ChatGPT

1. What is asynchronous function? Explain any two advantages of stream.
2. With appropriate example, explain how to create stream with necessary Dart code snippet.
3. What is single subscription stream? Explain it with appropriate example and Dart code.
4. What is broadcast stream? Explain it with appropriate example and Dart code.
5. Compare and differentiate single subscription streams and broadcast streams.

ANSWER DECRIPTIVELY WITH FULL CODE

1. What is an Asynchronous Function? Explain any two advantages of Stream.

Asynchronous Function:

An **asynchronous function** in Dart allows the program to perform non-blocking operations. This means the program can execute other tasks while waiting for a task (like reading a file or making an HTTP request) to finish. Instead of waiting for the task to complete and freezing the app's execution, an asynchronous function lets the app remain responsive.

In Dart, asynchronous functions are denoted with the `async` keyword, and they typically return a `Future`. A `Future` represents a result that might not be available yet but will be in the future.

Advantages of Stream:

A **Stream** is a sequence of asynchronous events or data. Streams in Dart are used to handle multiple asynchronous data, unlike `Future`, which only handles one asynchronous result.

1. **Efficient Handling of Multiple Data Events**: Streams allow handling multiple asynchronous events over time. Instead of waiting for one operation to complete before starting another, you can subscribe to a stream that sends events (data) as they become available. This is especially useful for scenarios like receiving real-time data from a server or listening to user input over time.
2. **Asynchronous Iteration**: Streams allow asynchronous iteration, meaning that you can receive and process each item of data as it becomes available. This can help prevent blocking the main thread of the application and maintain responsiveness.

---

2. How to Create a Stream in Dart?

A **Stream** in Dart can be created using the `` `Stream` `` class, and there are two types of streams: **Single-subscription** and **Broadcast** streams.

Example of creating a stream:

Here's an example of creating a stream using `` `Stream.periodic` `` to generate periodic events:

```dart
import 'dart:async';

void main() {
  // Creating a stream that emits a new value every second
  Stream<int> myStream = Stream.periodic(Duration(seconds: 1), (count) => count);

  // Listening to the stream
  myStream.listen((event) {
    print('Event: $event');
    if (event == 5) {
      // Cancel the subscription after 5 events
      print('Stream is completed');
      myStream.listen((event) {}).cancel();
    }
  });
}
```

In this example:

- The stream emits a new integer every second, starting from 0.
- `` `Stream.periodic` `` generates a stream of integers (count).
- The `` `listen()` `` method is used to listen for the emitted events, and after printing 5 events, we cancel the subscription.

---

3. What is Single Subscription Stream? Explain with Example and Dart Code.

Single Subscription Stream:

A **single-subscription stream** is a type of stream where only one listener can be attached at a time. Once the listener is done, the stream is closed, and no further listeners can be added.

These streams are typically used for data that is emitted once or in a limited sequence. They are often created using methods like `` `Stream.fromIterable()` `` or `` `Stream.periodic()` ``.

Example of Single Subscription Stream:

```dart
import 'dart:async';

void main() {
  // Create a single-subscription stream using Stream.fromIterable
  Stream<int> singleSubscriptionStream = Stream.fromIterable([1, 2, 3, 4, 5]);

  // Listen to the stream
  singleSubscriptionStream.listen((event) {
    print('Received: $event');
    });
  }
```

- The `Stream.fromIterable()` method creates a single-subscription stream from a list of numbers.
- Only one listener can listen to this stream. Once all values are emitted, the stream is closed.

Key Characteristics:

- It can only have one listener.
- The stream cannot be reused once the events have been consumed.

---

4. What is Broadcast Stream? Explain with Example and Dart Code.

Broadcast Stream:

A **broadcast stream** allows multiple listeners to subscribe to the same stream. It broadcasts the same events to all its listeners. This is useful when you need to send data to multiple parts of your application at the same time.

You can create a broadcast stream using the `.asBroadcastStream()` method on an existing stream.

Example of Broadcast Stream:

```dart
  import 'dart:async';

  void main() {
    // Create a broadcast stream
    Stream<int> broadcastStream = Stream.periodic(Duration(seconds: 1), (count) => count)
        .take(5)
        .asBroadcastStream();

    // First listener
    broadcastStream.listen((event) {
      print('Listener 1: $event');
    });

    // Second listener
    broadcastStream.listen((event) {
      print('Listener 2: $event');
    });
  }
```

- In this example, the `Stream.periodic` creates a stream that emits values every second.
- The `.asBroadcastStream()` method converts this stream into a broadcast stream, allowing multiple listeners to receive the same data.
- Both listeners will print the same emitted value at the same time.

Key Characteristics:

- It can have multiple listeners.
- It doesn't close automatically after the first listener has finished.
- Broadcast streams are useful when you want to share data among multiple consumers.

5. Comparison: Single Subscription Stream vs Broadcast Stream

Feature	Single Subscription Stream	Broadcast Stream
Listeners	Only one listener at a time.	Multiple listeners can subscribe.
Usage	Used for a sequence of data to be consumed by a single consumer.	Used for broadcasting data to multiple consumers.
Reusability	Once a listener has consumed all the data, the stream closes and cannot be reused.	The stream remains open and can continue to emit data for multiple listeners.
Example	**`Stream.fromIterable([...])`**, **`Stream.periodic()`**	**`Stream.periodic(...).asBroadcastStream()`**
Performance	More efficient for a single consumer.	More overhead as it supports multiple listeners.

Example: Usage of Both Streams in Dart:

```dart
import 'dart:async';

void main() {
  // Example of Single Subscription Stream
  Stream<int> singleStream = Stream.fromIterable([1, 2, 3]);
  singleStream.listen((data) {
    print('Single Subscription: $data');
  });

  // Example of Broadcast Stream
  Stream<int> broadcastStream = Stream.periodic(Duration(seconds: 1), (count) => count).take
(3).asBroadcastStream();

  // Listener 1 for Broadcast Stream
  broadcastStream.listen((data) {
    print('Broadcast Listener 1: $data');
  });

  // Listener 2 for Broadcast Stream
  broadcastStream.listen((data) {
    print('Broadcast Listener 2: $data');
  });
}
```

- The first part creates a **single-subscription stream**, and only one listener can be attached.
- The second part creates a **broadcast stream**, where multiple listeners can receive the same data simultaneously.

This comparison and code examples should give you a thorough understanding of streams, single-subscription streams, and broadcast streams in Dart.

6. With appropriate example and necessary code snippet, explain usage of StreamController, Stream, and StreamSink.
7. What is StreamBuilder? With appropriate example and necessary code snippet, explain usage of StreamBuilder.

8. With appropriate example and necessary code snippet, explain usage of StreamSubscription.
9. Analyze and explain following code snippet:

```
Class CounterApp extends StatefulWidget {
 const CounterApp({Key key}) : super(key: key);
 @override
 _CounterAppState createState() => _CounterAppState();
 }
 Class _CounterAppState extends State<CounterApp> {
 final StreamController _controller = StreamController();
 int _counter = 10;
 void startTimer() async {
 Timer.periodic(constant Duration(seconds: 1), (timer) {
 _counter—;
 // add event/data to stream controller using sink
 _controller.sink.add(_counter);
 if(_counter <= 0) {
 timer.cancel();
 }
 }); //Timer.periodic
 }
```

10. Analyze following code snippet and explain use of StreamBuilder.

```
 @override
 void dispose() {
 super.dispose(); _controller.close();
 }
 @override
 Widget build(BuildContext context) {
 return Scaffold(
 body: Center(
 child: Column(
 mainAxisAlignment: MainAxisAlignment.center,
 children: [
StreamBuilder( initialData: _counter,
 stream: _controller.stream,
 builder: (context, snapshot) {
 return Text('${snapshot.data}');
 }),
 const SizedBox(height:20,),
 ElevatedButton(
 onPressed: () { _counter = 10;
 startTimer(); },
 child: const Text('Start Count Down'))
 ]
```

6. Usage of StreamController, Stream, and StreamSink

StreamController:

A `StreamController` is used to create and manage a stream in Dart. It controls the flow of events from a producer (your code) to a consumer (a listener). A `StreamController` provides access to a **Stream** that listeners can subscribe to and a **StreamSink** that can be used to add data to the stream.

Stream:

A `Stream` represents a sequence of asynchronous events. It can emit multiple values over time, like user interactions, data from sensors, or network responses.

StreamSink:

A `StreamSink` is an object provided by the `StreamController` that allows you to add events or data to the stream. It is how you send data from your program to the stream. You can add data to the stream using `sink.add()`.

Example of StreamController Usage:

```dart
import 'dart:async';

void main() {
  // Creating a StreamController
  final StreamController<int> controller = StreamController<int>();

  // Create a stream and add data to it using the controller's sink
  controller.sink.add(1); // Adding the first value to the stream
  controller.sink.add(2); // Adding the second value to the stream

  // Listen to the stream
  controller.stream.listen((event) {
    print('Received: $event');
  });

  // Close the StreamController after usage
  controller.close();
}
```

In this example:

- A `StreamController` is created with a generic type of `int`.
- Data (`1` and `2`) is added to the stream using the `sink.add()` method.
- The stream is then listened to, and the values are printed to the console.

Key Steps:

1. Create a `StreamController`.
2. Use the `sink.add()` method to add data to the stream.
3. Listen to the stream using `.listen()`.

---

7. What is StreamBuilder?

StreamBuilder:

`StreamBuilder` is a Flutter widget that builds itself based on the latest snapshot of interaction with a stream. It listens to the stream and rebuilds the widget whenever the stream emits new data.

`StreamBuilder` is ideal for displaying real-time data that may change over time, like a countdown timer, chat messages, or data fetched from a network.

Usage of StreamBuilder:

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

void main() {
  runApp(MaterialApp(home: CounterApp()));
}

class CounterApp extends StatefulWidget {
  @override
  _CounterAppState createState() => _CounterAppState();
}

class _CounterAppState extends State<CounterApp> {
  final StreamController<int> _controller = StreamController<int>();
  int _counter = 10;

  void startTimer() {
    Timer.periodic(Duration(seconds: 1), (timer) {
      _counter--;
      _controller.sink.add(_counter);
      if (_counter <= 0) {
        timer.cancel();
      }
    });
  }

  @override
  void dispose() {
    super.dispose();
    _controller.close(); // Close the StreamController to avoid memory leaks
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('StreamBuilder Example')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            StreamBuilder<int>(
              initialData: _counter, // initial data
              stream: _controller.stream, // listening to the stream
              builder: (context, snapshot) {
                if (snapshot.connectionState == ConnectionState.waiting) {
                  return CircularProgressIndicator(); // show a loading indicator if the str
eam is waiting
                }
                if (snapshot.hasError) {
                  return Text('Error: ${snapshot.error}');
                }
                return Text('Counter: ${snapshot.data}', style: TextStyle(fontSize: 50));
              },
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {
                _counter = 10;
                startTimer(); // Start the countdown when the button is pressed
              },
              child: Text('Start Countdown'),
            ),
          ],
        ),
      ),
    );
  }
}
```

Explanation:

- `` `StreamBuilder` `` listens to `` `_controller.stream` ``, which is a stream of integer values representing the countdown.
- The `` `initialData` `` is set to the initial value of `` `_counter` ``, which is `` `10` ``.
- The builder function gets called every time the stream emits a new value. It updates the widget tree accordingly, showing the updated value of `` `_counter` ``.

Key Elements:

- `` `stream` ``: The stream that the `` `StreamBuilder` `` listens to.
- `` `builder` ``: A function that returns the widget based on the `` `snapshot` `` (the data from the stream).
- `` `initialData` ``: The initial value before any data is emitted from the stream.

---

8. Usage of StreamSubscription

A **StreamSubscription** is a handle to a stream subscription. It allows you to manage the stream's behavior, like canceling the subscription when no longer needed or pausing/resuming the stream.

Example of StreamSubscription:

```dart
import 'dart:async';

void main() {
  final StreamController<int> controller = StreamController<int>();
  final StreamSubscription subscription = controller.stream.listen(
    (data) {
      print('Received: $data');
    },
    onError: (error) {
      print('Error: $error');
    },
    onDone: () {
      print('Stream closed');
    },
  );

  // Adding data to the stream
  controller.sink.add(1);
  controller.sink.add(2);
  controller.sink.addError('An error occurred');
  controller.sink.add(3);

  // Canceling the subscription
  subscription.cancel();
```

```
    controller.close();
  }
```

Explanation:

- The `StreamSubscription` is used to listen to the stream.
- We can handle the data with `onData`, handle errors with `onError`, and call a function when the stream is closed with `onDone`.
- The `subscription.cancel()` method is used to stop receiving events from the stream.

---

9. Analyzing the CounterApp Code

```dart
class CounterApp extends StatefulWidget {
  const CounterApp({Key? key}) : super(key: key);
  @override
  _CounterAppState createState() => _CounterAppState();
}

class _CounterAppState extends State<CounterApp> {
  final StreamController _controller = StreamController();
  int _counter = 10;

  void startTimer() async {
    Timer.periodic(const Duration(seconds: 1), (timer) {
      _counter--;
      // Add event/data to stream controller using sink
      _controller.sink.add(_counter);
      if (_counter <= 0) {
        timer.cancel();
      }
    });
  }

  @override
  void dispose() {
    super.dispose();
    _controller.close();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            StreamBuilder<int>(
              initialData: _counter,
              stream: _controller.stream,
              builder: (context, snapshot) {
                return Text('${snapshot.data}');
              },
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {
                _counter = 10;
                startTimer();
              },
              child: const Text('Start Count Down'),
            ),
          ],
        ),
      ),
      );
    }
  }
```

Analysis:

- This is a simple Flutter app that counts down from 10 to 0 and updates the UI in real-time using `` `StreamController` `` and `` `StreamBuilder` ``.
- The `` `_controller` `` is a `` `StreamController` `` that handles the countdown data.
- The `` `startTimer()` `` method uses `` `Timer.periodic` `` to decrement `` `_counter` `` every second and add the updated value to the stream using `` `_controller.sink.add(_counter)` ``.
- The `` `StreamBuilder` `` widget listens to `` `_controller.stream` ``, displaying the current value of `` `_counter` `` as the stream emits new data.
- The `` `dispose()` `` method ensures that the `` `StreamController` `` is closed when the widget is disposed to avoid memory leaks.

---

10. Use of StreamBuilder in the Code

In this code:

```dart
  StreamBuilder<int>(
    initialData: _counter,
    stream: _controller.stream,
    builder: (context, snapshot) {
      return Text('${snapshot.data}');
    },
  )
```

- `` `StreamBuilder` `` listens to `` `_controller.stream` ``, which is updated every second when `` `startTimer()` `` is called.
- `` `initialData: _counter` `` sets the initial value of the counter to 10.
- `` `builder: (context, snapshot)` `` provides a `` `snapshot` `` of the stream's current state. It updates the UI with the current data emitted by the stream.
- `` `snapshot.data` `` provides the current value of `` `_counter` ``, which is displayed in the UI.

This detailed breakdown covers how to use `` `StreamController` ``, `` `Stream` ``, `` `StreamSink` ``, `` `StreamBuilder` ``, and `` `StreamSubscription` `` in Flutter, along with their integration to implement real-time UI updates like a countdown timer.

11. Analyze and explain following code snippet:
ElevatedButton(
onPressed: () {
setState(() { result = “\n”;
Stream<String> stream = broadcastController.stream;
StreamSubscription<String> subscriber1 =
stream.listen((String data) {

```
 result += “Subscriber1: $data\n”;
 }, onError: (error) { result += “Subscriber1: $error\n”;
 }, onDone: () { result += “Subscriber1: Stream Closed!\n”;
 });
 StreamSubscription<String> subscriber2 =
stream.listen((String data) {
 result += “Subscriber2: $data\n”;
 }, onError: (error) { result += “Subscriber2: $error\n”;
 }, onDone: () { result += “Subscriber2: Stream Closed!
 \n”;
 });
 broadcastController.sink.add(‘UTU’);
 broadcastController.addError(‘Error…’);
 broadcastController.close();
 });
 },
 child: const Text(‘Broadcast Stream’)), //ElevatedButton
 const SizedBox(height: 20, ),
 Text(result, style: const textStyle(fontSize : 20,),)
```

12. How to store data in SQLite databse using sqflite package? Write Dart code snippet and explain.
13. How to read data from SQLite databse using sqflite package? Write Dart code snippet and explain.
14. Analyze and explain following code snippet:

```
import ‘package:path/path.dart’;

 import ‘package:sqflite/sqflite.dart’;
 class MyDb{
 late Database db;
 Future open async {
 // Get a location using getDatabasesPath
 var databasePath  = await getDatabasesPath();
 String path = join(databasesPath, ‘demo.db’);
 // join is from path package
 print(path);
 // output /data/user/0/com.example.sqflight_student/
 databases/demo.db
 db = await openDatabase(path, version: 1,
 onCreate: (Database db, int version) async{
 // When creating the db, create the table
 await db.execute(‘‘’
 CREATE TABLE IF NOT EXISTS students(
 id primary key,
 name archer(255) not null,
 roll_no int not null,
 address varchar(255) not null
 );
 ‘‘’);
 // table students will be created if there is no table ‘students’
 print(“Table Created”);
 });
```

15. Write necessary Dart code and explain rawInsert() and rawDelete() with respect to sqflite.

11. Analyze and Explain the Code Snippet

```dart
ElevatedButton(
  onPressed: () {
    setState(() {
      result = "\n";
      Stream<String> stream = broadcastController.stream;

      StreamSubscription<String> subscriber1 =
        stream.listen((String data) {
          result += "Subscriber1: $data\n";
        }, onError: (error) {
          result += "Subscriber1: $error\n";
        }, onDone: () {
          result += "Subscriber1: Stream Closed!\n";
        });

      StreamSubscription<String> subscriber2 =
        stream.listen((String data) {
          result += "Subscriber2: $data\n";
        }, onError: (error) {
          result += "Subscriber2: $error\n";
        }, onDone: () {
          result += "Subscriber2: Stream Closed!\n";
        });

      broadcastController.sink.add('UTU');
      broadcastController.addError('Error...');
      broadcastController.close();
    });
  },
  child: const Text('Broadcast Stream'),
), // ElevatedButton
const SizedBox(height: 20,),
Text(result, style: const TextStyle(fontSize: 20,)),
```

Explanation:

- **`ElevatedButton`**: The button triggers the **`onPressed`** callback, which sets up the stream and its subscribers.
- **`broadcastController.stream`**: This is a broadcast stream that allows multiple listeners (subscribers) to listen to the same stream and receive the events emitted by it.
- **`StreamSubscription<String> subscriber1` and `subscriber2`**:
  - Each subscriber is subscribed to the same broadcast stream.
  - They both listen to the events (data) being emitted from the stream and process them using their own callbacks for **`onData`**, **`onError`**, and **`onDone`**.
  - **`onData`**: Handles the data received from the stream. In this case, it appends a string to the **`result`**.
  - **`onError`**: Handles errors. It appends the error message to **`result`**.

    - `onDone`: Handles the stream closure. It appends a message indicating that the stream has been closed.
- `broadcastController.sink.add('UTU')`: Adds the string `'UTU'` to the broadcast stream, which both subscribers will receive.
- `broadcastController.addError('Error...')`: An error is added to the stream. Both subscribers will receive the error in their `onError` callbacks.
- `broadcastController.close()`: Closes the stream after the data and error events are emitted. Both subscribers will trigger their `onDone` callbacks when the stream is closed.
- `Text(result)`: The result string is displayed as output in the UI. The `result` variable accumulates the events, errors, and stream closures processed by both subscribers.

Summary:

- The code demonstrates the use of a **broadcast stream**, where multiple listeners can subscribe to the same stream and handle the data, errors, and completion events.
- It uses `StreamSubscription` to manage the stream and listen to it, then displays the results in a `Text` widget.
- The button action simulates emitting data and errors, and then closes the stream.

---

12. How to Store Data in SQLite Database Using `sqflite` Package?

To store data in SQLite using the `sqflite` package, we need to:

1. Open or create the database.
2. Create a table (if it doesn't exist).
3. Insert data into the table.

Code Snippet:

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

void main() async {
  var db = await openDatabase(
    join(await getDatabasesPath(), 'my_database.db'),
    onCreate: (db, version) {
      return db.execute(
        "CREATE TABLE users(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)",
      );
    },
    version: 1,
  );

  // Insert a record
  await db.insert(
    'users',
    {'name': 'Alice', 'age': 30},
    conflictAlgorithm: ConflictAlgorithm.replace,
  );

  print('Data inserted');
```

```
    await db.close();
  }
```

Explanation:

- `` `openDatabase()` ``: Opens or creates a database. We specify the database name (`` `my_database.db` ``) and the `` `onCreate` `` function to create the table.
- `` `db.insert()` ``: Inserts a record into the `` `users` `` table. If a conflict occurs (e.g., trying to insert the same data), we use `` `ConflictAlgorithm.replace` `` to replace the existing row.

---

13. How to Read Data from SQLite Database Using `` `sqflite` `` Package?

To read data from SQLite, we use the `` `query()` `` method of the database.

Code Snippet:

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

  void main() async {
    var db = await openDatabase(
      join(await getDatabasesPath(), 'my_database.db'),
    );

    // Query all users
    List<Map<String, dynamic>> users = await db.query('users');

    // Print the data
    users.forEach((user) {
      print('User: ${user['name']}, Age: ${user['age']}');
    });

    await db.close();
  }
```

Explanation:

- `` `db.query('users')` ``: Queries all rows from the `` `users` `` table.
- The result is a `` `List<Map<String, dynamic>>` ``, where each `` `Map` `` represents a row in the database, and the map's keys correspond to the column names.

---

14. Analyze and Explain the Code Snippet for SQLite Database in `` `sqflite` ``

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

  class MyDb {
    late Database db;

    Future open() async {
```

```
    var databasePath = await getDatabasesPath();
    String path = join(databasePath, 'demo.db');
    print(path);

    db = await openDatabase(path, version: 1, onCreate: (Database db, int version) async {
      await db.execute('''
        CREATE TABLE IF NOT EXISTS students(
          id INTEGER PRIMARY KEY,
          name TEXT NOT NULL,
          roll_no INTEGER NOT NULL,
          address TEXT NOT NULL
        );
      ''');
      print("Table Created");
    });
  }
}
```

Explanation:

- `` `getDatabasesPath()` ``: This method returns the default location to store databases on the device.
- `` `join()` ``: This combines the database path with the database name (`` `demo.db` ``) to form the full path.
- `` `openDatabase()` ``: Opens the database at the specified path. If it doesn't exist, the `` `onCreate` `` callback is called to create the database and tables.
- `` `db.execute()` ``: Executes SQL commands to create a table named `` `students` `` if it doesn't already exist.
- `` `CREATE TABLE IF NOT EXISTS` ``: Ensures the `` `students` `` table is only created once if it doesn't already exist.

Summary:

This code demonstrates how to open a database, check for its existence, and create a table if needed using the `` `sqflite` `` package in Flutter.

---

15. Raw Insert and Raw Delete in `` `sqflite` ``

Code Snippet for `` `rawInsert()` `` and `` `rawDelete()` ``:

```
dart


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

  class MyDb {
    late Database db;

    Future open() async {
      var databasePath = await getDatabasesPath();
      String path = join(databasePath, 'demo.db');
      db = await openDatabase(path, version: 1, onCreate: (Database db, int version) async {
        await db.execute('''
          CREATE TABLE IF NOT EXISTS students(
            id INTEGER PRIMARY KEY,
            name TEXT NOT NULL,
            roll_no INTEGER NOT NULL,
            address TEXT NOT NULL
          );
        ''');
      });
    }

    Future<int> rawInsert() async {
      // Insert using raw SQL
      int id = await db.rawInsert(
        'INSERT INTO students(name, roll_no, address) VALUES(?, ?, ?)',
        ['John Doe', 123, '123 Main St'],
      );
      return id;  // Returns the inserted row id
    }

    Future<int> rawDelete(int id) async {
      // Delete using raw SQL
      int count = await db.rawDelete(
        'DELETE FROM students WHERE id = ?',
        [id],
      );
      return count;  // Returns the number of rows affected
    }
  }
```

Explanation:

- `` `rawInsert()` ``: Executes a raw SQL `` `INSERT` `` statement. The `` `?` `` placeholders are used for parameterized queries to prevent SQL injection. The method returns the ID of the inserted row.
- `` `rawDelete()` ``: Executes a raw SQL `` `DELETE` `` statement. The method deletes a row based on the provided `` `id` `` and returns the number of affected rows.

Summary:

- `` `rawInsert()` `` is used to insert data into the database using raw SQL queries.
- `` `rawDelete()` `` is used to delete data from the database using raw SQL queries. Both methods allow for flexibility in executing SQL commands directly.

16. Analyze and explain following code snippet:
 import 'package: flutter/material.dart';
 import 'db.dart';
 class EditStudent extends StatefulWidget {
 final int rollno;
 const EditStudent ({super.key, required this. rollno});
 @override
 State StatefulWidget> createState() {
 return _EditStudent();
 }
 }
 class _EditStudent extends State<EditStudent> {
TextEditingController name = TextEditingController();
 TextEditingController rollno
 TextEditingController();
 TextEditingController address = TextEditingController();
 MyDb mydb MyDb();
 @override
 void initState() {
 mydb.open();

Future.delayed (const Duration (milliseconds: 500), () async
{ var data = await mydb.getStudent (widget.rollno);
if (data != null) {
name×text = data["name"];
rollno.text = data["roll_no"].toString();
address×text = data["address"];
setState(() });
} else {
print("Student NOT found with roll no:
"+widget.rollno.toString());

17. What is Moor? Explain any four advantages of Moor.
18. Analyze and explain following code snippet (with respect to drift):

```
import 'dart: io';
import 'package: drift/drift.dart';
import 'package: drift/native.dart';
import 'package: path/path.dart' as p;
import 'package: sqflite/sqflite.dart' show getDatabasesPath;
part 'moor_database.g.dart';
class Tasks extends Table {
IntColumn get id => integer().autoIncrement ()0;
TextColumn get name => text().with Length (min: 1, max: 50)();
DateTimeColumn get dueDate => dateTime().nullable()();
BoolColumn get completed => boolean().withDefault (const
Constant(false))();
}
@DriftDatabase (tables: [Tasks], daos: [TaskDao])
class AppDatabase extends _$AppDatabase {
AppDatabase()
: super(_openConnection());
@override
int get schemaVersion => 2;
}
```

19. Analyze and explain following code snippet (with respect to drift):

```
LazyDatabase _openConnection() {
return LazyDatabase(() async {
final dbFolder = await getDatabasesPath();
final file File (p.join(db Folder, 'db.sqlite'));
return NativeDatabase(file);
}); // LazyDatabase
}
@DriftAccessor(
tables: [Tasks],
queries: {
'completedTasksGenerated':
'SELECT * FROM tasks WHERE completed = 1 ORDER BY
due_date DESC, name;'
},}
```

16. Analyze and Explain the Code Snippet for Editing a Student in Flutter

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

class EditStudent extends StatefulWidget {
  final int rollno;

  const EditStudent({super.key, required this.rollno});

  @override
  State<EditStudent> createState() {
    return _EditStudent();
  }
}

class _EditStudent extends State<EditStudent> {
  TextEditingController name = TextEditingController();
  TextEditingController rollno = TextEditingController();
  TextEditingController address = TextEditingController();
  MyDb mydb = MyDb();

  @override
  void initState() {
    super.initState();
    mydb.open();

    Future.delayed(const Duration(milliseconds: 500), () async {
      var data = await mydb.getStudent(widget.rollno);

      if (data != null) {
        name.text = data["name"];
        rollno.text = data["roll_no"].toString();
        address.text = data["address"];
        setState(() {});
      } else {
        print("Student NOT found with roll no: " + widget.rollno.toString());
      }
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Edit Student")),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: [
            TextField(
              controller: name,
              decoration: InputDecoration(labelText: 'Name'),
            ),
            TextField(
              controller: rollno,
              decoration: InputDecoration(labelText: 'Roll Number'),
            ),
            TextField(
              controller: address,
              decoration: InputDecoration(labelText: 'Address'),
            ),
            ElevatedButton(
              onPressed: () {
                mydb.updateStudent(widget.rollno, name.text, rollno.text, address.text);
              },
              child: Text("Update"),
            ),
          ],
        ),
      ),
    );
  }
}
```

Explanation:

1. `` `EditStudent` `` **Widget**:
   - This is a stateful widget that allows users to edit student details. It takes the student's `` `rollno` `` as a parameter to fetch the existing student data for editing.
2. `` `TextEditingController` ``:
   - The `` `TextEditingController` `` instances are used to manage text input fields for `` `name` ``, `` `rollno` ``, and `` `address` ``. They allow the widget to read the user's input and display the existing values in these fields.
3. `` `initState()` ``:
   - The `` `initState` `` method is called when the widget is first initialized. Here, a delayed `` `Future` `` is used to simulate fetching student data asynchronously.
   - The `` `getStudent()` `` method is called with `` `widget.rollno` `` to fetch the student's existing data from the database.
   - Once the data is retrieved, the `` `TextEditingController` `` fields are updated with the data, and `` `setState` `` is called to trigger a UI update.
4. `` `mydb.getStudent(widget.rollno)` ``:
   - This method is used to retrieve student data from the database based on the student's roll number. The fetched data is then used to populate the `` `TextField` `` controllers.
5. **Updating Student**:
   - The `` `ElevatedButton` `` triggers the `` `updateStudent()` `` method from `` `mydb` `` to update the student details in the database when pressed.

---

17. What is Moor? Explain Any Four Advantages of Moor

Moor is a **Flutter** library that simplifies working with SQLite databases by providing a reactive API. It is built on top of the SQLite database, providing an easy way to manage local data and allows working with structured data in a high-performance manner. Moor is similar to libraries like `` `drift` `` and provides a way to access databases using Dart code instead of SQL queries.

Four Advantages of Moor:

1. **Reactive Queries**:
   - Moor supports reactive queries. This means you can listen to database changes and update the UI accordingly without having to manually refresh or reload data.
2. **Type-safe API**:
   - Moor provides a type-safe API that prevents errors in SQL queries by using Dart objects instead of writing raw SQL strings, reducing the chances of runtime errors.
3. **Automatic Schema Migrations**:

   - Moor helps in handling database schema migrations automatically. You can define your schema in Dart, and Moor will generate the necessary SQL code for migration.
4. **Integrates with Dart Streams**:
   - Moor supports Dart streams, which allows you to easily manage the results of queries as streams. This enables you to listen to database changes and update the UI reactively.

---

18. Analyze and Explain the Code Snippet (with respect to Drift)

```dart
import 'dart:io';
import 'package:drift/drift.dart';
import 'package:drift/native.dart';
import 'package:path/path.dart' as p;
import 'package:sqflite/sqflite.dart' show getDatabasesPath;
part 'moor_database.g.dart';

class Tasks extends Table {
  IntColumn get id => integer().autoIncrement()();
  TextColumn get name => text().withLength(min: 1, max: 50)();
  DateTimeColumn get dueDate => dateTime().nullable()();
  BoolColumn get completed => boolean().withDefault(Constant(false))();
}

@DriftDatabase(tables: [Tasks], daos: [TaskDao])
class AppDatabase extends _$AppDatabase {
  AppDatabase() : super(_openConnection());

  @override
  int get schemaVersion => 2;
}
```

Explanation:

- **`Tasks` Class**:
  - This defines a database table **`Tasks`** with columns **`id`**, **`name`**, **`dueDate`**, and **`completed`**. The **`id`** column is set to auto-increment.
  - **`name`** is a **`TextColumn`** with a length constraint, **`dueDate`** is nullable, and **`completed`** is a boolean column with a default value of **`false`**.
- **`@DriftDatabase`**:
  - This annotation tells Drift to generate a database class (**`AppDatabase`**) based on the **`Tasks`** table and the **`TaskDao`** data access object (DAO).
  - The **`AppDatabase`** class extends **`_AppDatabase`** and provides access to the database.
  - **`schemaVersion`** specifies the version of the database schema. It is used to handle database migrations.
- **`_openConnection()`**:
  - A helper function that establishes the database connection. It uses **`LazyDatabase`** to open the SQLite database lazily.

---

19. Analyze and Explain the Code Snippet (with respect to Drift)

```dart
LazyDatabase _openConnection() {
  return LazyDatabase(() async {
    final dbFolder = await getDatabasesPath();
    final file = File(p.join(dbFolder, 'db.sqlite'));
    return NativeDatabase(file);
  });
}

@DriftAccessor(
  tables: [Tasks],
  queries: {
    'completedTasksGenerated': 'SELECT * FROM tasks WHERE completed = 1 ORDER BY due_date DE
SC, name;',
  },
)
```

Explanation:

- **`_openConnection()`**:
  - This function establishes the connection to the SQLite database using **`LazyDatabase`**. It ensures that the database is opened only when it's needed, improving startup performance.
  - **`getDatabasesPath()`** provides the default location for storing the database.
  - **`NativeDatabase(file)`** creates an instance of the **`NativeDatabase`** using the database file's path.
- **`@DriftAccessor`**:
  - This annotation is used to define a data access object (DAO) that works with the **`Tasks`** table.
  - The **`queries`** field defines a custom SQL query (**`completedTasksGenerated`**) to fetch tasks that are marked as completed. The query orders the tasks by their **`due_date`** in descending order and **`name`**.

Summary:

- **`LazyDatabase`**: Used to lazily initialize the database connection.
- **`@DriftAccessor`**: Customizes DAO by defining custom SQL queries and managing data interactions with the **`Tasks`** table.

20. Step-by-step explain how to set app icon and launch screen in flutter app
for deploying on Android platform.
21. Step-by-step explain how to set app icon and launch screen in flutter app
for deploying on iOS platform.

20. Step-by-Step Guide to Set App Icon and Launch Screen in Flutter App for Android Platform

Setting App Icon in Flutter for Android:

To set the app icon for an Android Flutter app, you can use the `` `flutter_launcher_icons` `` package.

1. **Add the `` `flutter_launcher_icons` `` dependency:**
   - Open your `` `pubspec.yaml` `` file and add the `` `flutter_launcher_icons` `` package under dependencies.

```yaml
yaml

dev_dependencies:
  flutter_launcher_icons: ^0.9.2

flutter_icons:
  android: true
  ios: false
  image_path: "assets/icon/app_icon.png"
```

   - The `` `image_path` `` should be the path to the image you want to use as your app icon. It must be a square image (e.g., 512x512 pixels).
2. **Install the package:**
   - Run the following command to install the `` `flutter_launcher_icons` `` package:

```bash
bash

flutter pub get
```

3. **Run the icon generation command:**
   - After setting up the configuration, run the following command to generate the icons:

```bash
bash

flutter pub run flutter_launcher_icons:main
```

   - This will generate app icons for all required Android densities (mdpi, hdpi, xhdpi, xxhdpi, xxxhdpi) and update the `` `AndroidManifest.xml` `` with the correct icon references.
4. **Verify the App Icon:**
   - Once the command finishes running, navigate to the `` `android/app/src/main/res` `` directory. You should see several drawable folders (`` `drawable-mdpi` ``, `` `drawable-hdpi` ``, etc.) containing the generated icons.

Setting Launch Screen in Flutter for Android:

For the launch screen, Android uses a splash screen. Here's how to set it up:

1. **Create a Launch Screen Image:**
   - Create an image (e.g., `` `launch_screen.png` ``) you want to display as the launch screen. Make sure the image has a high resolution and a transparent background if necessary.
2. **Edit the `` `launch_background.xml` `` file:**

   - Navigate to `` `android/app/src/main/res/drawable/` `` and create or edit a file called `` `launch_background.xml` ``.

```xml
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@android:color/white" />
    <item>
        <bitmap
            android:src="@drawable/launch_screen"
            android:gravity="center" />
    </item>
</layer-list>
```

3. **Modify the `` `styles.xml` `` File:**
   - Go to `` `android/app/src/main/res/values/styles.xml` `` and add the following theme to define the splash screen:

```xml
<style name="LaunchTheme" parent="Theme.AppCompat.Light.NoActionBar">
    <item name="android:windowBackground">@drawable/launch_background</item>
</style>
```

4. **Update the `` `AndroidManifest.xml` `` File:**
   - Open `` `android/app/src/main/AndroidManifest.xml` `` and add the `` `LaunchTheme` `` to the `` `android:theme` `` attribute of the `` `<activity>` `` tag for the main activity.

```xml
<activity
    android:name=".MainActivity"
    android:launchMode="singleTop"
    android:theme="@style/LaunchTheme"
    android:configChanges="orientation|keyboardHidden|keyboard|screenSize|smallestScreenSize"
    android:hardwareAccelerated="true"
    android:windowSoftInputMode="adjustResize">
    <!-- Other configurations -->
</activity>
```

5. **Build and Run:**
   - Build the app and run it on an Android device/emulator:

```bash
flutter run
```

Result:

- When the app starts, it should display the custom app icon and launch screen.

21. Step-by-Step Guide to Set App Icon and Launch Screen in Flutter App for iOS Platform

Setting App Icon in Flutter for iOS:

1. **Add the `flutter_launcher_icons` dependency:**
   - Open your `pubspec.yaml` file and add the `flutter_launcher_icons` package. Make sure to enable iOS support.

```yaml
dev_dependencies:
  flutter_launcher_icons: ^0.9.2

flutter_icons:
  android: false
  ios: true
  image_path: "assets/icon/app_icon.png"
```

   - Set the `image_path` to the path of your app icon image (e.g., `assets/icon/app_icon.png`).
2. **Install the Package:**
   - Run the following command to install the `flutter_launcher_icons` package:

```bash
flutter pub get
```

3. **Run the icon generation command:**
   - To generate the app icon for iOS, run the following command:

```bash
flutter pub run flutter_launcher_icons:main
```

4. **Verify the App Icon:**
   - After running the command, Flutter will generate the app icon for all required iOS resolutions and update the `ios/Runner/Assets.xcassets/AppIcon.appiconset` directory with the generated icons.

Setting Launch Screen in Flutter for iOS:

For iOS, the launch screen is typically a storyboard. Here's how to set it up:

1. **Create a Launch Screen Image:**
   - Prepare an image (e.g., `launch_screen.png`) that you want to use as the launch screen. This image should be high resolution.
2. **Modify the Launch Screen Storyboard:**

- Open the `ios/Runner/Assets.xcassets/LaunchImage.imageset` directory.
- Alternatively, open `ios/Runner/LaunchScreen.storyboard`. If this file doesn't exist, you can create one.

Example `LaunchScreen.storyboard`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<document type="com.apple.InterfaceBuilder3.CocoaTouch.Storyboard.XIB" version="3.0" toolsVersion="17425" systemVersion="20E241" targetRuntime="iOS.CocoaTouch" propertyAccessControl="none" useAutolayout="YES" initialViewController="BYj-l6-Qm5">
    <device id="ipad" orientation="portrait" layout="regular"/>
    <dependencies>
        <dependency identifier="com.apple.InterfaceBuilder.IBCocoaTouchPlugin" version="17406"/>
        <dependency identifier="com.apple.InterfaceBuilder.IBCocoaTouchPlugin" version="17406"/>
    </dependencies>
    <objects>
        <view key="view" contentMode="scaleToFill" id="BYj-l6-Qm5">
            <rect key="frame" x="0.0" y="0.0" width="375" height="667"/>
            <subviews>
                <imageView contentMode="scaleAspectFit" image="LaunchImage" translatesAutoresizingMaskIntoConstraints="NO" id="L4I-i6-e0B">
                    <rect key="frame" x="0.0" y="0.0" width="375" height="667"/>
                </imageView>
            </subviews>
        </view>
    </objects>
</document>
```

3. **Link Launch Image in Xcode:**
   - If you created a custom `LaunchScreen.storyboard`, ensure that it is linked in the `Xcode` project.
   - Open the project in Xcode (`ios/Runner.xcworkspace`).
   - Under the `General` tab for the `Runner` target, find the **Launch Screen File** setting and set it to the `LaunchScreen.storyboard` file.
4. **Update Info.plist (if necessary):**
   - If you are using a storyboard, ensure the `Info.plist` file is correctly configured to use the launch screen.

   Example from `Info.plist`:

```xml
<key>UILaunchStoryboardName</key>
<string>LaunchScreen</string>
```

5. **Build and Run:**
   - Finally, build and run your Flutter app on an iOS device or simulator:

```bash
flutter run
```

Result:

- Your iOS app should now show the custom app icon and launch screen when the app is launched.

Summary:

- **For Android:**
  - Use the `` `flutter_launcher_icons` `` package to set the app icon.
  - Create a custom `` `launch_background.xml` `` for the launch screen and modify the `` `AndroidManifest.xml` `` to use the custom theme.
- **For iOS:**
  - Use `` `flutter_launcher_icons` `` to set the app icon.
  - Modify `` `LaunchScreen.storyboard` `` or use an image in `` `Assets.xcassets` `` for the launch screen, and link it in `` `Info.plist` ``.