Saving Data with Local Persistence

Choosing the right database for your Flutter app

- We would focus primarily on two types of databases that can be used with mobile technologies. Databases can be classified based on a number of factors, which range from the type of data they support, how they scale, how they can be defined, and the location of their storage.

- There are lots of databases out there, but we will stick with these two:
- Nonrelational databases (NoSQL)
- Relational databases (SQL)

SQFLite

- Lightweight, embedded, open-source database engine
- Included as part of the Flutter framework
- Can store and manage data in a structured format
- Good for small to medium-sized data sets
- Not suitable for large-scale or complex projects

Hive

- Facebook
- Lightweight, NoSQL database engine optimized for mobile apps
- Provides fast and efficient data storage and retrieval
- Good for small to medium-sized data sets
- Easy integration with Flutter through the Hive package
- Limited feature set compared to other options

Firebase

- Firebase was acquired by Google on Oct 21, 2014
- Cloud-hosted NoSQL database
- Real-time synchronization
- Easy integration with Flutter through Firebase SDK
- Scalable and flexible
- Limited free tier, with pricing based on usage

Understanding The JavaScript Object Notation (JSON) format

- The **JSON** format is **text-based** and is independent of programming languages ‹ **JSON** uses the **key/value** pair

- **EXAMPLES**.

Key/Value Pairs

KEY	COLON	VALUE
"id"	:	"100"
"quantity"	:	3
"in_stock"	:	true

– Objects and Arrays

TYPE	SAMPLE
Object	{ "id": "100", "name": "Vacation" }
Array with values only	["Family", "Friends", "Fun"]
Array with key/value	[{ "id": "100", "name": "Vacation" }, { "id": "102", "name": "Birthday" }]
Object with array	{ "id": "100", "name": "Vacation", "tags": ["Family", "Friends", "Fun"] }
Multiple objects with arrays	{ "journals":[{ "id":"4710827", "mood":"Happy" }, { "id":"427836", "mood":"Surprised" },], "tags":[{ "id": "100", "name": "Family" }, { "id": "102", "name": "Friends" }] }

Data Persistence in Flutter

Data persistence is an essential aspect of any mobile application. It involves storing and retrieving data even after the application is closed or the device is restarted. In Flutter, data persistence can be achieved using various methods, including secured local storage and API calls.

Local Storage

Storing data in local storage is a popular method for data persistence in mobile applications. Local storage refers to the storage space available on the device where data can be stored and retrieved. In Flutter, local storage can be implemented using the shared_preferences package. This package provides a simple way to store key-value pairs in local storage.
To use the shared_preferences package, add it to your dependencies in the pubspec.yaml file as follows:

```
flutter pub add shared_preferences
```

```
dependencies:
shared_preferences: ^2.3.4
```

Once you have added the package to your dependencies, you can use it to store and retrieve data from local storage. For instance, consider the following example where we want to store and retrieve a user's name:

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

class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  String _name = '';

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

  Future<void> _loadName() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    setState(() {
      _name = prefs.getString('name') ?? '';
    });
  }
```

```dart
Future<void> _saveName(String name) async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    setState(() {
      _name = name;
    });
    await prefs.setString('name', name);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Data Persistence in Flutter'),
      ),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Text('Hello $_name!'),
          TextField(
            decoration: InputDecoration(
              hintText: 'Enter your name',
            ),
            onSubmitted: (String value) {
              _saveName(value);
            },
          ),
        ],
      ),    );  }}
```

we have defined two methods – _loadName and _saveName. _loadName loads the user's name from local storage, while _saveName saves the user's name to local storage.

When the app is launched, the _loadName method is called in the initState method. This method loads the user's name from local storage using the SharedPreferences.getInstance() method. If the user's name is not found in local storage, the default value of an empty string is used.

When the user enters their name in the TextField widget and submits it, the _saveName method is called. This method saves the user's name to local storage using the prefs.setString('name', name) method.

In this way, the user's name is persisted in local storage even after the app is closed or the device is restarted. The next time the user launches the app, their name will be loaded from local storage and displayed on the screen.

User Login Code

LoginScreen:
• We have two TextEditingControllers for capturing the username and password.
• The _login() method checks if the username and password are correct (for simplicity, we use 'user' and 'password' as the correct values).
• If successful, the username is saved in SharedPreferences, and the user is navigated to the HomeScreen.
• If the credentials are incorrect, an error dialog is shown.

HomeScreen:
Displays a welcome message with the logged-in username.The Logout button clears the username from SharedPreferences and navigates back to the LoginScreen.

SharedPreferences:
Used to persist the logged-in state (username) between app restarts.The getString() method retrieves the saved username, and the setString() method saves the username.

Navigation:
After a successful login, the app navigates to the HomeScreen.If the app is restarted, the _checkLoggedIn() method checks if a username is already saved and navigates to the HomeScreen if the user is logged in.

```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: LoginScreen(),
    );
  }
}

class LoginScreen extends StatefulWidget {
  @override
  _LoginScreenState createState() => _LoginScreenState();
}

class _LoginScreenState extends State<LoginScreen> {
  final TextEditingController _usernameController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();

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

```dart
_checkLoggedIn() async {
  SharedPreferences prefs = await SharedPreferences.getInstance();
  String? username = prefs.getString('username');

  if (username != null) {
    Navigator.pushReplacement(
      context,
      MaterialPageRoute(builder: (context) => HomeScreen(username: username)),
    );
  }
}

// Login function
_login() async {
  String username = _usernameController.text;
  String password = _passwordController.text;

  // Check if username and password are correct (simple example)
  if (username == 'user' && password == 'password') {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    await prefs.setString('username', username);

    Navigator.pushReplacement(
      context,
      MaterialPageRoute(builder: (context) => HomeScreen(username: username)),
    );
  } else {
    _showErrorDialog();
  }  }
```

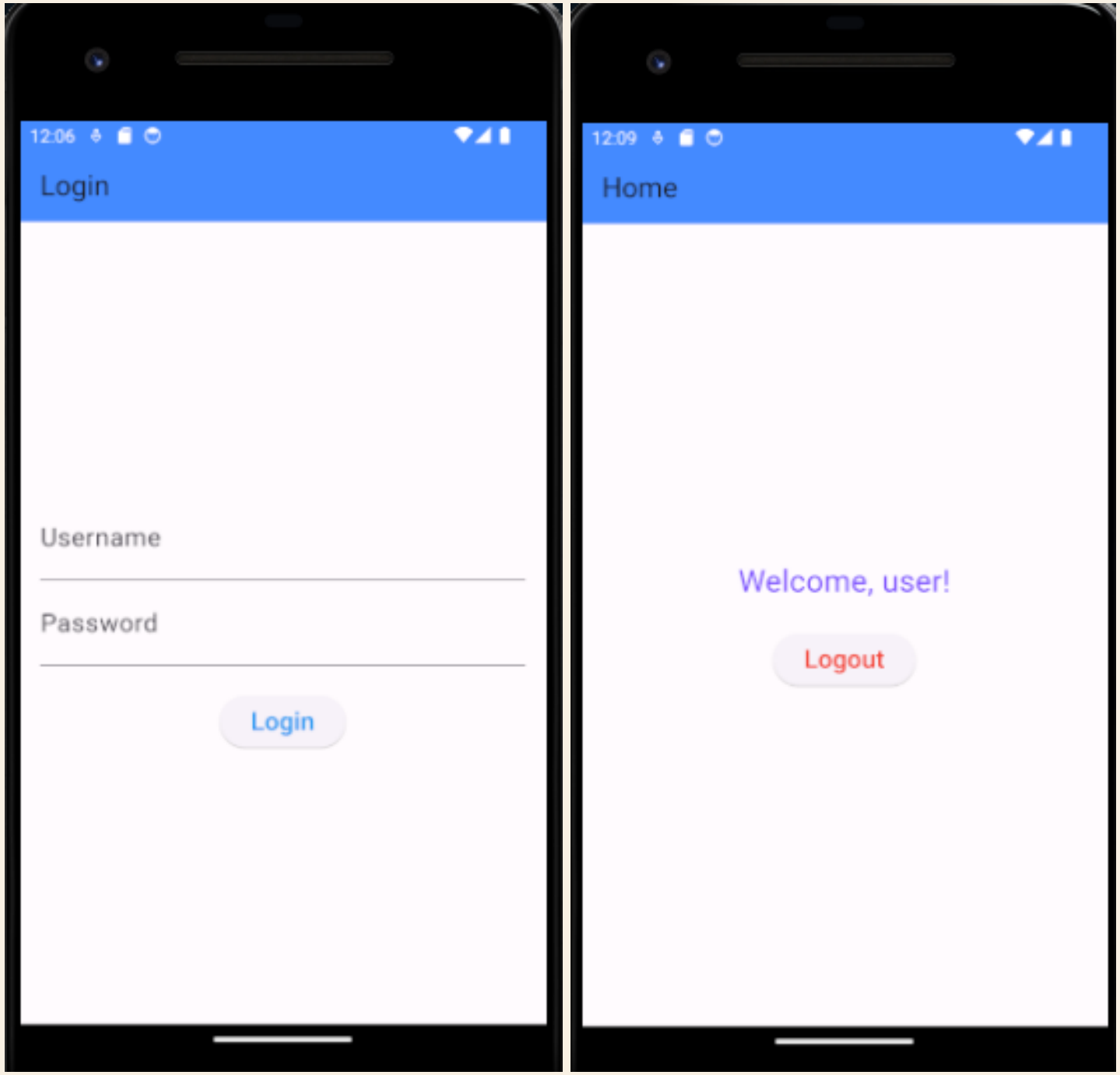
```
_showErrorDialog() {
    showDialog(
      context: context,
      builder: (BuildContext context) {
        return AlertDialog(
          title: Text('Login Failed'),
          content: Text('Invalid username or password'),
          actions: [
            TextButton(
              child: Text('OK'),
              onPressed: () {
                Navigator.of(context).pop();
              },
            ),
          ],
        );
      },
    );
  }
```

```dart
@override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(title: Text("Login")),
     body: Padding(
       padding: EdgeInsets.all(16),
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: [
           TextField(
             controller: _usernameController,
             decoration: InputDecoration(labelText: 'Username'),
           ),
           TextField(
             controller: _passwordController,
             obscureText: true,
             decoration: InputDecoration(labelText: 'Password'),
           ),
           SizedBox(height: 20),
           ElevatedButton(
             onPressed: _login,
             child: Text('Login'),
           ),
         ],
       ),
     ),
   );
 }
}
```

```dart
class HomeScreen extends StatelessWidget {
  final String username;

  HomeScreen({required this.username});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Home")),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text('Welcome, $username!', style: TextStyle(fontSize: 24)),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: () async {
                SharedPreferences prefs = await SharedPreferences.getInstance();
                await prefs.remove('username');
                Navigator.pushReplacement(
                  context,
                  MaterialPageRoute(builder: (context) => LoginScreen()),
                );
              },
              child: Text('Logout'),
            ),
          ],
        ),
      ),    );  }}
```


12:06
Login
Username
Password
Login
12:09
Home
Welcome, user!
Logout

Minimizing API Calls

Another way to achieve data persistence in Flutter is to minimize API calls. API calls refer to the process of retrieving data from a server using an API. Minimizing API calls involves reducing the number of times data is retrieved from the server, which can improve app performance and reduce data usage.

One way to minimize API calls is to cache data locally. Caching involves storing data in local storage for a certain period of time and retrieving it from local storage instead of making an API call every time the data is needed.

In Flutter, caching can be implemented using the flutter_cache_manager package. This package provides a cache manager that can be used to store and retrieve data from local storage.

To use the flutter_cache_manager package, add it to your dependencies in the pubspec.yaml file as follows:

```
dependencies:
flutter_cache_manager: ^3.1.2
```

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

class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  List<dynamic> _posts = [];

  @override
  void initState() {
    super.initState();
    _loadPosts();
  }
Future<void> _loadPosts() async {
    var cacheManager = await CacheManager.getInstance();
    var file = await
cacheManager.getFile('https://jsonplaceholder.typicode.com/posts');

    if (file != null && await file.exists()) {
      // Read data from cache
      var jsonString = await file.readAsString();
      var data = jsonDecode(jsonString);
      setState(() {
        _posts = data; });
```

```
    } else {
      // Make API call
      var response = await
http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));
      if (response.statusCode == 200) {
        // Cache data
        await cacheManager.putFile('https://jsonplaceholder.typicode.com/posts',
response.bodyBytes);
        var data = jsonDecode(response.body);
        setState(() {
          _posts = data;
        });
      } else {
        throw Exception('Failed to load posts');    }   }  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Data Persistence in Flutter'),
      ),
      body: ListView.builder(
        itemCount: _posts.length,
        itemBuilder: (context, index) {
          return ListTile(
            title: Text(_posts[index]['title']),
            subtitle: Text(_posts[index]['body']),
          );     },    ),     );   } }
```

In the above code, we have defined a _loadPosts method that retrieves a list of posts from an API. This method first checks if the data is available in local storage using the cacheManager.getFile() method. If the data is available in local storage, it is loaded from the cache and displayed on the screen. If the data is not available in local storage, an API call is made using the http.get() method. The data retrieved from the API is then cached using the cacheManager.putFile() method.

In this way, the data is retrieved from the server only when it is not available in local storage. Once the data is retrieved, it is cached in local storage for a certain period of time, reducing the number of API calls made by the app.

Overall, data persistence is a critical consideration for any mobile application. By using secure local storage and caching data, you can reduce the number of API calls made by your app, improve its performance, and provide a better user experience for your users. With Flutter, it is easy to implement data persistence in your app and ensure that your users' data is protected and accessible when needed.