Level 3: Advanced Concepts

1. Behavioral Components

- **Permissions**:
  - Managing app permissions (e.g., camera, storage, location) is crucial for accessing device features. The `permission_handler` package in Flutter provides a simple API for checking and requesting permissions.
  - **Example**:

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

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

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

class PermissionExample extends StatelessWidget {
  Future<void> _checkPermission() async {
    var status = await Permission.camera.status;
    if (status.isDenied) {
      if (await Permission.camera.request().isGranted) {
        print('Camera permission granted');
      }
    }
  }

  @override
```

```
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Permissions Example')),
      body: Center(
        child: ElevatedButton(
          onPressed: _checkPermission,
          child: Text('Check Camera Permission'),
        ),
      ),
    );
  }
}
```

- **Notifications**:
  - Push notifications allow apps to notify users about important events. The `firebase_messaging` package is commonly used in Flutter to integrate Firebase Cloud Messaging (FCM) for handling push notifications.
  - **Example**:

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

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

class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  final FirebaseMessaging _firebaseMessaging =
FirebaseMessaging.instance;

  @override
  void initState() {
    super.initState();
    _firebaseMessaging.requestPermission();
    FirebaseMessaging.onMessage.listen((RemoteMessage message) {
      print('Message received: ${message.notification?.title}');
    });
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text('Push Notifications Example')),
        body: Center(child: Text('Waiting for notifications...')),
      ),
    );
```

```
  }
}
```

- **Media**:
  - Handling media in Flutter involves tasks like playing videos, capturing images, and recording audio. The `video_player` and `image_picker` packages are widely used for these tasks.
  - **Example (Image Picker)**:

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

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

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

class ImagePickerExample extends StatefulWidget {
  @override
  _ImagePickerExampleState createState() => _ImagePickerExampleState();
}

class _ImagePickerExampleState extends State<ImagePickerExample> {
  final ImagePicker _picker = ImagePicker();
  XFile _image;

  Future<void> _pickImage() async {
    final pickedFile = await _picker.pickImage(source:
ImageSource.gallery);
    setState(() {
      _image = pickedFile;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Image Picker Example')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            _image == null ? Text('No image selected.') :
Image.file(File(_image.path)),
            ElevatedButton(
```

```
              onPressed: _pickImage,
              child: Text('Pick Image'),
            ),
          ],
        ),
      ),
    );
  }
}
```

2. Architecture Patterns

- **MVVM (Model-View-ViewModel)**:
  - MVVM is an architectural pattern that helps to separate the UI (View) from the business logic (Model) using ViewModel. It is popular in Flutter when using state management solutions like `Provider`.
  - **Theory**:
    - **Model**: Handles data and business logic.
    - **View**: Displays the UI and interacts with the user.
    - **ViewModel**: Acts as a bridge between the Model and the View, managing the state.
  - **Example**:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => CounterViewModel(),
      child: MaterialApp(
        home: CounterView(),
      ),
    );
  }
}

class CounterViewModel extends ChangeNotifier {
  int _counter = 0;

  int get counter => _counter;

  void incrementCounter() {
    _counter++;
    notifyListeners();
```

```
  }
}

class CounterView extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('MVVM Example')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('Counter Value:'),
            Consumer<CounterViewModel>(
              builder: (context, model, child) {
                return Text('${model.counter}', style:
Theme.of(context).textTheme.headline4);
              },
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          Provider.of<CounterViewModel>(context, listen:
false).incrementCounter();
        },
        child: Icon(Icons.add),
      ),
    );
  }
}
```

- **MVC (Model-View-Controller)**:
  - MVC is another architectural pattern that divides the application into three interconnected components. The `Model` represents the data, the `View` displays the data, and the `Controller` handles the input.
  - **Theory**:
    - **Model**: Manages the data and business logic.
    - **View**: Displays the UI and interacts with the user.
    - **Controller**: Acts as an intermediary between the Model and the View, handling user input.
  - **Example**:

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

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

class MyApp extends StatelessWidget {
```

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

class CounterModel {
  int _counter = 0;

  int get counter => _counter;

  void increment() {
    _counter++;
  }
}

class CounterController extends StatefulWidget {
  @override
  _CounterControllerState createState() => _CounterControllerState();
}

class _CounterControllerState extends State<CounterController> {
  final CounterModel _model = CounterModel();

  void _incrementCounter() {
    setState(() {
      _model.increment();
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('MVC Example')),
      body: CounterView(_model, _incrementCounter),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: Icon(Icons.add),
      ),
    );
  }
}

class CounterView extends StatelessWidget {
  final CounterModel model;
  final VoidCallback onIncrement;

  CounterView(this.model, this.onIncrement);

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
```

```
          Text('Counter Value:'),
          Text('${model.counter}', style:
Theme.of(context).textTheme.headline4),
        ],
      ),
    );
  }
}
```

- **Lifting State**:
  - "Lifting State Up" refers to moving state to the closest common ancestor widget when multiple widgets need to share the same state. It helps in keeping the state management more organized and localized to a part of the widget tree.

3. Storage

- **Local Storage**:
  - Flutter allows storing data locally using shared preferences, SQLite, or third-party libraries like Hive.
- **Shared Preferences (Continued)**:
  - Shared preferences are used to store simple data in key-value pairs. This is suitable for storing user settings or small amounts of data that need to persist between app launches.
  - **Example**:

```
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: SharedPreferencesExample(),
    );
  }
}

class SharedPreferencesExample extends StatefulWidget {
  @override
  _SharedPreferencesExampleState createState() =>
```

```
_SharedPreferencesExampleState();
}

class _SharedPreferencesExampleState extends
State<SharedPreferencesExample> {
  int _counter = 0;

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

  _loadCounter() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    setState(() {
      _counter = (prefs.getInt('counter') ?? 0);
    });
  }

  _incrementCounter() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    setState(() {
      _counter++;
      prefs.setInt('counter', _counter);
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Shared Preferences Example'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('You have pushed the button this many times:'),
            Text('$_counter', style:
Theme.of(context).textTheme.headline4),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

- **Hive**:

- Hive is a lightweight and fast key-value database written in pure Dart. It is a great alternative for local storage when you need to handle more complex or larger amounts of data compared to Shared Preferences.
- **Example**:

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

void main() async {
  await Hive.initFlutter();
  await Hive.openBox('myBox');
  runApp(MyApp());
}

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

class HiveExample extends StatefulWidget {
  @override
  _HiveExampleState createState() => _HiveExampleState();
}

class _HiveExampleState extends State<HiveExample> {
  Box box;

  @override
  void initState() {
    super.initState();
    box = Hive.box('myBox');
  }

  void _incrementCounter() {
    int currentValue = box.get('counter', defaultValue: 0);
    box.put('counter', currentValue + 1);
    setState(() {});
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Hive Example'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
```

```
            Text('You have pushed the button this many times:'),
            Text('${box.get('counter', defaultValue: 0)}', style:
Theme.of(context).textTheme.headline4),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

- **SQLite**:
  - SQLite is a C-language library that provides a lightweight, disk-based database. In Flutter, the `sqflite` package is commonly used to interact with SQLite databases. It is suitable for more complex queries and relational data storage.
  - **Example**:

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

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

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

class SQLiteExample extends StatefulWidget {
  @override
  _SQLiteExampleState createState() => _SQLiteExampleState();
}

class _SQLiteExampleState extends State<SQLiteExample> {
  Database database;
  List<Map<String, dynamic>> data = [];

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

```
  Future<void> _initDatabase() async {
    database = await openDatabase(
      join(await getDatabasesPath(), 'demo.db'),
      onCreate: (db, version) {
        return db.execute(
          "CREATE TABLE items(id INTEGER PRIMARY KEY, name TEXT)",
        );
      },
      version: 1,
    );
    _loadData();
  }

  Future<void> _insertData() async {
    await database.insert(
      'items',
      {'name': 'Item ${data.length + 1}'},
      conflictAlgorithm: ConflictAlgorithm.replace,
    );
    _loadData();
  }

  Future<void> _loadData() async {
    final List<Map<String, dynamic>> maps = await
database.query('items');
    setState(() {
      data = maps;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('SQLite Example'),
      ),
      body: Center(
        child: ListView.builder(
          itemCount: data.length,
          itemBuilder: (context, index) {
            return ListTile(
              title: Text(data[index]['name']),
            );
          },
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _insertData,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Level 4: Continuous Integration

1. FastLane:

- **Theory**:
  - FastLane is an open-source platform that automates the building and deployment of iOS and Android apps. It integrates well with Continuous Integration/Continuous Deployment (CI/CD) pipelines, enabling automated testing, screenshots, and more.
- **Example**:
  - Setup for Flutter typically involves configuring FastLane for both Android and iOS by setting up `Fastfile` and `Appfile` configurations. Tasks like app distribution, code signing, and more can be automated using lanes in the `Fastfile`.
  - **Fastfile Example**:

```
default_platform(:ios)

platform :ios do
  desc "Deploy iOS app to TestFlight"
  lane :beta do
    build_ios_app(scheme: "MyApp")
    upload_to_testflight
  end
end

platform :android do
  desc "Deploy Android app to Play Store"
  lane :beta do
    gradle(task: "assembleRelease")
    upload_to_play_store(track: "beta")
  end
end
```

2. CI Server:

- **Theory**:
  - A Continuous Integration (CI) server automates the testing and integration of code into the repository. Popular CI servers include Jenkins, CircleCI, and GitHub Actions. CI servers help maintain code quality by running automated tests on every commit.
- **Example**:

- Integrating a CI server involves creating configuration files that define the environment and steps for building and testing the app. For example, using GitHub Actions, a YAML file (`.github/workflows/ci.yml`) can be configured to automate the build and test process.
- **GitHub Actions Example**:

```
name: Flutter CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - uses: subosito/flutter-action@v2
      with:
        flutter-version: '2.5.3'
    - run: flutter pub get
    - run: flutter test
    - run: flutter build apk --release
```

3. Code Metrics:

- **Theory**:
  - Code metrics involve measuring the quality of code, including aspects like complexity, readability, and test coverage. Tools like `flutter analyze` and `flutter test` provide insight into code quality.
- **Example**:
  - Running `flutter analyze` in the terminal will check for coding issues, while `flutter test` runs unit tests to ensure code correctness. These tools can be integrated into the CI pipeline to enforce code quality standards.
  - **Using `flutter analyze`**:

```
flutter analyze
```