Future and Stream in Flutter

ASYNCHRONOUS PROGRAMMING

• In a mobile application, you will use a lot of asynchronous, or async, programming. Async functions perform time-consuming operations without waiting for the operation to complete.

Key terms:

synchronous operation: A synchronous operation blocks other operations from executing until it completes.

synchronous function: A synchronous function only performs synchronous operations.

asynchronous operation: Once initiated, an asynchronous operation allows other operations to execute before it completes.

asynchronous function: An asynchronous function performs at least one asynchronous operation and can also perform synchronous operations.

Why asynchronous code matters

- Asynchronous operations let your program complete work while waiting for another operation to finish.

Here are some common asynchronous operations:

- Fetching data over a network.
- Writing to a database.
- Reading data from a file.

- To perform asynchronous operations in Dart, you can use the Future class and the async and await keywords.

Working with futures: async and await

- The async and await keywords provide a declarative way to define asynchronous functions and use their results.

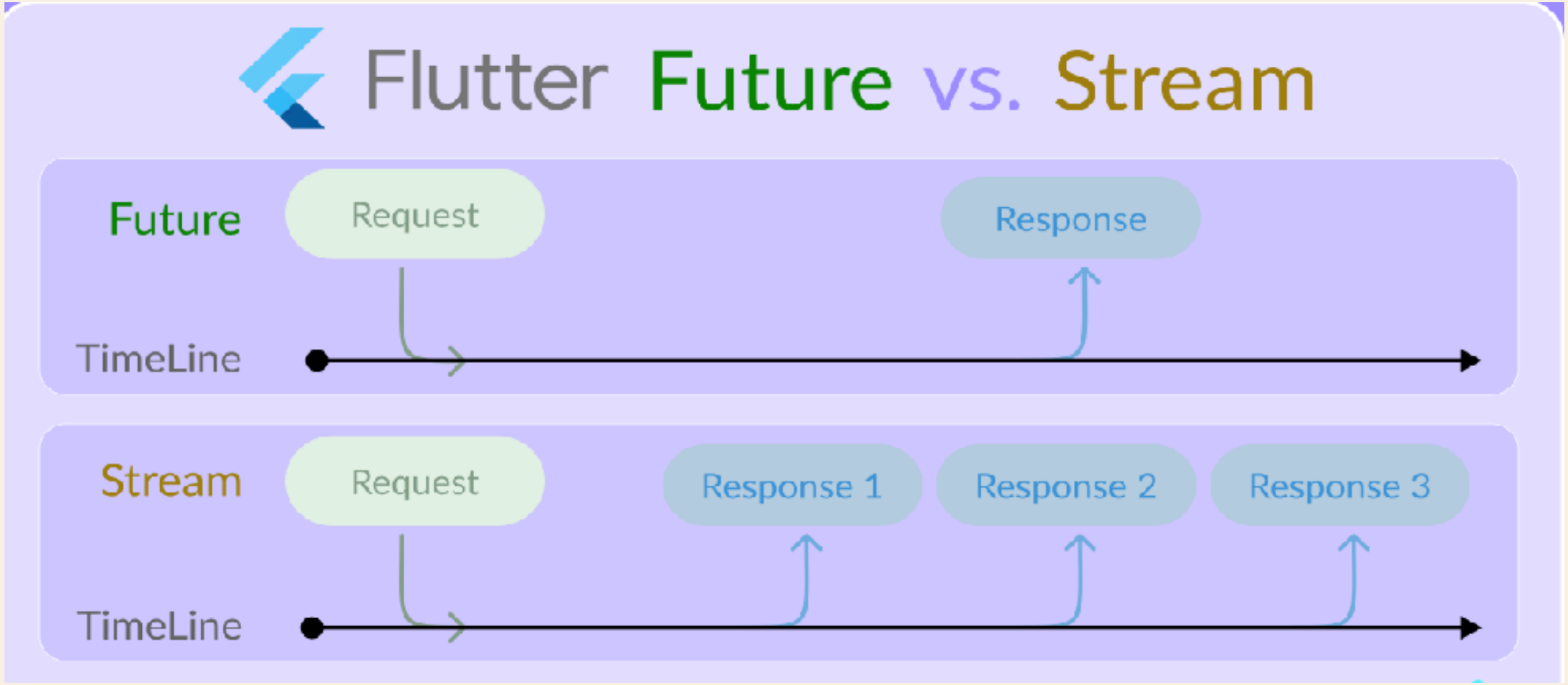
Remember these two basic guidelines when using async and await:

– To define an async function, add async before the function body:

– The await keyword works only in async functions.

In Flutter, both *Future* and *Stream* are fundamental to asynchronous programming, allowing developers to write code that can execute over time, responding to events or the completion of tasks.

In essence, if you're expecting a single result at some point in the future, you'd use a *Future*. If you're dealing with a series of values or events that will occur over time, you'd use a *Stream*.



1. Future

A **Future** represents a single asynchronous operation that will eventually produce a result or an error. It is used when you need to perform an action and receive a response sometime in the future. The result can be either successful or failed. Once a `Future` is created, it begins execution immediately.

Key Features:

- A Future completes once with a value or an error.
- Used for fetching data, waiting for a response, or delaying an action.

```dart
import 'dart:async';

void main() {
  print("Fetching data...");

  fetchData().then((result) {
    print("Result: $result");
  }).catchError((error) {
    print("Error: $error");
  });

  print("Waiting...");
}

Future<String> fetchData() async {
  // Simulate a delay
  await Future.delayed(Duration(seconds: 2));
  return "Data fetched successfully!";
}
```

```
Fetching data...
Waiting...
Result: Data fetched successfully!
```

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Future Example")),
        body: Center(
          child: FutureBuilder<String>(
            future: fetchData(),
            builder: (context, snapshot) {
              if (snapshot.connectionState == ConnectionState.waiting) {
                return CircularProgressIndicator();
              } else if (snapshot.hasError) {
                return Text("Error: ${snapshot.error}");
              } else {
                return Text("Result: ${snapshot.data}");
              }     },   ),   ),        ),     );   }

  Future<String> fetchData() async {
    await Future.delayed(Duration(seconds: 2));
    return "Data fetched successfully!";
  }
}
```

2. Stream

A Stream represents a sequence of asynchronous events. It is used when you expect multiple values to be emitted over time, rather than a single result. Streams are useful for handling continuous data flows, such as user input, network data, or sensor readings.

Key Features:

- Emits a sequence of events over time.
- Useful for real-time data updates (e.g., WebSockets, user input, or timers).

```dart
import 'dart:async';
Stream<int> generateNumbers() async* {
for (int i = 1; i <= 5; i++) {
await Future.delayed(Duration(seconds: 1));
yield i;
}
}
void main() {
print("Generating numbers...");
generateNumbers().listen((event) {
print(event);
});
print("Stream started.");
}
```

```
Generating numbers...
Stream started.
1
2
3
4
5
```

In this example, **generateNumbers()** produces numbers from 1 to 5 at one-second intervals. The `**listen()**` method is used to subscribe to the stream and receive the emitted values. As the numbers are generated asynchronously, the main function continues its execution without waiting for the stream to complete.

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text("Stream Example")),
        body: Center(
          child: StreamBuilder<int>(
            stream: countStream(5),
            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("No data");
              } else {
                return Text("Number: ${snapshot.data}");
              }        },   ),   ),   ),     );   }

  Stream<int> countStream(int count) async* {
    for (int i = 1; i <= count; i++) {
      await Future.delayed(Duration(seconds: 1));
      yield i;
    }   } }
```

Similarities:

• Both *Future* and *Stream* are used for handling asynchronous operations in Dart, which is the programming language for Flutter.

• They allow the execution of code sequences that depend on external events or data that will be available at a later time.

Differences:

• A *Future* represents a one-time asynchronous computation that will eventually complete with a value or an error. It's like to a promise of a single forthcoming result.

• A *Stream*, on the other hand, is a sequence of ongoing asynchronous events. It can emit multiple values over time, making it suitable for handling a series of related continuous events.