Lab Mocking en Azure functions

Inhoud

Introductie

Als eerste gaan we een kennismaken met nock (https://github.com/nock/nock).

Nock is een node library om http(s) requests te mocken. Dit gebruikt je om de afhankelijkheid van externe services die via HTTP(S) worden aangeroepen uit je test te halen. Bijvoorbeeld een externe service, maar het kan ook een eigen endpoint zijn, zoals een eigen API.

In de map 'nock' zie je een aantal voorbeelden.

De map 'user-api-function' bevat een Azure HTTP function met een zelf gebouwde REST interfaces met een 3 tal endpoints. Zie de open api definition.

Voor deze api gaan we een aantal unittesten maken.

Start

1. Check de SIG repository uit indien je dat nog niet hebt gedaan.
2. Open Visual Studio code en open de niet de root map, maar de *user=api-function* map.

Zoals gezegd is dit een eenvoudige REST API met 3 (werkende) endpoints.

Als je wilt kun je de API testen door de function lokaal te starten.

3. Hiervoor dien je de Azure Function Core Tools te installeren. Installeer versie 3.x. Zie https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local?tabs=v3%2Cwindows%2Ccsharp%2Cportal%2Cbash%2Ckeda voor de instructies.
4. Nadat je dit gedaan hebt, installeer je de Azure Functions extension. Zie https://docs.microsoft.com/en-us/azure/azure-functions/functions-develop-vs-code?tabs=nodejs voor meer informatie.
5. Nu dien je het project te initiëren voor gebruik met de Azure functions extension. Start het command palette (ctrl+shift+P op Windows) en kies voor *Azure Functions: Initialize Project for Use with VS Code*, selecteer de huidige map als daar om gevraagd wordt

Het project is nu klaar om gestart te worden.

Run als eerste in een terminal venster (ctrl+`):

```
npm clean-install
```

De relevante dependencies zijn al voorgedefinieerd in *package.json*.

Druk op **F5 (Run/Debug)** en de function zal lokaal gestart worden. Als je gebruik maakt van de REST client VS Code plugin kun je de voorbeeld request in de README uitvoeren door deze te selecteren en in de command palette te kiezen voor *Rest client: Send Request*.

Je zult dan zien dat je een foutmelding (met een 200 OK – wat eigenlijk niet juist is) krijgt omdat de function de database niet kan benaderen. Hiervoor kun je de *docker-compose.yml* file gebruiken om een database met een client te starten. Zie ook de instructies in de README.

Je zult merken dat het waarschijnlijk handig is om nog wat extra plugins in VS Code te installeren, zoals:

- Eslint
- Jest Runner
- Jest Snippets of Jest Snippets Standard Style
- npm Intellisense
- Swagger Viewer als je de api file in preview mode wilt kunnen zien
- Prettier – Code formatter

Het project heeft een eslint configuratie voor Javascript Standard style (https://standardjs.com/).

Mocking REST endpoints

In deze lab gaan we aan de slag met **nock**.

Start

Maak een map *test* aan in de root map van user-api-function.

Maak hierin een nieuw bestand aan: *user-api.test.js*

Kopieer het volgende code block er in:

```
const nock = require('nock')
const axios = require('axios').default

describe('Retrieving list of users', () => {
  beforeEach(async () => {
    nock.cleanAll()
  })

  it('should return an empty list', async () => {
    const expectedResponse = []
    const uri = 'https://api.amis.dev'
    nock(uri)
      .get('/users')
      .reply(200, expectedResponse)
    const resp = await axios.get(`${uri}/users`)
    expect(resp.status).toBe(200)
    expect(resp.data).toEqual(expectedResponse)
  })
})
```

We maken gebruik van de axios library voor het doen van REST calls.

Installeer nu eerst deze library plus nock:

```
npm i axios nock --save-dev
```

Run de unittest:

```
npm test
```

Als het goed is slaagt de test:

```
> npm test

> user-api@1.0.0 pretest C:\Projects\amis\sig-javascript-unittesting\user-api-function
> npx eslint --fix "**/*.js"


> user-api@1.0.0 test C:\Projects\amis\sig-javascript-unittesting\user-api-function
> npx jest --ci --reporters=default --reporters=jest-junit --detectOpenHandles

 PASS  test/user-api.test.js
  Retrieving list of users
    √ should return an empty list (86 ms)

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |       0 |        0 |       0 |       0 |
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.957 s
Ran all test suites.
```

Je zult zien dat er een file *junit.xml* is aangemaakt en een nieuwe map *coverage*.

Junit.xml bevat het testresult en de coverage map de coverage van de code als de unittest ook code raakt. Dit is nu nog niet het geval. In de submap lcovreport zal in dat geval een html file per javascript module komen te staan.

Opdrachten

Opdracht 1

Voeg nu zelf 2 unittesten toe voor het endpoint **GET /users**.

1. Als de api wel 1 of meer rijen teruggeeft.
   Check ook op het aantal rijen.
2. Als de api een error status (zoals 400, 404, 500) teruggeeft
   Check op zowel de status code als de error tekst.

Let op: axios geeft een exception als de status geen 2xx is.

Tip: Bij exceptions icm met async code is het verstandig om aan te geven dat je een x aantal assertions verwacht.

```
expect.assertions(#)
```

```
try {
   await axios.get(<URL>)
} catch (error) {
   …
}
```

Opdracht 2

Maak 3 unittesten voor het endpoint **GET /user**.

Maak een nieuwe groep aan:

```
describe('Retrieve a single user', () => {

})
```

En plaats hier de 3 unittesten:

1. Als je een id meegeeft, dat er een record terugkomt
2. Als je een id meegeeft en het record wordt niet gevonden
3. Als er een niet numerieke id meegeeft en je een fout terugkrijgt uit de api

Opdracht 3

Maak 2 unittesten voor het endpoint **POST /user**.

Maak een nieuwe groep aan:

```
describe('Add a new user', () => {

})
```

En plaats hierin de 3 unittesten:

1. Als je nieuw record succesvol hebt aangemaakt
2. Als het aanmaken van het record fout gaat omdat verplichte velden niet worden meegegeven
3. Als het aanmaken van het record fout gaat omdat er al een record bestaat met dezelfde voornaam, achternaam en adres combinatie

Een uitgewerkt voorbeeld staat in de branch *mocking-tests* in de file *user-api.test.js*

Azure function unittests

Start

De volgende stap is het toevoegen van unittesten voor de Azure function zelf.

Het uiteindelijke doel is om alle Javascript modules te raken.

De Azure function is een HTTP function die bestaat uit een map per endpoint (createUser, getUser en getUsers) en daarnaast een map voor de gemeenschappelijke code (common).

Laten we beginnen met getUsers. Dat is gekoppeld aan het endpoint GET /users. Zie ook het bestand *function.json* in de map *getUsers.* Er zijn 2 variabelen gedefieerd: req en res. De variabele *req* zal de request krijgen en die vind je ook terug als parameter van de main function in *index.js*. De variabele *res* bevat de response die je zelf kunt vullen.

Opdrachten

Introductie

We gaan nu een unittest maken voor de main function, vergelijkbaar met wat je in de eerste opdracht in de vorige paragraaf hebt gedaan.

Maak een nieuw bestand *getusers.test.js* aan in de map *test*.

Begin weer met de groep definitie:

```
describe('Retrieving list of users', () => {
})
```

En hierbinnen de eerste test (vergelijkbaar met de 1e opdracht:

```
  it('should return an empty list', async () => {
  })
```

Alleen nu gaan we niet de http call mocken en deze aanroepen, maar we roepen de echte function aan.

Deze moet dan eerst geïmporteerd worden:

```
const getUsers = require('../getUsers')
```

Voeg de test code toe:

```
  it('should return an empty list', async () => {
    const expectedResponse = []
    await getUsers(context, request)
  })
```

Type de regel met de await getUsers met de hand in. Je zult dan zien dat VS Code al aangeeft dat er 2 parameters zijn: context en req. En beide zul je moeten meegeven.

We beginnen met de *req* variabele. Hier geef je de query parameters, body en header variabelen op.

Voor het ophalen van de lijst met users is het niet nodig om deze variabelen een waarden te geven. Dus initialiseer de variabele maar als een leef object:

```
const request = {}
```

De context parameter

De volgende variabele is wat complexer: *context*.

Deze variabele wordt door de Azure functions runtime automatisch toegekend en bevat metadata van de functions, zoals de de naam van de function en functionapp, maar ook de methodes om logging te doen: context.log(…). En de *res* variabele die we nodig hebben om de http status en eventueel een body en header terug te geven.

Deze variabele zou je kunnen mocken, alleen dan kun je lastig controleren of bepaalde meldingen wel worden aangemaakt of dat de function het juiste resultaat teruggeeft. Daarom introduceren we een helper mock class (deze methode wordt ook bij het Eneco project gebruikt).

Mocking context parameter

Maak in de *test* map een nieuwe bestand aan: *testhelper.js*. En zet er de volgende code in:

```
/* istanbul ignore file */
class Res {
  constructor () {
    this.keys = []
    this.status = 200
  }

  setHeader (key, value) {
    this.keys[key] = [value]
  }
}

/**
   * Helper class as replacement of context
   * Only implemented log method, as redirect to console.log
   * And log property
   *
   * @class ContextLogger
   */
class ContextLogger {
  constructor () {
    const self = this
    this.log_messages = []
    this.res = new Res()

    this.log = function (message) {
      throw new Error('Do not use default log level')
    }
```

```
    this.log.info = function (msg) {
      self.log_messages.push(msg)
    }

    this.log.warn = function (msg) {
      self.log_messages.push(msg)
    }

    this.log.error = function (msg) {
      self.log_messages.push(msg)
    }
  }
}

module.exports = {
  ContextLogger
}
```

De class *Res* is voor het mock van de *res* variabele, waarbij ook de method setHeader is geimplementeerd. De class *ContextLogger* is voor de mock van de *context* variabele, waarbij de *res* variabele wordt geinitialiseerd en de log method is geimplementeerd zoals deze ook in de echte method geimplementeerd is.

De bovenstaande implementatie geeft bij het gebruik van context.log een exception omdat we ervoor gekozen hebben om de logging altijd via de nested logging methodes info, warn en error te laten lopen. Wil je toch ook het gebruik van context.log toestaan, dan zul je de bovenstaande code moeten aanpassen.

Nog 1 laatste opmerking over deze code: de 1e regel met het commentaar is belangrijk zodat de code coverage bepaling dit bestand niet meeneemt.

Om de mock context te gebruiken voeg je nu aan je test de volgende regel toe:

```
const context = new ContextLogger()
```

Als je de code met de hand intypt (en niet copy-paste doet), dan zal VS code de suggestie geven om de class uit de testhelper.js te importeren. Anders zul je dit zelf moeten doen:

```
const { ContextLogger } = require('./testhelper')
```

Nu willen we checken of de function een status 200 teruggeeft en het gewenste resultaat: een lege lijst.

Voeg de volgende 2 assertions toe:

```
expect(context.res.status).toBe(200)
expect(context.res.body).toEqual(expectedResponse)
```

Je ziet: het lijkt om de eerdere test, maar niet wat anders. Dit komt omdat het resultaat van de function altijd in **context.res** terecht komt.

Uitvoeren test

Start de test maar eens:

```
npx jest test/getusers.test.js
```

Hé, dat is niet het verwachte resultaat: de test faalt op de http status code. Deze is 500 ipv 200.

```
> npx jest test/getusers.test.js
 FAIL  test/getusers.test.js
  Retrieving list of users
    × should return an empty list (16 ms)

  ● Retrieving list of users › should return an empty list

    expect(received).toBe(expected) // Object.is equality

    Expected: 200
    Received: 500

       8 |     const context = new ContextLogger()
       9 |     await getUsers(context, request)
    > 10 |     expect(context.res.status).toBe(200)
         |                                ^
      11 |     expect(context.res.body).toEqual(expectedResponse)
      12 |   })
      13 | })

      at Object.<anonymous> (test/getusers.test.js:10:32)
```

Waarom zou dat zijn?

Als je naar de code kijkt of de debugger gebruikt om door de code heen te lopen, zul je zien dat het komt door de aanroep van de function executeQuery(). Deze geeft een exception. Die weliswaar netjes wordt afgevangen, maar niet het gewenste resultaat geeft.

Deze function voert een query op de database uit. Omdat je de database op dit moment nog niet tot onze beschikking hebben, zullen we de aanroep van de database moeten mocken. In dit geval kunnen we dat doen door de aanroep van onze eigen function executeQuery() te mocken.

Voeg de volgende code toe aan het begin van de file:

```
const db = require('../common/database')
jest.mock('../common/database', () => ({
  ...(jest.requireActual('../common/database')),
  executeQuery: jest.fn()
}))
```

En voeg de volgende regel toe aan het begin van de unittest:

```
db.executeQuery.mockReturnValue(expectedResponse)
```

En probeer de test opnieuw.

Nu zou de test wel goed moeten gaan:

```
> npx jest test/getusers.test.js
 PASS  test/getusers.test.js
  Retrieving list of users
    √ should return an empty list (3 ms)
```

Als je wilt testen of de function daadwerkelijk is aangeroepen zul je een spy moeten toevoegen. De totale unittest wordt daarmee:

```
describe('Retrieving list of users', () => {
  it('should return an empty list', async () => {
    const expectedResponse = []
    db.executeQuery.mockReturnValue(expectedResponse)
    const executeQueryCall = jest.spyOn(db, 'executeQuery')
    const request = {}
    const context = new ContextLogger()
    await getUsers(context, request)
    expect(context.res.status).toBe(200)
    expect(context.res.body).toEqual(expectedResponse)
    expect(executeQueryCall).toHaveBeenCalled()
    expect(executeQueryCall).toHaveBeenCalledTimes(1)
  })
})
```

Als je tegen dingen aanloopt met mocking kijk ook naar: https://jestjs.io/docs/jest-object

Opdracht 1

Voeg nu aan deze testgroep ook de andere 2 testen toe die je in de nock test hebt gemaakt:

1. Als de api wel 1 of meer rijen teruggeeft.
   Check ook op het aantal rijen.
2. Als de api een error status (zoals 400, 404, 500) teruggeeft
   Check op zowel de status code als de error tekst.

Opdracht 2

Maak een nieuw unittest aan voor de getUser function met de volgende testen:

1. Als je een id meegeeft, dat er een record terugkomt
2. Als je een id meegeeft en het record wordt niet gevonden
3. Als er een niet numerieke id meegeeft en je een fout terugkrijgt uit de api

Plaats de unittest in een eigen file.

Je zult zien dat je bij deze unittest de *req* variabele wilt zult moeten vullen om de test werkend te krijgen. De ontstane fout geeft je een hint wat je moet doen.

Opdracht 3

Maak een nieuwe unittest aan voor de createUser function met de volgende testen:

1. Als je nieuw record succesvol hebt aangemaakt
2. Als het aanmaken van het record fout gaat omdat verplichte velden niet worden meegegeven
3. Als het aanmaken van het record fout gaat omdat er al een record bestaat met dezelfde voornaam, achternaam en adres combinatie

Plaats de unittest in een eigen file.

Omdat createUser gebruik maakt van een andere function in de database module, zul je deze hier moeten mocken.

Bij de 2e test zul je ook zien dat je iets nieuws moet bedenken om deze test goed te laten verlopen.

Zorg ervoor dat uiteindelijk bij **`npm test`** alles succesvol is.

Mogelijk gaan er nog testen fout omdat oude mock instellingen worden onthouden. Zorg ervoor dat je altijd met een schone lei begint. Bijvoorbeeld door een beforeEach toe te voegen aan de testgroep waarin je de mock opschoont. Check je jest documentatie om te achterhalen wat je hiervoor moet doen.

In de branch *function-unittests* vind je een uitgewerkt voorbeeld.

Opdracht 4

Een extra opdracht is om bij een aantal testen die je gedaan hebt ook te checken op de info dan wel error logging die je kan terugvinden in het *context* object.

Een handige test helper function zou de volgende code kunnen zijn:

```
function countNoOfLogLines (context, text) {
  return context.log_messages.filter(m => m.indexOf(text) >= 0).length
}
```

Plaats deze function in het *testhelper.js* bestand en exporteer de function. Daarna kun je die in je unitest files weer importeren en gebruiken.

Opdracht 5

Als je nog tijd en zin hebt kun je kijken of je de code coverage omhoog kan schroeven.

Nu zul op zo'n 63% coverage op statement, 42% op branch en 58% op function niveau uitkomen:

```
----------------|---------|----------|---------|---------|-------------------
File            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------|---------|----------|---------|---------|-------------------
All files       |   63.35 |    41.93 |   57.14 |   63.35 |
 common         |   48.93 |       50 |      50 |   48.93 |
  database.js   |   23.07 |        0 |       0 |   23.07 | 22-31,45-58,67-77
  logging.js    |   78.94 |       80 |   66.66 |   78.94 | 36,47-48,75
  utils.js      |     100 |       50 |     100 |     100 | 20-24
 createUser     |   57.77 |    23.07 |      50 |   57.77 |
  index.js      |    92.3 |       75 |     100 |    92.3 | 54-55
  validation.js |   10.52 |        0 |       0 |   10.52 | 30-61
 getUser        |    91.3 |       75 |     100 |    91.3 |
  index.js      |    91.3 |       75 |     100 |    91.3 | 42-43
 getUsers       |   81.25 |       50 |     100 |   81.25 |
  index.js      |   81.25 |       50 |     100 |   81.25 | 25-27
----------------|---------|----------|---------|---------|-------------------
Test Suites: 4 passed, 4 total
Tests:       16 passed, 16 total
Snapshots:   0 total
Time:        1.759 s
Ran all test suites.
```

Het mooiste is als je minimaal alle rode tekst weg kan krijgen.

Als je het bestand <workspace folder>\coverage\lcov-report\index.html opent in een browser kun je mooi zien waar welke code wel en niet geraakt wordt.