Click to prove you're human

RxDart is a reactive programming library for Dart and Flutter that provides a way to handle asynchronous and event-based programming using the principles of ReactiveX. It brings the power of observables, observers, and operators to simplify the management of asynchronous streams of data. If you are new to RxDart, here are some key concepts and tips to get started: Understanding Observables and Observers Observables are sources of data that emit values over time. They can represent streams of events, user inputs, network requests, or any other asynchronous data source. Observers, on the other hand, listen to these observables and react to the emitted values. Key Concepts in RxDart Streams In RxDart, an Observable is represented by a Stream. A Stream is a sequence of asynchronous events that can be listened to and processed. Subscriptions When you listen to a Stream, you create a Subscription. A Subscription allows you to control the flow of events by canceling or pausing the subscription. Operators RxDart provides a rich set of operators that allow you to transform, filter, combine, and manipulate streams of data. These operators provide powerful and expressive ways to handle complex asynchronous scenarios. Getting Started with RxDart To use RxDart in your Dart or Flutter project, you need to add the rxdart package to your pubspec.yaml file and import it into your code. Once you have imported the package, you can start using RxDart in your project. Here's a basic example to help you understand the usage:import 'package:rxdart/rxdart.dart'; void main() { final numbers = Stream.fromIterable([1, 2, 3, 4, 5]); final doubledNumbers = numbers.map((number) => number * 2); final subscription = doubledNumbers.listen(print); // Output: 2, 4, 6, 8, 10 subscription.cancel();} In the above example, we created an Observable from an iterable of numbers. We then applied a transformation using the map operator to double each number in the stream.Finally, we listened to the resulting stream and print the doubled numbers. Don't forget to cancel the subscription to release resources when you're done with the stream. Further Learning and Resources To deepen your understanding of RxDart, here are some recommended resources: Official RxDart GitHub repository: documentation: documentation: By exploring these resources and experimenting with RxDart in your own projects, you'll quickly grasp the power and flexibility that reactive programming with RxDart brings to your Dart and Flutter applications. Remember, practice and hands-on experience are key to mastering any new technology. Happy coding! Hello again! Have you already heard about reactive programming? RxDart is a reactive functional programming library for Dart language, based on ReactiveX. Dart already has a decent package to work with Streams, but RxDart comes to adds functionality on top of it. But now you might be asking, whats Stream?Streams and SinksStreams represent flux of data and events, and what its important for? With Streams, you can listen to data and event changes, and just as well, deal with whats coming from the Stream with listeners. How it can be applied to Flutter? For example, we have a Widget in Flutter called StreamBuilder that builds itself based on the latest snapshot of interaction with a Stream, and when theres a new flux of data the Widget reload to deal with the new data. Widget Weekly of Flutter Dev Channel offers great content about how the StreamBuilder works. And about Sinks? If we have an output of a data flux, we also need an input, thats what Sinks is used for, seems simple right? Now lets see about the BLoC pattern and how can we combine both concepts into a great Flutter app.Stream of Cats The BLoC(Bussiness Logic Component) Pattern was announced officially by Paolo Soares in the Dart Conference 2018. If you saw the announcement video, probably you realized that the initial proposal was to reuse the code related to the business logic in other platforms, in this case, Angular Dart. Shortly what the pattern seeks for, is take all business logic code off the UI, and using it only in the BLoC classes. It brings to the project and code, independence of environment and platform, besides put the responsibilities in the correct component. And now our talk will make much more sense, because BLoC Pattern only relies on the use of Streams. at the image above, we can realize the flux. The Widgets send data/event to the BLoC class through Sink and are notified by Stream. See that theres no business logic in the widget, that means what happened in BLoC is not the concern of UI. This architecture improves even easier tests, in which the business logic tests cases needed to be applied only to the BLoC classes.RxDart is now (at the moment of this post) in the version 0.21.0. And here Im going to talk about some objects that the library brings to us.Observable classObservable allow us to send a notification to Widgets which is observing it and then deal with the flux of data. Observable class in RxDart extends from Stream, which implies in some great things:All methods defined on the Stream class exist on Observable as well.All Observable can be passed to any API that expects a Dart Stream as an input (including for example StreamBuilder Widget).PublishSubject classThis one is pretty simple. This Subject allows sending data, error and done events to the listener. Here it will work with Sinks, which we were talking about before. See the example above:PublishSubject subject = new PublishSubject();/*this listener below will print every integer added to the subject: 1, 2, 3, ...*/subject.stream.listen(print);subject.add(1);subject.add(2);/*but this listener below will print only the integer added after his initialization: 3, .../*subject.stream.listen(print);subject.add(3);BehaviorSubject classThis one is similar to the PublishSubject. It also allows sending data, error and done events to the listener, but the latest item that has been added to the subject will be sent to any new listeners of the subject. But dont you worry, after that, any new events will be appropriately sent to the listeners. See the example above:BehaviorSubject subject = new BehaviorSubject();subject.stream.listen(print); // prints 1,2,3 subject.add(1);subject.add(2);subject.add(3);subject.stream.listen(print); // prints 3ReplaySubject classThe ReplaySubject allow us the same: sending data, error and done events to the listener. But with a crucial difference here. As items are added to the subject, the ReplaySubject will store them and when the stream is listened to, those recorded items will be emitted to the listener. See the example above:ReplaySubject subject = new ReplaySubject();subject.add(1);subject.add(2);subject.add(3);subject.stream.listen(print); // prints 1, 2, 3Now lets see it in practiceIn this article I will show to you a simple example of using RxDart and principles of BLoC pattern. Lets start it.Now I really need your attentionA great way to start it, is from the beginning: Flutter Hello World. Probably you are familiarized with the increment function on the app, but to make more lets create the decrement function as well. So first of all, create a flutter project and import rxdart to your project. Lets code:As you can see, this code implements the increment and decrement function, but still doesnt apply the BLoC pattern or even Streams. This code works and its pretty simple, but if you took attention youll see that we have two logic business function in the UI code: increment and decrement. So imagine if this app was a big app that you was working hard, but now the requirement has been changed and the increment needs to add two at time. Do you agree with me (that in this case) a requirement changing in the business logic shouldnt affect UI code, right? If yes, great! You got it, that is the point to separate responsibilities.Now lets separate it and use what we have learned so far. Lets create our CounterBloc class:bloc/CounterBloc.dartGreat! Now let me explain the code above. We created a class called CounterBloc which imports the rxdart library. In this case, we need to receive the initialCount, that allow us to know from which number our counter should begin. I choose for this example the BehaviorSubeject, and then I initialized the Subject with the data passed by parameter, in other words, when the Widget become a listener of the Subject the first value passed through the stream will be the initialCount which was set in the CounterBloc constructor. Now lets talk about the methods. In this case, we have four methods in the class:increment(): increment the initialCount and send to the Subject listeners by Sink the new value.decrement(): decrement the initialCount and send to the Subject listeners by Sink the new value.dispose(): close the opened subject.counterObeservable(): return an Observable of the Subject, in other words, the object which will be used to notify the Widgets when changes happen in the Stream.Now that we have the BLoC class created lets see integrating it with the UI.We changed some few things in the UI:Now we initialize the CounterBloc with initialCount = 0.Then we removed the increment and decrement methods. Those method implementations are not the responsibility of UI anymore.When the both FloatingActionButton is clicked, it calls the correspondent method in the CounterBloc.Now we use StreamBuilder to show our data on the screen. We called StreamBuilder passing as Stream our counterObservable method available by the CounterBloc class, and we call the builder which must deal with the data which comes from the Strem and return the appropriate Widget.At this moment our well-structured app will look like this:App runningNotes and conclusions:Thats it, guys. Theres a lot of alternatives to structure your Flutter app and patterns to help with state management like BLoC, Redux, ScopedModel, and others. I confess BLoC is my favorite, but tell me if you liked it too. Thank you for reading the article so far, and please let your feedback. Tell me if you want part 2 with a more complex example.Theres my social network: LinkedIn, GitHub, Twitter. (Feel free to contact me).References:Updated (November 2020)There are many way to use Stream in Flutter and also many way to write the same code.In this article we will see three different way to write a Counter App using Streams, RxDart and Flutter Hooks.Table of Contents What is a StreamStreams are just a sequence of data that flows in a asynchronous way, in Flutter are widely use by many Widgets to listen for new data/changes to then rebuild part of the widget tree.The most common Streams are:Stream: just listen for new data;StreamController: allow to both listen and add/emit data to a streamWhat are Flutter Hooks?In short, Flutter Hooks are basically a simplify version of StatefulWidget, they will handle the lifecycle of an Object inside the build method of a simple StatelessWidget.If you are familiar with React Native it will just take seconds to get started with Flutter Hooks, if you are not, don't worry, the concept is simple, but you still need to be careful when using it to avoid unnecessary rebuild of the widget tree.Flutter hooks package already provide a full list of reusable hooks and creating Custom Hooks is so a simple task that can easily create new ones that fit our need.In our case, we need a custom Hook, but first let's see some code should we?Use Streams with HooksLet's start from an example, the classic counter app, but implemented using Streams.Pure FlutterFirst we use a Pure Flutter approach, no third party library needed.with_pure_flutter.dartsclass CounterApp extends StatefulWidget { const CounterApp({Key key}) : super(key: key); @override _CounterAppState createState() => _CounterAppState();}class _CounterAppState extends State { StreamController controller; int count = 0; @override void initState() { super.initState(); controller = StreamController.broadcast(); } @override void dispose() { super.dispose(); controller.dispose(); } @override Widget build(BuildContext context) { return Scaffold(appBar: AppBar(title: Text('Counter App'),), body: GestureDetector(onTap: () => controller.add(count++), child: StreamBuilder(stream: controller.stream, initialData: 0, builder: (context, snapshot) => Text('You tapped me ${snapshot.data} times.'),),),); }}We need to use a StatefulWidget because we need to safely dispose the StreamController, we don't want any memory leaks.We also need a count variable to store the current count since we can't access the current StreamController value inside the onTap function.Some people will be "fine" with this is code, it works, is not that verbose, but we can do better.Using Flutter HooksAmong the Flutter hooks example there is similar counter app that uses Stream and StreamController hooks, it also use shared_preferences to store the counter value, but we don't need that for this example, so the simplified code will be like this:with_flutter_hooks.dartimport 'package:flutter_hooks/flutter_hooks.dart';class CounterApp extends HookWidget { const CounterApp({Key key}) : super(key: key); @override Widget build(BuildContext context) { final controller = useStreamController(); return Scaffold(appBar: AppBar(title: Text('Counter App'),), body: HookBuilder(builder: (context) { final count = useStream(controller.stream); return GestureDetector(onTap: () => controller.add(count.data + 1), child: Text('You tapped me ${count.data} times.'),); }),); }}We simplified a bit, we don't need to dispose the StreamController, Flutter Hooks will handle this for us.We also don't need a count variable, we can use the useStream, but we need to use HookBuilder widget to avoid rebuilding the whole app when the stream changes.Can we do even better? let's try 8)Enhanced Streams with RxDartThere is another type of Stream, provided by the RxDart package called BehaviorSubject. If you are familiar with the Reactive Extensions for Async Programming you might already know if not, this is a Stream with Memory, once subscribed, it will emit the previous last value.It also provide a way to access the current value of the stream.In order to use it in a HookWidget, we need to create Custom Widgetuse_behavior_stream_controller_hook.dartimport 'package:rxdart/rxdart.dart';import 'package:flutter_hooks/flutter_hooks.dart';BehaviorSubject useBehaviorStreamController({bool sync = false, VoidCallback onListen, VoidCallback onCancel, List keys}) { return use(_BehaviorStreamControllerHook(onCancel: onCancel, onListen: onListen, sync: sync, keys: keys,));}class _BehaviorStreamControllerHook extends Hook { const _BehaviorStreamControllerHook({this.sync = false, this.onListen, this.onCancel, List keys}) : super(keys: keys); final bool sync; final VoidCallback onListen; final VoidCallback onCancel; @override _BehaviorStreamControllerHookState createState() => _BehaviorStreamControllerHookState();}class _BehaviorStreamControllerHookState extends HookState { BehaviorSubject _controller; @override void initHook() { super.initHook(); _controller = BehaviorSubject(sync: hook.sync, onCancel: hook.onCancel, onListen: hook.onListen,); } @override void didUpdateHook(_BehaviorStreamControllerHook oldHook) { super.didUpdateHook(oldHook); if (oldHook.onListen != hook.onListen) { _controller.onListen = hook.onListen; } if (oldHook.onCancel != hook.onCancel) { _controller.onCancel = hook.onCancel; } } @override BehaviorSubject build(BuildContext context) { return _controller; } @override void dispose() { _controller.close(); } @override String get debugLabel => 'useBehaviorStreamController';}We create a new hooks called useBehaviorStreamController, the code is exactly like the useStreamController hooks, but it use the RxDart BehaviorSubject instead of the Flutter StreamController.The final optimized resultWith is useBehaviorStreamController hooks, we can write our counter app in a more compact way:with_behavior_stream_controller.dartimport 'package:flutter_hooks/flutter_hooks.dart';class CounterApp extends HookWidget { const CounterApp({Key key}) : super(key: key); @override Widget build(BuildContext context) { final controller = useBehaviorStreamController(); return Scaffold(appBar: AppBar(title: Text('Counter App'),), body: GestureDetector(onTap: () => controller.add(controller.value + 1), child: StreamBuilder(stream: controller.stream, initialData: 0, builder: (context, snapshot) => Text('You tapped me ${snapshot.data} times.'),),),); }}We are using a StreamBuilder, but this time in the onTap function we can directly access the current value of the stream.With this last optimization our code now is much shorter, we don't need the HookBuilder and we will just rebuild the Text widget using a StreamBuilder.Additional ResourcesRxDartFlutter HooksStreamController Ayoub Ali Posted on May 30, 2023 Reactive programming is a popular paradigm that enables developers to build highly responsive and scalable applications. When combined with the Flutter framework, it empowers developers to create dynamic and reactive user interfaces. One of the most powerful libraries for reactive programming in Flutter is RxDart. In this blog post, we will explore the fundamentals of reactive programming and demonstrate how RxDart can be leveraged to enhance your Flutter applications. Outline Core Concepts of Reactive Programming At its core, reactive programming revolves around three fundamental concepts: Observables:Observables represent a sequence of values that can change over time. They can emit data or events, and other parts of the application can subscribe to these observables to be notified when new values or events are available.Streams:Streams are a type of observables in reactive programming. They provide a continuous flow of data or events over time. Developers can listen to streams and react to the data or events emitted by them.Data Flow:Reactive programming encourages a unidirectional data flow, where changes in the observables trigger updates in dependent components or operations. This ensures that the application remains responsive and efficiently handles changes without causing unnecessary side effects. By leveraging these core concepts, reactive programming enables developers to build applications that can react to user interactions, data updates, and other events in a scalable and efficient manner. It promotes a more declarative and event-driven style of programming, making it easier to handle complex asynchronous operations and maintain a responsive user interface. RxDart Installation and Setup To install RxDart and set it up in your Flutter project, follow these steps: Open your Flutter project in an IDE or text editor.Open the pubspec.yaml file located in the root directory of your Flutter project.In the dependencies section of the pubspec.yaml file, add the following line:dependencies: rxdart: ^0.27.1 This line specifies that your project will depend on the RxDart library and uses version 0.27.1 (or the latest version available). Save the pubspec.yaml file.In your IDE or terminal, run the following command to fetch and install the RxDart library: This command will download the RxDart library and make it available for use in your Flutter project. Once the installation is complete, you can start using RxDart in your Flutter code. Import the RxDart package in the relevant files where you intend to use it:import 'package:rxdart/rxdart.dart'; You are now ready to utilize RxDart and its reactive programming capabilities in your Flutter project. Refer to the RxDart documentation and examples to learn about different observables, streams, and operators provided by the library. Remember to import the necessary classes from the RxDart package whenever you need to use them in your code. Note: Make sure to follow the Flutter and Dart version compatibility requirements specified by the RxDart library. Key concept of Observable, Stream, StreamController and Subjects in RxDart Key Concepts in RxDart: Observable: Observables represent a stream of data or events that can change over time.They allow you to emit values or events and enable other parts of the application to subscribe and react to those emissions.Observables can be created from various sources such as lists, futures, or streams. Example: // Creating an Observable from a List final numbers = Observable.fromIterable([1, 2, 3, 4, 5]); // Subscribing to the Observable final subscription = numbers.listen((number) { print('Received number: $number'); }); // Output: Received number: 1, Received number: 2, ... Stream and StreamController: A stream represents a sequence of asynchronous events. It is a core concept in Dart's async programming model.StreamController acts as a source of events for a stream. It allows you to add events to the stream and control its flow.Streams provide a way to handle asynchronous data and enable listening to events emitted by the stream. Example: // Creating a StreamController final controller = StreamController(); // Adding events to the stream controller.add(1); controller.add(2); controller.add(3); // Listening to the stream final subscription = controller.stream.listen((event) { print('Received event: $event'); }); // Output: Received event: 1, Received event: 2, ... Subjects: Subjects are a type of Observable and StreamController combined. They can act as both a source of events and a stream to listen to those events.RxDart provides different types of subjects, such as BehaviorSubject, PublishSubject, and ReplaySubject, each with unique characteristics.Subjects are often used for managing state and broadcasting events within reactive programming. Example: // Creating a BehaviorSubject final subject = BehaviorSubject(); // Subscribing to the BehaviorSubject final subscription = subject.listen((value) { print('Received value: $value'); }); // Emitting values through the BehaviorSubject subject.add(1); subject.add(2); subject.add(3); // Output: Received value: 1, Received value: 2, ... Reactive event handling Combining streams and observables in RxDart 1. Reactive Event Handling: Reactive event handling refers to the ability of RxDart to handle and react to events in a reactive and efficient manner. It allows you to listen to events from various sources, such as user interactions, network responses, or timer events, and perform actions based on those events. RxDart provides operators and techniques to handle events reactively, enabling you to build responsive and dynamic applications. Example of Reactive Event Handling:// Creating an Observable for button pressesfinal buttonPresses = Observable(controller.stream); // Subscribing to button presses and reacting to the eventsfinal subscription = buttonPresses.listen((event) { print('Button pressed!');}); // Simulating button presses by adding events to the streamcontroller.add(true);controller.add(false); // Output: Button pressed!, Button pressed! 2. Combining Streams and Observables: Combining streams and observables in RxDart allows you to merge, combine, or transform multiple streams or observables into a single stream or observable. This capability is useful when you need to handle multiple data sources or perform complex operations on data emitted by different streams or observables. Example of Combining Streams and Observables:// Creating two streamsfinal stream1 = Stream.fromIterable([1, 2, 3]);final stream2 = Stream.fromIterable([4, 5, 6]); // Combining the streams into a single streamfinal combinedStream = Rx.concat([stream1, stream2]); // Subscribing to the combined stream and reacting to the eventsfinal subscription = combinedStream.listen((event) { print('Combined event: $event');}); // Output: Combined event: 1, Combined event: 2, ..., Combined event: 6 In this example, the concat operator from RxDart combines two streams into a single stream, merging their events in the order they occur. The resulting combined stream emits events from both stream1 and stream2. Advanced Techniques and Best PracticesThrottling and Debouncing: Throttling and debouncing are techniques used to control the rate at which events are emitted.Throttling limits the number of events emitted within a specified time interval.Debouncing delays emitting events until a specified quiet period occurs, discarding any previous events within that period. Example of Throttling: // Creating an Observable from button presses final buttonPresses = Observable(controller.stream); // Throttling the button presses to emit at most one event per 500 milliseconds final throttledButtonPresses = buttonPresses.throttleTime(Duration(milliseconds: 500)); // Subscribing to the throttled button presses final subscription = throttledButtonPresses.listen((event) { print('Button pressed!'); }); // Simulating multiple button presses for (int i = 0; i < 10; i++) { controller.add(true); await Future.delayed(Duration(milliseconds: 100)); } // Output: Button pressed! Example of Debouncing: // Creating an Observable from search queries final searchQueries = Observable(controller.stream); // Debouncing the search queries to emit events only after 500 milliseconds of quiet period final debouncedSearchQueries = searchQueries.debounceTime(Duration(milliseconds: 500)); // Subscribing to the debounced search queries final subscription = debouncedSearchQueries.listen((query) { print('Search query: $query'); // Perform search operation here }); // Simulating search queries controller.add('flutter'); await Future.delayed(Duration(milliseconds: 200)); controller.add('rx'); await Future.delayed(Duration(milliseconds: 200)); controller.add('dart'); await Future.delayed(Duration(milliseconds: 800)); // Output: Search query: dart Error Handling and Retries: RxDart provides operators to handle errors emitted by observables or streams.Error-handling operators like onErrorResumeNext or catchError allow you to handle errors gracefully and provide fallback mechanisms.You can also implement retry mechanisms using operators like retry or retryWhen to automatically retry failed operations. Example of Error Handling and Retries: // Creating an Observable from a network request final request = Observable.fromFuture(fetchDataFromNetwork()); // Handling errors and providing a fallback value final response = request.onErrorResumeNext(Observable.just('Fallback response')); // Subscribing to the response final subscription = response.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }); // Output: Received data: Fallback response (in case of error) Memory Management and Resource Disposal: It is essential to manage resources and dispose of subscriptions and subjects properly to avoid memory leaks.Use the takeUntil or takeWhile operators to automatically dispose of subscriptions when certain conditions are met.Dispose of subscriptions and subjects explicitly using the subscription.cancel() or subject.close() methods when they are no longer needed. Example of Memory Management and Resource Disposal: // Creating an Observable from a timer final timer = Observable(Stream.periodic(Duration(seconds: 1), (value) => value)); // Subscribing to the timer and automatically disposing of the subscription after 5 seconds final subscription = timer.takeUntil(Observable.timer(null, Duration(seconds: 5))).listen((value) { print('Timer value: $value'); }); // Output: Timer value: 0, Timer value: 1, Timer value: 2, Timer value: 3, Timer value: 4 // Disposing of the subscription explicitly after it is no longer needed subscription.cancel(); Testing and Debugging with RxDart Testing and debugging are crucial aspects of any software development process. Here's how you can approach testing and debugging with RxDart, along with an example: Testing with RxDart: RxDart provides various utilities and techniques to test observables, streams, and operators.Use the TestWidgetsFlutterBinding.ensureInitialized() method to initialize the test environment before running RxDart tests.Utilize the TestStream class from the rxdart/testing.dart package to create testable streams and observables.Use test-specific operators like materialize() and dematerialize() to convert events into notifications that can be easily asserted. Example of Testing with RxDart: import 'package:rxdart/rxdart.dart'; import 'package:rxdart/testing.dart'; import 'package:test/test.dart'; void main() { test('Test observable emits correct values', () { // Initialize the test environment TestWidgetsFlutterBinding.ensureInitialized(); // Create a TestStream final stream = TestStream(); // Emit values to the stream stream.emit(1); stream.emit(2); stream.emit(3); stream.close(); // Create an observable from the TestStream final observable = Observable(stream); // Assert the emitted values expect(observable, emitsInOrder([1, 2, 3])); }); } Debugging with RxDart: RxDart provides debugging operators that help analyze and debug observables and streams during development.The doOnData() operator allows you to inspect each emitted data item, enabling you to log or perform other debugging operations.The doOnError() and doOnDone() operators allow you to handle error and completion events respectively for debugging purposes. Example of Debugging with RxDart: // Creating an observable from a list final observable = Observable.fromIterable([1, 2, 3, 4, 5]); // Adding the doOnData operator for debugging final debugObservable = observable.doOnData((data) { print('Data: $data'); }); // Subscribing to the debugObservable final subscription = debugObservable.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }, onDone: () { print('Stream completed'); }); // Output: // Data: 1 // Received data: 1 // Data: 2 // Received data: 2 // Data: 3 // Received data: 3 // Data: 4 // Received data: 4 // Data: 5 // Received data: 5 // Stream completed By applying testing and debugging techniques, you can ensure the correctness and reliability of your RxDart code. Test your observables and streams using the provided testing utilities and leverage debugging operators to gain insights into the behavior of your reactive code during development and troubleshooting processes. Real-World Example In this example we will build a fully functional app that search a world from JSON API. Step -1 In first step we will generate a model for our JSON API.Use can use QuikeType.io to generate it just paste the JSON schema and you have the model @immutableclass Words { final List names; const Words({ required this.names, }); Words copyWith({ List? names, }) => Words(names: names ?? this.names,); factory Words.fromJson(Map json) => Words(names: List.from(json["names"].map((x) => x)),); Map toJson() => { "names": List.from(names.map((x) => x)), };} This code provides a way to convert Words objects to JSON and vice versa, making it easy to serialize and deserialize the data for communication or storage purposes. Step -2 Let's build a heraricary that focuses on creating of classes representing different search result states. Each class represents a specific state, such as loading, no result, error, or with a successful result.import 'package:flutter/foundation.dart' show immutable; @immutableabstract class SearchResult { const SearchResult();} @immutableclass SearchResultLoading implements SearchResult { const SearchResultLoading();} @immutableclass SearchResultNoResult implements SearchResult { const SearchResultNoResult();} @immutableclass SearchResultWithError implements SearchResult { final Object? error; const SearchResultWithError(this.error);} @immutableclass SearchResultWithResult implements SearchResult { final List result; const SearchResultWithResult(this.result);} Step - 3 Let's Write code that demonstrates and performs a search operation on a list of words fetched from an API. This code demonstrates the basic steps involved in performing a search operation using a remote API, caching the results, and extracting the matching words based on a search term.import 'dart:convert'; import 'package:http/http.dart' as http; class Api { List? _words; Api(); // Step - 3 Future search(String searchTerm) async { final term = searchTerm.trim().toLowerCase(); final cachedResult = _extractWordsUsingSearchTerm(searchTerm); if (cachedResult != null) { return cachedResult; } // api calling final words = await _getData(" _words = words; return _extractWordsUsingSearchTerm(term) ?? []; } // Step - 2 List? _extractWordsUsingSearchTerm(String word) { final cachedWords = _words; if (cachedWords != null) { List result = []; for (final worded in cachedWords) { if (worded.contains(word.trim().toLowerCase())) { result.add(worded); } } return result; } else { return null; } } // Step - 1 // Future _getData(String url) => HttpClient() // .getUrl(Uri.parse(url)) // .then((request) => request.close()) // .then((response) => response.transform(utf8.decoder).join()) // .then((jsonString) => json.decode(jsonString) as List); Future _getData(String url) async { final response = await http.Client().get(Uri.parse(url)); final parsed = jsonDecode(response.body)['names']; List? names = parsed != null ? List.from(parsed) : null; return names; }} // work only on String not listextension TrimmedCaseInsensitiveContain on String { bool trimmedContains(String other) => trim().toLowerCase().contains(other.trim().toLowerCase(),);} Step - 4 This code demonstrates the setup of a reactive search bloc using RxDart. It establishes a bidirectional communication channel for searching by providing a sink to add search terms and a stream to receive search results. The search terms undergo stream transformations to control the search behavior, and the results are emitted as a stream of SearchResult objects with different states. The result stream is created by chaining several stream transformations on textChanges. It performs the following operations: - distinct() ensures that only distinct search terms are processed. - debounceTime(const Duration(milliseconds: 350)) delays the processing of the search term stream, allowing a brief duration (350 milliseconds) of inactivity before emitting the latest search term. This helps to reduce unnecessary API calls for rapidly changing search terms. - switchMap((String searchTerm) maps each search term to a stream of SearchResult objects based on the search operation. If the search term is empty, it immediately emits a null result. Otherwise, it performs the actual search operation using the api.search method, which returns a Future>. This future is wrapped using Rx.fromCallable and then delayed by 1 second using delay to introduce a delay before emitting the result. The mapped stream is further transformed using map to convert the search results into appropriate SearchResult objects (SearchResultNoResult, SearchResultWithResult, or SearchResultLoading) based on the conditions. - startWith(const SearchResultLoading()) emits a loading result as the initial value when the search begins. - onErrorReturnWith((error, _) => SearchResultWithError(error)) handles any errors that occur during the search operation and emits a SearchResultWithError object.import 'dart:async'; import 'package:flutter/foundation.dart' show immutable;import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:rxdart/rxdart.dart'; @immutableclass SearchBloc { final Sink search; final Stream results; void dispose() { search.close(); } factory SearchBloc({required Api api}) { final textChanges = BehaviorSubject(); final result = textChanges .distinct() .debounceTime(const Duration(milliseconds: 350)) .switchMap((String searchTerm) { if (searchTerm.isEmpty) { return Stream.value(null); } else { return Rx.fromCallable(() => api.search(searchTerm)) .delay(const Duration(seconds: 1)) .map((results) => results.isEmpty ? const SearchResultNoResult() : SearchResultWithResult(results),) .startWith(const SearchResultLoading()) .onErrorReturnWith((error, _) => SearchResultWithError(error)); } }); return SearchBloc._(search: textChanges.sink, results: result,); } const SearchBloc._({ required this.search, required this.results, });} Step - 5 UI Development Custom widget to display out search result in GridViewimport 'package:flutter/material.dart'; class GridViewWidget extends StatelessWidget { const GridViewWidget({ Key? key, required this.results, }) : super(key: key); final List results; @override Widget build(BuildContext context) { return Expanded(child: GridView.builder(itemCount: results.length, gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent: 200, childAspectRatio: 3 / 2, crossAxisSpacing: 20, mainAxisSpacing: 20), itemBuilder: (context, index) { return Container(alignment: Alignment.center, decoration: BoxDecoration(color: Colors.white, borderRadius: BorderRadius.circular(15)), child: Text(results[index], textAlign: TextAlign.center, style: const TextStyle(fontSize: 20, fontWeight: FontWeight.bold),),); },),); }} Step - 6 This code provides a UI representation of the different states of the search results and handles the appropriate rendering based on the received data.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:infinite_words/widgets/grid_view_widget.dart'; class SearchResultView extends StatelessWidget { final Stream searchResults; const SearchResultView({ Key? key, required this.searchResults, }) : super(key: key); @override Widget build(BuildContext context) { return StreamBuilder(stream: searchResults, builder: (BuildContext context, AsyncSnapshot snapshot,) { if (snapshot.hasData) { final result = snapshot.data; if (result is SearchResultWithError) { return const Center(child: Text('Error')); } else if (result is SearchResultLoading) { return const Center(child: CircularProgressIndicator()); } else if (result is SearchResultNoResult) { return const Center(child: Text('No Result Found')); } else if (result is SearchResultWithResult) { final results = result.result; return GridViewWidget(results: results); } else { return const Center(child: Text("Unknown State")); } } else { return const Center(child: Text("Waiting.....", style: TextStyle(color: Colors.white, fontSize: 18),),); } },); }} Step -7 This code sets up the home page of the application with a text input field for searching words and displays the search results using the SearchResultView widget. The SearchBloc manages the search functionality and emits the search results to the UI.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_bloc.dart';import 'package:infinite_words/view/search_result_view.dart'; class HomePage extends StatefulWidget { const HomePage({Key? key}) : super(key: key); @override State createState() => _HomePageState();} class _HomePageState extends State { late final SearchBloc _bloc; @override void initState() { _bloc = SearchBloc(api: Api()); super.initState(); } @override void dispose() { super.dispose(); _bloc.dispose(); } @override Widget build(BuildContext context) { return Scaffold(backgroundColor: Colors.black, // appBar: AppBar(// title: const Text('Search'), //), body: Padding(padding: const EdgeInsets.all(10), child: Column(children: [const SizedBox(height: 50,), TextField(decoration: InputDecoration(border: InputBorder.none, filled: true, fillColor: Colors.white, contentPadding: const EdgeInsets.only(left: 14.0, bottom: 6.0, top: 8.0), focusedBorder: OutlineInputBorder(borderSide: const BorderSide(color: Colors.grey), borderRadius: BorderRadius.circular(10.0),), enabledBorder: UnderlineInputBorder(borderSide: const BorderSide(color: Colors.white), borderRadius: BorderRadius.circular(10.0),), hintText: "Write a word", hintStyle: const TextStyle(fontSize: 20, // color: Colors.white,)), onChanged: _bloc.search.add), const SizedBox(height: 10,), SearchResultView(searchResults: _bloc.results)],),),); }} SourceCode Ayoub Ali Posted on May 30, 2023 Reactive programming is a popular paradigm that enables developers to build highly responsive and scalable applications. When combined with the Flutter framework, it empowers developers to create dynamic and reactive user interfaces. One of the most powerful libraries for reactive programming in Flutter is RxDart. In this blog post, we will explore the fundamentals of reactive programming and demonstrate how RxDart can be leveraged to enhance your Flutter applications. Outline Core Concepts of Reactive Programming At its core, reactive programming revolves around three fundamental concepts: Observables:Observables represent a sequence of values that can change over time. They can emit data or events, and other parts of the application can subscribe to these observables to be notified when new values or events are available.Streams:Streams are a type of observables in reactive programming. They provide a continuous flow of data or events over time. Developers can listen to streams and react to the data or events emitted by them.Data Flow:Reactive programming encourages a unidirectional data flow, where changes in the observables trigger updates in dependent components or operations. This ensures that the application remains responsive and efficiently handles changes without causing unnecessary side effects. By leveraging these core concepts, reactive programming enables developers to build applications that can react to user interactions, data updates, and other events in a scalable and efficient manner. It promotes a more declarative and event-driven style of programming, making it easier to handle complex asynchronous operations and maintain a responsive user interface. RxDart Installation and Setup To install RxDart and set it up in your Flutter project, follow these steps: Open your Flutter project in an IDE or text editor.Open the pubspec.yaml file located in the root directory of your Flutter project.In the dependencies section of the pubspec.yaml file, add the following line:dependencies: rxdart: ^0.27.1 This line specifies that your project will depend on the RxDart library and uses version 0.27.1 (or the latest version available). Save the pubspec.yaml file.In your IDE or terminal, run the following command to fetch and install the RxDart library: This command will download the RxDart library and make it available for use in your Flutter project. Once the installation is complete, you can start using RxDart in your Flutter code. Import the RxDart package in the relevant files where you intend to use it:import 'package:rxdart/rxdart.dart'; You are now ready to utilize RxDart and its reactive programming capabilities in your Flutter project. Refer to the RxDart documentation and examples to learn about different observables, streams, and operators provided by the library. Remember to import the necessary classes from the RxDart package whenever you need to use them in your code. Note: Make sure to follow the Flutter and Dart version compatibility requirements specified by the RxDart library. Key concept of Observable, Stream, StreamController and Subjects in RxDart Key Concepts in RxDart: Observable: Observables represent a stream of data or events that can change over time.They allow you to emit values or events and enable other parts of the application to subscribe and react to those emissions.Observables can be created from various sources such as lists, futures, or streams. Example: // Creating an Observable from a List final numbers = Observable.fromIterable([1, 2, 3, 4, 5]); // Subscribing to the Observable final subscription = numbers.listen((number) { print('Received number: $number'); }); // Output: Received number: 1, Received number: 2, ... Stream and StreamController: A stream represents a sequence of asynchronous events. It is a core concept in Dart's async programming model.StreamController acts as a source of events for a stream. It allows you to add events to the stream and control its flow.Streams provide a way to handle asynchronous data and enable listening to events emitted by the stream. Example: // Creating a StreamController final controller = StreamController(); // Adding events to the stream controller.add(1); controller.add(2); controller.add(3); // Listening to the stream final subscription = controller.stream.listen((event) { print('Received event: $event'); }); // Output: Received event: 1, Received event: 2, ... Subjects: Subjects are a type of Observable and StreamController combined. They can act as both a source of events and a stream to listen to those events.RxDart provides different types of subjects, such as BehaviorSubject, PublishSubject, and ReplaySubject, each with unique characteristics.Subjects are often used for managing state and broadcasting events within reactive programming. Example: // Creating a BehaviorSubject final subject = BehaviorSubject(); // Subscribing to the BehaviorSubject final subscription = subject.listen((value) { print('Received value: $value'); }); // Emitting values through the BehaviorSubject subject.add(1); subject.add(2); subject.add(3); // Output: Received value: 1, Received value: 2, ... Reactive event handling Combining streams and observables in RxDart 1. Reactive Event Handling: Reactive event handling refers to the ability of RxDart to handle and react to events in a reactive and efficient manner. It allows you to listen to events from various sources, such as user interactions, network responses, or timer events, and perform actions based on those events. RxDart provides operators and techniques to handle events reactively, enabling you to build responsive and dynamic applications. Example of Reactive Event Handling:// Creating an Observable for button pressesfinal buttonPresses = Observable(controller.stream); // Subscribing to button presses and reacting to the eventsfinal subscription = buttonPresses.listen((event) { print('Button pressed!');}); // Simulating button presses by adding events to the streamcontroller.add(true);controller.add(false); // Output: Button pressed!, Button pressed! 2. Combining Streams and Observables: Combining streams and observables in RxDart allows you to merge, combine, or transform multiple streams or observables into a single stream or observable. This capability is useful when you need to handle multiple data sources or perform complex operations on data emitted by different streams or observables. Example of Combining Streams and Observables:// Creating two streamsfinal stream1 = Stream.fromIterable([1, 2, 3]);final stream2 = Stream.fromIterable([4, 5, 6]); // Combining the streams into a single streamfinal combinedStream = Rx.concat([stream1, stream2]); // Subscribing to the combined stream and reacting to the eventsfinal subscription = combinedStream.listen((event) { print('Combined event: $event');}); // Output: Combined event: 1, Combined event: 2, ..., Combined event: 6 In this example, the concat operator from RxDart combines two streams into a single stream, merging their events in the order they occur. The resulting combined stream emits events from both stream1 and stream2. Advanced Techniques and Best PracticesThrottling and Debouncing: Throttling and debouncing are techniques used to control the rate at which events are emitted.Throttling limits the number of events emitted within a specified time interval.Debouncing delays emitting events until a specified quiet period occurs, discarding any previous events within that period. Example of Throttling: // Creating an Observable from button presses final buttonPresses = Observable(controller.stream); // Throttling the button presses to emit at most one event per 500 milliseconds final throttledButtonPresses = buttonPresses.throttleTime(Duration(milliseconds: 500)); // Subscribing to the throttled button presses final subscription = throttledButtonPresses.listen((event) { print('Button pressed!'); }); // Simulating multiple button presses for (int i = 0; i < 10; i++) { controller.add(true); await Future.delayed(Duration(milliseconds: 100)); } // Output: Button pressed! Example of Debouncing: // Creating an Observable from search queries final searchQueries = Observable(controller.stream); // Debouncing the search queries to emit events only after 500 milliseconds of quiet period final debouncedSearchQueries = searchQueries.debounceTime(Duration(milliseconds: 500)); // Subscribing to the debounced search queries final subscription = debouncedSearchQueries.listen((query) { print('Search query: $query'); // Perform search operation here }); // Simulating search queries controller.add('flutter'); await Future.delayed(Duration(milliseconds: 200)); controller.add('rx'); await Future.delayed(Duration(milliseconds: 200)); controller.add('dart'); await Future.delayed(Duration(milliseconds: 800)); // Output: Search query: dart Error Handling and Retries: RxDart provides operators to handle errors emitted by observables or streams.Error-handling operators like onErrorResumeNext or catchError allow you to handle errors gracefully and provide fallback mechanisms.You can also implement retry mechanisms using operators like retry or retryWhen to automatically retry failed operations. Example of Error Handling and Retries: // Creating an Observable from a network request final request = Observable.fromFuture(fetchDataFromNetwork()); // Handling errors and providing a fallback value final response = request.onErrorResumeNext(Observable.just('Fallback response')); // Subscribing to the response final subscription = response.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }); // Output: Received data: Fallback response (in case of error) Memory Management and Resource Disposal: It is essential to manage resources and dispose of subscriptions and subjects properly to avoid memory leaks.Use the takeUntil or takeWhile operators to automatically dispose of subscriptions when certain conditions are met.Dispose of subscriptions and subjects explicitly using the subscription.cancel() or subject.close() methods when they are no longer needed. Example of Memory Management and Resource Disposal: // Creating an Observable from a timer final timer = Observable(Stream.periodic(Duration(seconds: 1), (value) => value)); // Subscribing to the timer and automatically disposing of the subscription after 5 seconds final subscription = timer.takeUntil(Observable.timer(null, Duration(seconds: 5))).listen((value) { print('Timer value: $value'); }); // Output: Timer value: 0, Timer value: 1, Timer value: 2, Timer value: 3, Timer value: 4 // Disposing of the subscription explicitly after it is no longer needed subscription.cancel(); Testing and Debugging with RxDart Testing and debugging are crucial aspects of any software development process. Here's how you can approach testing and debugging with RxDart, along with an example: Testing with RxDart: RxDart provides various utilities and techniques to test observables, streams, and operators.Use the TestWidgetsFlutterBinding.ensureInitialized() method to initialize the test environment before running RxDart tests.Utilize the TestStream class from the rxdart/testing.dart package to create testable streams and observables.Use test-specific operators like materialize() and dematerialize() to convert events into notifications that can be easily asserted. Example of Testing with RxDart: import 'package:rxdart/rxdart.dart'; import 'package:rxdart/testing.dart'; import 'package:test/test.dart'; void main() { test('Test observable emits correct values', () { // Initialize the test environment TestWidgetsFlutterBinding.ensureInitialized(); // Create a TestStream final stream = TestStream(); // Emit values to the stream stream.emit(1); stream.emit(2); stream.emit(3); stream.close(); // Create an observable from the TestStream final observable = Observable(stream); // Assert the emitted values expect(observable, emitsInOrder([1, 2, 3])); }); } Debugging with RxDart: RxDart provides debugging operators that help analyze and debug observables and streams during development.The doOnData() operator allows you to inspect each emitted data item, enabling you to log or perform other debugging operations.The doOnError() and doOnDone() operators allow you to handle error and completion events respectively for debugging purposes. Example of Debugging with RxDart: // Creating an observable from a list final observable = Observable.fromIterable([1, 2, 3, 4, 5]); // Adding the doOnData operator for debugging final debugObservable = observable.doOnData((data) { print('Data: $data'); }); // Subscribing to the debugObservable final subscription = debugObservable.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }, onDone: () { print('Stream completed'); }); // Output: // Data: 1 // Received data: 1 // Data: 2 // Received data: 2 // Data: 3 // Received data: 3 // Data: 4 // Received data: 4 // Data: 5 // Received data: 5 // Stream completed By applying testing and debugging techniques, you can ensure the correctness and reliability of your RxDart code. Test your observables and streams using the provided testing utilities and leverage debugging operators to gain insights into the behavior of your reactive code during development and troubleshooting processes. Real-World Example In this example we will build a fully functional app that search a world from JSON API. Step -1 In first step we will generate a model for our JSON API.Use can use QuikeType.io to generate it just paste the JSON schema and you have the model @immutableclass Words { final List names; const Words({ required this.names, }); Words copyWith({ List? names, }) => Words(names: names ?? this.names,); factory Words.fromJson(Map json) => Words(names: List.from(json["names"].map((x) => x)),); Map toJson() => { "names": List.from(names.map((x) => x)), };} This code provides a way to convert Words objects to JSON and vice versa, making it easy to serialize and deserialize the data for communication or storage purposes. Step -2 Let's build a heraricary that focuses on creating of classes representing different search result states. Each class represents a specific state, such as loading, no result, error, or with a successful result.import 'package:flutter/foundation.dart' show immutable; @immutableabstract class SearchResult { const SearchResult();} @immutableclass SearchResultLoading implements SearchResult { const SearchResultLoading();} @immutableclass SearchResultNoResult implements SearchResult { const SearchResultNoResult();} @immutableclass SearchResultWithError implements SearchResult { final Object? error; const SearchResultWithError(this.error);} @immutableclass SearchResultWithResult implements SearchResult { final List result; const SearchResultWithResult(this.result);} Step - 3 Let's Write code that demonstrates and performs a search operation on a list of words fetched from an API. This code demonstrates the basic steps involved in performing a search operation using a remote API, caching the results, and extracting the matching words based on a search term.import 'dart:convert'; import 'package:http/http.dart' as http; class Api { List? _words; Api(); // Step - 3 Future search(String searchTerm) async { final term = searchTerm.trim().toLowerCase(); final cachedResult = _extractWordsUsingSearchTerm(searchTerm); if (cachedResult != null) { return cachedResult; } // api calling final words = await _getData(" _words = words; return _extractWordsUsingSearchTerm(term) ?? []; } // Step - 2 List? _extractWordsUsingSearchTerm(String word) { final cachedWords = _words; if (cachedWords != null) { List result = []; for (final worded in cachedWords) { if (worded.contains(word.trim().toLowerCase())) { result.add(worded); } } return result; } else { return null; } } // Step - 1 // Future _getData(String url) => HttpClient() // .getUrl(Uri.parse(url)) // .then((request) => request.close()) // .then((response) => response.transform(utf8.decoder).join()) // .then((jsonString) => json.decode(jsonString) as List); Future _getData(String url) async { final response = await http.Client().get(Uri.parse(url)); final parsed = jsonDecode(response.body)['names']; List? names = parsed != null ? List.from(parsed) : null; return names; }} // work only on String not listextension TrimmedCaseInsensitiveContain on String { bool trimmedContains(String other) => trim().toLowerCase().contains(other.trim().toLowerCase(),);} Step - 4 This code demonstrates the setup of a reactive search bloc using RxDart. It establishes a bidirectional communication channel for searching by providing a sink to add search terms and a stream to receive search results. The search terms undergo stream transformations to control the search behavior, and the results are emitted as a stream of SearchResult objects with different states. The result stream is created by chaining several stream transformations on textChanges. It performs the following operations: - distinct() ensures that only distinct search terms are processed. - debounceTime(const Duration(milliseconds: 350)) delays the processing of the search term stream, allowing a brief duration (350 milliseconds) of inactivity before emitting the latest search term. This helps to reduce unnecessary API calls for rapidly changing search terms. - switchMap((String searchTerm) maps each search term to a stream of SearchResult objects based on the search operation. If the search term is empty, it immediately emits a null result. Otherwise, it performs the actual search operation using the api.search method, which returns a Future>. This future is wrapped using Rx.fromCallable and then delayed by 1 second using delay to introduce a delay before emitting the result. The mapped stream is further transformed using map to convert the search results into appropriate SearchResult objects (SearchResultNoResult, SearchResultWithResult, or SearchResultLoading) based on the conditions. - startWith(const SearchResultLoading()) emits a loading result as the initial value when the search begins. - onErrorReturnWith((error, _) => SearchResultWithError(error)) handles any errors that occur during the search operation and emits a SearchResultWithError object.import 'dart:async'; import 'package:flutter/foundation.dart' show immutable;import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:rxdart/rxdart.dart'; @immutableclass SearchBloc { final Sink search; final Stream results; void dispose() { search.close(); } factory SearchBloc({required Api api}) { final textChanges = BehaviorSubject(); final result = textChanges .distinct() .debounceTime(const Duration(milliseconds: 350)) .switchMap((String searchTerm) { if (searchTerm.isEmpty) { return Stream.value(null); } else { return Rx.fromCallable(() => api.search(searchTerm)) .delay(const Duration(seconds: 1)) .map((results) => results.isEmpty ? const SearchResultNoResult() : SearchResultWithResult(results),) .startWith(const SearchResultLoading()) .onErrorReturnWith((error, _) => SearchResultWithError(error)); } }); return SearchBloc._(search: textChanges.sink, results: result,); } const SearchBloc._({ required this.search, required this.results, });} Step - 5 UI Development Custom widget to display out search result in GridViewimport 'package:flutter/material.dart'; class GridViewWidget extends StatelessWidget { const GridViewWidget({ Key? key, required this.results, }) : super(key: key); final List results; @override Widget build(BuildContext context) { return Expanded(child: GridView.builder(itemCount: results.length, gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent: 200, childAspectRatio: 3 / 2, crossAxisSpacing: 20, mainAxisSpacing: 20), itemBuilder: (context, index) { return Container(alignment: Alignment.center, decoration: BoxDecoration(color: Colors.white, borderRadius: BorderRadius.circular(15)), child: Text(results[index], textAlign: TextAlign.center, style: const TextStyle(fontSize: 20, fontWeight: FontWeight.bold),),); },),); }} Step - 6 This code provides a UI representation of the different states of the search results and handles the appropriate rendering based on the received data.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:infinite_words/widgets/grid_view_widget.dart'; class SearchResultView extends StatelessWidget { final Stream searchResults; const SearchResultView({ Key? key, required this.searchResults, }) : super(key: key); @override Widget build(BuildContext context) { return StreamBuilder(stream: searchResults, builder: (BuildContext context, AsyncSnapshot snapshot,) { if (snapshot.hasData) { final result = snapshot.data; if (result is SearchResultWithError) { return const Center(child: Text('Error')); } else if (result is SearchResultLoading) { return const Center(child: CircularProgressIndicator()); } else if (result is SearchResultNoResult) { return const Center(child: Text('No Result Found')); } else if (result is SearchResultWithResult) { final results = result.result; return GridViewWidget(results: results); } else { return const Center(child: Text("Unknown State")); } } else { return const Center(child: Text("Waiting.....", style: TextStyle(color: Colors.white, fontSize: 18),),); } },); }} Step -7 This code sets up the home page of the application with a text input field for searching words and displays the search results using the SearchResultView widget. The SearchBloc manages the search functionality and emits the search results to the UI.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_bloc.dart';import 'package:infinite_words/view/search_result_view.dart'; class HomePage extends StatefulWidget { const HomePage({Key? key}) : super(key: key); @override State createState() => _HomePageState();} class _HomePageState extends State { late final SearchBloc _bloc; @override void initState() { _bloc = SearchBloc(api: Api()); super.initState(); } @override void dispose() { super.dispose(); _bloc.dispose(); } @override Widget build(BuildContext context) { return Scaffold(backgroundColor: Colors.black, // appBar: AppBar(// title: const Text('Search'), //), body: Padding(padding: const EdgeInsets.all(10), child: Column(children: [const SizedBox(height: 50,), TextField(decoration: InputDecoration(border: InputBorder.none, filled: true, fillColor: Colors.white, contentPadding: const EdgeInsets.only(left: 14.0, bottom: 6.0, top: 8.0), focusedBorder: OutlineInputBorder(borderSide: const BorderSide(color: Colors.grey), borderRadius: BorderRadius.circular(10.0),), enabledBorder: UnderlineInputBorder(borderSide: const BorderSide(color: Colors.white), borderRadius: BorderRadius.circular(10.0),), hintText: "Write a word", hintStyle: const TextStyle(fontSize: 20, // color: Colors.white,)), onChanged: _bloc.search.add), const SizedBox(height: 10,), SearchResultView(searchResults: _bloc.results)],),),); }} SourceCode Ayoub Ali Posted on May 30, 2023 Reactive programming is a popular paradigm that enables developers to build highly responsive and scalable applications. When combined with the Flutter framework, it empowers developers to create dynamic and reactive user interfaces. One of the most powerful libraries for reactive programming in Flutter is RxDart. In this blog post, we will explore the fundamentals of reactive programming and demonstrate how RxDart can be leveraged to enhance your Flutter applications. Outline Core Concepts of Reactive Programming At its core, reactive programming revolves around three fundamental concepts: Observables:Observables represent a sequence of values that can change over time. They can emit data or events, and other parts of the application can subscribe to these observables to be notified when new values or events are available.Streams:Streams are a type of observables in reactive programming. They provide a continuous flow of data or events over time. Developers can listen to streams and react to the data or events emitted by them.Data Flow:Reactive programming encourages a unidirectional data flow, where changes in the observables trigger updates in dependent components or operations. This ensures that the application remains responsive and efficiently handles changes without causing unnecessary side effects. By leveraging these core concepts, reactive programming enables developers to build applications that can react to user interactions, data updates,

and other events in a scalable and efficient manner. It promotes a more declarative and event-driven style of programming, making it easier to handle complex asynchronous operations and maintain a responsive user interface. RxDart Installation and Setup To install RxDart and set it up in your Flutter project, follow these steps: Open your Flutter project in an IDE or text editor.Open the pubspec.yaml file located in the root directory of your Flutter project.In the dependencies section of the pubspec.yaml file, add the following line:dependencies: rxdart: ^0.27.1 This line specifies that your project will depend on the RxDart library and uses version 0.27.1 (or the latest version available). Save the pubspec.yaml file.In your IDE or terminal, run the following command to fetch and install the RxDart library: This command will download the RxDart library and make it available for use in your Flutter project. Once the installation is complete, you can start using RxDart in your Flutter code. Import the RxDart package in the relevant files where you intend to use it:import 'package:rxdart/rxdart.dart'; You are now ready to utilize RxDart and its reactive programming capabilities in your Flutter project. Refer to the RxDart documentation and examples to learn about different observables, streams, and operators provided by the library. Remember to import the necessary classes from the RxDart package whenever you need to use them in your code. Note: Make sure to follow the Flutter and Dart version compatibility requirements specified by the RxDart library. Key concept of Observable, Stream, StreamController and Subjects in RxDart Key Concepts in RxDart: Observable: Observables represent a stream of data or events that can change over time.They allow you to emit values or events and enable other parts of the application to subscribe and react to those emissions.Observables can be created from various sources such as lists, futures, or streams. Example: // Creating an Observable from a List final numbers = Observable.fromIterable([1, 2, 3, 4, 5]); // Subscribing to the Observable final subscription = numbers.listen((number) { print('Received number: $number'); }); // Output: Received number: 1, Received number: 2, ... Stream and StreamController: A stream represents a sequence of asynchronous events. It is a core concept in Dart's async programming model.StreamController acts as a source of events for a stream. It allows you to add events to the stream and control its flow.Streams provide a way to handle asynchronous data and enable listening to events emitted by the stream. Example: // Creating a StreamController final controller = StreamController(); // Adding events to the stream controller.add(1); controller.add(2); controller.add(3); // Listening to the stream final subscription = controller.stream.listen((event) { print('Received event: $event'); }); // Output: Received event: 1, Received event: 2, ... Subjects: Subjects are a type of Observable and StreamController combined. They can act as both a source of events and a stream to listen to those events.RxDart provides different types of subjects, such as BehaviorSubject, PublishSubject, and ReplaySubject, each with unique characteristics.Subjects are often used for managing state and broadcasting events within reactive programming. Example: // Creating a BehaviorSubject final subject = BehaviorSubject(); // Subscribing to the BehaviorSubject final subscription = subject.listen((value) { print('Received value: $value'); }); // Emitting values through the BehaviorSubject subject.add(1); subject.add(2); subject.add(3); // Output: Received value: 1, Received value: 2, ... Reactive event handling Combining streams and observables in RxDart 1. Reactive Event Handling: Reactive event handling refers to the ability of RxDart to handle and react to events in a reactive and efficient manner. It allows you to listen to events from various sources, such as user interactions, network responses, or timer events, and perform actions based on those events. RxDart provides operators and techniques to handle events reactively, enabling you to build responsive and dynamic applications. Example of Reactive Event Handling:// Creating an Observable for button pressesfinal buttonPresses = Observable(controller.stream); // Subscribing to button presses and reacting to the eventsfinal subscription = buttonPresses.listen((event) { print('Button pressed!');}); // Simulating button presses by adding events to the streamcontroller.add(true);controller.add(false); // Output: Button pressed!, Button pressed! 2. Combining Streams and Observables: Combining streams and observables in RxDart allows you to merge, combine, or transform multiple streams or observables into a single stream or observable. This capability is useful when you need to handle multiple data sources or perform complex operations on data emitted by different streams or observables. Example of Combining Streams and Observables:// Creating two streamsfinal stream1 = Stream.fromIterable([1, 2, 3]);final stream2 = Stream.fromIterable([4, 5, 6]); // Combining the streams into a single streamfinal combinedStream = Rx.concat([stream1, stream2]); // Subscribing to the combined stream and reacting to the eventsfinal subscription = combinedStream.listen((event) { print('Combined event: $event');}); // Output: Combined event: 1, Combined event: 2, ..., Combined event: 6 In this example, the concat operator from RxDart combines two streams into a single stream, merging their events in the order they occur. The resulting combined stream emits events from both stream1 and stream2. Advanced Techniques and Best PracticesThrottling and Debouncing: Throttling and debouncing are techniques used to control the rate at which events are emitted.Throttling limits the number of events emitted within a specified time interval.Debouncing delays emitting events until a specified quiet period occurs, discarding any previous events within that period. Example of Throttling: // Creating an Observable from button presses final buttonPresses = Observable(controller.stream); // Throttling the button presses to emit at most one event per 500 milliseconds final throttledButtonPresses = buttonPresses.throttleTime(Duration(milliseconds: 500)); // Subscribing to the throttled button presses final subscription = throttledButtonPresses.listen((event) { print('Button pressed!'); }); // Simulating multiple button presses for (int i = 0; i < 10; i++) { controller.add(true); await Future.delayed(Duration(milliseconds: 100)); } // Output: Button pressed! Example of Debouncing: // Creating an Observable from search queries final searchQueries = Observable(controller.stream); // Debouncing the search queries to emit events only after 500 milliseconds of quiet period final debouncedSearchQueries = searchQueries.debounceTime(Duration(milliseconds: 500)); // Subscribing to the debounced search queries final subscription = debouncedSearchQueries.listen((query) { print('Search query: $query'); // Perform search operation here }); // Simulating search queries controller.add('flutter'); await Future.delayed(Duration(milliseconds: 200)); controller.add('rx'); await Future.delayed(Duration(milliseconds: 200)); controller.add('dart'); await Future.delayed(Duration(milliseconds: 800)); // Output: Search query: dart Error Handling and Retries: RxDart provides operators to handle errors emitted by observables or streams.Error-handling operators like onErrorResumeNext or catchError allow you to handle errors gracefully and provide fallback mechanisms.You can also implement retry mechanisms using operators like retry or retryWhen to automatically retry failed operations. Example of Error Handling and Retries: // Creating an Observable from a network request final request = Observable.fromFuture(fetchDataFromNetwork()); // Handling errors and providing a fallback value final response = request.onErrorResumeNext(Observable.just('Fallback response')); // Subscribing to the response final subscription = response.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }); // Output: Received data: Fallback response (in case of error) Memory Management and Resource Disposal: It is essential to manage resources and dispose of subscriptions and subjects properly to avoid memory leaks.Use the takeUntil or takeWhile operators to automatically dispose of subscriptions when certain conditions are met.Dispose of subscriptions and subjects explicitly using the subscription.cancel() or subject.close() methods when they are no longer needed. Example of Memory Management and Resource Disposal: // Creating an Observable from a timer final timer = Observable(Stream.periodic(Duration(seconds: 1), (value) => value)); // Subscribing to the timer and automatically disposing of the subscription after 5 seconds final subscription = timer.takeUntil(Observable.timer(null, Duration(seconds: 5))).listen((value) { print('Timer value: $value'); }); // Output: Timer value: 0, Timer value: 1, Timer value: 2, Timer value: 3, Timer value: 4 // Disposing of the subscription explicitly after it is no longer needed subscription.cancel(); Testing and Debugging with RxDart Testing and debugging are crucial aspects of any software development process. Here's how you can approach testing and debugging with RxDart, along with an example: Testing with RxDart: RxDart provides various utilities and techniques to test observables, streams, and operators.Use the TestWidgetsFlutterBinding.ensureInitialized() method to initialize the test environment before running RxDart tests.Utilize the TestStream class from the rxdart/testing.dart package to create testable streams and observables.Use test-specific operators like materialize() and dematerialize() to convert events into notifications that can be easily asserted. Example of Testing with RxDart: import 'package:rxdart/rxdart.dart'; import 'package:rxdart/testing.dart'; import 'package:test/test.dart'; void main() { test('Test observable emits correct values', () { // Initialize the test environment TestWidgetsFlutterBinding.ensureInitialized(); // Create a TestStream final stream = TestStream(); // Emit values to the stream stream.emit(1); stream.emit(2); stream.emit(3); stream.close(); // Create an observable from the TestStream final observable = Observable(stream); // Assert the emitted values expect(observable, emitsInOrder([1, 2, 3])); }); } Debugging with RxDart: RxDart provides debugging operators that help analyze and debug observables and streams during development.The doOnData() operator allows you to inspect each emitted data item, enabling you to log or perform other debugging operations.The doOnError() and doOnDone() operators allow you to handle error and completion events respectively for debugging purposes. Example of Debugging with RxDart: // Creating an observable from a list final observable = Observable.fromIterable([1, 2, 3, 4, 5]); // Adding the doOnData operator for debugging final debugObservable = observable.doOnData((data) { print('Data: $data'); }); // Subscribing to the debugObservable final subscription = debugObservable.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }, onDone: () { print('Stream completed'); }); // Output: // Data: 1 // Received data: 1 // Data: 2 // Received data: 2 // Data: 3 // Received data: 3 // Data: 4 // Received data: 4 // Data: 5 // Received data: 5 // Stream completed By applying testing and debugging techniques, you can ensure the correctness and reliability of your RxDart code. Test your observables and streams using the provided testing utilities and leverage debugging operators to gain insights into the behavior of your reactive code during development and troubleshooting processes. Real-World Example In this example we will build a fully functional app that search a world from JSON API. Step -1 In first step we will generate a model for our JSON API.Use can use QuikeType.io to generate it just paste the JSON schema and you have the model @immutableclass Words { final List names; const Words({ required this.names, }); Words copyWith({ List? names, }) => Words(names: names ?? this.names,); factory Words.fromJson(Map json) => Words(names: List.from(json["names"].map((x) => x)),); Map toJson() => { "names": List.from(names.map((x) => x)), };} This code provides a way to convert Words objects to JSON and vice versa, making it easy to serialize and deserialize the data for communication or storage purposes. Step -2 Let's build a heraricary that focuses on creating of classes representing different search result states. Each class represents a specific state, such as loading, no result, error, or with a successful result.import 'package:flutter/foundation.dart' show immutable; @immutableabstract class SearchResult { const SearchResult();} @immutableclass SearchResultLoading implements SearchResult { const SearchResultLoading();} @immutableclass SearchResultNoResult implements SearchResult { const SearchResultNoResult();} @immutableclass SearchResultWithError implements SearchResult { final Object? error; const SearchResultWithError(this.error);} @immutableclass SearchResultWithResult implements SearchResult { final List result; const SearchResultWithResult(this.result);} Step - 3 Let's Write code that demonstrates and performs a search operation on a list of words fetched from an API. This code demonstrates the basic steps involved in performing a search operation using a remote API, caching the results, and extracting the matching words based on a search term.import 'dart:convert'; import 'package:http/http.dart' as http; class Api { List? _words; Api(); // Step - 3 Future search(String searchTerm) async { final term = searchTerm.trim().toLowerCase(); final cachedResult = _extractWordsUsingSearchTerm(searchTerm); if (cachedResult != null) { return cachedResult; } // api calling final words = await _getData(" _words = words; return _extractWordsUsingSearchTerm(term) ?? []; } // Step - 2 List? _extractWordsUsingSearchTerm(String word) { final cachedWords = _words; if (cachedWords != null) { List result = []; for (final worded in cachedWords) { if (worded.contains(word.trim().toLowerCase())) { result.add(worded); } } return result; } else { return null; } } // Step - 1 // Future _getData(String url) => HttpClient() // .getUrl(Uri.parse(url)) // .then((request) => request.close()) // .then((response) => response.transform(utf8.decoder).join()) // .then((jsonString) => json.decode(jsonString) as List); Future _getData(String url) async { final response = await http.Client().get(Uri.parse(url)); final parsed = jsonDecode(response.body)['names']; List? names = parsed != null ? List.from(parsed) : null; return names; }} // work only on String not listextension TrimmedCaseInsensitiveContain on String { bool trimmedContains(String other) => trim().toLowerCase().contains(other.trim().toLowerCase(),);} Step - 4 This code demonstrates the setup of a reactive search bloc using RxDart. It establishes a bidirectional communication channel for searching by providing a sink to add search terms and a stream to receive search results. The search terms undergo stream transformations to control the search behavior, and the results are emitted as a stream of SearchResult objects with different states. The result stream is created by chaining several stream transformations on textChanges. It performs the following operations: - distinct() ensures that only distinct search terms are processed. - debounceTime(const Duration(milliseconds: 350)) delays the processing of the search term stream, allowing a brief duration (350 milliseconds) of inactivity before emitting the latest search term. This helps to reduce unnecessary API calls for rapidly changing search terms. - switchMap((String searchTerm) maps each search term to a stream of SearchResult objects based on the search operation. If the search term is empty, it immediately emits a null result. Otherwise, it performs the actual search operation using the api.search method, which returns a Future>. This future is wrapped using Rx.fromCallable and then delayed by 1 second using delay to introduce a delay before emitting the result. The mapped stream is further transformed using map to convert the search results into appropriate SearchResult objects (SearchResultNoResult, SearchResultWithResult, or SearchResultLoading) based on the conditions. - startWith(const SearchResultLoading()) emits a loading result as the initial value when the search begins. - onErrorReturnWith((error, _) => SearchResultWithError(error)) handles any errors that occur during the search operation and emits a SearchResultWithError object.import 'dart:async'; import 'package:flutter/foundation.dart' show immutable;import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:rxdart/rxdart.dart'; @immutableclass SearchBloc { final Sink search; final Stream results; void dispose() { search.close(); } factory SearchBloc({required Api api}) { final textChanges = BehaviorSubject(); final result = textChanges .distinct() .debounceTime(const Duration(milliseconds: 350)) .switchMap((String searchTerm) { if (searchTerm.isEmpty) { return Stream.value(null); } else { return Rx.fromCallable(() => api.search(searchTerm)) .delay(const Duration(seconds: 1)) .map((results) => results.isEmpty ? const SearchResultNoResult() : SearchResultWithResult(results),) .startWith(const SearchResultLoading()) .onErrorReturnWith((error, _) => SearchResultWithError(error)); } }); return SearchBloc._(search: textChanges.sink, results: result,); } const SearchBloc._({ required this.search, required this.results, });} Step - 5 UI Development Custom widget to display out search result in GridViewimport 'package:flutter/material.dart'; class GridViewWidget extends StatelessWidget { const GridViewWidget({ Key? key, required this.results, }) : super(key: key); final List results; @override Widget build(BuildContext context) { return Expanded(child: GridView.builder(itemCount: results.length, gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent: 200, childAspectRatio: 3 / 2, crossAxisSpacing: 20, mainAxisSpacing: 20), itemBuilder: (context, index) { return Container(alignment: Alignment.center, decoration: BoxDecoration(color: Colors.white, borderRadius: BorderRadius.circular(15)), child: Text(results[index], textAlign: TextAlign.center, style: const TextStyle(fontSize: 20, fontWeight: FontWeight.bold),),); },),); }} Step - 6 This code provides a UI representation of the different states of the search results and handles the appropriate rendering based on the received data.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:infinite_words/widgets/grid_view_widget.dart'; class SearchResultView extends StatelessWidget { final Stream searchResults; const SearchResultView({ Key? key, required this.searchResults, }) : super(key: key); @override Widget build(BuildContext context) { return StreamBuilder(stream: searchResults, builder: (BuildContext context, AsyncSnapshot snapshot,) { if (snapshot.hasData) { final result = snapshot.data; if (result is SearchResultWithError) { return const Center(child: Text('Error')); } else if (result is SearchResultLoading) { return const Center(child: CircularProgressIndicator()); } else if (result is SearchResultNoResult) { return const Center(child: Text('No Result Found')); } else if (result is SearchResultWithResult) { final results = result.result; return GridViewWidget(results: results); } else { return const Center(child: Text("Unknown State")); } } else { return const Center(child: Text("Waiting.....", style: TextStyle(color: Colors.white, fontSize: 18),),); } },); }} Step -7 This code sets up the home page of the application with a text input field for searching words and displays the search results using the SearchResultView widget. The SearchBloc manages the search functionality and emits the search results to the UI.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_bloc.dart';import 'package:infinite_words/view/search_result_view.dart'; class HomePage extends StatefulWidget { const HomePage({Key? key}) : super(key: key); @override State createState() => _HomePageState();} class _HomePageState extends State { late final SearchBloc _bloc; @override void initState() { _bloc = SearchBloc(api: Api()); super.initState(); } @override void dispose() { super.dispose(); _bloc.dispose(); } @override Widget build(BuildContext context) { return Scaffold(backgroundColor: Colors.black, // appBar: AppBar(// title: const Text('Search'), //), body: Padding(padding: const EdgeInsets.all(10), child: Column(children: [const SizedBox(height: 50,), TextField(decoration: InputDecoration(border: InputBorder.none, filled: true, fillColor: Colors.white, contentPadding: const EdgeInsets.only(left: 14.0, bottom: 6.0, top: 8.0), focusedBorder: OutlineInputBorder(borderSide: const BorderSide(color: Colors.grey), borderRadius: BorderRadius.circular(10.0),), enabledBorder: UnderlineInputBorder(borderSide: const BorderSide(color: Colors.white), borderRadius: BorderRadius.circular(10.0),), hintText: "Write a word", hintStyle: const TextStyle(fontSize: 20, // color: Colors.white,)), onChanged: _bloc.search.add), const SizedBox(height: 10,), SearchResultView(searchResults: _bloc.results)],),),); }} SourceCode Ayoub Ali Posted on May 30, 2023 Reactive programming is a popular paradigm that enables developers to build highly responsive and scalable applications. When combined with the Flutter framework, it empowers developers to create dynamic and reactive user interfaces. One of the most powerful libraries for reactive programming in Flutter is RxDart. In this blog post, we will explore the fundamentals of reactive programming and demonstrate how RxDart can be leveraged to enhance your Flutter applications. Outline Core Concepts of Reactive Programming At its core, reactive programming revolves around three fundamental concepts: Observables:Observables represent a sequence of values that can change over time. They can emit data or events, and other parts of the application can subscribe to these observables to be notified when new values or events are available.Streams:Streams are a type of observables in reactive programming. They provide a continuous flow of data or events over time. Developers can listen to streams and react to the data or events emitted by them.Data Flow:Reactive programming encourages a unidirectional data flow, where changes in the observables trigger updates in dependent components or operations. This ensures that the application remains responsive and efficiently handles changes without causing unnecessary side effects. By leveraging these core concepts, reactive programming enables developers to build applications that can react to user interactions, data updates, and other events in a scalable and efficient manner. It promotes a more declarative and event-driven style of programming, making it easier to handle complex asynchronous operations and maintain a responsive user interface. RxDart Installation and Setup To install RxDart and set it up in your Flutter project, follow these steps: Open your Flutter project in an IDE or text editor.Open the pubspec.yaml file located in the root directory of your Flutter project.In the dependencies section of the pubspec.yaml file, add the following line:dependencies: rxdart: ^0.27.1 This line specifies that your project will depend on the RxDart library and uses version 0.27.1 (or the latest version available). Save the pubspec.yaml file.In your IDE or terminal, run the following command to fetch and install the RxDart library: This command will download the RxDart library and make it available for use in your Flutter project. Once the installation is complete, you can start using RxDart in your Flutter code. Import the RxDart package in the relevant files where you intend to use it:import 'package:rxdart/rxdart.dart'; You are now ready to utilize RxDart and its reactive programming capabilities in your Flutter project. Refer to the RxDart documentation and examples to learn about different observables, streams, and operators provided by the library. Remember to import the necessary classes from the RxDart package whenever you need to use them in your code. Note: Make sure to follow the Flutter and Dart version compatibility requirements specified by the RxDart library. Key concept of Observable, Stream, StreamController and Subjects in RxDart Key Concepts in RxDart: Observable: Observables represent a stream of data or events that can change over time.They allow you to emit values or events and enable other parts of the application to subscribe and react to those emissions.Observables can be created from various sources such as lists, futures, or streams. Example: // Creating an Observable from a List final numbers = Observable.fromIterable([1, 2, 3, 4, 5]); // Subscribing to the Observable final subscription = numbers.listen((number) { print('Received number: $number'); }); // Output: Received number: 1, Received number: 2, ... Stream and StreamController: A stream represents a sequence of asynchronous events. It is a core concept in Dart's async programming model.StreamController acts as a source of events for a stream. It allows you to add events to the stream and control its flow.Streams provide a way to handle asynchronous data and enable listening to events emitted by the stream. Example: // Creating a StreamController final controller = StreamController(); // Adding events to the stream controller.add(1); controller.add(2); controller.add(3); // Listening to the stream final subscription = controller.stream.listen((event) { print('Received event: $event'); }); // Output: Received event: 1, Received event: 2, ... Subjects: Subjects are a type of Observable and StreamController combined. They can act as both a source of events and a stream to listen to those events.RxDart provides different types of subjects, such as BehaviorSubject, PublishSubject, and ReplaySubject, each with unique characteristics.Subjects are often used for managing state and broadcasting events within reactive programming. Example: // Creating a BehaviorSubject final subject = BehaviorSubject(); // Subscribing to the BehaviorSubject final subscription = subject.listen((value) { print('Received value: $value'); }); // Emitting values through the BehaviorSubject subject.add(1); subject.add(2); subject.add(3); // Output: Received value: 1, Received value: 2, ... Reactive event handling Combining streams and observables in RxDart 1. Reactive Event Handling: Reactive event handling refers to the ability of RxDart to handle and react to events in a reactive and efficient manner. It allows you to listen to events from various sources, such as user interactions, network responses, or timer events, and perform actions based on those events. RxDart provides operators and techniques to handle events reactively, enabling you to build responsive and dynamic applications. Example of Reactive Event Handling:// Creating an Observable for button pressesfinal buttonPresses = Observable(controller.stream); // Subscribing to button presses and reacting to the eventsfinal subscription = buttonPresses.listen((event) { print('Button pressed!');}); // Simulating button presses by adding events to the streamcontroller.add(true);controller.add(false); // Output: Button pressed!, Button pressed! 2. Combining Streams and Observables: Combining streams and observables in RxDart allows you to merge, combine, or transform multiple streams or observables into a single stream or observable. This capability is useful when you need to handle multiple data sources or perform complex operations on data emitted by different streams or observables. Example of Combining Streams and Observables:// Creating two streamsfinal stream1 = Stream.fromIterable([1, 2, 3]);final stream2 = Stream.fromIterable([4, 5, 6]); // Combining the streams into a single streamfinal combinedStream = Rx.concat([stream1, stream2]); // Subscribing to the combined stream and reacting to the eventsfinal subscription = combinedStream.listen((event) { print('Combined event: $event');}); // Output: Combined event: 1, Combined event: 2, ..., Combined event: 6 In this example, the concat operator from RxDart combines two streams into a single stream, merging their events in the order they occur. The resulting combined stream emits events from both stream1 and stream2. Advanced Techniques and Best PracticesThrottling and Debouncing: Throttling and debouncing are techniques used to control the rate at which events are emitted.Throttling limits the number of events emitted within a specified time interval.Debouncing delays emitting events until a specified quiet period occurs, discarding any previous events within that period. Example of Throttling: // Creating an Observable from button presses final buttonPresses = Observable(controller.stream); // Throttling the button presses to emit at most one event per 500 milliseconds final throttledButtonPresses = buttonPresses.throttleTime(Duration(milliseconds: 500)); // Subscribing to the throttled button presses final subscription = throttledButtonPresses.listen((event) { print('Button pressed!'); }); // Simulating multiple button presses for (int i = 0; i < 10; i++) { controller.add(true); await Future.delayed(Duration(milliseconds: 100)); } // Output: Button pressed! Example of Debouncing: // Creating an Observable from search queries final searchQueries = Observable(controller.stream); // Debouncing the search queries to emit events only after 500 milliseconds of quiet period final debouncedSearchQueries = searchQueries.debounceTime(Duration(milliseconds: 500)); // Subscribing to the debounced search queries final subscription = debouncedSearchQueries.listen((query) { print('Search query: $query'); // Perform search operation here }); // Simulating search queries controller.add('flutter'); await Future.delayed(Duration(milliseconds: 200)); controller.add('rx'); await Future.delayed(Duration(milliseconds: 200)); controller.add('dart'); await Future.delayed(Duration(milliseconds: 800)); // Output: Search query: dart Error Handling and Retries: RxDart provides operators to handle errors emitted by observables or streams.Error-handling operators like onErrorResumeNext or catchError allow you to handle errors gracefully and provide fallback mechanisms.You can also implement retry mechanisms using operators like retry or retryWhen to automatically retry failed operations. Example of Error Handling and Retries: // Creating an Observable from a network request final request = Observable.fromFuture(fetchDataFromNetwork()); // Handling errors and providing a fallback value final response = request.onErrorResumeNext(Observable.just('Fallback response')); // Subscribing to the response final subscription = response.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }); // Output: Received data: Fallback response (in case of error) Memory Management and Resource Disposal: It is essential to manage resources and dispose of subscriptions and subjects properly to avoid memory leaks.Use the takeUntil or takeWhile operators to automatically dispose of subscriptions when certain conditions are met.Dispose of subscriptions and subjects explicitly using the subscription.cancel() or subject.close() methods when they are no longer needed. Example of Memory Management and Resource Disposal: // Creating an Observable from a timer final timer = Observable(Stream.periodic(Duration(seconds: 1), (value) => value)); // Subscribing to the timer and automatically disposing of the subscription after 5 seconds final subscription = timer.takeUntil(Observable.timer(null, Duration(seconds: 5))).listen((value) { print('Timer value: $value'); }); // Output: Timer value: 0, Timer value: 1, Timer value: 2, Timer value: 3, Timer value: 4 // Disposing of the subscription explicitly after it is no longer needed subscription.cancel(); Testing and Debugging with RxDart Testing and debugging are crucial aspects of any software development process. Here's how you can approach testing and debugging with RxDart, along with an example: Testing with RxDart: RxDart provides various utilities and techniques to test observables, streams, and operators.Use the TestWidgetsFlutterBinding.ensureInitialized() method to initialize the test environment before running RxDart tests.Utilize the TestStream class from the rxdart/testing.dart package to create testable streams and observables.Use test-specific operators like materialize() and dematerialize() to convert events into notifications that can be easily asserted. Example of Testing with RxDart: import 'package:rxdart/rxdart.dart'; import 'package:rxdart/testing.dart'; import 'package:test/test.dart'; void main() { test('Test observable emits correct values', () { // Initialize the test environment TestWidgetsFlutterBinding.ensureInitialized(); // Create a TestStream final stream = TestStream(); // Emit values to the stream stream.emit(1); stream.emit(2); stream.emit(3); stream.close(); // Create an observable from the TestStream final observable = Observable(stream); // Assert the emitted values expect(observable, emitsInOrder([1, 2, 3])); }); } Debugging with RxDart: RxDart provides debugging operators that help analyze and debug observables and streams during development.The doOnData() operator allows you to inspect each emitted data item, enabling you to log or perform other debugging operations.The doOnError() and doOnDone() operators allow you to handle error and completion events respectively for debugging purposes. Example of Debugging with RxDart: // Creating an observable from a list final observable = Observable.fromIterable([1, 2, 3, 4, 5]); // Adding the doOnData operator for debugging final debugObservable = observable.doOnData((data) { print('Data: $data'); }); // Subscribing to the debugObservable final subscription = debugObservable.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }, onDone: () { print('Stream completed'); }); // Output: // Data: 1 // Received data: 1 // Data: 2 // Received data: 2 // Data: 3 // Received data: 3 // Data: 4 // Received data: 4 // Data: 5 // Received data: 5 // Stream completed By applying testing and debugging techniques, you can ensure the correctness and reliability of your RxDart code. Test your observables and streams using the provided testing utilities and leverage debugging operators to gain insights into the behavior of your reactive code during development and troubleshooting processes. Real-World Example In this example we will build a fully functional app that search a world from JSON API. Step -1 In first step we will generate a model for our JSON API.Use can use QuikeType.io to generate it just paste the JSON schema and you have the model @immutableclass Words { final List names; const Words({ required this.names, }); Words copyWith({ List? names, }) => Words(names: names ?? this.names,); factory Words.fromJson(Map json) => Words(names: List.from(json["names"].map((x) => x)),); Map toJson() => { "names": List.from(names.map((x) => x)), };} This code provides a way to convert Words objects to JSON and vice versa, making it easy to serialize and deserialize the data for communication or storage purposes. Step -2 Let's build a heraricary that focuses on creating of classes representing different search result states. Each class represents a specific state, such as loading, no result, error, or with a successful result.import 'package:flutter/foundation.dart' show immutable; @immutableabstract class SearchResult { const SearchResult();} @immutableclass SearchResultLoading implements SearchResult { const SearchResultLoading();} @immutableclass SearchResultNoResult implements SearchResult { const SearchResultNoResult();} @immutableclass SearchResultWithError implements SearchResult { final Object? error; const SearchResultWithError(this.error);} @immutableclass SearchResultWithResult implements SearchResult { final List result; const SearchResultWithResult(this.result);} Step - 3 Let's Write code that demonstrates and performs a search operation on a list of words fetched from an API. This code demonstrates the basic steps involved in performing a search operation using a remote API, caching the results, and extracting the matching words based on a search term.import 'dart:convert'; import 'package:http/http.dart' as http; class Api { List? _words; Api(); // Step - 3 Future search(String searchTerm) async { final term = searchTerm.trim().toLowerCase(); final cachedResult = _extractWordsUsingSearchTerm(searchTerm); if (cachedResult != null) { return cachedResult; } // api calling final words = await _getData(" _words = words; return _extractWordsUsingSearchTerm(term) ?? []; } // Step - 2 List? _extractWordsUsingSearchTerm(String word) { final cachedWords = _words; if (cachedWords != null) { List result = []; for (final worded in cachedWords) { if (worded.contains(word.trim().toLowerCase())) { result.add(worded); } } return result; } else { return null; } } // Step - 1 // Future _getData(String url) => HttpClient() // .getUrl(Uri.parse(url)) // .then((request) => request.close()) // .then((response) => response.transform(utf8.decoder).join()) // .then((jsonString) => json.decode(jsonString) as List); Future _getData(String url) async { final response = await http.Client().get(Uri.parse(url)); final parsed = jsonDecode(response.body)['names']; List? names = parsed != null ? List.from(parsed) : null; return names; }} // work only on String not listextension TrimmedCaseInsensitiveContain on String { bool trimmedContains(String other) => trim().toLowerCase().contains(other.trim().toLowerCase(),);} Step - 4 This code demonstrates the setup of a reactive search bloc using RxDart. It establishes a bidirectional communication channel for searching by providing a sink to add search terms and a stream to receive search results. The search terms undergo stream transformations to control the search behavior, and the results are emitted as a stream of SearchResult objects with different states. The result stream is created by chaining several stream transformations on textChanges. It performs the following operations: - distinct() ensures that only distinct search terms are processed. - debounceTime(const Duration(milliseconds: 350)) delays the processing of the search term stream, allowing a brief duration (350 milliseconds) of inactivity before emitting the latest search term. This helps to reduce unnecessary API calls for rapidly changing search terms. - switchMap((String searchTerm) maps each search term to a stream of SearchResult objects based on the search operation. If the search term is empty, it immediately emits a null result. Otherwise, it performs the actual search operation using the api.search method, which returns a Future>. This future is wrapped using Rx.fromCallable and then delayed by 1 second using delay to introduce a delay before emitting the result. The mapped stream is further transformed using map to convert the search results into appropriate SearchResult objects (SearchResultNoResult, SearchResultWithResult, or SearchResultLoading) based on the conditions. - startWith(const SearchResultLoading()) emits a loading result as the initial value when the search begins. - onErrorReturnWith((error, _) => SearchResultWithError(error)) handles any errors that occur during the search operation and emits a SearchResultWithError object.import 'dart:async'; import 'package:flutter/foundation.dart' show immutable;import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:rxdart/rxdart.dart'; @immutableclass SearchBloc { final Sink search; final Stream results; void dispose() { search.close(); } factory SearchBloc({required Api api}) { final textChanges = BehaviorSubject(); final result = textChanges .distinct() .debounceTime(const Duration(milliseconds: 350)) .switchMap((String searchTerm) { if (searchTerm.isEmpty) { return Stream.value(null); } else { return Rx.fromCallable(() => api.search(searchTerm)) .delay(const Duration(seconds: 1)) .map((results) => results.isEmpty ? const SearchResultNoResult() : SearchResultWithResult(results),) .startWith(const SearchResultLoading()) .onErrorReturnWith((error, _) => SearchResultWithError(error)); } }); return SearchBloc._(search: textChanges.sink, results: result,); } const SearchBloc._({ required this.search, required this.results, });} Step - 5 UI Development Custom widget to display out search result in GridViewimport 'package:flutter/material.dart'; class GridViewWidget extends StatelessWidget { const GridViewWidget({ Key? key, required this.results, }) : super(key: key); final List results; @override Widget build(BuildContext context) { return Expanded(child: GridView.builder(itemCount: results.length, gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent: 200, childAspectRatio: 3 / 2, crossAxisSpacing: 20, mainAxisSpacing: 20), itemBuilder: (context, index) { return Container(alignment: Alignment.center, decoration: BoxDecoration(color: Colors.white, borderRadius: BorderRadius.circular(15)), child: Text(results[index], textAlign: TextAlign.center, style: const TextStyle(fontSize: 20, fontWeight: FontWeight.bold),),); },),); }} Step - 6 This code provides a UI representation of the different states of the search results and handles the appropriate rendering based on the received data.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:infinite_words/widgets/grid_view_widget.dart'; class SearchResultView extends StatelessWidget { final Stream searchResults; const SearchResultView({ Key? key, required this.searchResults, }) : super(key: key); @override Widget build(BuildContext context) { return StreamBuilder(stream: searchResults, builder: (BuildContext context, AsyncSnapshot snapshot,) { if (snapshot.hasData) { final result = snapshot.data; if (result is SearchResultWithError) { return const Center(child: Text('Error')); } else if (result is SearchResultLoading) { return const Center(child: CircularProgressIndicator()); } else if (result is SearchResultNoResult) { return const Center(child: Text('No Result Found')); } else if (result is SearchResultWithResult) { final results = result.result; return GridViewWidget(results: results); } else { return const Center(child: Text("Unknown State")); } } else { return const Center(child: Text("Waiting.....", style: TextStyle(color: Colors.white, fontSize: 18),),); } },); }} Step -7 This code sets up the home page of the application with a text input field for searching words and displays the search results using the SearchResultView widget. The SearchBloc manages the search functionality and emits the search results to the UI.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_bloc.dart';import 'package:infinite_words/view/search_result_view.dart'; class HomePage extends StatefulWidget { const HomePage({Key? key}) : super(key: key); @override State createState() => _HomePageState();} class _HomePageState extends State { late final SearchBloc _bloc; @override void initState() { _bloc = SearchBloc(api: Api()); super.initState(); } @override void dispose() { super.dispose(); _bloc.dispose(); } @override Widget build(BuildContext context) { return Scaffold(backgroundColor: Colors.black, // appBar: AppBar(// title: const Text('Search'), //), body: Padding(padding: const EdgeInsets.all(10), child: Column(children: [const SizedBox(height: 50,), TextField(decoration: InputDecoration(border: InputBorder.none, filled: true, fillColor: Colors.white, contentPadding: const EdgeInsets.only(left: 14.0, bottom: 6.0, top: 8.0), focusedBorder: OutlineInputBorder(borderSide: const BorderSide(color: Colors.grey), borderRadius: BorderRadius.circular(10.0),), enabledBorder: UnderlineInputBorder(borderSide: const BorderSide(color: Colors.white), borderRadius: BorderRadius.circular(10.0),), hintText: "Write a word", hintStyle: const TextStyle(fontSize: 20, // color: Colors.white,)), onChanged: _bloc.search.add), const SizedBox(height: 10,), SearchResultView(searchResults: _bloc.results)],),),); }} SourceCodeDeveloping highly responsive, fault-tolerant, event-driven asynchronous applications that are scalable requires a different way of thinking from the traditional synchronous programming architecture.This is why the reactive programming principle exists. It uses streams to build these types of reactive applications.In this tutorial, youll use RxDart, a Dart implementation of the reactive programming principle, to develop Gravity Pop, a very simple game based off a very popular block falling videogame. In this game, blocks fall and disappear if a row is filled Along the way, youll learn:RxDart streams.Extension functions.Subjects.The concept of Backpressure. Getting StartedClick the Download Materials button at the top or bottom of this tutorial to download the starter project. The starter project has a few models to represent the various states/events youll be manipulating using RxDart. These are:GameState: An enum to store the game state: whether the game has started, is playing or is over.GameData: A model that represents the current state of the game and all the collected tetriminos.Input: A model representing change events that affect the current tetrimino in play. This includes xOffset, yOffset and angle.Piece: An enum that represents the type of block from the available seven.Tetrimino: represents the current block animating in the screen.Note: A group of falling blocks is also known as a tetrimino.Along with the models, here are some other things to note about the starter project:The project contains utility functions for common use cases.There are classes for the board, the player and a layer that only handles user interactions.We paint each tetrimino using the custom painter class.Now, get packages by running flutter pub get in your terminal/PowerShell. Then build and run to see the starter app:Tap the Play button to enter play mode. Youll see the screen below:Youll notice that it only contains a white screen; there are no blocks, and the buttons dont work. Youll change that and make this into a simple game using Reactive streams in RxDart.At the end, you will get to see a game like this:Are you ready? Time to get stacking.The Reactive Programming ParadigmAn event is a general term used to represent the press of a button or any of the many sensors in your device recording their data. Various components process events to produce something meaningful in the end, to make updates in the UI, for instance.But this process is not a one-time deal. As long as events are produced, the process repeats itself. A stream in programming is this flow of any raw input event to a useful result:Above is a basic illustration of a stream showing relation between a source and a sink.All streams have two required components: a source (for example, an event) and a sink or receiver (a logical or UI component that can consume that stream.) Reactive programming uses such streams as a backbone to build applications.Flutter offers built-in support for streams using the Stream class. This class handles creating streams from various sources such as I/O updates, sensor data capture, UI event capture and so much more.A sink can be any component that has the power to consume stream events. Moving forward, youll use StreamBuilder to consume streams and update UI.For your first task, open lib/player.dart and replace //TODO: add a stream builder with the following code snippet: //1child: StreamBuilder(//2 //TODO: replace with staticPlayerStream stream: _engine.blankPlayerStream, //3 builder: ((context, snapshot) { if (snapshot.hasData) { if (snapshot.data?.current == Piece.Empty) return const SizedBox.shrink(); return ClipRect(child: CustomPaint(painter: _getNextPiece(snapshot.data!, _engine.extent.toDouble()), child: SizedBox(width: _engine.effectiveWidth.toDouble(), height: _engine.effectiveHeight.toDouble(),),),); } return const SizedBox.shrink(); }),),Heres whats happening in the preceding code snippets:You used Flutter StreamBuilder as a sink to accept events from a stream, _engine.blankPlayerStream._engine.blankPlayerStream creates a stream that returns an empty stream at the moment. Youll expand on this as you get further into the tutorial.The snapshot variable references stream events and processes them. In this case, you use each event in a Text widget.Build and run the app and you will see the same white board. But now the building blocks for streams are in place with the source(stream instances) and sink(StreamBuilder widget). At this point, it may seem trivial, but as you go on you will implement more features of the reactive programming paradigm using RxDart.Creating RxDart StreamsDart Stream API fulfils the basic requirements to follow the reactive programming style. RxDart offers various new features that increase its usefulness in real-world applications to enhance it more.As an example of this, consider the Stream.fromIterable() constructor returned from gridStateStream in lib/engine.dart. As simple as it may be, you can improve it further by specifying the start and end of the iterable and turn each numbers in between into events.The RangeStream stream from RxDart fulfills this exact usecase. Open lib/player.dart and replace TODO: replace with staticPlayerStream and the line beneath it with the following code://TODO: replace with animatingPlayerStreamstream: _engine.staticPlayerStream(),Next, open lib/engine.dart and take a look at the implementation for staticPlayerStream:return RangeStream(0, effectiveHeight ~/ extent - 1).map((value) => Tetrimino(current: Piece.I, origin: Point(0, value * extent.toDouble())));RangeStream emits numbers within its input range as events. Each event is mapped into a tetrimino object in the above code snippet.Build and run the app and you will see that since RangeStream has completed emitting all its events, the block has come to rest at the bottom of the board which is represented by the final event emitted by RangeStream.Along with RangeStream, RxDart also offers various other types of pre-built streams. Youll see more of them as you continue with this tutorial.Using RxDart ExtensionsThe map function at the end of the code above is an example of an extension function. Extension functions extend the operations you can apply on streams. You can use them when you need to manipulate an event before it reaches the sink.For example, what if you only need odd events from a stream that emits integer values? Or, you need to create a gap between each event emitted in a stream.Thats what map does in the previous code. It takes each integer and transforms it into a tetrimino. These functions have the power to transform the emitted events in any shape or form and can even create other streams from the events.Note: A higher-order stream is a special stream that emits other streams.Lets make the previous example a bit more exciting using interval. The interval function emits events of the stream after the specified time interval. Open lib/player.dart and replace //TODO: replace with animatingPlayerStream and the code statement below it with the following://TODO: replace with animatingPlayerWithCompletionStreamstream: _engine.animatingPlayerStream(),Check staticPlayerStream in lib/engine.dart to see how you used the interval extension.return RangeStream(0, effectiveHeight ~/ extent - 1) .interval(const Duration(milliseconds: 500)) .map((value) => Tetrimino(current: Piece.I, origin: Point(0, value * extent.toDouble())));The only difference between this RangeStream and the previous stream is the addition of the interval extension that emits each event from its input stream in predefined intervals, 500 milliseconds in this case. Build and run the app now. You will now see the block fall in intervals of 500 ms. As you can see, youre not limited to just one extension function between the source and a sink. You can chain multiple extension functions to manipulate the data between the source and the sink. Chaining various functions one after the other is an important part of reactive programming, used to achieve high degrees of data manipulation.This short tutorial shows how to implement search with RxDart in Flutter.We will use the GitHub Search API, but the same concepts are valid for any other search REST APIs.Our goal is to have a good search user experience, without putting too much load on the server, or compromising bandwidth and battery life on the client.GitHub Search Example AppSo let's get started by looking at a simple working app.This uses the SearchDelegate class to show a list of users matching the input search query:In order to build this we need a few components:a GitHubSearchAPIWrapper class to pull the data from the GitHub REST APIa GitHubSearchResult model class that contains the API response dataa GitHubSearchDelegate class that shows the search UI with a grid of resultsa GitHubSearchService class with all the logic for wiring up the API wrapper with the UIHere's how everything is connected:This is a short tutorial, so we will focus only on the GitHubSearchService class.But you can check the full source code in GitHub for all the remaining details.GitHubSearchServiceLet's start with some code that represents a starting point for this class:class GitHubSearchService { GitHubSearchService({@required this.apiWrapper}); final GitHubSearchAPIWrapper apiWrapper; // Input stream (search terms) final _searchTerms = BehaviorSubject(); void searchUser(String query) => _searchTerms.add(query); // Output stream (search results) final _results = BehaviorSubject(); Stream get results => _results.stream; void dispose() { _results.close(); _searchTerms.close(); }}We have an input stream for the search terms (generated as we type on the search box), and an output stream for the result data (shown as a grid of items in the UI).This class takes a GitHubSearchAPIWrapper as a constructor argument.And we need to work out how to add values to the output stream when the input changes.Mapping inputs to outputs: first attemptAs a first attempt, let's add a listener to the _searchTerms stream:GitHubSearchService({@required this.apiWrapper}) { _searchTerms.listen((query) async { print('searching: $query'); // get new result from the api final result = await apiWrapper.searchUser(query); print('received result for: $query'); // add to the output stream _results.add(result); });}Every time the search query changes, this code calls the API and adds the result to the output stream.While this approach seems logical, it doesn't work very well in practice.Why? Because there is no guarantee that the results come back in the same order as the search terms.This is a problem when we have an unreliable connection, and can lead to our-of-order results and a bad user experience:// example logsearching: bsearching: bisearching: bizsearching: bizzsearching: bizz8searching: bizz84received result for: breceived result for: bireceived result for: bizzreceived result for: bizz8received result for: bizz84received result for: bizasyncMapLet's try using the asyncMap operator instead:GitHubSearchService({@required this.apiWrapper}) { _results = _searchTerms.asyncMap((query) async { print('searching: $query'); return await apiWrapper.searchUser(query); }); // discard previous events}// To make this work, `_results` is now decleared as a `Stream` Stream _results;asyncMap guarantees that the output events are emitted in the same order as the inputs.This solution solves the out-of-order problem, but it has one major drawback.If one of the output arrives late, then all the subsequent outputs are delayed too.Instead, we should discard any in-flight requests as soon as the search query changes.This can be done with the switchMap operator.switchMapswitchMap does the right thing for us, and can be used like this:GitHubSearchService({@required this.apiWrapper}) { _results = _searchTerms.switchMap((query) async* { print('searching: $query'); yield await apiWrapper.searchUser(query); }); // discard previous events}Note that in this case we're using a stream generator with the async* syntax.With this setup, we can discard any in-flight requests as soon as a new search term comes in.But there is still one problem that is particularly noticeable with the GitHub Search API.If we submit too many queries too quickly, we get a "rate limit exceeded" error:{ "message": "API rate limit exceeded for 82.37.171.3. (But here's the good news: Authenticated requests get a higher rate limit. Check out the documentation for more details.)", "documentation_url": " }debounceDebounce alleviates pressure on the server by putting all input events "on hold" for a given duration.We can easily debounce the input stream by adding one line:_results = _searchTerms .debounce((_) => TimerStream(true, Duration(milliseconds: 500))) .switchMap((query) async* { print('searching: $query'); yield await apiWrapper.searchUser(query);}); // discard previous eventsWe can control the debounce duration to our liking (500ms is a good default value).And there we have it! An efficient search implementation that feels snappy, without overloading the server or the client.CreditsThis tutorial was heavily inspired by this talk by Brian Egan & Filip Hracek at ReactiveConf 2018:ReactiveConf 2018 - Brian Egan & Filip Hracek: Practical Rx with FlutterThe full source code is available for the GitHub search example - feel free to use this as reference in your own apps.Happy coding! Ayoub Ali Posted on May 30, 2023 Reactive programming is a popular paradigm that enables developers to build highly responsive and scalable applications. When combined with the Flutter framework, it empowers developers to create dynamic and reactive user interfaces. One of the most powerful libraries for reactive programming in Flutter is RxDart. In this blog post, we will explore the fundamentals of reactive programming and demonstrate how RxDart can be leveraged to enhance your Flutter applications. Outline Core Concepts of Reactive Programming At its core, reactive programming revolves around three fundamental concepts: Observables:Observables represent a sequence of values that can change over time. They can emit data or events, and other parts of the application can subscribe to these observables to be notified when new values or events are available.Streams:Streams are a type of observables in reactive programming. They provide a continuous flow of data or events over time. Developers can listen to streams and react to the data or events emitted by them.Data Flow:Reactive programming encourages a unidirectional data flow, where changes in the observables trigger updates in dependent components or operations. This ensures that the application remains responsive and efficiently handles changes without causing unnecessary side effects. By leveraging these core concepts, reactive programming enables developers to build applications that can react to user interactions, data updates, and other events in a scalable and efficient manner. It promotes a more declarative and event-driven style of programming, making it easier to handle complex asynchronous operations and maintain a responsive user interface. RxDart Installation and Setup To install RxDart and set it up in your Flutter project, follow these steps: Open your Flutter project in an IDE or text editor.Open the pubspec.yaml file located in the root directory of your Flutter project.In the dependencies section of the pubspec.yaml file, add the following line:dependencies: rxdart: ^0.27.1 This line specifies that your project will depend on the RxDart library and uses version 0.27.1 (or the latest version available). Save the pubspec.yaml file.In your IDE or terminal, run the following command to fetch and install the RxDart library: This command will download the RxDart library and make it available for use in your Flutter project. Once the installation is complete, you can start using RxDart in your Flutter code. Import the RxDart package in the relevant files where you intend to use it:import 'package:rxdart/rxdart.dart'; You are now ready to utilize RxDart and its reactive programming capabilities in your Flutter project. Refer to the RxDart documentation and examples to learn about different observables, streams, and operators provided by the library. Remember to import the necessary classes from the RxDart package whenever you need to use them in your code. Note: Make sure to follow the Flutter and Dart version compatibility requirements specified by the RxDart library. Key concept of Observable, Stream, StreamController and Subjects in RxDart Key Concepts in RxDart: Observable: Observables represent a stream of data or events that can change over time.They allow you to emit values or events and enable other parts of the application to subscribe and react to those emissions.Observables can be created from various sources such as lists, futures, or streams. Example: // Creating an Observable from a List final numbers = Observable.fromIterable([1, 2, 3, 4, 5]); // Subscribing to the Observable final subscription = numbers.listen((number) { print('Received number: $number'); }); // Output: Received number: 1, Received number: 2, ... Stream and StreamController: A stream represents a sequence of asynchronous events. It is a core concept in Dart's async programming model.StreamController acts as a source of events for a stream. It allows you to add events to the stream and control its flow.Streams provide a way to handle asynchronous data and enable listening to events emitted by the stream. Example: // Creating a StreamController final controller = StreamController(); // Adding events to the stream controller.add(1); controller.add(2); controller.add(3); // Listening to the stream final subscription = controller.stream.listen((event) { print('Received event: $event'); }); // Output: Received event: 1, Received event: 2, ... Subjects: Subjects are a type of Observable and StreamController combined. They can act as both a source of events and a stream to listen to those events.RxDart provides different types of subjects, such as BehaviorSubject, PublishSubject, and ReplaySubject, each with unique characteristics.Subjects are often used for managing state and broadcasting events within reactive programming. Example: // Creating a BehaviorSubject final subject = BehaviorSubject(); // Subscribing to the BehaviorSubject final subscription = subject.listen((value) { print('Received value: $value'); }); // Emitting values through the BehaviorSubject subject.add(1); subject.add(2); subject.add(3); // Output: Received value: 1, Received value: 2, ... Reactive event handling Combining streams and observables in RxDart 1. Reactive Event Handling: Reactive event handling refers to the ability of RxDart to handle and react to events in a reactive and efficient manner. It allows you to listen to events from various sources, such as user interactions, network responses, or timer events, and perform actions based on those events. RxDart provides operators and techniques to handle events reactively, enabling you to build responsive and dynamic applications. Example of Reactive Event Handling:// Creating an Observable for button pressesfinal buttonPresses = Observable(controller.stream); // Subscribing to button presses and reacting to the eventsfinal subscription = buttonPresses.listen((event) { print('Button pressed!');}); // Simulating button presses by adding events to the streamcontroller.add(true);controller.add(false); // Output: Button pressed!, Button pressed! 2. Combining Streams and Observables: Combining streams and observables in RxDart allows you to merge, combine, or transform multiple streams or observables into a single stream or observable. This capability is useful when you need to handle multiple data sources or perform complex operations on data emitted by different streams or observables. Example of Combining Streams and Observables:// Creating two streamsfinal stream1 = Stream.fromIterable([1, 2, 3]);final stream2 = Stream.fromIterable([4, 5, 6]); // Combining the streams into a single streamfinal combinedStream = Rx.concat([stream1, stream2]); // Subscribing to the combined stream and reacting to the eventsfinal subscription = combinedStream.listen((event) { print('Combined event: $event');}); // Output: Combined event: 1, Combined event: 2, ..., Combined event: 6 In this example, the concat operator from RxDart combines two streams into a single stream, merging their events in the order they occur. The resulting combined stream emits events from both stream1 and stream2. Advanced Techniques and Best PracticesThrottling and Debouncing: Throttling and debouncing are techniques used to control the rate at which events are emitted.Throttling limits the number of events emitted within a specified time interval.Debouncing delays emitting events until a specified quiet period occurs, discarding any previous events within that period. Example of Throttling: // Creating an Observable from button presses final buttonPresses = Observable(controller.stream); // Throttling the button presses to emit at most one event per 500 milliseconds final throttledButtonPresses = buttonPresses.throttleTime(Duration(milliseconds: 500)); // Subscribing to the throttled button presses final subscription = throttledButtonPresses.listen((event) { print('Button pressed!'); }); // Simulating multiple button presses for (int i = 0; i < 10; i++) { controller.add(true); await Future.delayed(Duration(milliseconds: 100)); } // Output: Button pressed! Example of Debouncing: // Creating an Observable from search queries final searchQueries = Observable(controller.stream); // Debouncing the search queries to emit events only after 500 milliseconds of quiet period final debouncedSearchQueries = searchQueries.debounceTime(Duration(milliseconds: 500)); // Subscribing to the debounced search queries final subscription = debouncedSearchQueries.listen((query) { print('Search query: $query'); // Perform search operation here }); // Simulating search queries controller.add('flutter'); await Future.delayed(Duration(milliseconds: 200)); controller.add('rx'); await Future.delayed(Duration(milliseconds: 200)); controller.add('dart'); await Future.delayed(Duration(milliseconds: 800)); // Output: Search query: dart Error Handling and Retries: RxDart provides operators to handle errors emitted by observables or streams.Error-handling operators like onErrorResumeNext or catchError allow you to handle errors gracefully and provide fallback mechanisms.You can also implement retry mechanisms using operators like retry or retryWhen to automatically retry failed operations. Example of Error Handling and Retries: // Creating an Observable from a network request final request = Observable.fromFuture(fetchDataFromNetwork()); // Handling errors and providing a fallback value final response = request.onErrorResumeNext(Observable.just('Fallback response')); // Subscribing to the response final subscription = response.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }); // Output: Received data: Fallback response (in case of error) Memory Management and Resource Disposal: It is essential to manage resources and dispose of subscriptions and subjects properly to avoid memory leaks.Use the takeUntil or takeWhile operators to automatically dispose of subscriptions when certain conditions are met.Dispose of subscriptions and subjects explicitly using the subscription.cancel() or subject.close() methods when they are no longer needed. Example of Memory Management and Resource Disposal: // Creating an Observable from a timer final timer = Observable(Stream.periodic(Duration(seconds: 1), (value) => value)); // Subscribing to the timer and automatically disposing of the subscription after 5 seconds final subscription = timer.takeUntil(Observable.timer(null, Duration(seconds: 5))).listen((value) { print('Timer value: $value'); }); // Output: Timer value: 0, Timer value: 1, Timer value: 2, Timer value: 3, Timer value: 4 // Disposing of the subscription explicitly after it is no longer needed subscription.cancel(); Testing and Debugging with RxDart Testing and debugging are crucial aspects of any software development process. Here's how you can approach testing and debugging with RxDart, along with an example: Testing with RxDart: RxDart provides various utilities and techniques to test observables, streams, and operators.Use the TestWidgetsFlutterBinding.ensureInitialized() method to initialize the test environment before running RxDart tests.Utilize the TestStream class from the rxdart/testing.dart package to create testable streams and observables.Use test-specific operators like materialize() and dematerialize() to convert events into notifications that can be easily asserted. Example of Testing with RxDart: import 'package:rxdart/rxdart.dart'; import 'package:rxdart/testing.dart'; import 'package:test/test.dart'; void main() { test('Test observable emits correct values', () { // Initialize the test environment TestWidgetsFlutterBinding.ensureInitialized(); // Create a TestStream final stream = TestStream(); // Emit values to the stream stream.emit(1); stream.emit(2); stream.emit(3); stream.close(); // Create an observable from the TestStream final observable = Observable(stream); // Assert the emitted values expect(observable, emitsInOrder([1, 2, 3])); }); } Debugging with RxDart: RxDart provides debugging operators that help analyze and debug observables and streams during development.The doOnData() operator allows you to inspect each emitted data item, enabling you to log or perform other debugging operations.The doOnError() and doOnDone() operators allow you to handle error and completion events respectively for debugging purposes. Example of Debugging with RxDart: // Creating an observable from a list final observable = Observable.fromIterable([1, 2, 3, 4, 5]); // Adding the doOnData operator for debugging final debugObservable = observable.doOnData((data) { print('Data: $data'); }); // Subscribing to the debugObservable final subscription = debugObservable.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }, onDone: () { print('Stream completed'); }); // Output: // Data: 1 // Received data: 1 // Data: 2 // Received data: 2 // Data: 3 // Received data: 3 // Data: 4 // Received data: 4 // Data: 5 // Received data: 5 // Stream completed By applying testing and debugging techniques, you can ensure the correctness and reliability of your RxDart code. Test your observables and streams using the provided testing utilities and leverage debugging operators to gain insights into the behavior of your reactive code during development and troubleshooting processes. Real-World Example In this example we will build a fully functional app that search a world from JSON API. Step -1 In first step we will generate a model for our JSON API.Use can use QuikeType.io to generate it just paste the JSON schema and you have the model @immutableclass Words { final List names; const Words({ required this.names, }); Words copyWith({ List? names, }) => Words(names: names ?? this.names,); factory Words.fromJson(Map json) => Words(names: List.from(json["names"].map((x) => x)),); Map toJson() => { "names": List.from(names.map((x) => x)), };} This code provides a way to convert Words objects to JSON and vice versa, making it easy to serialize and deserialize the data for communication or storage purposes. Step -2 Let's build a heraricary that focuses on creating of classes representing different search result states. Each class represents a specific state, such as loading, no result, error, or with a successful result.import 'package:flutter/foundation.dart' show immutable; @immutableabstract class SearchResult { const SearchResult();} @immutableclass SearchResultLoading implements SearchResult { const SearchResultLoading();} @immutableclass SearchResultNoResult implements SearchResult { const SearchResultNoResult();} @immutableclass SearchResultWithError implements SearchResult { final Object? error; const SearchResultWithError(this.error);} @immutableclass SearchResultWithResult implements SearchResult { final List result; const SearchResultWithResult(this.result);} Step - 3 Let's Write code that demonstrates and performs a search operation on a list of words fetched from an API. This code demonstrates the basic steps involved in performing a search operation using a remote API, caching the results, and extracting the matching words based on a search term.import 'dart:convert'; import 'package:http/http.dart' as http; class Api { List? _words; Api(); // Step - 3 Future search(String searchTerm) async { final term = searchTerm.trim().toLowerCase(); final cachedResult = _extractWordsUsingSearchTerm(searchTerm); if (cachedResult != null) { return cachedResult; } // api calling final words = await _getData("

_words = words; return _extractWordsUsingSearchTerm(term) ?? []; } // Step - 2 List? _extractWordsUsingSearchTerm(String word) { final cachedWords = _words; if (cachedWords != null) { List result = []; for (final worded in cachedWords) { if (worded.contains(word.trim().toLowerCase())) { result.add(worded); } } return result; } else { return null; } } // Step - 1 // Future _getData(String url) => HttpClient() // .getUrl(Uri.parse(url)) // .then((request) => request.close()) // .then((response) => response.transform(utf8.decoder).join()) // .then((jsonString) => json.decode(jsonString) as List); Future _getData(String url) async { final response = await http.Client().get(Uri.parse(url)); final parsed = jsonDecode(response.body)['names']; List? names = parsed != null ? List.from(parsed) : null; return names; }} // work only on String not listextension TrimmedCaseInsensitiveContain on String { bool trimmedContains(String other) => trim().toLowerCase().contains(other.trim().toLowerCase(),);} Step - 4 This code demonstrates the setup of a reactive search bloc using RxDart. It establishes a bidirectional communication channel for searching by providing a sink to add search terms and a stream to receive search results. The search terms undergo stream transformations to control the search behavior, and the results are emitted as a stream of SearchResult objects with different states. The result stream is created by chaining several stream transformations on textChanges. It performs the following operations: - distinct() ensures that only distinct search terms are processed. - debounceTime(const Duration(milliseconds: 350)) delays the processing of the search term stream, allowing a brief duration (350 milliseconds) of inactivity before emitting the latest search term. This helps to reduce unnecessary API calls for rapidly changing search terms. - switchMap((String searchTerm) maps each search term to a stream of SearchResult objects based on the search operation. If the search term is empty, it immediately emits a null result. Otherwise, it performs the actual search operation using the api.search method, which returns a Future>. This future is wrapped using Rx.fromCallable and then delayed by 1 second using delay to introduce a delay before emitting the result. The mapped stream is further transformed using map to convert the search results into appropriate SearchResult objects (SearchResultNoResult, SearchResultWithResult, or SearchResultLoading) based on the conditions. - startWith(const SearchResultLoading()) emits a loading result as the initial value when the search begins. - onErrorReturnWith((error, _) => SearchResultWithError(error)) handles any errors that occur during the search operation and emits a SearchResultWithError object.import 'dart:async'; import 'package:flutter/foundation.dart' show immutable;import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:rxdart/rxdart.dart'; @immutableclass SearchBloc { final Sink search; final Stream results; void dispose() { search.close(); } factory SearchBloc({required Api api}) { final textChanges = BehaviorSubject(); final result = textChanges .distinct() .debounceTime(const Duration(milliseconds: 350)) .switchMap((String searchTerm) { if (searchTerm.isEmpty) { return Stream.value(null); } else { return Rx.fromCallable(() => api.search(searchTerm)) .delay(const Duration(seconds: 1)) .map((results) => results.isEmpty ? const SearchResultNoResult() : SearchResultWithResult(results),) .startWith(const SearchResultLoading()) .onErrorReturnWith((error, _) => SearchResultWithError(error)); } }); return SearchBloc._(search: textChanges.sink, results: result,); } const SearchBloc._({ required this.search, required this.results, });} Step - 5 UI Development Custom widget to display out search result in GridViewimport 'package:flutter/material.dart'; class GridViewWidget extends StatelessWidget { const GridViewWidget({ Key? key, required this.results, }) : super(key: key); final List results; @override Widget build(BuildContext context) { return Expanded(child: GridView.builder(itemCount: results.length, gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent: 200, childAspectRatio: 3 / 2, crossAxisSpacing: 20, mainAxisSpacing: 20), itemBuilder: (context, index) { return Container(alignment: Alignment.center, decoration: BoxDecoration(color: Colors.white, borderRadius: BorderRadius.circular(15)), child: Text(results[index], textAlign: TextAlign.center, style: const TextStyle(fontSize: 20, fontWeight: FontWeight.bold),),); },),); }} Step - 6 This code provides a UI representation of the different states of the search results and handles the appropriate rendering based on the received data.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:infinite_words/widgets/grid_view_widget.dart'; class SearchResultView extends StatelessWidget { final Stream searchResults; const SearchResultView({ Key? key, required this.searchResults, }) : super(key: key); @override Widget build(BuildContext context) { return StreamBuilder(stream: searchResults, builder: (BuildContext context, AsyncSnapshot snapshot,) { if (snapshot.hasData) { final result = snapshot.data; if (result is SearchResultWithError) { return const Center(child: Text('Error')); } else if (result is SearchResultLoading) { return const Center(child: CircularProgressIndicator()); } else if (result is SearchResultNoResult) { return const Center(child: Text('No Result Found')); } else if (result is SearchResultWithResult) { final results = result.result; return GridViewWidget(results: results); } else { return const Center(child: Text("Unknown State")); } } else { return const Center(child: Text("Waiting.....", style: TextStyle(color: Colors.white, fontSize: 18),),); } },); }} Step -7 This code sets up the home page of the application with a text input field for searching words and displays the search results using the SearchResultView widget. The SearchBloc manages the search functionality and emits the search results to the UI.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_bloc.dart';import 'package:infinite_words/view/search_result_view.dart'; class HomePage extends StatefulWidget { const HomePage({Key? key}) : super(key: key); @override State createState() => _HomePageState();} class _HomePageState extends State { late final SearchBloc _bloc; @override void initState() { _bloc = SearchBloc(api: Api()); super.initState(); } @override void dispose() { super.dispose(); _bloc.dispose(); } @override Widget build(BuildContext context) { return Scaffold(backgroundColor: Colors.black, // appBar: AppBar(// title: const Text('Search'), //), body: Padding(padding: const EdgeInsets.all(10), child: Column(children: [const SizedBox(height: 50,), TextField(decoration: InputDecoration(border: InputBorder.none, filled: true, fillColor: Colors.white, contentPadding: const EdgeInsets.only(left: 14.0, bottom: 6.0, top: 8.0), focusedBorder: OutlineInputBorder(borderSide: const BorderSide(color: Colors.grey), borderRadius: BorderRadius.circular(10.0),), enabledBorder: UnderlineInputBorder(borderSide: const BorderSide(color: Colors.white), borderRadius: BorderRadius.circular(10.0),), hintText: "Write a word", hintStyle: const TextStyle(fontSize: 20, // color: Colors.white,)), onChanged: _bloc.search.add), const SizedBox(height: 10,), SearchResultView(searchResults: _bloc.results)],),),); }} SourceCode RxDart extends the capabilities of DartStreams andStreamControllers.Dart comes with a very decentStreams APIout-of-the-box; rather than attempting to provide an alternative to this API,RxDart adds functionality from the reactive extensions specification on top ofit.RxDart does not provide its Observable class as a replacement for DartStreams. Instead, it offers several additional Stream classes, operators(extension methods on the Stream class), and Subjects.If you are familiar with Observables from other languages, please see the RxObservables vs. Dart Streams comparison chartfor notable distinctions between the two. RxDart 0.23.x moves away from the Observable class, utilizing Dart 2.6's newextension methods instead. This requires several small refactors that can beasily automated -- which is just what we've done!Please follow the instructions on therxdart_codemod package toautomatically upgrade your code to support RxDart 0.23.x. import 'package:rxdart/rxdart.dart';void main() { const konamiKeyCodes = [KeyCode.UP, KeyCode.UP, KeyCode.DOWN, KeyCode.DOWN, KeyCode.LEFT, KeyCode.RIGHT, KeyCode.LEFT, KeyCode.RIGHT, KeyCode.B, KeyCode.A,]; final result = querySelector('#result')!; document.onKeyUp .map((event) => event.keyCode) .bufferCount(10, 1) // An extension method provided by rxdart .where((lastTenKeyCodes) => const IterableEquality().equals(lastTenKeyCodes, konamiKeyCodes)) .listen((_) => result.innerHtml = 'KONAMI!');} RxDart adds functionality to Dart Streams in three ways:Stream Classes - create Streams with specific capabilities, such as combining or merging many Streams.Extension Methods - transform a source Stream into a new Stream with different capabilities, such as throttling or buffering events.Subjects - StreamControllers with additional powers The Stream class provides different ways to create a Stream: Stream.fromIterable or Stream.periodic. RxDart provides additional Stream classes for a variety of tasks, such as combining or merging Streams!You can construct the Streams provided by RxDart in two ways. The following examples are equivalent in terms of functionality:Instantiating the Stream class directly.Example: final mergedStream = MergeStream([myFirstStream, mySecondStream]);Using static factories from the Rx class, which are useful for discovering which types of Streams are provided by RxDart. Under the hood, these factories call the corresponding Stream constructor.Example: final mergedStream = Rx.merge([myFirstStream, mySecondStream]);List of Classes / Static Factories The extension methods provided by RxDart can be used on any Stream. They convert a source Stream into a new Stream with additional capabilities, such as buffering or throttling events.ExampleStream.fromIterable([1, 2, 3]) .throttleTime(Duration(seconds: 1)) .listen(print); // prints 1List of Extension Methods Dart provides the StreamController class to create and manage a Stream. RxDart offers two additional StreamControllers with additional capabilities, known as Subjects:BehaviorSubject - A broadcast StreamController that caches the latest added value or error. When a new listener subscribes to the Stream, the latest value or error will be emitted to the listener. Furthermore, you can synchronously read the last emitted value.ReplaySubject - A broadcast StreamController that caches the added values. When a new listener subscribes to the Stream, the cached values will be emitted to the listener. In many situations, Streams and Observables work the same way. However, if you're used to standard Rx Observables, some features of the Stream API may surprise you. We've included a table below to help folks understand the differences.Additional information about the following situations can be found by reading the Rx class documentation.SituationRx ObservablesDart StreamsAn error is raisedObservable Terminates with ErrorError is emitted and Stream continuesCold ObservablesMultiple subscribers can listen to the same cold Observable, and each subscription will receive a unique Stream of dataSingle subscriber onlyHot ObservablesYesYes, known as Broadcast StreamsIs {Publish, Behavior, Replay}Subject hot?YesYesSingle/Maybe/Completable ?YesYes, uses rxdart_ext Single (Completable = Single, Maybe = Single)Support back pressureYesYesCan emit null?Yes, except RxJavaYesSync by defaultYesNoCan pause/resume a subscription*?NoYes Web and command-line examples can be found in the example folder. In order to run the web examples, please follow these steps: In order to run the command line example, please follow these steps:Clone this repo and enter the directoryRun pub getRun dart examples/fibonacci/lib/example.dart 10 Install FlutterTo run the flutter example, you must have Flutter installed. For installation instructions, view the onlinedocumentation.Run the appOpen up an Android Emulator, the iOS Simulator, or connect an appropriate mobile device for debugging.Open up a terminalcd into the examples/flutter/github_search directoryRun flutter doctor to ensure you have all Flutter dependencies working.Run flutter packages getRun flutter run Refer to the Changelog to get all release notes. Check out rxdart_ext, which provides many extension methods and classes built on top of RxDart. Ayoub Ali Posted on May 30, 2023 Reactive programming is a popular paradigm that enables developers to build highly responsive and scalable applications. When combined with the Flutter framework, it empowers developers to create dynamic and reactive user interfaces. One of the most powerful libraries for reactive programming in Flutter is RxDart. In this blog post, we will explore the fundamentals of reactive programming and demonstrate how RxDart can be leveraged to enhance your Flutter applications. Outline Core Concepts of Reactive Programming At its core, reactive programming revolves around three fundamental concepts: Observables:Observables represent a sequence of values that can change over time. They can emit data or events, and other parts of the application can subscribe to these observables to be notified when new values or events are available.Streams:Streams are a type of observables in reactive programming. They provide a continuous flow of data or events over time. Developers can listen to streams and react to the data or events emitted by them.Data Flow:Reactive programming encourages a unidirectional data flow, where changes in the observables trigger updates in dependent components or operations. This ensures that the application remains responsive and efficiently handles changes without causing unnecessary side effects. By leveraging these core concepts, reactive programming enables developers to build applications that can react to user interactions, data updates, and other events in a scalable and efficient manner. It promotes a more declarative and event-driven style of programming, making it easier to handle complex asynchronous operations and maintain a responsive user interface. RxDart Installation and Setup To install RxDart and set it up in your Flutter project, follow these steps: Open your Flutter project in an IDE or text editor.Open the pubspec.yaml file located in the root directory of your Flutter project.In the dependencies section of the pubspec.yaml file, add the following line:dependencies: rxdart: ^0.27.1 This line specifies that your project will depend on the RxDart library and uses version 0.27.1 (or the latest version available). Save the pubspec.yaml file.In your IDE or terminal, run the following command to fetch and install the RxDart library: This command will download the RxDart library and make it available for use in your Flutter project. Once the installation is complete, you can start using RxDart in your Flutter code. Import the RxDart package in the relevant files where you intend to use it:import 'package:rxdart/rxdart.dart'; You are now ready to utilize RxDart and its reactive programming capabilities in your Flutter project. Refer to the RxDart documentation and examples to learn about different observables, streams, and operators provided by the library. Remember to import the necessary classes from the RxDart package whenever you need to use them in your code. Note: Make sure to follow the Flutter and Dart version compatibility requirements specified by the RxDart library. Key concept of Observable, Stream, StreamController and Subjects in RxDart Key Concepts in RxDart: Observable: Observables represent a stream of data or events that can change over time.They allow you to emit values or events and enable other parts of the application to subscribe and react to those emissions.Observables can be created from various sources such as lists, futures, or streams. Example: // Creating an Observable from a List final numbers = Observable.fromIterable([1, 2, 3, 4, 5]); // Subscribing to the Observable final subscription = numbers.listen((number) { print('Received number: $number'); }); // Output: Received number: 1, Received number: 2, ... Stream and StreamController: A stream represents a sequence of asynchronous events. It is a core concept in Dart's async programming model.StreamController acts as a source of events for a stream. It allows you to add events to the stream and control its flow.Streams provide a way to handle asynchronous data and enable listening to events emitted by the stream. Example: // Creating a StreamController final controller = StreamController(); // Adding events to the stream controller.add(1); controller.add(2); controller.add(3); // Listening to the stream final subscription = controller.stream.listen((event) { print('Received event: $event'); }); // Output: Received event: 1, Received event: 2, ... Subjects: Subjects are a type of Observable and StreamController combined. They can act as both a source of events and a stream to listen to those events.RxDart provides different types of subjects, such as BehaviorSubject, PublishSubject, and ReplaySubject, each with unique characteristics.Subjects are often used for managing state and broadcasting events within reactive programming. Example: // Creating a BehaviorSubject final subject = BehaviorSubject(); // Subscribing to the BehaviorSubject final subscription = subject.listen((value) { print('Received value: $value'); }); // Emitting values through the BehaviorSubject subject.add(1); subject.add(2); subject.add(3); // Output: Received value: 1, Received value: 2, ... Reactive event handling Combining streams and observables in RxDart 1. Reactive Event Handling: Reactive event handling refers to the ability of RxDart to handle and react to events in a reactive and efficient manner. It allows you to listen to events from various sources, such as user interactions, network responses, or timer events, and perform actions based on those events. RxDart provides operators and techniques to handle events reactively, enabling you to build responsive and dynamic applications. Example of Reactive Event Handling:// Creating an Observable for button pressesfinal buttonPresses = Observable(controller.stream); // Subscribing to button presses and reacting to the eventsfinal subscription = buttonPresses.listen((event) { print('Button pressed!');}); // Simulating button presses by adding events to the streamcontroller.add(true);controller.add(false); // Output: Button pressed!, Button pressed! 2. Combining Streams and Observables: Combining streams and observables in RxDart allows you to merge, combine, or transform multiple streams or observables into a single stream or observable. This capability is useful when you need to handle multiple data sources or perform complex operations on data emitted by different streams or observables. Example of Combining Streams and Observables:// Creating two streamsfinal stream1 = Stream.fromIterable([1, 2, 3]);final stream2 = Stream.fromIterable([4, 5, 6]); // Combining the streams into a single streamfinal combinedStream = Rx.concat([stream1, stream2]); // Subscribing to the combined stream and reacting to the eventsfinal subscription = combinedStream.listen((event) { print('Combined event: $event');}); // Output: Combined event: 1, Combined event: 2, ..., Combined event: 6 In this example, the concat operator from RxDart combines two streams into a single stream, merging their events in the order they occur. The resulting combined stream emits events from both stream1 and stream2. Advanced Techniques and Best PracticesThrottling and Debouncing: Throttling and debouncing are techniques used to control the rate at which events are emitted.Throttling limits the number of events emitted within a specified time interval.Debouncing delays emitting events until a specified quiet period occurs, discarding any previous events within that period. Example of Throttling: // Creating an Observable from button presses final buttonPresses = Observable(controller.stream); // Throttling the button presses to emit at most one event per 500 milliseconds final throttledButtonPresses = buttonPresses.throttleTime(Duration(milliseconds: 500)); // Subscribing to the throttled button presses final subscription = throttledButtonPresses.listen((event) { print('Button pressed!'); }); // Simulating multiple button presses for (int i = 0; i < 10; i++) { controller.add(true); await Future.delayed(Duration(milliseconds: 100)); } // Output: Button pressed! Example of Debouncing: // Creating an Observable from search queries final searchQueries = Observable(controller.stream); // Debouncing the search queries to emit events only after 500 milliseconds of quiet period final debouncedSearchQueries = searchQueries.debounceTime(Duration(milliseconds: 500)); // Subscribing to the debounced search queries final subscription = debouncedSearchQueries.listen((query) { print('Search query: $query'); // Perform search operation here }); // Simulating search queries controller.add('flutter'); await Future.delayed(Duration(milliseconds: 200)); controller.add('rx'); await Future.delayed(Duration(milliseconds: 200)); controller.add('dart'); await Future.delayed(Duration(milliseconds: 800)); // Output: Search query: dart Error Handling and Retries: RxDart provides operators to handle errors emitted by observables or streams.Error-handling operators like onErrorResumeNext or catchError allow you to handle errors gracefully and provide fallback mechanisms.You can also implement retry mechanisms using operators like retry or retryWhen to automatically retry failed operations. Example of Error Handling and Retries: // Creating an Observable from a network request final request = Observable.fromFuture(fetchDataFromNetwork()); // Handling errors and providing a fallback value final response = request.onErrorResumeNext(Observable.just('Fallback response')); // Subscribing to the response final subscription = response.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }); // Output: Received data: Fallback response (in case of error) Memory Management and Resource Disposal: It is essential to manage resources and dispose of subscriptions and subjects properly to avoid memory leaks.Use the takeUntil or takeWhile operators to automatically dispose of subscriptions when certain conditions are met.Dispose of subscriptions and subjects explicitly using the subscription.cancel() or subject.close() methods when they are no longer needed. Example of Memory Management and Resource Disposal: // Creating an Observable from a timer final timer = Observable(Stream.periodic(Duration(seconds: 1), (value) => value)); // Subscribing to the timer and automatically disposing of the subscription after 5 seconds final subscription = timer.takeUntil(Observable.timer(null, Duration(seconds: 5))).listen((value) { print('Timer value: $value'); }); // Output: Timer value: 0, Timer value: 1, Timer value: 2, Timer value: 3, Timer value: 4 // Disposing of the subscription explicitly after it is no longer needed subscription.cancel(); Testing and Debugging with RxDart Testing and debugging are crucial aspects of any software development process. Here's how you can approach testing and debugging with RxDart, along with an example: Testing with RxDart: RxDart provides various utilities and techniques to test observables, streams, and operators.Use the TestWidgetsFlutterBinding.ensureInitialized() method to initialize the test environment before running RxDart tests.Utilize the TestStream class from the rxdart/testing.dart package to create testable streams and observables.Use test-specific operators like materialize() and dematerialize() to convert events into notifications that can be easily asserted. Example of Testing with RxDart: import 'package:rxdart/rxdart.dart'; import 'package:rxdart/testing.dart'; import 'package:test/test.dart'; void main() { test('Test observable emits correct values', () { // Initialize the test environment TestWidgetsFlutterBinding.ensureInitialized(); // Create a TestStream final stream = TestStream(); // Emit values to the stream stream.emit(1); stream.emit(2); stream.emit(3); stream.close(); // Create an observable from the TestStream final observable = Observable(stream); // Assert the emitted values expect(observable, emitsInOrder([1, 2, 3])); }); } Debugging with RxDart: RxDart provides debugging operators that help analyze and debug observables and streams during development.The doOnData() operator allows you to inspect each emitted data item, enabling you to log or perform other debugging operations.The doOnError() and doOnDone() operators allow you to handle error and completion events respectively for debugging purposes. Example of Debugging with RxDart: // Creating an observable from a list final observable = Observable.fromIterable([1, 2, 3, 4, 5]); // Adding the doOnData operator for debugging final debugObservable = observable.doOnData((data) { print('Data: $data'); }); // Subscribing to the debugObservable final subscription = debugObservable.listen((data) { print('Received data: $data'); }, onError: (error) { print('Error occurred: $error'); }, onDone: () { print('Stream completed'); }); // Output: // Data: 1 // Received data: 1 // Data: 2 // Received data: 2 // Data: 3 // Received data: 3 // Data: 4 // Received data: 4 // Data: 5 // Received data: 5 // Stream completed By applying testing and debugging techniques, you can ensure the correctness and reliability of your RxDart code. Test your observables and streams using the provided testing utilities and leverage debugging operators to gain insights into the behavior of your reactive code during development and troubleshooting processes. Real-World Example In this example we will build a fully functional app that search a world from JSON API. Step -1 In first step we will generate a model for our JSON API.Use can use QuikeType.io to generate it just paste the JSON schema and you have the model @immutableclass Words { final List names; const Words({ required this.names, }); Words copyWith({ List? names, }) => Words(names: names ?? this.names,); factory Words.fromJson(Map json) => Words(names: List.from(json["names"].map((x) => x)),); Map toJson() => { "names": List.from(names.map((x) => x)), };} This code provides a way to convert Words objects to JSON and vice versa, making it easy to serialize and deserialize the data for communication or storage purposes. Step -2 Let's build a hierarchy that focuses on creating of classes representing different search result states. Each class represents a specific state, such as loading, no result, error, or with a successful result.import 'package:flutter/foundation.dart' show immutable; @immutableabstract class SearchResult { const SearchResult();} @immutableclass SearchResultLoading implements SearchResult { const SearchResultLoading();} @immutableclass SearchResultNoResult implements SearchResult { const SearchResultNoResult();} @immutableclass SearchResultWithError implements SearchResult { final Object? error; const SearchResultWithError(this.error);} @immutableclass SearchResultWithResult implements SearchResult { final List result; const SearchResultWithResult(this.result);} Step - 3 Let's Write code that demonstrates and performs a search operation on a list of words fetched from an API. This code demonstrates the basic steps involved in performing a search operation using a remote API, caching the results, and extracting the matching words based on a search term.import 'dart:convert'; import 'package:http/http.dart' as http; class Api { List? _words; Api(); // Step - 3 Future search(String searchTerm) async { final term = searchTerm.trim().toLowerCase(); final cachedResult = _extractWordsUsingSearchTerm(searchTerm); if (cachedResult != null) { return cachedResult; } // api calling final words = await _getData(" _words = words; return _extractWordsUsingSearchTerm(term) ?? []; } // Step - 2 List? _extractWordsUsingSearchTerm(String word) { final cachedWords = _words; if (cachedWords != null) { List result = []; for (final worded in cachedWords) { if (worded.contains(word.trim().toLowerCase())) { result.add(worded); } } return result; } else { return null; } } // Step - 1 // Future _getData(String url) => HttpClient() // .getUrl(Uri.parse(url)) // .then((request) => request.close()) // .then((response) => response.transform(utf8.decoder).join()) // .then((jsonString) => json.decode(jsonString) as List); Future _getData(String url) async { final response = await http.Client().get(Uri.parse(url)); final parsed = jsonDecode(response.body)['names']; List? names = parsed != null ? List.from(parsed) : null; return names; }} // work only on String not listextension TrimmedCaseInsensitiveContain on String { bool trimmedContains(String other) => trim().toLowerCase().contains(other.trim().toLowerCase(),);} Step - 4 This code demonstrates the setup of a reactive search bloc using RxDart. It establishes a bidirectional communication channel for searching by providing a sink to add search terms and a stream to receive search results. The search terms undergo stream transformations to control the search behavior, and the results are emitted as a stream of SearchResult objects with different states. The result stream is created by chaining several stream transformations on textChanges. It performs the following operations: - distinct() ensures that only distinct search terms are processed. - debounceTime(const Duration(milliseconds: 350)) delays the processing of the search term stream, allowing a brief duration (350 milliseconds) of inactivity before emitting the latest search term. This helps to reduce unnecessary API calls for rapidly changing search terms. - switchMap((String searchTerm) maps each search term to a stream of SearchResult objects based on the search operation. If the search term is empty, it immediately emits a null result. Otherwise, it performs the actual search operation using the api.search method, which returns a Future>. This future is wrapped using Rx.fromCallable and then delayed by 1 second using delay to introduce a delay before emitting the result. The mapped stream is further transformed using map to convert the search results into appropriate SearchResult objects (SearchResultNoResult, SearchResultWithResult, or SearchResultLoading) based on the conditions. - startWith(const SearchResultLoading()) emits a loading result as the initial value when the search begins. - onErrorReturnWith((error, _) => SearchResultWithError(error)) handles any errors that occur during the search operation and emits a SearchResultWithError object.import 'dart:async'; import 'package:flutter/foundation.dart' show immutable;import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:rxdart/rxdart.dart'; @immutableclass SearchBloc { final Sink search; final Stream results; void dispose() { search.close(); } factory SearchBloc({required Api api}) { final textChanges = BehaviorSubject(); final result = textChanges .distinct() .debounceTime(const Duration(milliseconds: 350)) .switchMap((String searchTerm) { if (searchTerm.isEmpty) { return Stream.value(null); } else { return Rx.fromCallable(() => api.search(searchTerm)) .delay(const Duration(seconds: 1)) .map((results) => results.isEmpty ? const SearchResultNoResult() : SearchResultWithResult(results),) .startWith(const SearchResultLoading()) .onErrorReturnWith((error, _) => SearchResultWithError(error)); } }); return SearchBloc._(search: textChanges.sink, results: result,); } const SearchBloc._({ required this.search, required this.results, });} Step - 5 UI Development Custom widget to display out search result in GridViewimport 'package:flutter/material.dart'; class GridViewWidget extends StatelessWidget { const GridViewWidget({ Key? key, required this.results, }) : super(key: key); final List results; @override Widget build(BuildContext context) { return Expanded(child: GridView.builder(itemCount: results.length, gridDelegate: const SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent: 200, childAspectRatio: 3 / 2, crossAxisSpacing: 20, mainAxisSpacing: 20), itemBuilder: (context, index) { return Container(alignment: Alignment.center, decoration: BoxDecoration(color: Colors.white, borderRadius: BorderRadius.circular(15)), child: Text(results[index], textAlign: TextAlign.center, style: const TextStyle(fontSize: 20, fontWeight: FontWeight.bold),),); },),); }} Step - 6 This code provides a UI representation of the different states of the search results and handles the appropriate rendering based on the received data.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/search_result.dart';import 'package:infinite_words/widgets/grid_view_widget.dart'; class SearchResultView extends StatelessWidget { final Stream searchResults; const SearchResultView({ Key? key, required this.searchResults, }) : super(key: key); @override Widget build(BuildContext context) { return StreamBuilder(stream: searchResults, builder: (BuildContext context, AsyncSnapshot snapshot,) { if (snapshot.hasData) { final result = snapshot.data; if (result is SearchResultWithError) { return const Center(child: Text('Error')); } else if (result is SearchResultLoading) { return const Center(child: CircularProgressIndicator()); } else if (result is SearchResultNoResult) { return const Center(child: Text('No Result Found')); } else if (result is SearchResultWithResult) { final results = result.result; return GridViewWidget(results: results); } else { return const Center(child: Text("Unknown State")); } } else { return const Center(child: Text("Waiting.....", style: TextStyle(color: Colors.white, fontSize: 18),),); } },); }} Step -7 This code sets up the home page of the application with a text input field for searching words and displays the search results using the SearchResultView widget. The SearchBloc manages the search functionality and emits the search results to the UI.import 'package:flutter/material.dart';import 'package:infinite_words/bloc/api.dart';import 'package:infinite_words/bloc/search_bloc.dart';import 'package:infinite_words/view/search_result_view.dart'; class HomePage extends StatefulWidget { const HomePage({Key? key}) : super(key: key); @override State createState() => _HomePageState();} class _HomePageState extends State { late final SearchBloc _bloc; @override void initState() { _bloc = SearchBloc(api: Api()); super.initState(); } @override void dispose() { super.dispose(); _bloc.dispose(); } @override Widget build(BuildContext context) { return Scaffold(backgroundColor: Colors.black, // appBar: AppBar(// title: const Text('Search'), //), body: Padding(padding: const EdgeInsets.all(10), child: Column(children: [const SizedBox(height: 50,), TextField(decoration: InputDecoration(border: InputBorder.none, filled: true, fillColor: Colors.white, contentPadding: const EdgeInsets.only(left: 14.0, bottom: 6.0, top: 8.0), focusedBorder: OutlineInputBorder(borderSide: const BorderSide(color: Colors.grey), borderRadius: BorderRadius.circular(10.0),), enabledBorder: UnderlineInputBorder(borderSide: const BorderSide(color: Colors.white), borderRadius: BorderRadius.circular(10.0),), hintText: "Write a word", hintStyle: const TextStyle(fontSize: 20, // color: Colors.white,)), onChanged: _bloc.search.add), const SizedBox(height: 10,), SearchResultView(searchResults: _bloc.results)],),),); }} SourceCodeReactive Programming is a paradigm that has gained substantial recognition for managing asynchronous data streams and handling reactive user interfaces, among other things. Think of user interactions, API responses, or even animations - Reactive Programming handles them all.Introduction to RxDartEnter RxDart - a reactive streaming extension library built on top of Dart Streams. Its not an attempt to replace Dart Streams but embellishes them with the goodness of reactive programming. RxDart in Flutter has become increasingly popular due to its seamless capabilities in managing asynchronous data stream challenges.Bridging RxDart and Dart StreamsSo, how does RxDart connect with Dart Streams? Dart comes equipped with a powerful Streams API out-of-the-box. However, RxDart adds that extra sparkle with more functionalities based on reactive extensions for Dart Streams, bringing in more flexibility and control.Why Choose RxDart for Flutter?Flutter applications usually have to deal with asynchronous data streams extensively - be it user interactions, network responses, or data changes. By leveraging RxDart, we can handle these data streams more lively, responsive, and controlled. Especially when dealing with complex functionalities, Flutter RxDart provides ease of use and maintains the flow of data in a precise reactive way.Getting Started with RxDartBefore diving into the usage of RxDart in Flutter, we need to set things up.Installing RxDartSetting up RxDart in your Flutter project is a breeze. All you need to do is include it as a dependency in your pubspec.yaml file.Here is how you add RxDart:1 dependencies:2 flutter:3 sdk: flutter4 rxdart: ^0.27.25 Remember to replace the package version with the latest version of RxDart available. Dont forget to run flutter pub get in your terminal to ensure all dependencies are fetched.Basic Usage of RxDart in FlutterNow that we've added RxDart to our project, let's see a simple example that depicts the usage of RxDart and Dart Streams. RxDart does not provide its Observable class as a replacement for Dart Streams, but it offers several additional Stream classes, operators (extension methods on the Stream class), and Subjects.Below is a simple example of how we can use RxDart's capabilities:1 import 'package:rxdart/rxdart.dart';2 3 void main() {4 final subject = BehaviorSubject();5 6 // observer7 subject.stream.listen(print); // prints 1, 2, 38 9 // producer10 subject.sink.add(1);11 subject.sink.add(2);12 subject.sink.add(3);13 14 subject.close();15 }16 In the above Flutter RxDart example, we're creating a BehaviorSubject that deals with integers. We listen to the stream of data and print whatever data is added into the BehaviorSubject.Deeper Dive into RxDartAfter setting up RxDart and understanding its basics, we now venture into more exciting functionalities that the Flutter RxDart combo provides. Let's break it down into Stream Classes, Extension Methods, and Subjects.Understanding Stream Classes in RxDartIn RxDart, Stream Classes allow us to create Streams with specific capabilities, such as combining or merging many Streams. Dart itself provides a Stream class with ways to create a Stream, like Stream.fromIterable or Stream.periodic. RxDart, however, takes it up a notch and extends some cool Stream classes for different use-cases.Here's how you can merge two streams using RxDart's MergeStream:1 final myFirstStream = Stream.fromIterable([1, 2, 3]);2 final mySecondStream = Stream.fromIterable([4, 5, 6]);3 4 final mergedStream = MergeStream([myFirstStream, mySecondStream]);5 6 mergedStream.listen(print); // prints 1, 2, 3, 4, 5, 67 In this code, we create two Streams and merge them using the MergeStream class which results in a single Stream that merges the events from both input Streams.Utilizing RxDart Extension MethodsRxDart extension methods are just methods that may be applied to any Stream. They empower an existing Stream and change it into a new Stream with enhanced capabilities. Throttling or buffering events, for example.Let's see an example where we buffer a Stream of integers to groups of two:1 Stream.fromIterable([1, 2, 3, 4, 5])2 .bufferCount(2)3 .listen(print); // prints [1, 2], [3, 4], [5]4 The bufferCount is an extension method provided by RxDart that buffers a Stream into a specified count. Another powerful feature of RxDart is Subjects - a type of StreamController with added powers! Dart's StreamController creates and manages a Stream, while RxDart offers two additional types, BehaviorSubject and ReplaySubject.The BehaviorSubject is a type of StreamController that caches the latest added value or error. So, when a new listener subscribes to the Stream, the latest value or error will be emitted to the listener. This can be extremely useful in scenarios where you want to share a single value (or its latest status) with multiple components in your Flutter application.An example of a BehaviorSubject would be:1 final behaviorSubject = BehaviorSubject();2 3 // Adding data to the stream4 behaviorSubject.add(1);5 behaviorSubject.add(2);6 behaviorSubject.add(3);7 8 // This will print 3, as BehaviorSubject always returns the last added item9 behaviorSubject.stream.listen(print);10 11 behaviorSubject.close();12 RxDart provides a gamut of functionalities that play a significant role in enhancing the way we work with reactive programming in Flutter. Streams, Extension methods, and Subjects are the core elements to making RxDart one of the most efficient means to handle complex asynchronous data streams in Flutter.RxDart Observables vs Flutter StreamsWhen starting with RxDart, developers coming from other Rx libraries might question the differences between Observables that they used previously and Dart Streams. While they often work similarly, there are a few variations worth mentioning.RxDart Observables vs Flutter StreamsKey DifferencesDart Streams can be thought of as asynchronous Iterables spread out over time, and when an error occurs in a Stream, the Stream emits an error event and then is finished. On the other hand, Observables, as in the standard Rx scenarios, terminate when an error occurs.Dart Streams can have a single subscriber (Single-subscription streams) or multiple subscribers (broadcast streams), while Rx Observables (Cold Observables) do allow multiple subscribers, and each subscription will receive all events.Transitioning from Observables to StreamsWhile transitioning from Observables to Dart Streams, developers need to keep in mind that Dart Streams emit their events asynchronously, and are not synchronized by default.For example, last, length, and other methods always return Futures:1 Stream.fromIterable([1, 2, 3, 4])2 .last3 .then(print); // prints '4'4 The given differences illuminate the contrast and help in recognizing when to employ Observables or Streams. With Flutter RxDart, you start with Dart Streams, then enhance them with extension methods provided by RxDart, offering the much-needed boost for managing data streams.Integrating RxDart in FlutterOne aspect that makes Flutter applications stand out in the field of cross-platform application development is state management. If managed efficiently, apps can perform exceptionally with a smooth UI experience. That's where Flutter RxDart steps in, facilitating state management in a more effortless and streamlined manner.Importance of RxDart in State ManagementRxDart brings the power of streams and reactive programming to Flutter, making state management a breeze. States in Flutter equate to the values that can change over time. Guess what? That's exactly what streams are all about! A vital concept in this scenario is BehaviorSubject. As discussed before, BehaviorSubject is a special type of stream provided by RxDart that holds the most recent value, and it can be accessed synchronously.Implementing StreamBuilder with RxDartFlutter provides a handy out-of-the-box widget called StreamBuilder that automatically registers a listener to a Stream and invokes the builder whenever an event is emitted. StreamBuilder and RxDart harmoniously work hand-in-hand to create a reactive Flutter application.Let's look at a simple Flutter RxDart usage with StreamBuilder:1 import 'package:flutter/material.dart';2 import 'package:rxdart/rxdart.dart';3 4 void main() {5 runApp(MaterialApp(home: MyApp()));6 }7 8 class MyApp extends StatelessWidget {9 final BehaviorSubject _subject = BehaviorSubject.seeded("Hello, RxDart!");10 11 @override12 Widget build(BuildContext context) {13 return Scaffold(14 appBar: AppBar(15 title: Text('RxDart with StreamBuilder'),16),17 body: Padding(18 padding: EdgeInsets.all(16.0),19 child: StreamBuilder(20 stream: _subject.stream,21 builder: (context, snapshot) {22 if (snapshot.hasData) {23 return Text(24 snapshot.data!,25 style: Theme.of(context).textTheme.headline4,26);27 } else {28 return CircularProgressIndicator();29 }30 },31),32),33);34 }35 }36 In the example above, a StreamBuilder is implemented that listens to the _subject stream. Whenever data is added to the stream, it automatically builds the widget.Dealing with Backpressure in Flutter using RxDartWhen working with asynchronous programming, specifically streams of data, one common issue that might arise is backpressure. It's a condition where the stream is producing data faster than its consumer (subscriber) can handle. Backpressure can lead to performance issues or even crashes.RxDart introduces several operators that assist in managing backpressure scenarios in Flutter applications. Let's check out an example using the debounceTime operator to tackle backpressure:1 // Keyboard input stream that emits every keyup event2 final inputStream = querySelector('#input')!.onKeyUp;3 4 inputStream5 .map((event) => (event.currentTarget as InputElement).value)6 .debounceTime(Duration(milliseconds: 200)) // RxDart extension method7 .listen((value) => print('User is typing: $value'));8 In this example, if the user is typing too fast, we don't want to handle every single keyup event as it may put an unwanted load on our application. Thus, we debounce the keyup stream and listen to it only if the user hasn't typed anything for the last 200 milliseconds. This way, RxDart helps us in managing backpressure.Advanced RxDart Concepts for Flutter developersRxDart goes beyond just managing asynchronous data streams. It also supports advanced features like combining, merging, and switching between different streams which can be useful for complex Flutter applications.Advanced RxDart Concepts for Flutter developersCombining, Merging, and Switching between StreamsLet's look at some examples for these advanced concepts:Combining StreamsIn RxDart, you can combine multiple streams into one stream that will emit all values from every given stream in order of subscription. Let's look at how to apply the CombineLatestStream operator to two streams:1 final firstStream = Stream.fromIterable([1, 2]);2 final secondStream = Stream.fromIterable(['A', 'B']);3 4 Rx.combineLatest2(firstStream, secondStream, (a, b) => '$a $b')5 .listen(print); // prints '1 A', '2 B'6 Merging StreamsMergeStream merges multiple streams into one stream that emits all data from the given streams in the exact order they were emitted.1 final firstStream = Stream.fromIterable([1, 2]);2 final secondStream = Stream.fromIterable(['A', 'B']);3 4 Rx.merge([firstStream, secondStream])5 .listen(print); // prints 1, 2, A, B6 Switching Between StreamsSwitchLatestStream takes a Stream of Streams (high-order Stream) as input and always emits values from the most recently provided Stream.1 final triggerSwitch = PublishSubject();2 final firstStream = Rx.timer('A', Duration(seconds: 3));3 final secondStream = Rx.timer('B', Duration(seconds: 1));4 5 triggerSwitch6 .switchMap((value) => value == 1 ? firstStream : secondStream)7 .listen(print); // if switch is triggered before 3 seconds, it will print B8 9 triggerSwitch.add(1);10 11 // After 2 seconds, trigger to switch to the faster stream12 Future.delayed(Duration(seconds: 2), () => triggerSwitch.add(2));13 One of the strong suits of RxDart that Flutter developers can capitalize on is these advanced functionalities which can help in handling complex Flutter apps elegantly and effectively.Issues and Solutions in Using RxDart with FlutterWhile using RxDart with Flutter, developers can face some challenges. But don't worry, we're here to address these common pitfalls and provide concise solutions.Common Mistakes and How to Prevent ThemNot Closing Streams: One common mistake is not closing streams when they're no longer needed. This can lead to memory leaks. Just like opening a Stream, it's important to close them too.Solution: Always remember to close your streams, typically in dispose method in Flutter:1 BehaviorSubject counter;2 3 @override4 void dispose() {5 counter.close();6 super.dispose();7 }8 Not Handling Stream Errors: When dealing with streams, error handling is often overlooked. Your stream might throw an error in certain cases, and if not caught, it can result in a crash.Solution: Always wrap your stream in a try-catch block or use onError as a method to handle errors gracefully:1 stream.listen(2 (data) { /* handle data */ },3 onError: (error) { /* handle error */ },4 onDone: () { /* handle done */ },5);6 Debugging RxDart ApplicationsDebugging RxDart applications can be challenging due to the asynchronous nature of streams. However, Dart provides a debugging utility to Dump Stack Traces that helps in debugging.Ideally, you would need to handle errors gracefully and write tests for error scenarios to ensure better debugging and error handling in your Flutter RxDart applications.Real-World Example: Reading Konami Code with RxDartTo fully understand the potential of RxDart in Flutter, let's see how we can use it in a real-world scenario. In this scenario, we'll be reading the Konami Code as user keyboard input. For those unfamiliar, the Konami Code is a secret code sequence historically used in video games, almost like an easter egg!Building an App with RxDartFirst, let's import RxDart and define the ASCII values that correspond to the key codes in the Konami Code:1 import 'package:rxdart/rxdart.dart';2 3 const konamiKeyCodes = const [4 KeyCode.UP,5 KeyCode.UP,6 KeyCode.DOWN,7 KeyCode.DOWN,8 KeyCode.LEFT,9 KeyCode.RIGHT,10 KeyCode.LEFT,11 KeyCode.RIGHT,12 KeyCode.B,13 KeyCode.A14];15 Next, we need to listen for key-up events. We'll use RxDart to buffer the last ten key codes, and then check if they match the Konami Code sequence:1 document.onKeyUp2 .map((event) => event.keyCode)3 .bufferCount(10, 1)4 .where((lastTenKeyCodes) => const IterableEquality().equals(lastTenKeyCodes, konamiKeyCodes))5 .listen((_) => result.innerHtml = 'KONAMI!')6 In this example, we are listening for keyup events, transforming the event to emit only the key code, buffering the last ten emitted key codes, and then checking to see if the last ten key codes match the Konami Code!This example demonstrates how elegantly RxDart handles complex asynchronous data sequences allowing Flutter developers to create robust and efficient applications.Upgrading from RxDart 0.22.x to 0.23.xUpgrades are required as technology advances, and we must adapt our code to support these modifications. The same is true for RxDart. The Observable class has been deprecated since the release of RxDart 0.23.x, and Dart 2.6's extension methods are being utilized instead.In order to upgrade your code to support the latest version, RxDart provides an automatic upgrade solution using the rxdart_codemod package.To automatically update your code, follow the instructions included with the rxdart_codemod package. Simply running the package will assist in refactoring the code to support RxDart 0.23.x.This way, you can ensure your code is always up-to-speed with the latest updates of RxDart in Flutter and harness the benefits of new features and performance improvements.Future of RxDart in FlutterThroughout this post, we traversed the diverse landscape of RxDart in Flutter. We started from the basics, went through advanced concepts, and also looked at a real-world example to cap it off.Future of RxDart in FlutterWith the ever-growing Flutter ecosystem and the consistent evolution of reactive programming, RxDart has a promising future. Being capable of providing more refined solutions for handling asynchronous data streams in Flutter, RxDart is bound to become even more popular among Flutter developers.Start Reactive Programming with Flutter RxDart!RxDart captures the essence of reactive programming and integrates smoothly with Flutter, making it an influential asset for mobile app developers. It embraces Dart Streams' power and enhances them with additional classes, methods, and functionalities that streamlined asynchronous stream management. With mastery over RxDart, you can create reactive applications in Flutter that are robust, efficient, and provide a smooth user experience.Remember, regardless of the technology you work with, constant learning and practising are the keys to becoming a proficient software developer. Be it Flutter RxDart or any other tech stack, keep exploring and keep coding!Your journey into the world of reactive programming with Flutter RxDart starts here. Good luck!1. What is BLoC in Flutter?BLoC stands for Business Logic Component. It helps separate business logic from the UI using streams.2. How does RxDart enhance BLoC?RxDart provides advanced stream manipulation capabilities, improving BLoCs functionality.3. What are the core concepts of RxDart?Streams, Sinks, Subjects.4. How do I set up BLoC in a Flutter project?Define BLoC classes, create events and states, and use streams to manage state.5. What is the purpose of StreamTransformers in BLoC?They help in transforming and manipulating stream data efficiently.6. How can I test BLoC classes?Use Flutters testing framework to check BLoCs stream outputs and event handling.7. What are some common mistakes with BLoC and RxDart?Avoid managing too many streams and ensure streams are properly disposed of.8. How can I debug BLoC streams?Use debugging tools and monitor streams to ensure they emit the correct states.9. What are the benefits of using BLoC?Improved separation of concerns and easier state management.10. How does BLoC improve app performance?By managing the state efficiently and reducing UI rebuilds.11. Can I use BLoC without RxDart?Yes, but RxDart adds powerful features for stream manipulation.12. What are the best practices for using BLoC?Keep BLoC classes focused, manage streams carefully, and test thoroughly.Learning Resources for Flutter State ManagementOfficial Flutter Documentation: State ManagementFlutter BLoC Library: BLoC PackageRxDart Documentation: RxDart GuideOnline Tutorials: State Management with BLoCVideo Courses: Flutter State ManagementReactive programming is programming with asynchronous observable streams. In Dart, stream provide an asynchronous sequence of data. In this first part, we will be looking on how to do reactive programming for our flutter applications by extending the capabilities of dart stream with RxDart using its Subjects and Stream Classes. In next part we will check the Extension Methods on the Stream Classes aka Operators of Rx.tPhoto by Jordan on UnsplashIntroductionDart streams were developed with reactive programming in mind, Instead of providing an alternative to it, RxDart adds additional functionality using the reactive approach on top of it.SetupAdd rxdart to your flutter projects pubspec.yaml file:dependencies: rxdart:Now in our code, we can use it by importing it:import 'package:rxdart/rxdart.dart';RxDart does not provide Rxs main class Observable as alternative for Dart Streams. Instead, it offers several additional Subjects, Stream Classes and Extension Methods on the Stream Classes (Operators of Rx). lets look over it,SubjectsIn Dart, if weve to send data, error and done events to its stream we use StreamController but, in RxDart we have to use Subject which is the same as StreamController but with additional stuff.all the subjects of RxDart is similar to broadcast StreamController which means we can listened to the subjects stream as many time as we want.BehaviorSubjectIn BehaviorSubject the most recent item that we added to our subject is dispatched to its new listeners. When we listen to our new listener it will receive the latest stored item from the subject and after that new event will be sent to all other listeners. lets see with example,We can also assign an initial value to our BehaviorSubject. like this,PublishSubjectIn PublishSubject all the items that we added to our subject is dispatched to its listeners. this is most easiest subject among other subjects provided by rxdart.for this, the sequence of listener matters for adding and listening to the items of PublishSubject. lets see with example,ReplaySubjectIn ReplaySubject all the items we added to our subject is dispatched to its listeners doesnt matter the sequence when we add first or listen first to that items of ReplaySubject, unlike PublishSubject and BehaviorSubject. lets see with example,We can also specify a maxSize value at the time of initializing the ReplaySubject variable. this means we can only listen to the number of latest items that equal to the maxSize. like this,But maxSize did not work when we added items later / or after the listener.The Stream Class is a source of asynchronous data events. RxDart gives us different ways to create a Stream with Stream Classes that have additional capabilities to create a variety of tasks as per our requirement, such as combining or merging Streams!!We can create the Streams provided by RxDart in two ways. by instantiating the stream class directly like this,final mergedStream = MergeStream([firstStream, secondStream]);and using static factories from the Rx class like this,final mergedStream = Rx.merge([firstStream, secondStream]);There are so many Stream Classes are provided by RxDart lets see them with example,CombineLatestStream : This stream joins all the given streams into one single stream sequence by combining them when any of the source stream sequences send forth an item. this stream will emit items after all others streams that we want to combine have emitted at least one item. If any of the streams that we want to combine is empty then the resulting sequence completes instantly without emitting any items.ConcatStream : This stream is used when we want to concatenates all stream sequences. ConcatStream concat the next stream sequence after the previous stream sequence is terminated successfully. In case where the streams is empty then its completes instantly without emitting any items.ConcatEagerStream : This stream is similar with ConcatStream only the difference is that instead of subscribing to stream one by one, all the streams are immediately subscribed with correct time.DeferStream : This stream is used when we want to constructs a stream lazily when we subscribes to it. In some case we have to wait until the last minute to generate the stream that have latest data. its a single subscription stream but we can make it reusable.ForkJoinStream : This stream is used when we only want sequence that contains only final emitted value of each stream. in the case where any of the inner streams have some error then we will lose the value of any other streams that already completed if we not catch that error correctly on the inner stream.FromCallableStream : This stream is used when we want to return a stream that is based on the result of some function. the stream emits the value thats returned from that function.MergeStream : This stream is used when we have to flattens the items that emitted by the given streams into a single sequence of stream. In this the items is emitted one after another from all the given streams.NeverStream : This stream returns a non-terminating stream sequence, which can be used for infinite duration. this stream have very specific and limited behavior. as per my knowledge this is used for testing purposes only.RaceStream : This stream returns a stream that emits all of its items before any other streams emits its items.RangeStream : This stream class basically used when we want to returns a resulting stream that emits a sequence of integers within a particular range that we added.RepeatStream : This stream creates a stream that will recreate it self and re- listen to the source stream for the specified number of times until the stream terminates successfully. In case if we forget to specify the count then it repeats indefinitely.RetryStream : This stream is similar with RepeatStream only the difference is that if the retry count is not specified, it retries indefinitely and if the retry count is met, but the stream has not terminated successfully then all of the errors that caused the failure will be emitted at the end.RetryWhenStream : This stream is somehow similar with the RetryStream the difference is it will take two stream as an argument (streamFactory and retryWhenFactory). if the retryWhenFactory throws an error or returns a stream that emits an error then the original error will be emitted and then the error from retryWhenFactory will be emitted if it is not same as the original error.SequenceEqualStream : This stream is used to check that whether the two streams emit the same sequence of items or not.SwitchLatestStream : This stream is used when we only want the single streams items that emitted most recently from the multiple streams.TimerStream : This stream is useful when we want to emits an item after a some specific amount of time.UsingStream : This stream is used to create a way so we can instruct an stream to create a resource for us that exists only during the lifetime of the stream and is disposed when the stream terminates.ZipStream : This stream is used to merge all the specified streams into a one single stream sequence using the zipper function whenever all of the stream sequences have produced an element at a corresponding index.Well done! Youve survived Part 1 of Reactive Programming Using RxDart in which we check RxDarts Subjects and Stream Classes.All explanation and example code in this part is based on rxdart version 0.27.1.Now youre ready to check Part 2 of Reactive Programming Using RxDart. In that we will check the Extension Methods on the Stream Classes aka Operators of Rx.

Rxdart flutter. Flutter dart define. Rxdart.