

Section : IT Systems Development (DSI)
Module : Framework cross-platform workshop
Teaching unit : Mobile Development and Web
Level : 3rd Year (Applied license: LMD)

Workshop 8: Testing in flutter

Objective

Generate some tests in flutter applications

Technical requirements

In order to start with Flutter, we need a few tools:

- A PC with a recent Windows version, or a Mac with a recent version of the macOS or Linux operating system. we can also use a Chrome OS machine, with a few tweaks.
- An Android/iOS setup.
- The Flutter SDK. It's free, light, and open source.
- Physical device/emulator/simulator
- Android Studio/IntelliJ IDEA or Visual Studio Code

content

We will cover the following recipe:

- Unit testing
- Widget testing
- Integration testing

Unit testing

An introduction to unit testing

Unit tests are handy for verifying the behaviour of a :

- 1- Single method
- 2- Single function
- 3- Single class

The test package provides the core framework for writing unit tests, and the flutter test package provides additional utilities for testing widgets.

Add the test dependency

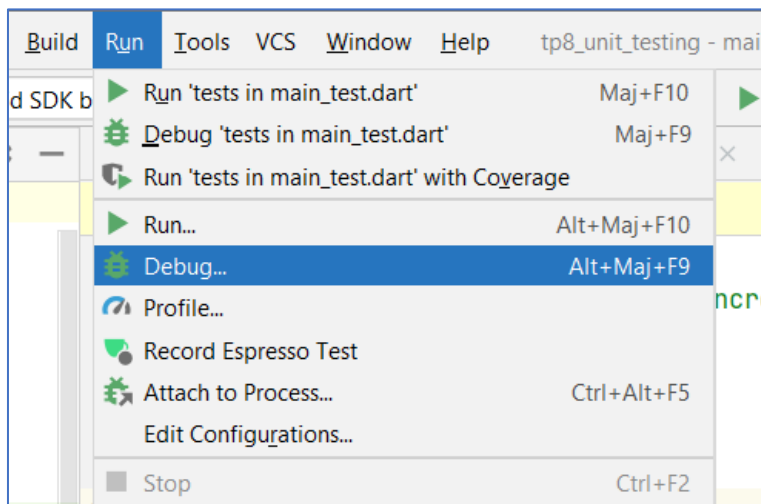
```
dev_dependencies:  
  test: ^1.19.2
```

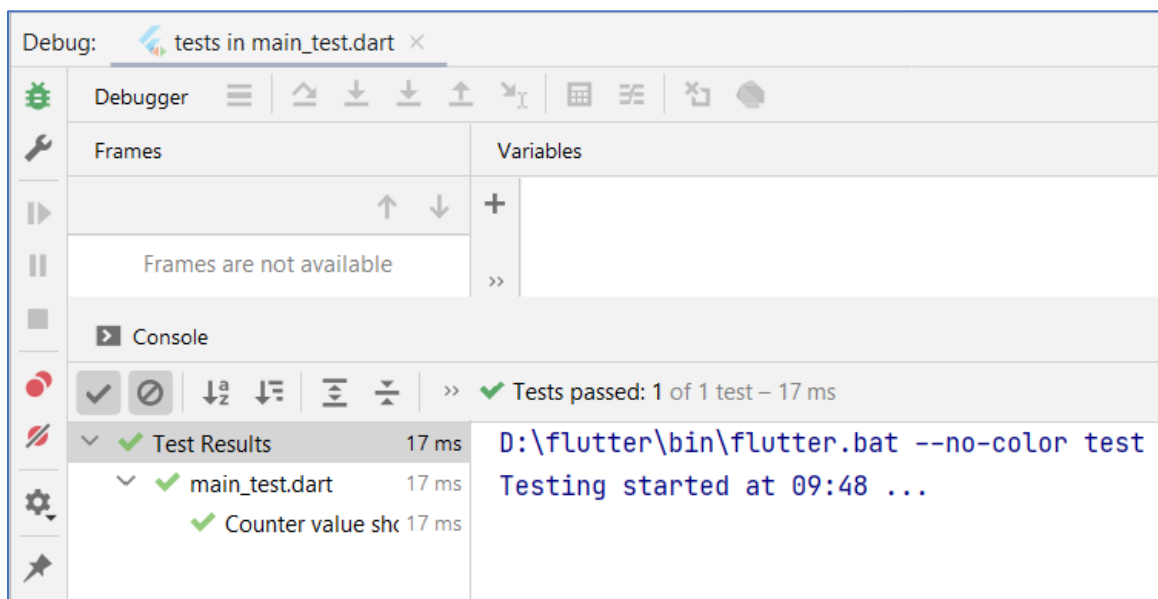
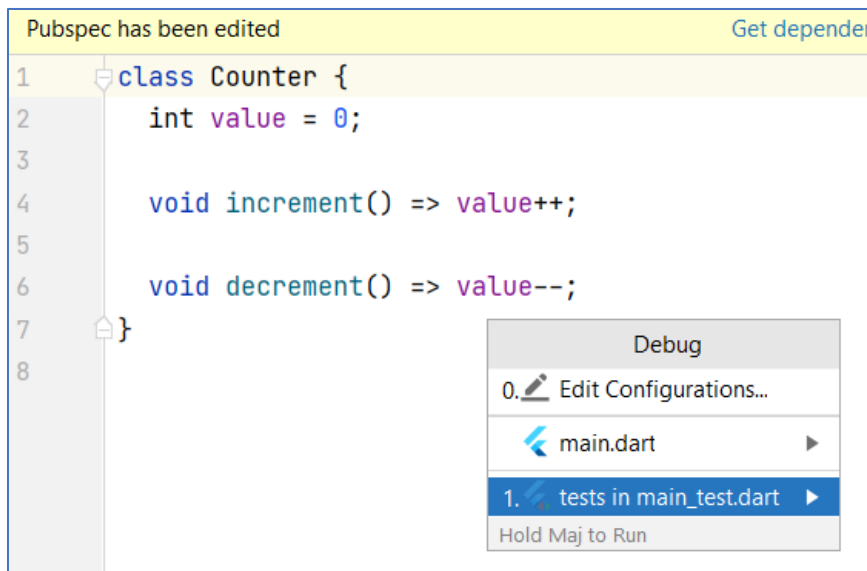
Create a test file

```
lib/  
  counter.dart  
test/  
  counter_test.dart  
  
class Counter {  
  int value = 0;  
  
  void increment() => value++;  
  
  void decrement() => value--;  
}  
  
import 'package:test/test.dart';  
import 'package:tp8_unit_testing/Counter.dart';  
  
void main() {  
  test('Counter value should be incremented', () {  
    final counter = Counter();  
  
    counter.increment();  
  
    expect(counter.value, 1);  
  });  
}
```

Run the tests

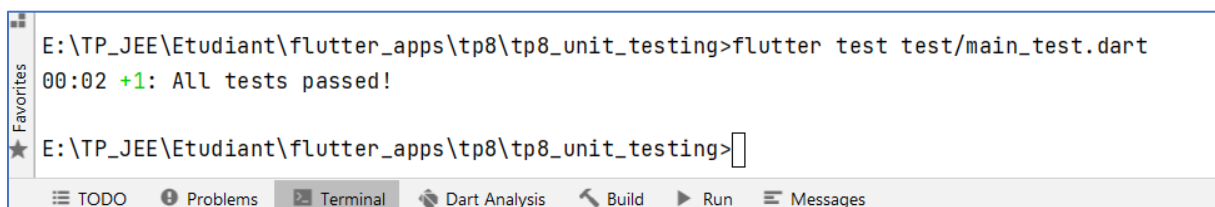
1- Run tests using Android studio





2- Run tests in a terminal

`flutter test test/main_test.dart`



Change the value by 0 in expect function

```
expect(counter.value, 0);
```

After execution you will find a result:

```
E:\TP_JEE\Etudiant\flutter_apps\tp8\tp8_unit_testing>flutter test test/main_test.dart
00:01 +0 -1: Counter value should be incremented [E]
  Expected: <0>
  Actual: <-1>

package:test_api          expect
test\main_test.dart 13:5  main.<fn>

00:01 +0 -1: Some tests failed.
```

Mock dependencies using Mockito

Sometimes, unit tests might depend on classes that fetch data from live web services or databases. This is inconvenient for a few reasons:

- Calling live services or databases slows down test execution.
- A passing test might start failing if a web service or database returns unexpected results. This is known as a “flaky test.”
- It is difficult to test all possible success and failure scenarios by using a live web service or database.

Therefore, rather than relying on a live web service or database, you can “mock” these dependencies. Mocks allow emulating a live web service or database and return specific results depending on the situation.

Generally speaking, you can mock dependencies by creating an alternative implementation of a class. Write these alternative implementations by hand or make use of the [Mockito package](#) as a shortcut.

Mock objects simulate the behavior of real (often complex) objects. They allow us to create an object that will replace the real one used in the implementation code. A mocked object will expect a defined method with defined arguments to return the expected result. It knows in advance what is supposed to happen and how we expect it to react. Mockito is a mocking framework with a clean and simple API. Tests produced with Mockito are readable, easy to write, and intuitive

Add the package dependencies

dependencies:

```
flutter:
  sdk: flutter
http: ^0.13.4
cupertino_icons: ^1.0.3
```

dev_dependencies:

```
flutter_test:
  sdk: flutter
mockito: ^5.0.16
build_runner: ^2.1.4
```

Create a function to test

1. Provide an `http.Client` to the function. This allows providing the correct `http.Client` depending on the situation. For Flutter and server-side projects, provide an `http.IOCClient`. For Browser apps, provide an `http.BrowserClient`. For tests, provide a mock `http.Client`.
2. Use the provided `client` to fetch data from the internet, rather than the static `http.get()` method, which is difficult to mock.

```
import 'dart:async';
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;

Future<Album> fetchAlbum(http.Client client) async {
  final response = await client
    .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));

  if (response.statusCode == 200) {
    // If the server did return a 200 OK response,
    // then parse the JSON.
    return Album.fromJson(jsonDecode(response.body));
  } else {
    // If the server did not return a 200 OK response,
    // then throw an exception.
    throw Exception('Failed to load album');
  }
}

class Album {
  final int userId;
  final int id;
  final String title;

  const Album({required this.userId, required this.id, required
this.title});

  factory Album.fromJson(Map<String, dynamic> json) {
    return Album(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
    );
  }
}

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

class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  late final Future<Album> futureAlbum;

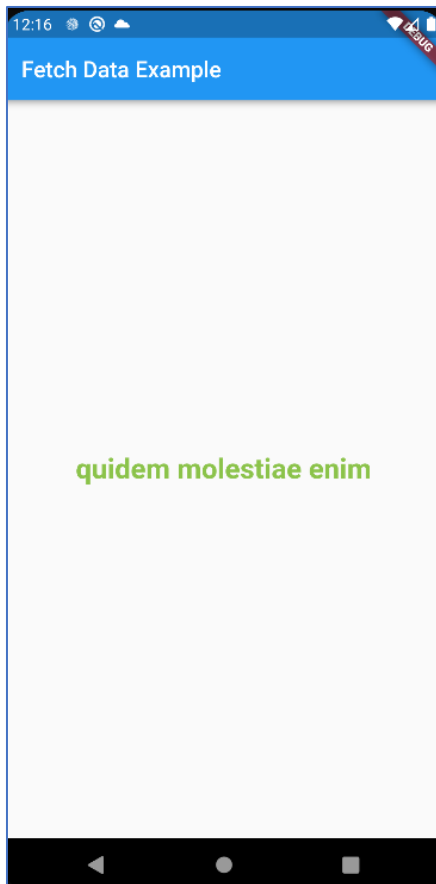
  @override
  void initState() {
    super.initState();
    futureAlbum = fetchAlbum(http.Client());
  }
}
```

```

}

@override
Widget build(BuildContext context) {
  return MaterialApp(
    title: 'Fetch Data Example',
    theme: ThemeData(
      primarySwatch: Colors.blue,
    ),
    home: Scaffold(
      appBar: AppBar(
        title: const Text('Fetch Data Example'),
      ),
      body: Center(
        child: FutureBuilder<Album>(
          future: futureAlbum,
          builder: (context, snapshot) {
            if (snapshot.hasData) {
              return Text(
                snapshot.data!.title,
                style: TextStyle(
                  color: Colors.lightGreen,
                  fontWeight: FontWeight.w700,
                  fontSize: 25,
                ),
              );
            } else if (snapshot.hasError) {
              return Text('${snapshot.error}');
            }
            // By default, show a loading spinner.
            return const CircularProgressIndicator();
          },
        ),
      ),
    ),
  );
}

```



Create a test file with a mock `http.Client`

- 1- Create a file called `fetch_album_test.dart` in the root `test` folder.
- 2- Add the annotation `@GenerateMocks([http.Client])` to the main function to generate a `MockClient` class with `mockito`.

```
import 'package:http/http.dart' as http;
import 'package:mockito/annotations.dart';
// Generate a MockClient using the Mockito package.
// Create new instances of this class in each test.
@GenerateMocks([http.Client])
void main() {}
```

- 3- Next, generate the mocks running the following command:

```
flutter pub run build_runner build
```

The generated `MockClient` class implements the `http.Client` class. This allows you to pass the `MockClient` to the `fetchAlbum` function, and return different http responses in each test.

The generated mocks will be located in `fetch_album_test.mocks.dart`. Import this file to use them.

Write a test for each condition in `fetch_album_test.dart`

```

import 'package:flutter_test/flutter_test.dart';
import 'package:http/http.dart' as http;
import 'package:mockito/annotations.dart';
import 'package:mockito/mockito.dart';
import 'package:tp8_unit_mockito/main.dart';

import 'fetch_album_test.mocks.dart';

// Generate a MockClient using the Mockito package.
// Create new instances of this class in each test.
@GenerateMocks([http.Client])
void main() {
  group('fetchAlbum', () {
    test('returns an Album if the http call completes successfully', ()
    async {
      final client = MockClient();

      // Use Mockito to return a successful response when it calls the
      // provided http.Client.
      when(client

.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1')))
      .thenAnswer((_) async =>
        http.Response({'userId': 1, "id": 2, "title": "mock"},
200));

      expect(await fetchAlbum(client), isA<Album>());
    });

    test('throws an exception if the http call completes with an error', ()
    {
      final client = MockClient();

      // Use Mockito to return an unsuccessful response when it calls the
      // provided http.Client.
      when(client

.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1')))
      .thenAnswer((_) async => http.Response('Not Found', 404));

      expect(fetchAlbum(client), throwsException);
    });
  });
}

```

Run the tests

```
flutter test test/fetch_album_test.dart
```

```

E:\TP_JEE\Etudiant\flutter_apps\tp8\tp8_unit_mockito>flutter test test/fetch_album_test.dart
00:07 +2: All tests passed!
E:\TP_JEE\Etudiant\flutter_apps\tp8\tp8_unit_mockito>

```


Widget testing

- To test that your codes in creating your widgets is working

Integration testing

- Test that for example tapping on a button will cause that changes or not