following code snippet:

```dart
  Future<Person> fetchPerson() async {
    final response = await http
      .get(Uri.parse('http://192.168.0.103/getStudentService.php/'));

    if (response.statusCode == 200) {
      // If the server did return a 200 OK response,
      // then parse the JSON.
      return Person.fromJson(jsonDecode(response.body));
    } else {
      // If the server did not return a 200 OK response,
      // then throw an exception.
      throw Exception('Failed to load person');
    }
  }
```

Explanation:

- `` `fetchPerson()` ``: This is an asynchronous function that returns a `` `Future<Person>` ``. This means that it will eventually return a `` `Person` `` object once the network request completes.
- `` `await http.get(...)` ``: The function makes an HTTP GET request to the provided URL (`` `'http://192.168.0.103/getStudentService.php/'` ``). The `` `await` `` keyword makes the code wait for the HTTP request to complete before moving forward.
- `` `if (response.statusCode == 200)` ``: This checks if the HTTP request was successful (i.e., the server responded with a 200 status code, which means "OK").
- `` `Person.fromJson(jsonDecode(response.body))` ``: If the response is successful, the body of the response (which is in JSON format) is decoded into a Dart object using `` `jsonDecode()` ``. Then, the decoded data is passed to the `` `fromJson` `` method of the `` `Person` `` class, which converts the JSON data into a `` `Person` `` object.

- **Exception Handling**: If the HTTP request doesn't return a 200 status code, the function throws an exception with the message `'Failed to load person'`.

12. Write and explain Dart code for a Flutter application to download data from a web API using the `http` package and display it on the screen.

Example:

```dart
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

class Person {
  final String name;
  final int age;

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

  factory Person.fromJson(Map<String, dynamic> json) {
    return Person(
      name: json['name'],
      age: json['age'],
    );
  }
}

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Fetch Data Example")),
        body: PersonScreen(),
      ),
    );
  }
}

class PersonScreen extends StatefulWidget {
  @override
  _PersonScreenState createState() => _PersonScreenState();
}

class _PersonScreenState extends State<PersonScreen> {
  late Future<Person> person;

  @override
  void initState() {
    super.initState();
    person = fetchPerson();
  }

  Future<Person> fetchPerson() async {
    final response = await http.get(Uri.parse('http://192.168.0.103/getStudentService.ph
p/'));

    if (response.statusCode == 200) {
      return Person.fromJson(jsonDecode(response.body));
    } else {
      throw Exception('Failed to load person');
    }
  }

  @override
  Widget build(BuildContext context) {
      return Center(
        child: FutureBuilder<Person>(
          future: person,
          builder: (context, snapshot) {
            if (snapshot.connectionState == ConnectionState.waiting) {
              return CircularProgressIndicator();
            } else if (snapshot.hasError) {
              return Text('Error: ${snapshot.error}');
            } else if (snapshot.hasData) {
              return Text('Name: ${snapshot.data!.name}, Age: ${snapshot.data!.age}');
            } else {
              return Text('No data');
            }
          },
        ),
      );
    }
  }
```

Explanation:

- **`Person` Class**: This class represents a `Person` with a `name` and `age`. It also has a `fromJson` factory constructor to convert JSON into a `Person` object.
- **Fetching Data**: The `fetchPerson` function makes an HTTP request to the provided API endpoint. If successful, it returns a `Person` object; if the request fails, it throws an error.
- **`FutureBuilder` Widget**: This widget listens for the `Future` to complete and rebuilds the UI based on the current state (waiting, error, or data). It shows a loading spinner while waiting, an error message if something goes wrong, and the person's name and age when the data is successfully retrieved.

13. What is the use of the `chopper` package? Explain at least two advantages of `Chopper` compared to the `http` package.

The `chopper` package is a powerful HTTP client for Flutter and Dart that helps in making network requests more manageable and efficient. It simplifies API requests by allowing for easier model serialization, request/response interceptors, and dependency injection.

Two advantages of `Chopper` compared to the `http` package:

1. **Automatic Serialization**: Chopper integrates well with `json_serializable`, making it easy to automatically convert JSON data to Dart objects. With `http`, you need to manually handle JSON parsing and object creation, which can be cumbersome for larger projects.
2. **Request and Response Interceptors**: Chopper provides interceptors that allow you to manipulate requests and responses globally. For example, you can automatically add authentication tokens to every request or log every response. This is much harder to achieve with the basic `http` package.

---

14. Analyze and explain the following code snippet:

```
dart


  import 'package:chopper/chopper.dart';
  part 'student_service.chopper.dart';
```

```
@ChopperApi()
abstract class StudentService extends ChopperService {
  static StudentService create() {
    final client = ChopperClient(
      baseUrl: Uri.parse('http://192.168.0.103/getStudentService.php/'),
      interceptors: [_addQuery],
      services: [_$StudentService()],
      converter: const JsonConverter(),
    );
    return _$StudentService(client);
  }

  static Request _addQuery(Request req) {
    final params = Map<String, dynamic>.from(req.parameters);
    return req.copyWith(parameters: params);
  }

  @Get()
  Future<Response> getStudent(@Query() String rank);
}
```

Explanation:

- `` `@ChopperApi()` ``: This annotation marks the class as a Chopper API service.
- `` `create()` `` **method**: This static method creates a `` `ChopperClient` `` with a base URL, interceptors, services, and converters. It sets up the client for making API requests and returns an instance of `` `StudentService` ``.
- `` `_addQuery()` `` **method**: This static method defines an interceptor that modifies requests by adding or manipulating query parameters. It ensures that every request has the appropriate parameters.
- `` `getStudent()` `` **method**: This method defines an endpoint (`` `GET` ``) for fetching a student's data based on a `` `rank` `` query parameter. It returns a `` `Future<Response>` `` where `` `Response` `` is the result of the HTTP request.
- **Chopper Client**: The `` `ChopperClient` `` is used to configure network requests. In this case, it includes the base URL, interceptors for adding query parameters, a `` `JsonConverter` `` for serializing data, and the `` `StudentService` `` class for defining the API endpoints.

---

15. Write and explain Dart code to demonstrate the use of a Chopper interceptor.

Interceptors in Chopper allow you to manipulate requests and responses globally. You can use interceptors to add headers, log requests, or handle authentication.

Example Code for Chopper Interceptor:

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

part 'auth_service.chopper.dart';

@ChopperApi()
abstract class AuthService extends ChopperService {
  static AuthService create() {
    final client = ChopperClient(
      baseUrl: Uri.parse('https://api.example.com'),
      interceptors: [_addAuthHeader],
      services: [_$AuthService()],
      converter: const JsonConverter(),
    );
    return _$AuthService(client);
  }

  @Post(path: '/login')
  Future<Response> login(@Body() Map<String, dynamic> body);

  // Interceptor that adds an Authorization header to every request
  static Request _addAuthHeader(Request request) {
    final updatedRequest = request.copyWith(
      headers: {
        'Authorization': 'Bearer your_api_token',
        ...request.headers,
      },
    );
    return updatedRequest;
  }
}
```

Explanation:

- **`_addAuthHeader`**: This interceptor adds an **`Authorization`** header with a bearer token to every request. It uses the **`copyWith`** method to modify the request before it is sent.
- **`login()` method**: This method represents a **`POST`** request for the login endpoint. The **`@Body()`** annotation indicates that the method will send data as the request body.
- **Using Interceptors**: Interceptors can be added globally to the **`ChopperClient`**, and they allow manipulation of requests before they are sent and responses after they are received.