Core Concepts (package:bloc)

Please make sure to carefully read the following sections before working with package:bloc.

There are several core concepts that are critical to understanding how to use the bloc package.

In the upcoming sections, we're going to discuss each of them in detail as well as work through how they would apply to a counter app.

Streams

Check out the official Dart Documentation for more information about `Streams`.

> A stream is a sequence of asynchronous data.

In order to use the bloc library, it is critical to have a basic understanding of `Streams` and how they work.

> If you're unfamiliar with `Streams` just think of a pipe with water flowing through it. The pipe is the `Stream` and the water is the asynchronous data.

We can create a `Stream` in Dart by writing an `async*` (async generator) function.

```dart
Stream<int> countStream(int max) async* {
    for (int i = 0; i < max; i++) {
        yield i;
```

```
    }
}
```

By marking a function as `async*` we are able to use the `yield` keyword and return a `Stream` of data. In the above example, we are returning a `Stream` of integers up to the `max` integer parameter.

Every time we `yield` in an `async*` function we are pushing that piece of data through the `Stream`.

We can consume the above `Stream` in several ways. If we wanted to write a function to return the sum of a `Stream` of integers it could look something like:

```dart
Future<int> sumStream(Stream<int> stream) async {
    int sum = 0;
    await for (int value in stream) {
        sum += value;
    }
    return sum;
}
```

By marking the above function as `async` we are able to use the `await` keyword and return a `Future` of integers. In this example, we are awaiting each value in the stream and returning the sum of all integers in the stream.

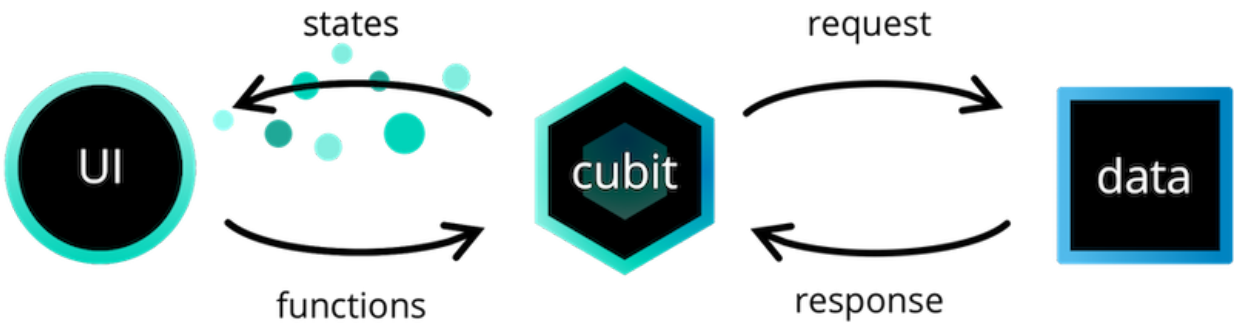
We can put it all together like so:

```dart
void main() async {
    /// Initialize a stream of integers 0-9
    Stream<int> stream = countStream(10);
    /// Compute the sum of the stream of integers
    int sum = await sumStream(stream);
    /// Print the sum
    print(sum); // 45
}
```

Now that we have a basic understanding of how `Streams` work in Dart we're ready to learn about the core component of the bloc package: a `Cubit`.

Cubit

> A `Cubit` is a class which extends `BlocBase` and can be extended to manage any type of state.



A `Cubit` can expose functions which can be invoked to trigger state changes.

> States are the output of a `Cubit` and represent a part of your application's state. UI components can be notified of states and redraw portions of themselves based on the current state.

> **Note**: For more information about the origins of `Cubit` checkout the following issue.

Creating a Cubit

We can create a `CounterCubit` like:

```dart
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);
}
```

When creating a `Cubit`, we need to define the type of state which the `Cubit` will be managing. In the case of the `CounterCubit` above, the state can be represented via an `int` but in more complex cases it might be necessary to use a `class` instead of a primitive type.

The second thing we need to do when creating a `Cubit` is specify the initial state. We can do this by calling `super` with the value of the initial state. In the snippet above, we are setting the initial state to `0` internally but we can also allow the `Cubit` to be more flexible by accepting an external value:

```dart
class CounterCubit extends Cubit<int> {
  CounterCubit(int initialState) : super(initialState);
}
```

This would allow us to instantiate `CounterCubit` instances with different initial states like:

```dart
final cubitA = CounterCubit(0); // state starts at 0
final cubitB = CounterCubit(10); // state starts at 10
```

State Changes

> Each `Cubit` has the ability to output a new state via `emit`.

```dart
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increment() => emit(state + 1);
}
```

In the above snippet, the `CounterCubit` is exposing a public method called `increment` which can be called externally to notify the `CounterCubit` to increment its state. When `increment` is called, we can access the current state of the `Cubit` via the `state` getter and `emit` a new state by adding 1 to the current state.

> The `emit` method is protected, meaning it should only be used inside of a `Cubit`.

Using a Cubit

We can now take the `CounterCubit` we've implemented and put it to use!

Basic Usage

```dart
void main() {
  final cubit = CounterCubit();
  print(cubit.state); // 0
  cubit.increment();
  print(cubit.state); // 1
  cubit.close();
}
```

In the above snippet, we start by creating an instance of the `CounterCubit`. We then print the current state of the cubit which is the initial state (since no new states have been emitted yet). Next, we call the `increment` function to trigger a state change. Finally, we print the state of the `Cubit` again which went from `0` to `1` and close the `Cubit` to close the internal state stream.

Stream Usage

Since a `Cubit` is a special type of `Stream`, we can also subscribe to a `Cubit` for real-time updates to its state:

```dart
Future<void> main() async {
  final cubit = CounterCubit();
  final subscription = cubit.stream.listen(print); // 1
  cubit.increment();
  await Future.delayed(Duration.zero);
  await subscription.cancel();
  await cubit.close();
}
```

In the above snippet, we are subscribing to the `CounterCubit` and calling print on each state change. We are then invoking the `increment` function which will emit a

new state. Lastly, we are calling `cancel` on the `subscription` when we no longer want to receive updates and closing the `Cubit`.

> **Note**: `await Future.delayed(Duration.zero)` is added for this example to avoid canceling the subscription immediately.

> ! Only subsequent state changes will be received when calling `listen` on a `Cubit`.

Observing a Cubit

> When a `Cubit` emits a new state, a `Change` occurs. We can observe all changes for a given `Cubit` by overriding `onChange`.

```dart
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increment() => emit(state + 1);

  @override
  void onChange(Change<int> change) {
    print(change);
    super.onChange(change);
  }
}
```

We can then interact with the `Cubit` and observe all changes output to the console.

```dart
void main() {
  CounterCubit()
    ..increment()
    ..close();
}
```

The above example would output:

```sh
Change { currentState: 0, nextState: 1 }
```

Note: A `Change` occurs just before the state of the `Cubit` is updated. A `Change` consists of the `currentState` and the `nextState`.

BlocObserver

One added bonus of using the bloc library is that we can have access to all `Changes` in one place. Even though in this application we only have one `Cubit`, it's fairly common in larger applications to have many `Cubits` managing different parts of the application's state.

If we want to be able to do something in response to all `Changes` we can simply create our own `BlocObserver`.

```dart
class SimpleBlocObserver extends BlocObserver {
  @override
  void onChange(BlocBase bloc, Change change) {
    super.onChange(bloc, change);
    print('${bloc.runtimeType} $change');
  }
}
```

Note: All we need to do is extend `BlocObserver` and override the `onChange` method.

In order to use the `SimpleBlocObserver`, we just need to tweak the `main` function:

```dart
void main() {
  Bloc.observer = SimpleBlocObserver();
  CounterCubit()
    ..increment()
```

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

The above snippet would then output:

```sh
Change { currentState: 0, nextState: 1 }
CounterCubit Change { currentState: 0, nextState: 1 }
```

Note: The internal `onChange` override is called first, followed by `onChange` in `BlocObserver`.

Tip: In `BlocObserver` we have access to the `Cubit` instance in addition to the `Change` itself.

Error Handling

> Every `Cubit` has an `addError` method which can be used to indicate that an error has occurred.

```dart
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increment() {
    addError(Exception('increment error!'), StackTrace.current);
    emit(state + 1);
  }

  @override
  void onChange(Change<int> change) {
    super.onChange(change);
    print(change);
  }

  @override
```

```
  void onError(Object error, StackTrace stackTrace) {
    print('$error, $stackTrace');
    super.onError(error, stackTrace);
  }
}
```

Note: `onError` can be overridden within the `Cubit` to handle all errors for a specific `Cubit`.

`onError` can also be overridden in `BlocObserver` to handle all reported errors globally.

```dart
class SimpleBlocObserver extends BlocObserver {
  @override
  void onChange(BlocBase bloc, Change change) {
    super.onChange(bloc, change);
    print('${bloc.runtimeType} $change');
  }

  @override
  void onError(BlocBase bloc, Object error, StackTrace stackTrace) {
    print('${bloc.runtimeType} $error $stackTrace');
    super.onError(bloc, error, stackTrace);
  }
}
```

If we run the same program again we should see the following output:

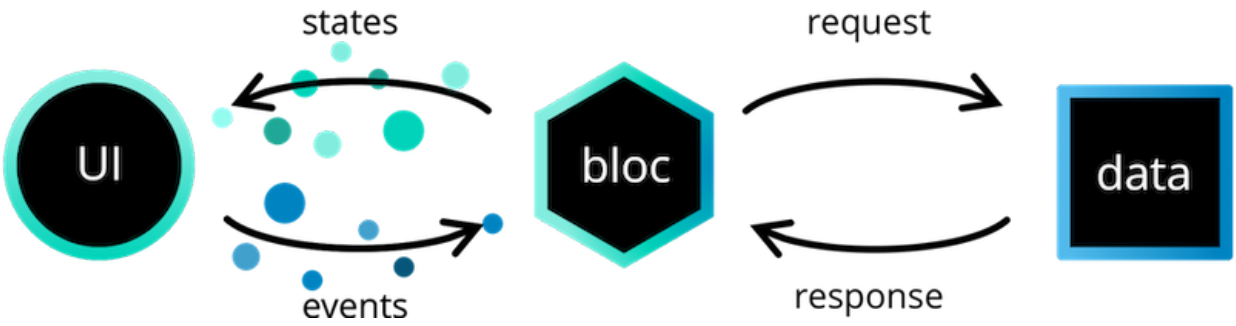
```sh
Exception: increment error!, #0      CounterCubit.increment (file:///main
#1      main (file:///main.dart:41:7)
#2      _startIsolate.<anonymous closure> (dart:isolate-patch/isolate_pat
#3      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_pa

CounterCubit Exception: increment error! #0      CounterCubit.increment (
#1      main (file:///main.dart:41:7)
#2      _startIsolate.<anonymous closure> (dart:isolate-patch/isolate_pat
#3      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_pa
```

```
Change { currentState: 0, nextState: 1 }
CounterCubit Change { currentState: 0, nextState: 1 }
```

Note: Just as with `onChange`, the internal `onError` override is invoked before the global `BlocObserver` override.

Bloc

> A `Bloc` is a more advanced class which relies on `events` to trigger `state` changes rather than functions. `Bloc` also extends `BlocBase` which means it has a similar public API as `Cubit`. However, rather than calling a `function` on a `Bloc` and directly emitting a new `state`, `Blocs` receive `events` and convert the incoming `events` into outgoing `states`.



Creating a Bloc

Creating a `Bloc` is similar to creating a `Cubit` except in addition to defining the state that we'll be managing, we must also define the event that the `Bloc` will be able to process.

> Events are the input to a Bloc. They are commonly added in response to user interactions such as button presses or lifecycle events like page loads.

```dart
enum CounterEvent { increment }
```

```
class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);
}
```

Just like when creating the `CounterCubit`, we must specify an initial state by passing it to the superclass via `super`.

State Changes

Unlike using `CounterCubit` directly where we define functions to trigger state changes, using `Bloc` requires us to instead override `mapEventToState`. This will be responsible for converting any incoming events into one or more outgoing states.

dart

```
enum CounterEvent { increment }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {}
}
```

Tip: `async*` means the function is an **async generator** which is capable of emitting states via the `yield` keyword.

We can then update `mapEventToState` to handle the `CounterEvent.increment` event:

dart

```
enum CounterEvent { increment }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
```

```
    switch (event) {
      case CounterEvent.increment:
        yield state + 1;
        break;
    }
  }
}
```

In the above snippet, we are switching on the incoming event and if it is an increment event, we are yielding a new state (similar to `emit`).

> **Note**: Since the `Bloc` class extends `Cubit`, we have access to the current state of the bloc at any point in time via the `state` getter.

> Blocs should never directly `emit` new states. Instead every state change must be output in response to an incoming event within `mapEventToState`.

> Both blocs and cubits will ignore duplicate states. If we yield or emit `State nextState` where `state == nextState`, then no state change will occur.

Using a Bloc

At this point, we can create an instance of our `CounterBloc` and put it to use!

Basic Usage

```dart
Future<void> main() async {
  final bloc = CounterBloc();
  print(bloc.state); // 0
  bloc.add(CounterEvent.increment);
  await Future.delayed(Duration.zero);
  print(bloc.state); // 1
  await bloc.close();
}
```

In the above snippet, we start by creating an instance of the `CounterBloc`. We then print the current state of the `Bloc` which is the initial state (since no new states have been emitted yet). Next, we add the increment event to trigger a state change. Finally, we print the state of the `Bloc` again which went from 0 to 1 and close the `Bloc` to close the internal state stream.

> **Note**: `await Future.delayed(Duration.zero)` is added to ensure we wait for the next event-loop iteration (allowing `mapEventToState` to process the increment event).

Stream Usage

Just like with `Cubit`, a `Bloc` is a special type of `Stream`, which means we can also subscribe to a `Bloc` for real-time updates to its state:

```dart
Future<void> main() async {
  final bloc = CounterBloc();
  final subscription = bloc.stream.listen(print); // 1
  bloc.add(CounterEvent.increment);
  await Future.delayed(Duration.zero);
  await subscription.cancel();
  await bloc.close();
}
```

In the above snippet, we are subscribing to the `CounterBloc` and calling print on each state change. We are then adding the increment event which triggers `mapEventToState` and yields a new state. Lastly, we are calling cancel on the subscription when we no longer want to receive updates and closing the `Bloc`.

> **Note**: `await Future.delayed(Duration.zero)` is added for this example to avoid canceling the subscription immediately.

Observing a Bloc

Since `Bloc` extends `Cubit` (meaning all blocs are also cubits), we can observe all state changes for a `Bloc` using `onChange`.

```dart
enum CounterEvent { increment }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.increment:
        yield state + 1;
        break;
    }
  }

  @override
  void onChange(Change<int> change) {
    print(change);
    super.onChange(change);
  }
}
```

We can then update `main.dart` to:

```dart
void main() {
  CounterBloc()
    ..add(CounterEvent.increment)
    ..close();
}
```

Now if we run the above snippet, the output will be:

```sh
Change { currentState: 0, nextState: 1 }
```

One key differentiating factor between `Bloc` and `Cubit` is that because `Bloc` is event-driven, we are also able to capture information about what triggered the state change.

We can do this by overriding `onTransition`.

> The change from one state to another is called a `Transition`. A `Transition` consists of the current state, the event, and the next state.

```dart
enum CounterEvent { increment }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.increment:
        yield state + 1;
        break;
    }
  }

  @override
  void onChange(Change<int> change) {
    super.onChange(change);
    print(change);
  }

  @override
  void onTransition(Transition<CounterEvent, int> transition) {
    super.onTransition(transition);
    print(transition);
  }
}
```

If we then rerun the same `main.dart` snippet from before, we should see the following output:

```sh
Transition { currentState: 0, event: CounterEvent.increment, nextState: 1
Change { currentState: 0, nextState: 1 }
```

Note: `onTransition` is invoked before `onChange` and contains the event which triggered the change from `currentState` to `nextState`.

BlocObserver

Just as before, we can override `onTransition` in a custom `BlocObserver` to observe all transitions that occur from a single place.

```dart
class SimpleBlocObserver extends BlocObserver {
  @override
  void onChange(BlocBase bloc, Change change) {
    super.onChange(bloc, change);
    print('${bloc.runtimeType} $change');
  }

  @override
  void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    print('${bloc.runtimeType} $transition');
  }

  @override
  void onError(BlocBase bloc, Object error, StackTrace stackTrace) {
    print('${bloc.runtimeType} $error $stackTrace');
    super.onError(bloc, error, stackTrace);
  }
}
```

We can initialize the `SimpleBlocObserver` just like before:

```dart
void main() {
  Bloc.observer = SimpleBlocObserver();
```

```
  CounterBloc()
    ..add(CounterEvent.increment)
    ..close();
}
```

Now if we run the above snippet, the output should look like:

```sh
Transition { currentState: 0, event: CounterEvent.increment, nextState: 1
CounterBloc Transition { currentState: 0, event: CounterEvent.increment,
Change { currentState: 0, nextState: 1 }
CounterBloc Change { currentState: 0, nextState: 1 }
```

Note: `onTransition` is invoked first (local before global) followed by `onChange`.

Another unique feature of `Bloc` instances is that they allow us to override `onEvent` which is called whenever a new event is added to the `Bloc`. Just like with `onChange` and `onTransition`, `onEvent` can be overridden locally as well as globally.

```dart
enum CounterEvent { increment }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.increment:
        yield state + 1;
        break;
    }
  }

  @override
  void onEvent(CounterEvent event) {
    super.onEvent(event);
```

```
    print(event);
  }

  @override
  void onChange(Change<int> change) {
    super.onChange(change);
    print(change);
  }

  @override
  void onTransition(Transition<CounterEvent, int> transition) {
    super.onTransition(transition);
    print(transition);
  }
}
```

dart

```
class SimpleBlocObserver extends BlocObserver {
  @override
  void onEvent(Bloc bloc, Object? event) {
    super.onEvent(bloc, event);
    print('${bloc.runtimeType} $event');
  }

  @override
  void onChange(BlocBase bloc, Change change) {
    super.onChange(bloc, change);
    print('${bloc.runtimeType} $change');
  }

  @override
  void onTransition(Bloc bloc, Transition transition) {
    super.onTransition(bloc, transition);
    print('${bloc.runtimeType} $transition');
  }
}
```

We can run the same `main.dart` as before and should see the following output:

```sh
CounterEvent.increment
CounterBloc CounterEvent.increment
Transition { currentState: 0, event: CounterEvent.increment, nextState: 1
CounterBloc Transition { currentState: 0, event: CounterEvent.increment,
Change { currentState: 0, nextState: 1 }
CounterBloc Change { currentState: 0, nextState: 1 }
```

Note: `onEvent` is called as soon as the event is added. The local `onEvent` is invoked before the global `onEvent` in `BlocObserver`.

Error Handling

Just like with `Cubit`, each `Bloc` has an `addError` and `onError` method. We can indicate that an error has occurred by calling `addError` from anywhere inside our `Bloc`. We can then react to all errors by overriding `onError` just as with `Cubit`.

```dart
enum CounterEvent { increment }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.increment:
        addError(Exception('increment error!'), StackTrace.current);
        yield state + 1;
        break;
    }
  }

  @override
  void onChange(Change<int> change) {
    print(change);
    super.onChange(change);
  }
```

```
  @override
  void onTransition(Transition<CounterEvent, int> transition) {
    print(transition);
    super.onTransition(transition);
  }

  @override
  void onError(Object error, StackTrace stackTrace) {
    print('$error, $stackTrace');
    super.onError(error, stackTrace);
  }
}
```

If we rerun the same `main.dart` as before, we can see what it looks like when an error is reported:

```sh
Exception: increment error!, #0      CounterBloc.mapEventToState (file://
<asynchronous suspension>
#1      Bloc._bindEventsToStates.<anonymous closure> (package:bloc/src/bl
#2      Stream.asyncExpand.onListen.<anonymous closure> (dart:async/strea
#3      _RootZone.runUnaryGuarded (dart:async/zone.dart:1374:10)
#4      _BufferingStreamSubscription._sendData (dart:async/stream_impl.da
#5      _DelayedData.perform (dart:async/stream_impl.dart:594:14)
#6      _StreamImplEvents.handleNext (dart:async/stream_impl.dart:710:11)
#7      _PendingEvents.schedule.<anonymous closure> (dart:async/stream_im
#8      _microtaskLoop (dart:async/schedule_microtask.dart:43:21)
#9      _startMicrotaskLoop (dart:async/schedule_microtask.dart:52:5)
#10     _runPendingImmediateCallback (dart:isolate-patch/isolate_patch.da
#11     _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_pa

CounterBloc Exception: increment error! #0      CounterBloc.mapEventToSta
<asynchronous suspension>
#1      Bloc._bindEventsToStates.<anonymous closure> (package:bloc/src/bl
#2      Stream.asyncExpand.onListen.<anonymous closure> (dart:async/strea
#3      _RootZone.runUnaryGuarded (dart:async/zone.dart:1374:10)
#4      _BufferingStreamSubscription._sendData (dart:async/stream_impl.da
#5      _DelayedData.perform (dart:async/stream_impl.dart:594:14)
#6      _StreamImplEvents.handleNext (dart:async/stream_impl.dart:710:11)
#7      _PendingEvents.schedule.<anonymous closure> (dart:async/stream_im
#8      _microtaskLoop (dart:async/schedule_microtask.dart:43:21)
#9      _startMicrotaskLoop (dart:async/schedule_microtask.dart:52:5)
```

```
#10     _runPendingImmediateCallback (dart:isolate-patch/isolate_patch.da
#11     _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_pa

Transition { currentState: 0, event: CounterEvent.increment, nextState: 1
CounterBloc Transition { currentState: 0, event: CounterEvent.increment,
Change { currentState: 0, nextState: 1 }
CounterBloc Change { currentState: 0, nextState: 1 }
```

Note: The local `onError` is invoked first followed by the global `onError` in `BlocObserver`.

Note: `onError` and `onChange` work the exact same way for both `Bloc` and `Cubit` instances.

! Any unhandled exceptions that occur within `mapEventToState` are also reported to `onError`.

Cubit vs. Bloc

Now that we've covered the basics of the `Cubit` and `Bloc` classes, you might be wondering when you should use `Cubit` and when you should use `Bloc`.

Cubit Advantages

Simplicity

One of the biggest advantages of using `Cubit` is simplicity. When creating a `Cubit`, we only have to define the state as well as the functions which we want to expose to change the state. In comparison, when creating a `Bloc`, we have to define the states, events, and the `mapEventToState` implementation. This makes `Cubit` easier to understand and there is less code involved.

Now let's take a look at the two counter implementations:

CounterCubit

```dart
class CounterCubit extends Cubit<int> {
  CounterCubit() : super(0);

  void increment() => emit(state + 1);
}
```

CounterBloc

```dart
enum CounterEvent { increment }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async* {
    switch (event) {
      case CounterEvent.increment:
        yield state + 1;
        break;
    }
  }
}
```

The `Cubit` implementation is a lot more concise and instead of defining events separately, the functions act like events. In addition, when using a `Cubit`, we don't have to use async generators (`async*`) or have a strong understanding of the `yield` and `yield*` keywords because we can simply call `emit` from anywhere in order to trigger a state change.

Bloc Advantages

Traceability

One of the biggest advantages of using `Bloc` is knowing the sequence of state changes as well as exactly what triggered those changes. For state that is critical

to the functionality of an application, it might be very beneficial to use a more event-driven approach in order to capture all events in addition to state changes.

A common use case might be managing `AuthenticationState`. For simplicity, let's say we can represent `AuthenticationState` via an `enum`:

```dart
enum AuthenticationState { unknown, authenticated, unauthenticated }
```

There could be many reasons as to why the application's state could change from `authenticated` to `unauthenticated`. For example, the user might have tapped a logout button and requested to be signed out of the application. On the other hand, maybe the user's access token was revoked and they were forcefully logged out. When using `Bloc` we can clearly trace how the application state got to a certain state.

```sh
Transition {
  currentState: AuthenticationState.authenticated,
  event: LogoutRequested,
  nextState: AuthenticationState.unauthenticated
}
```

The above `Transition` gives us all the information we need to understand why the state changed. If we had used a `Cubit` to manage the `AuthenticationState`, our logs would look like:

```sh
Change {
  currentState: AuthenticationState.authenticated,
  nextState: AuthenticationState.unauthenticated
}
```

This tells us that the user was logged out but it doesn't explain why which might be critical to debugging and understanding how the state of the application is changing over time.

Advanced ReactiveX Operations

Another area in which `Bloc` excels over `Cubit` is when we need to take advantage of reactive operators such as `buffer`, `debounceTime`, `throttle`, etc.

`Bloc` has an event sink that allows us to control and transform the incoming flow of events.

For example, if we were building a real-time search, we would probably want to debounce the requests to the backend in order to avoid getting rate-limited as well as to cut down on cost/load on the backend.

With `Bloc` we can override `transformEvents` to change the way incoming events are processed by the `Bloc`.

```dart
@override
Stream<Transition<CounterEvent, int>> transformEvents(
  Stream<CounterEvent> events,
  TransitionFunction<CounterEvent, int> transitionFn,
) {
  return super.transformEvents(
    events.debounceTime(const Duration(milliseconds: 300)),
    transitionFn,
  );
}
```

With the above code, we can easily debounce the incoming events with very little additional code.

> **Tip**: If you are still unsure about which to use, start with `Cubit` and you can later refactor or scale-up to a `Bloc` as needed.

NEXT >
package:flutter_bloc