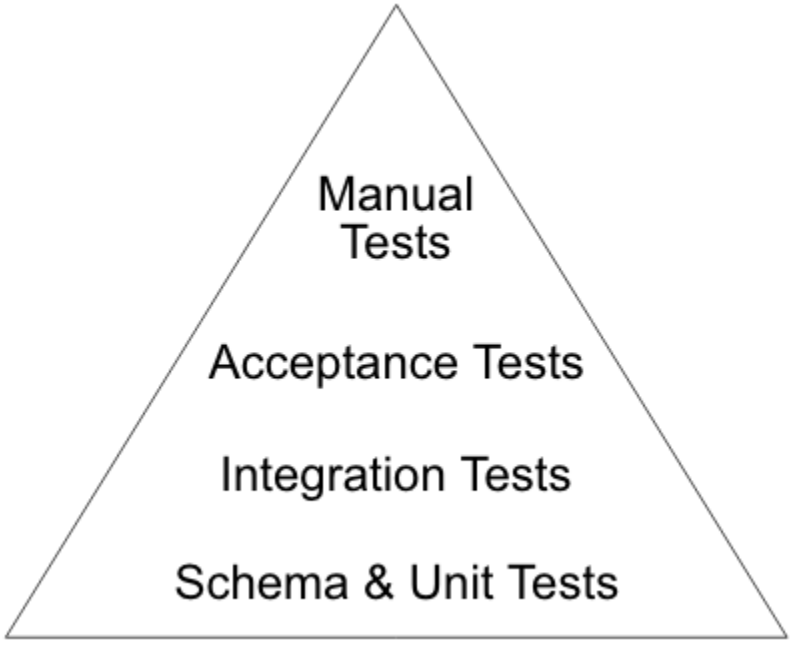
Testing GraphQL

Introduction

To provide quality and reliability of the GraphQL API we need to make sure the code is tested on each level of the testing triangle.



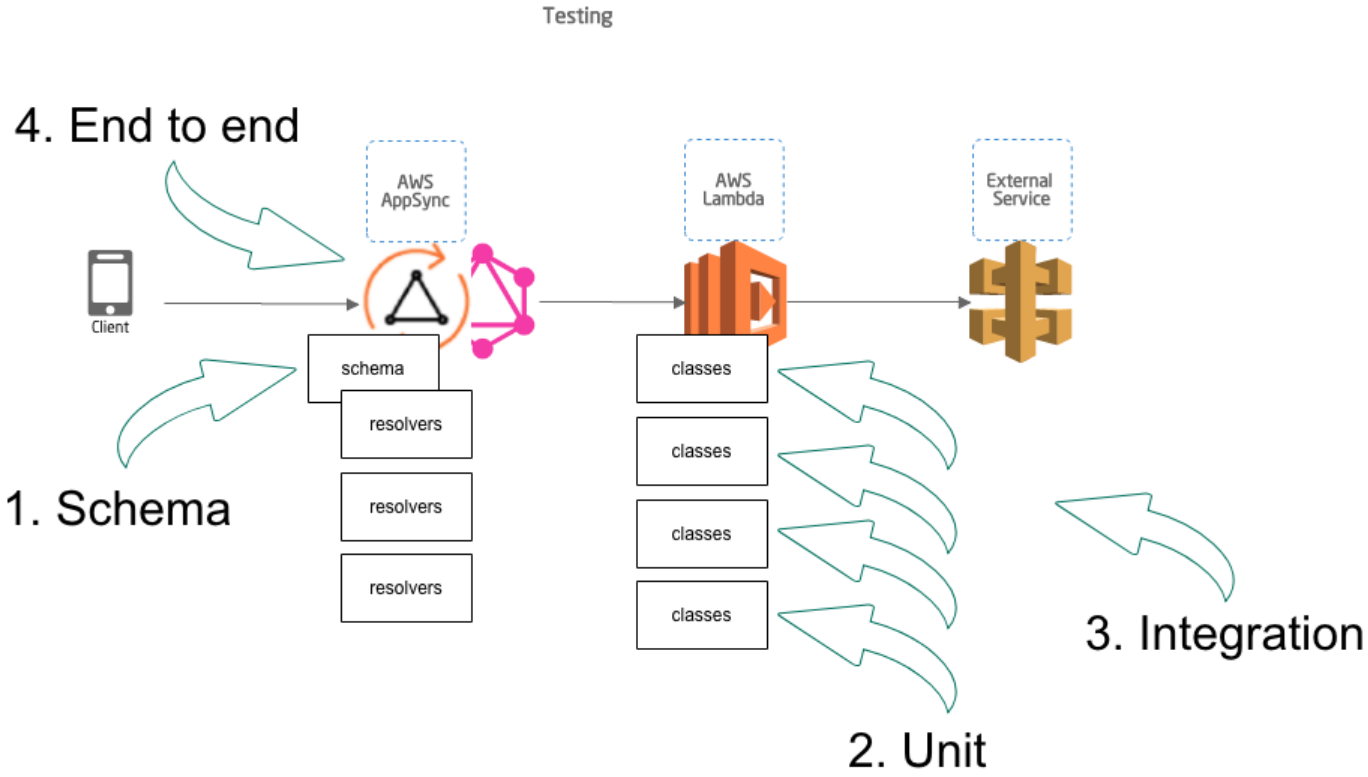
That means we need to test the code on the level of:

- classes and functions (unit tests)
- multiple classes working together (integration tests)
- GraphQL API treated as a black box (acceptance tests)

The tests mentioned above should be automated to speed up the development and detect any problems early. Manual tests should be considered as only an additional and final step.

In terms of GraphQL, on the unit tests level, we should add tests of the schema for the schema validation.

Let's see the diagram below.



Challenges

There's a number of challenges we encounter in relation to the serverless architecture, just to quote a few sentences from the Serverless.com website:

> *"While the Serverless Architecture introduces a lot of simplicity when it comes to serving business logic, some of its characteristics present challenges for testing. They are:*
>
> - *The Serverless Architecture is an integration of separate, distributed services, which must be tested both independently, and together.*
> - *The Serverless Architecture is dependent on internet/cloud services, which are hard to emulate locally.*
> - *The Serverless Architecture can feature event-driven, asynchronous workflows, which are hard to emulate entirely.*
>
> *To get through these challenges, and to keep the test pyramid in mind, keep the following principles in mind:*
>
> - *Write your business logic so that it is separate from your FaaS provider (e.g., AWS Lambda), to keep it provider-independent, reusable and more easily testable.*
> - *When your business logic is written separately from the FaaS provider, you can write traditional Unit Tests to ensure it is working properly.*
> - *Write Integration Tests to verify integrations with other services are working correctly.* "

On top of this, there are a number of challenges in terms of using the AppSync plugin with the Serverless Framework (see this), these include:

- AppSync resolvers are defined as a configuration in the serverless.yml file, but the both: business logic code and configuration form the working service - *how to test the configuration?*
- AppSync as a GraphQL API serving service cannot be run locally* which proves difficulties in testing the setup - *how do we test the API before deployment?*
- Request and response mapping templates are a part of the AppSync setup and cannot be easily tested in isolation - *how do we test the mappings?*

- Lambda functions can be invoked locally but the output needs to be parsed - *how do we parse the outputs?*
- Deployment and execution is dependent on provider permissions - *how do we test the permissions?*

*as per August 2018

How to achieve testing on different levels?

Unit testing

This can be easily achieved with one of the JavaScript testing frameworks. The recommended ones are Mocha + Sinon + Chai.

Example:

```
describe('Identity', function () {

    beforeEach(function () {
        tenant = 'orc';
        callback = sinon.fake;
        rest = new Rest();
        identity = new Identity(rest);
        sinon.stub(rest, 'post');
    });

    describe('#authenticate()', function () {
        it('should call post() on the REST client', function () {
            // given
            const username = 'user';
            const password = 'pass';
            const allowRefresh = false;

            // when
            identity.authenticate(tenant, username, password,
allowRefresh, callback);

            // then
            assert(rest.post.calledOnce);
        });
    });
});
```

Schema testing

In the form of a unit test we can easily verify the schema.

Example:

```
describe('SCHEMA', function () {
    beforeEach(function () {
        const schemaString = readFileSync('schema.graphql', 'utf8');
        schema = makeExecutableSchema({ typeDefs: schemaString });
        addMockFunctionsToSchema({ schema });
    });

    describe('verify the schema', function () {
        it('should make a successful authenticate query request', async
function () {
            // given
            const query = readFileSync('test/schema/query-authenticate.
graphql', 'utf8');

            // when
            const result = await graphql(schema, query);

            // then
            expect(result).toHaveProperty("data.authenticate");
            expect(result).toHaveProperty("data.authenticate.token",
"Hello World");
            expect(result).toHaveProperty("data.authenticate.createdAt");
            expect(result).toHaveProperty("data.authenticate.expiresAt");
            expect(result).toHaveProperty("data.authenticate.userUuid");
            expect(result).toHaveProperty("data.authenticate.
refreshToken");
        });
});
```

Integration testing

Unfortunately, there isn't just yet a nice way to test AppSync Serverless configuration in conjunction with the code to prove the setup will be working as expected after the deployment (we can consider this also as 'local acceptance tests'). This is because AppSync resolvers are stored as YAML definition, so we can't directly use local Apollo/Express servers to handle this setup. What we can do however is to use Serverless Framework to trigger Lambda functions locally and move the full setup testing to remote (end-to-end) tests.

To invoke Serverless tests we simply trigger the following (where the JSON file contains the Lambda event data):

```
serverless invoke local --function authentication --path ./test/invoke
/events/data.json
```

The above can be triggered from a unit test and the output can be parsed giving us half-way-through integration/local-remote test.

We can also mock external services with one of the mock servers e.g. https://www.npmjs.com/package/mockttp

In this way, we're able to introduce integration tests on two levels:

- local integration tests (calling mocks)
- remote integration tests (calling real services).

Remote acceptance testing

End to end tests (remote acceptance) can be implemented with one of the following Node modules:

- https://www.npmjs.com/package/graphql-request
- https://www.npmjs.com/package/graphql-client

..or any other GraphQL client that can be used in tests.

These tests will use an existing (deployed) GraphQL API and can be triggered as a deployment step. Separate repository is recommended.

Examples of how to use the clients are provided in the links above.