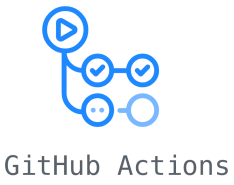
BITBYBIT

Atbash

Testing Policy

Continuous Integration

Testing tool



- We made use of Github Actions for continuous integration.
- The reason we chose to use Github Actions is because:
    - It is free for all public repositories.
    - It supports a wide range of languages.
    - It allows for the building, testing and deploying straight from Github.
    - It makes it easier to manage branching and code reviews.
    - It is easy to set up.
    - It is already integrated into Github so it makes it easy to view any errors and the status of each merge without having to leave the website to view it on an external continuous integration tool.

Procedure

- The testing is automated when a merge request is created, which means that the tests are run with the software when it is pushed onto the repository. This makes it easy to view the status of each build.

Front-end Testing

Testing tool

- We made use of Flutters built-in testing framework as well as mockito. The flutter framework provides core testing functionality and the flutter_test provides additional widget testing functionality. Mockito is used to mock out data for the unit tests. We also make use of the many services that we have created. We didn't make use of any other tools, because flutters built in testing functionality worked well and allowed for easy integration.

Procedure

- We created a “test” directory on the root of the project and each page would then have its own directory. The naming convention for the testing files is PageName_test.dart. An example would be “AnalyticsPage_test.dart”. Each page's individual functionality would make use of the testWidgets function (provided by flutters built in testing).

```
testWidgets(
    "clicking 'REGISTER' shows loading icon and succeeds, displaying ProfileSetupPage",
    (WidgetTester tester) async {
  when(registrationService.register(any)).thenAnswer((_) => Future.value(""));

  await tester.pumpWidget(
    MaterialApp(
      home: RegistrationPage(),
    ),  // MaterialApp
  );

  final buttonFinder = find.byType(MaterialButton);
  final loadingIconFinder = find.byType(SpinKitThreeBounce);

  expect(buttonFinder, findsOneWidget);
  expect(loadingIconFinder, findsNothing);

  await tester.tap(buttonFinder);
  await tester.pump();

  expect(buttonFinder, findsNothing);
  expect(loadingIconFinder, findsOneWidget);

  await tester.pump();
});
```

    - The testing assertion is at the end of each individual test as seen above by the “expect” function. It takes in mock data and compares it to what it expects to find.

Backend Testing

Testing tool

- We made use of a javascript testing framework called Jest. Jest natively supports testing, asserting and mocking.

Advantages:

- Jest is incredibly easy to setup and integrate into our installation of NodeJs
- It is an open source library designed for unit testing, and allows for asserting and mocking all in one package.
- Jest creates coverage reports so that we can ensure every aspect of our backend has been tested and approved.
- Jest is efficient at testing asynchronous code which the backend of Atbash relies heavily on, and it's as simple as declaring the test function with the async tag and using await in front of functions that are done asynchronously.

Procedure

Test Suites :

Because our backend made use of microservices, every function needed its own test directory which was always named __tests__ and inside it was the file with the naming convention "FileName.test.js" or in most cases "index.test.js". This convention is used to allow Jest to automatically detect and run the files.

In each test file the microservice was individually tested using Jest's test function which allows for a description and a function call to actually run the test code. Assertions are achieved through the expect function which accepts the calculated value and compares it to a set value in the toBe function, additionally similar tests are grouped in a describe function as seen below.

```
describe('unit tests for index.handler', () => {
    test('When handler is called with an undefined id, should return status code 400', async () => {
        const response = await handler({pathParameters: {}})
        expect(response.statusCode).toBe(400)
    })

    test('When no element with id exists, should return status code 404', async () => {
        existsMessageById.mockImplementation(() => Promise.resolve(false))

        const response = await handler({pathParameters: {id: "123"}})
        expect(response.statusCode).toBe(404)
    })

    test('When existsMessageById fails, should return status code 500', async () => {
        existsMessageById.mockImplementation(() => Promise.reject())

        const response = await handler({pathParameters: {id: "123"}})
        expect(response.statusCode).toBe(500)
    })
```

Mocking:

Jest allows for mocking services used in your tests so that you can return a specific value and test all edge cases. This is achieved by calling jest.mock at the beginning of the file and specifying the functions which Jest should mock.

```
1  jest.mock("../db_access", () => ({
2      existsMessageById: jest.fn(),
3      deleteMessageById: jest.fn()
4  }))
5
6  const {handler} = require("../index")
7  const {existsMessageById, deleteMessageById} = require("../db_access")
8
9  describe('unit tests for index.handler', () => {
10     test('When handler is called with an undefined id, should return status code 400', async () => {
11         const response = await handler({pathParameters: {}})
12         expect(response.statusCode).toBe(400)
13     })
14
15     test('When no element with id exists, should return status code 404', async () => {
16         existsMessageById.mockImplementation(() => Promise.resolve(false))
17
18         const response = await handler({pathParameters: {id: "123"}})
19         expect(response.statusCode).toBe(404)
20     })
21
```