Testing

> Bloc was designed to be extremely easy to test.

For the sake of simplicity, let's write tests for the `CounterBloc` we created in Core Concepts.

To recap, the `CounterBloc` implementation looks like

```dart
enum CounterEvent { increment, decrement }

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

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

Before we start writing our tests we're going to need to add a testing framework to our dependencies.

We need to add test and bloc_test to our `pubspec.yaml`.

```yaml
dev_dependencies:
  test: ^1.16.0
  bloc_test: ^8.0.0
```

Let's get started by creating the file for our `CounterBloc` Tests, `counter_bloc_test.dart` and importing the test package.

```dart
import 'package:test/test.dart';
import 'package:bloc_test/bloc_test.dart';
```

Next, we need to create our `main` as well as our test group.

```dart
void main() {
    group('CounterBloc', () {

    });
}
```

Note: groups are for organizing individual tests as well as for creating a context in which you can share a common `setUp` and `tearDown` across all of the individual tests.

Let's start by creating an instance of our `CounterBloc` which will be used across all of our tests.

```dart
group('CounterBloc', () {
    late CounterBloc counterBloc;

    setUp(() {
        counterBloc = CounterBloc();
    });
});
```

Now we can start writing our individual tests.

```dart
group('CounterBloc', () {
    CounterBloc counterBloc;

    setUp(() {
        counterBloc = CounterBloc();
    });

    test('initial state is 0', () {
        expect(counterBloc.state, 0);
    });
});
```

Note: We can run all of our tests with the `pub run test` command.

At this point we should have our first passing test! Now let's write a more complex test using the bloc_test package.

```dart
blocTest(
    'emits [1] when CounterEvent.increment is added',
    build: () => counterBloc,
    act: (bloc) => bloc.add(CounterEvent.increment),
    expect: () => [1],
);

blocTest(
    'emits [-1] when CounterEvent.decrement is added',
    build: () => counterBloc,
    act: (bloc) => bloc.add(CounterEvent.decrement),
    expect: () => [-1],
);
```

We should be able to run the tests and see that all are passing.

That's all there is to it, testing should be a breeze and we should feel confident when making changes and refactoring our code.

You can refer to the Todos App for an example of a fully tested application.