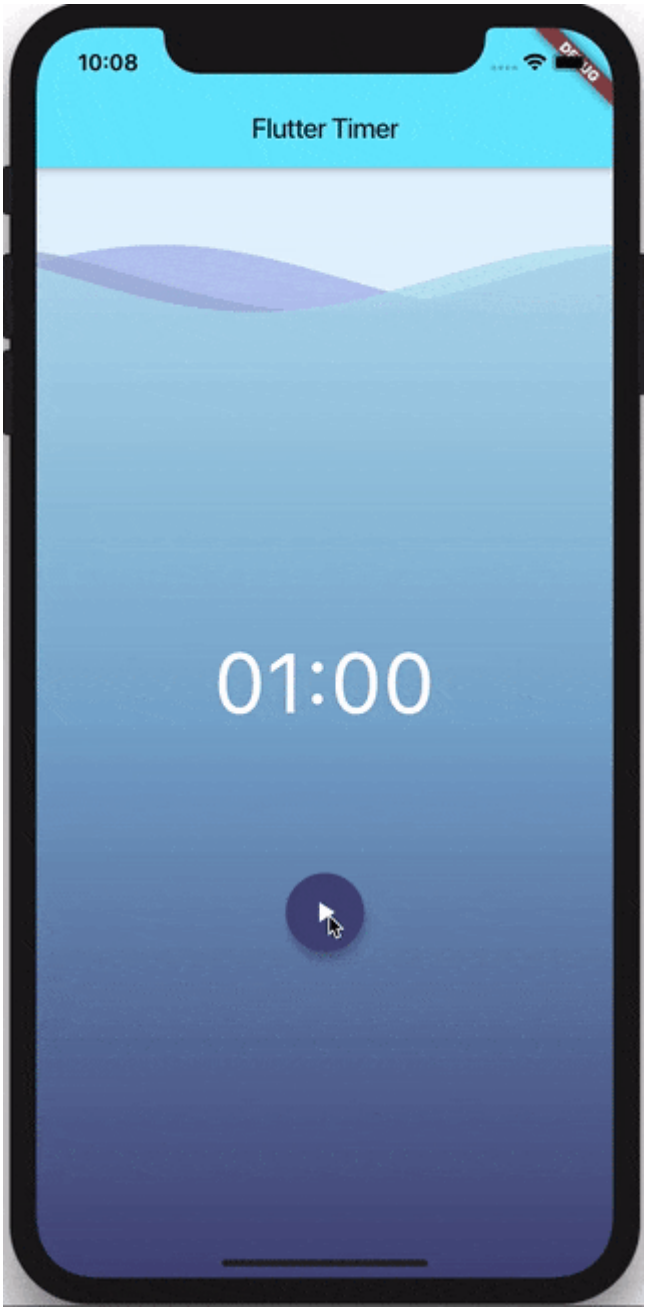
Flutter Timer Tutorial

level beginner

> In the following tutorial we're going to cover how to build a timer application using the bloc library. The finished application should look like this:



Key Topics

- Observe state changes with BlocObserver.
- BlocProvider, Flutter widget which provides a bloc to its children.
- BlocBuilder, Flutter widget that handles building the widget in response to new states.
- Using Cubit instead of Bloc. What's the difference?
- Prevent unnecessary rebuilds with Equatable.
- Learn to use `StreamSubscription` in a Bloc.
- Prevent unnecessary rebuilds with `buildWhen`.

Setup

We'll start off by creating a brand new Flutter project

```sh
flutter create flutter_timer
```

We can then replace the contents of pubspec.yaml with:

```yaml
name: flutter_timer
description: A new Flutter project.

version: 1.0.0+1

environment:
  sdk: ">=2.6.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  flutter_bloc: ^5.0.0
  equatable: ^1.0.0
  wave: ^0.0.8

flutter:
  uses-material-design: true
```

Note: We'll be using the flutter_bloc, equatable, and wave packages in this app.

Next, run `flutter packages get` to install all the dependencies.

Ticker

> The ticker will be our data source for the timer application. It will expose a stream of ticks which we can subscribe and react to.

Start off by creating `ticker.dart`.

```dart
class Ticker {
  Stream<int> tick({int ticks}) {
    return Stream.periodic(Duration(seconds: 1), (x) => ticks - x - 1)
        .take(ticks);
  }
}
```

All our `Ticker` class does is expose a tick function which takes the number of ticks (seconds) we want and returns a stream which emits the remaining seconds every second.

Next up, we need to create our `TimerBloc` which will consume the `Ticker`.

Timer Bloc

TimerState

We'll start off by defining the `TimerStates` which our `TimerBloc` can be in.

Our `TimerBloc` state can be one of the following:

- TimerInitial — ready to start counting down from the specified duration.
- TimerRunInProgress—actively counting down from the specified duration.
- TimerRunPause—paused at some remaining duration.

- TimerRunComplete—completed with a remaining duration of 0.

Each of these states will have an implication on what the user sees. For example:

- if the state is `TimerInitial` the user will be able to start the timer.
- if the state is `TimerRunInProgress` the user will be able to pause and reset the timer as well as see the remaining duration.
- if the state is `TimerRunPause` the user will be able to resume the timer and reset the timer.
- if the state is `TimerRunComplete` the user will be able to reset the timer.

In order to keep all of our bloc files together, let's create a bloc directory with `bloc/timer_state.dart`.

> **Tip:** You can use the IntelliJ or VSCode extensions to autogenerate the following bloc files for you.

```dart
import 'package:equatable/equatable.dart';

abstract class TimerState extends Equatable {
  final int duration;

  const TimerState(this.duration);

  @override
  List<Object> get props => [duration];
}

class TimerInitial extends TimerState {
  const TimerInitial(int duration) : super(duration);

  @override
  String toString() => 'TimerInitial { duration: $duration }';
}

class TimerRunPause extends TimerState {
  const TimerRunPause(int duration) : super(duration);

  @override
  String toString() => 'TimerRunPause { duration: $duration }';
}
```

```
class TimerRunInProgress extends TimerState {
  const TimerRunInProgress(int duration) : super(duration);

  @override
  String toString() => 'TimerRunInProgress { duration: $duration }';
}

class TimerRunComplete extends TimerState {
  const TimerRunComplete() : super(0);
}
```

Note that all of the `TimerStates` extend the abstract base class `TimerState` which has a duration property. This is because no matter what state our `TimerBloc` is in, we want to know how much time is remaining.

Next up, let's define and implement the `TimerEvents` which our `TimerBloc` will be processing.

TimerEvent

Our `TimerBloc` will need to know how to process the following events:

- TimerStarted—informs the TimerBloc that the timer should be started.
- TimerPaused—informs the TimerBloc that the timer should be paused.
- TimerResumed—informs the TimerBloc that the timer should be resumed.
- TimerReset—informs the TimerBloc that the timer should be reset to the original state.
- TimerTicked—informs the TimerBloc that a tick has occurred and that it needs to update its state accordingly.

If you didn't use the IntelliJ or VSCode extensions, then create `bloc/timer_event.dart` and let's implement those events.

```dart
import 'package:equatable/equatable.dart';
import 'package:meta/meta.dart';

abstract class TimerEvent extends Equatable {
  const TimerEvent();
```

```
  @override
  List<Object> get props => [];
}

class TimerStarted extends TimerEvent {
  final int duration;

  const TimerStarted({@required this.duration});

  @override
  String toString() => "TimerStarted { duration: $duration }";
}

class TimerPaused extends TimerEvent {}

class TimerResumed extends TimerEvent {}

class TimerReset extends TimerEvent {}

class TimerTicked extends TimerEvent {
  final int duration;

  const TimerTicked({@required this.duration});

  @override
  List<Object> get props => [duration];

  @override
  String toString() => "TimerTicked { duration: $duration }";
}
```

Next up, let's implement the `TimerBloc`!

TimerBloc

If you haven't already, create `bloc/timer_bloc.dart` and create an empty `TimerBloc`.

dart

```
import 'package:bloc/bloc.dart';
import 'package:flutter_timer/bloc/bloc.dart';
```

```
class TimerBloc extends Bloc<TimerEvent, TimerState> {
  // TODO: set initial state
  TimerBloc(): super();

  @override
  Stream<TimerState> mapEventToState(
    TimerEvent event,
  ) async* {
    // TODO: implement mapEventToState
  }
}
```

The first thing we need to do is define the initial state of our `TimerBloc`. In this case, we want the `TimerBloc` to start off in the `TimerInitial` state with a preset duration of 1 minute (60 seconds).

dart

```
import 'package:bloc/bloc.dart';
import 'package:flutter_timer/bloc/bloc.dart';

class TimerBloc extends Bloc<TimerEvent, TimerState> {
  static const int _duration = 60;

  TimerBloc() : super(TimerInitial(_duration));

  @override
  Stream<TimerState> mapEventToState(
    TimerEvent event,
  ) async* {
    // TODO: implement mapEventToState
  }
}
```

Next, we need to define the dependency on our `Ticker`.

dart

```
import 'dart:async';
import 'package:meta/meta.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_timer/bloc/bloc.dart';
```

```
import 'package:flutter_timer/ticker.dart';

class TimerBloc extends Bloc<TimerEvent, TimerState> {
  final Ticker _ticker;
  static const int _duration = 60;

  StreamSubscription<int> _tickerSubscription;

  TimerBloc({@required Ticker ticker})
      : assert(ticker != null),
        _ticker = ticker,
        super(TimerInitial(_duration));

  @override
  Stream<TimerState> mapEventToState(
    TimerEvent event,
  ) async* {
    // TODO: implement mapEventToState
  }
}
```

We are also defining a `StreamSubscription` for our `Ticker` which we will get to in a bit.

At this point, all that's left to do is implement `mapEventToState`. For improved readability, I like to break out each event handler into its own helper function. We'll start with the `TimerStarted` event.

dart

```
import 'dart:async';
import 'package:meta/meta.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_timer/bloc/bloc.dart';
import 'package:flutter_timer/ticker.dart';

class TimerBloc extends Bloc<TimerEvent, TimerState> {
  final Ticker _ticker;
  static const int _duration = 60;

  StreamSubscription<int> _tickerSubscription;

  TimerBloc({@required Ticker ticker})
```

```
        : assert(ticker != null),
          _ticker = ticker,
          super(TimerInitial(_duration));

  @override
  Stream<TimerState> mapEventToState(
    TimerEvent event,
  ) async* {
    if (event is TimerStarted) {
      yield* _mapTimerStartedToState(event);
    }
  }

  @override
  Future<void> close() {
    _tickerSubscription?.cancel();
    return super.close();
  }

  Stream<TimerState> _mapTimerStartedToState(TimerStarted start) async* {
     yield TimerRunInProgress(start.duration);
    _tickerSubscription?.cancel();
    _tickerSubscription = _ticker
        .tick(ticks: start.duration)
        .listen((duration) => add(TimerTicked(duration: duration)));
  }
}
```

If the `TimerBloc` receives a `TimerStarted` event, it pushes a `TimerRunInProgress` state with the start duration. In addition, if there was already an open `_tickerSubscription` we need to cancel it to deallocate the memory. We also need to override the `close` method on our `TimerBloc` so that we can cancel the `_tickerSubscription` when the `TimerBloc` is closed. Lastly, we listen to the `_ticker.tick` stream and on every tick we add a `TimerTicked` event with the remaining duration.

Next, let's implement the `TimerTicked` event handler.

```dart
import 'dart:async';
import 'package:meta/meta.dart';
```

```dart
import 'package:bloc/bloc.dart';
import 'package:flutter_timer/bloc/bloc.dart';
import 'package:flutter_timer/ticker.dart';

class TimerBloc extends Bloc<TimerEvent, TimerState> {
  final Ticker _ticker;
  static const int _duration = 60;

  StreamSubscription<int> _tickerSubscription;

  TimerBloc({@required Ticker ticker})
      : assert(ticker != null),
        _ticker = ticker,
        super(TimerInitial(_duration));

  @override
  Stream<TimerState> mapEventToState(
    TimerEvent event,
  ) async* {
    if (event is TimerStarted) {
      yield* _mapTimerStartedToState(event);
    } else if (event is TimerTicked) {
      yield* _mapTimerTickedToState(event);
    }
  }

  @override
  Future<void> close() {
    _tickerSubscription?.cancel();
    return super.close();
  }

  Stream<TimerState> _mapTimerStartedToState(TimerStarted start) async* {
     yield TimerRunInProgress(start.duration);
    _tickerSubscription?.cancel();
    _tickerSubscription = _ticker
        .tick(ticks: start.duration)
        .listen((duration) => add(TimerTicked(duration: duration)));
  }

  Stream<TimerState> _mapTimerTickedToState(TimerTicked tick) async* {
    yield tick.duration > 0 ? TimerRunInProgress(tick.duration) : TimerRu
```

```
    }
  }
```

Every time a `TimerTicked` event is received, if the tick's duration is greater than 0, we need to push an updated `TimerRunInProgress` state with the new duration. Otherwise, if the tick's duration is 0, our timer has ended and we need to push a `TimerRunComplete` state.

Now let's implement the `TimerPaused` event handler.

dart

```dart
import 'dart:async';
import 'package:meta/meta.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_timer/bloc/bloc.dart';
import 'package:flutter_timer/ticker.dart';

class TimerBloc extends Bloc<TimerEvent, TimerState> {
  final Ticker _ticker;
  static const int _duration = 60;

  StreamSubscription<int> _tickerSubscription;

  TimerBloc({@required Ticker ticker})
      : assert(ticker != null),
        _ticker = ticker,
        super(TimerInitial(_duration));

  @override
  Stream<TimerState> mapEventToState(
    TimerEvent event,
  ) async* {
    if (event is TimerStarted) {
      yield* _mapTimerStartedToState(event);
    } else if (event is TimerPaused) {
      yield* _mapTimerPausedToState(event);
    } else if (event is TimerTicked) {
      yield* _mapTimerTickedToState(event);
    }
  }
```

```
  @override
  Future<void> close() {
    _tickerSubscription?.cancel();
    return super.close();
  }

  Stream<TimerState> _mapTimerStartedToState(TimerStarted start) async* {
     yield TimerRunInProgress(start.duration);
    _tickerSubscription?.cancel();
    _tickerSubscription = _ticker
        .tick(ticks: start.duration)
        .listen((duration) => add(TimerTicked(duration: duration)));
  }

  Stream<TimerState> _mapTimerPausedToState(TimerPaused pause) async* {
    if (state is TimerRunInProgress) {
      _tickerSubscription?.pause();
      yield TimerRunPause(state.duration);
    }
  }

  Stream<TimerState> _mapTimerTickedToState(TimerTicked tick) async* {
    yield tick.duration > 0 ? TimerRunInProgress(tick.duration) : TimerRu
  }
}
```

In `_mapTimerPausedToState` if the `state` of our `TimerBloc` is `TimerRunInProgress`, then we can pause the `_tickerSubscription` and push a `TimerRunPause` state with the current timer duration.

Next, let's implement the `TimerResumed` event handler so that we can unpause the timer.

dart

```
import 'dart:async';
import 'package:meta/meta.dart';
import 'package:bloc/bloc.dart';
import 'package:flutter_timer/bloc/bloc.dart';
import 'package:flutter_timer/ticker.dart';

class TimerBloc extends Bloc<TimerEvent, TimerState> {
```

```
final Ticker _ticker;
static const int _duration = 60;

StreamSubscription<int> _tickerSubscription;

TimerBloc({@required Ticker ticker})
    : assert(ticker != null),
      _ticker = ticker,
      super(TimerInitial(_duration));

@override
Stream<TimerState> mapEventToState(
  TimerEvent event,
) async* {
  if (event is TimerStarted) {
    yield* _mapTimerStartedToState(event);
  } else if (event is TimerPaused) {
    yield* _mapTimerPausedToState(event);
  } else if (event is TimerResumed) {
    yield* _mapTimerResumedToState(event);
  } else if (event is TimerTicked) {
    yield* _mapTimerTickedToState(event);
  }
}

@override
Future<void> close() {
  _tickerSubscription?.cancel();
  return super.close();
}

Stream<TimerState> _mapTimerStartedToState(TimerStarted start) async* {
   yield TimerRunInProgress(start.duration);
  _tickerSubscription?.cancel();
  _tickerSubscription = _ticker
      .tick(ticks: start.duration)
      .listen((duration) => add(TimerTicked(duration: duration)));
}

Stream<TimerState> _mapTimerPausedToState(TimerPaused pause) async* {
  if (state is TimerRunInProgress) {
    _tickerSubscription?.pause();
    yield TimerRunPause(state.duration);
```

```
    }
  }

  Stream<TimerState> _mapTimerResumedToState(TimerResumed resume) async*
    if (state is TimerRunPause) {
      _tickerSubscription?.resume();
      yield TimerRunInProgress(state.duration);
    }
  }

  Stream<TimerState> _mapTimerTickedToState(TimerTicked tick) async* {
    yield tick.duration > 0 ? TimerRunInProgress(tick.duration) : TimerRu
  }
}
```

The `TimerResumed` event handler is very similar to the `TimerPaused` event handler. If the `TimerBloc` has a `state` of `TimerRunPause` and it receives a `TimerResumed` event, then it resumes the `_tickerSubscription` and pushes a `TimerRunInProgress` state with the current duration.

Lastly, we need to implement the `TimerReset` event handler.

dart

```
import 'dart:async';
import 'package:meta/meta.dart';
import 'package:bloc/bloc.dart';
import 'package:equatable/equatable.dart';
import 'package:flutter_timer/ticker.dart';

part 'timer_event.dart';
part 'timer_state.dart';

class TimerBloc extends Bloc<TimerEvent, TimerState> {
  final Ticker _ticker;
  static const int _duration = 60;

  StreamSubscription<int> _tickerSubscription;

  TimerBloc({@required Ticker ticker})
      : assert(ticker != null),
        _ticker = ticker,
```

```
      super(TimerInitial(_duration));

  @override
  void onTransition(Transition<TimerEvent, TimerState> transition) {
    print(transition);
    super.onTransition(transition);
  }

  @override
  Stream<TimerState> mapEventToState(
    TimerEvent event,
  ) async* {
    if (event is TimerStarted) {
      yield* _mapTimerStartedToState(event);
    } else if (event is TimerPaused) {
      yield* _mapTimerPausedToState(event);
    } else if (event is TimerResumed) {
      yield* _mapTimerResumedToState(event);
    } else if (event is TimerReset) {
      yield* _mapTimerResetToState(event);
    } else if (event is TimerTicked) {
      yield* _mapTimerTickedToState(event);
    }
  }

  @override
  Future<void> close() {
    _tickerSubscription?.cancel();
    return super.close();
  }

  Stream<TimerState> _mapTimerStartedToState(TimerStarted start) async* {
    yield TimerRunInProgress(start.duration);
    _tickerSubscription?.cancel();
    _tickerSubscription = _ticker
        .tick(ticks: start.duration)
        .listen((duration) => add(TimerTicked(duration: duration)));
  }

  Stream<TimerState> _mapTimerPausedToState(TimerPaused pause) async* {
    if (state is TimerRunInProgress) {
      _tickerSubscription?.pause();
      yield TimerRunPause(state.duration);
```

```
    }
  }

  Stream<TimerState> _mapTimerResumedToState(TimerResumed resume) async*
    if (state is TimerRunPause) {
      _tickerSubscription?.resume();
      yield TimerRunInProgress(state.duration);
    }
  }

  Stream<TimerState> _mapTimerResetToState(TimerReset reset) async* {
    _tickerSubscription?.cancel();
    yield TimerInitial(_duration);
  }

  Stream<TimerState> _mapTimerTickedToState(TimerTicked tick) async* {
    yield tick.duration > 0
        ? TimerRunInProgress(tick.duration)
        : TimerRunComplete();
  }
}
```

If the `TimerBloc` receives a `TimerReset` event, it needs to cancel the current `_tickerSubscription` so that it isn't notified of any additional ticks and pushes a `TimerInitial` state with the original duration.

If you didn't use the IntelliJ or VSCode extensions be sure to create `bloc/bloc.dart` in order to export all the bloc files and make it possible to use a single import for convenience.

```dart
export 'timer_bloc.dart';
export 'timer_event.dart';
export 'timer_state.dart';
```

That's all there is to the `TimerBloc`. Now all that's left is implement the UI for our Timer Application.

Application UI

MyApp

We can start off by deleting the contents of `main.dart` and creating our `MyApp` widget which will be the root of our application.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:flutter_timer/bloc/bloc.dart';
import 'package:flutter_timer/ticker.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(
        primaryColor: Color.fromRGBO(109, 234, 255, 1),
        accentColor: Color.fromRGBO(72, 74, 126, 1),
        brightness: Brightness.dark,
      ),
      title: 'Flutter Timer',
      home: BlocProvider(
        create: (context) => TimerBloc(ticker: Ticker()),
        child: Timer(),
      ),
    );
  }
}
```

`MyApp` is a `StatelessWidget` which will manage initializing and closing an instance of `TimerBloc`. In addition, it's using the `BlocProvider` widget in order to make our `TimerBloc` instance available to the widgets in our subtree.

Next, we need to implement our `Timer` widget.

Timer

Our `Timer` widget will be responsible for displaying the remaining time along with the proper buttons which will enable users to start, pause, and reset the

timer.

```dart
class Timer extends StatelessWidget {
  static const TextStyle timerTextStyle = TextStyle(
    fontSize: 60,
    fontWeight: FontWeight.bold,
  );

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Flutter Timer')),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        crossAxisAlignment: CrossAxisAlignment.center,
        children: <Widget>[
          Padding(
            padding: EdgeInsets.symmetric(vertical: 100.0),
            child: Center(
              child: BlocBuilder<TimerBloc, TimerState>(
                builder: (context, state) {
                  final String minutesStr = ((state.duration / 60) % 60)
                      .floor()
                      .toString()
                      .padLeft(2, '0');
                  final String secondsStr =
                      (state.duration % 60).floor().toString().padLeft(2,
                  return Text(
                    '$minutesStr:$secondsStr',
                    style: Timer.timerTextStyle,
                  );
                },
              ),
            ),
          ),
        ],
      ),
    );
  }
}
```

So far, we're just using `BlocProvider` to access the instance of our `TimerBloc` and using a `BlocBuilder` widget in order to rebuild the UI every time we get a new `TimerState`.

Next, we're going to implement our `Actions` widget which will have the proper actions (start, pause, and reset).

Actions

```dart
class Actions extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: _mapStateToActionButtons(
        timerBloc: BlocProvider.of<TimerBloc>(context),
      ),
    );
  }

  List<Widget> _mapStateToActionButtons({
    TimerBloc timerBloc,
  }) {
    final TimerState currentState = timerBloc.state;
    if (currentState is TimerInitial) {
      return [
        FloatingActionButton(
          child: Icon(Icons.play_arrow),
          onPressed: () =>
              timerBloc.add(TimerStarted(duration: currentState.duration)
        ),
      ];
    }
    if (currentState is TimerRunInProgress) {
      return [
        FloatingActionButton(
          child: Icon(Icons.pause),
          onPressed: () => timerBloc.add(TimerPaused()),
        ),
        FloatingActionButton(
          child: Icon(Icons.replay),
```

```
            onPressed: () => timerBloc.add(TimerReset()),
          ),
        ];
      }
      if (currentState is TimerRunPause) {
        return [
          FloatingActionButton(
            child: Icon(Icons.play_arrow),
            onPressed: () => timerBloc.add(TimerResumed()),
          ),
          FloatingActionButton(
            child: Icon(Icons.replay),
            onPressed: () => timerBloc.add(TimerReset()),
          ),
        ];
      }
      if (currentState is TimerRunComplete) {
        return [
          FloatingActionButton(
            child: Icon(Icons.replay),
            onPressed: () => timerBloc.add(TimerReset()),
          ),
        ];
      }
      return [];
    }
  }
```

The `Actions` widget is just another `StatelessWidget` which uses `BlocProvider` to access the `TimerBloc` instance and then returns different `FloatingActionButtons` based on the current state of the `TimerBloc`. Each of the `FloatingActionButtons` adds an event in its `onPressed` callback to notify the `TimerBloc`.

Now we need to hook up the `Actions` to our `Timer` widget.

dart

```
class Timer extends StatelessWidget {
  static const TextStyle timerTextStyle = TextStyle(
    fontSize: 60,
    fontWeight: FontWeight.bold,
```

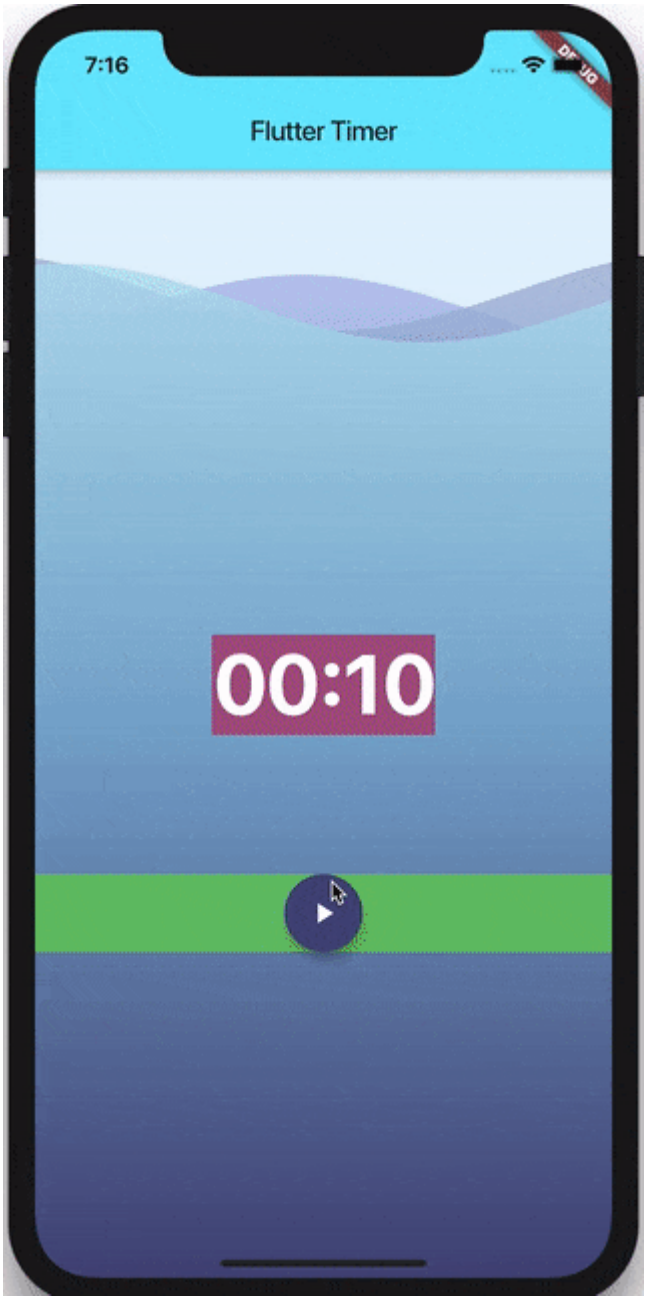
```
  );

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Flutter Timer')),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        crossAxisAlignment: CrossAxisAlignment.center,
        children: <Widget>[
          Padding(
            padding: EdgeInsets.symmetric(vertical: 100.0),
            child: Center(
              child: BlocBuilder<TimerBloc, TimerState>(
                builder: (context, state) {
                  final String minutesStr = ((state.duration / 60) % 60)
                      .floor()
                      .toString()
                      .padLeft(2, '0');
                  final String secondsStr =
                      (state.duration % 60).floor().toString().padLeft(2,
                  return Text(
                    '$minutesStr:$secondsStr',
                    style: Timer.timerTextStyle,
                  );
                },
              ),
            ),
          ),
          BlocBuilder<TimerBloc, TimerState>(
            buildWhen: (previousState, state) =>
                state.runtimeType != previousState.runtimeType,
            builder: (context, state) => Actions(),
          ),
        ],
      ),
    );
  }
}
```

We added another `BlocBuilder` which will render the `Actions` widget; however, this time we're using a newly introduced **flutter_bloc** feature to control how frequently the `Actions` widget is rebuilt (introduced in `v0.15.0`).

If you want fine-grained control over when the `builder` function is called you can provide an optional `buildWhen` to `BlocBuilder`. The `buildWhen` takes the previous bloc state and current bloc state and returns a `boolean`. If `buildWhen` returns `true`, `builder` will be called with `state` and the widget will rebuild. If `buildWhen` returns `false`, `builder` will not be called with `state` and no rebuild will occur.

In this case, we don't want the `Actions` widget to be rebuilt on every tick because that would be inefficient. Instead, we only want `Actions` to rebuild if the `runtimeType` of the `TimerState` changes (TimerInitial => TimerRunInProgress, TimerRunInProgress => TimerRunPause, etc...).

As a result, if we randomly colored the widgets on every rebuild, it would look like:



Notice: Even though the `Text` widget is rebuilt on every tick, we only rebuild the `Actions` if they need to be rebuilt.

Lastly, we need to add the super cool wave background using the wave package.

Waves Background

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

class Background extends StatelessWidget {
```

```
    @override
    Widget build(BuildContext context) {
      return WaveWidget(
        config: CustomConfig(
          gradients: [
            [
              Color.fromRGBO(72, 74, 126, 1),
              Color.fromRGBO(125, 170, 206, 1),
              Color.fromRGBO(184, 189, 245, 0.7)
            ],
            [
              Color.fromRGBO(72, 74, 126, 1),
              Color.fromRGBO(125, 170, 206, 1),
              Color.fromRGBO(172, 182, 219, 0.7)
            ],
            [
              Color.fromRGBO(72, 73, 126, 1),
              Color.fromRGBO(125, 170, 206, 1),
              Color.fromRGBO(190, 238, 246, 0.7)
            ],
          ],
          durations: [19440, 10800, 6000],
          heightPercentages: [0.03, 0.01, 0.02],
          gradientBegin: Alignment.bottomCenter,
          gradientEnd: Alignment.topCenter,
        ),
        size: Size(double.infinity, double.infinity),
        waveAmplitude: 25,
        backgroundColor: Colors.blue[50],
      );
    }
  }
```

Putting it all together

Our finished, `main.dart` should look like:

```dart
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:flutter_timer/bloc/bloc.dart';
```

```dart
import 'package:flutter_timer/ticker.dart';
import 'package:wave/wave.dart';
import 'package:wave/config.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(
        primaryColor: Color.fromRGBO(109, 234, 255, 1),
        accentColor: Color.fromRGBO(72, 74, 126, 1),
        brightness: Brightness.dark,
      ),
      title: 'Flutter Timer',
      home: BlocProvider(
        create: (context) => TimerBloc(ticker: Ticker()),
        child: Timer(),
      ),
    );
  }
}

class Timer extends StatelessWidget {
  static const TextStyle timerTextStyle = TextStyle(
    fontSize: 60,
    fontWeight: FontWeight.bold,
  );

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Flutter Timer')),
      body: Stack(
        children: [
          Background(),
          Column(
            mainAxisAlignment: MainAxisAlignment.center,
            crossAxisAlignment: CrossAxisAlignment.center,
            children: <Widget>[
              Padding(
                padding: EdgeInsets.symmetric(vertical: 100.0),
```

```
                child: Center(
                  child: BlocBuilder<TimerBloc, TimerState>(
                    builder: (context, state) {
                      final String minutesStr = ((state.duration / 60) %
                          .floor()
                          .toString()
                          .padLeft(2, '0');
                      final String secondsStr = (state.duration % 60)
                          .floor()
                          .toString()
                          .padLeft(2, '0');
                      return Text(
                        '$minutesStr:$secondsStr',
                        style: Timer.timerTextStyle,
                      );
                    },
                  ),
                ),
              ),
              BlocBuilder<TimerBloc, TimerState>(
                buildWhen: (previousState, currentState) =>
                    currentState.runtimeType != previousState.runtimeType
                builder: (context, state) => Actions(),
              ),
            ],
          ),
        ],
      ),
    );
  }
}

class Actions extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: _mapStateToActionButtons(
        timerBloc: BlocProvider.of<TimerBloc>(context),
      ),
    );
  }
```

```
  List<Widget> _mapStateToActionButtons({
    TimerBloc timerBloc,
  }) {
    final TimerState currentState = timerBloc.state;
    if (currentState is TimerInitial) {
      return [
        FloatingActionButton(
          child: Icon(Icons.play_arrow),
          onPressed: () =>
              timerBloc.add(TimerStarted(duration: currentState.duration)
        ),
      ];
    }
    if (currentState is TimerRunInProgress) {
      return [
        FloatingActionButton(
          child: Icon(Icons.pause),
          onPressed: () => timerBloc.add(TimerPaused()),
        ),
        FloatingActionButton(
          child: Icon(Icons.replay),
          onPressed: () => timerBloc.add(TimerReset()),
        ),
      ];
    }
    if (currentState is TimerRunPause) {
      return [
        FloatingActionButton(
          child: Icon(Icons.play_arrow),
          onPressed: () => timerBloc.add(TimerResumed()),
        ),
        FloatingActionButton(
          child: Icon(Icons.replay),
          onPressed: () => timerBloc.add(TimerReset()),
        ),
      ];
    }
    if (currentState is TimerRunComplete) {
      return [
        FloatingActionButton(
          child: Icon(Icons.replay),
          onPressed: () => timerBloc.add(TimerReset()),
        ),
```

```
        ];
      }
      return [];
    }
  }

  class Background extends StatelessWidget {
    @override
    Widget build(BuildContext context) {
      return WaveWidget(
        config: CustomConfig(
          gradients: [
            [
              Color.fromRGBO(72, 74, 126, 1),
              Color.fromRGBO(125, 170, 206, 1),
              Color.fromRGBO(184, 189, 245, 0.7)
            ],
            [
              Color.fromRGBO(72, 74, 126, 1),
              Color.fromRGBO(125, 170, 206, 1),
              Color.fromRGBO(172, 182, 219, 0.7)
            ],
            [
              Color.fromRGBO(72, 73, 126, 1),
              Color.fromRGBO(125, 170, 206, 1),
              Color.fromRGBO(190, 238, 246, 0.7)
            ],
          ],
          durations: [19440, 10800, 6000],
          heightPercentages: [0.03, 0.01, 0.02],
          gradientBegin: Alignment.bottomCenter,
          gradientEnd: Alignment.topCenter,
        ),
        size: Size(double.infinity, double.infinity),
        waveAmplitude: 25,
        backgroundColor: Colors.blue[50],
      );
    }
  }
```

That's all there is to it! At this point we have a pretty solid timer application which efficiently rebuilds only widgets that need to be rebuilt.

The full source for this example can be found here.

PREVIOUS
Counter ✨

NEXT
Infinite List