Veebiarendus

Back-End arendus

Martti Raavel

martti.raavel@tlu.ee

Tänased teemad

- Node API automaattestimine

Express API automaattestimine

Testimine

Testimine on protsess, mille käigus hinnatakse tarkvara kvaliteeti, kontrollides selle funktsionaalsust, jõudlust ja turvalisust. Testimise eesmärk on leida vead ja puudused ning kinnitada, et tarkvara töötab ootuspäraselt ja vastab määratletud nõuetele.

Testimise liigid

- Ühiktestimine (Unit Testing)
- Integratsioonitestimine (Integration Testing)
- Süsteemitestimine (System Testing)
- Vastuvõtutestimine (Acceptance Testing)
- jne

Testimise meetodid

- Käsitsi testimine
- Automaatne testimine

Node.js testiraamistikud

- Mocha
- Jest
- SuperTest
- jne

Mocha

Mocha on paindlik ja lihtne testiraamistik Node.js jaoks, mis võimaldab teste kirjutada ja käivitada.

Mocha paigaldamine ja kasutamine

```
npm install mocha
```

Loo projekti juurkausta kataloog nimega `test` ja selle sisse testifail, näiteks `test.js`.

> Mocha otsib testifaile vaikimisi kataloogist `test`.
>
> Edaspidi võib testifaile nimetada näiteks `ressurss.test.js`.

`package.json` skripti lisamine

```
{
  "scripts": {
    "test": "mocha --exit"
  }
}
```

> Edaspidi saab teste käivitada käsurealt käsklusega `npm test`.

Mocha testide struktuur

Mocha kasutab BDD stiilis testide kirjutamiseks funktsioone `describe`, `it` ja `before`, `after`, `beforeEach`, `afterEach`.

`describe` funktsioon võimaldab teste grupeerida ja kirjeldada.

`it` funktsioon kirjeldab üksikuid teste.

Mocha testide struktuuri näide

```
describe('Testi grupi nimetus', function() {
  it('Testi kirjeldus', function() {
    // Testi kood
  });
  it('Testi kirjeldus', function() {
    // Testi kood
  });
});
```

Mocha testide näide

```
const assert = require('assert');
const sum = require('./sum');

describe('sum', () => {
  it('should return 3 when the input is 1 and 2', () => {
    assert.strictEqual(sum(1, 2), 3);
  });

  it('should return 5 when the input is 2 and 3', () => {
    assert.strictEqual(sum(2, 3), 5);
  });

  it('should return 0 when the input is 0 and 0', () => {
    assert.strictEqual(sum(0, 0), 0);
  });
});
```

> `assert` on Node.js standardne moodul, mis võimaldab teha 'väiteid'.

Chai

Chai on populaarne asertsiooniraamatukogu JavaScripti jaoks, mida kasutatakse koos testimisraamistikega nagu Mocha ja Jest.

Chai paigaldamine

```
npm install chai@4.4.1
```

> Märkus: Kasutame Chai versiooni 4.4.1, kuna see on viimane versioon, mis toetab Node.js `require` süntaksit.

Chai kasutamine

```
it('should return 3 when the input is 1 and 2', () => {
  expect(sum(1, 2)).to.equal(3);
});
```

Chai asertsioonistiilid

- `expect`
- `should`
- `assert`

Chai cheatsheet: https://devhints.io/chai

Expect asertsioonistiil

```
expect(object)
  .to.equal(expected)
  .to.deep.equal(expected)
  .to.be.a('string')
  .to.include(val)
  .be.ok(val)
  .be.true
  .be.false
  .to.exist
  .to.be.null
  .to.be.undefined
  .to.be.empty
  .to.be.arguments
  .to.be.function
  .to.be.instanceOf
  .to.have.property
```

Supertest

SuperTest on Node.js raamistik, mis võimaldab testida HTTP päringuid ja vastuseid.

Supertest'i eelised võrreldes teiste raamistikega

- Lihtne integreerida
- Ei vaja serveri käivitamist
- Rikkalikud päringu võimalused
- Kergesti loetavad `assertionid`
- Toetab asünkroonseid teste

Supertest paigaldamine

```
npm install supertest
```

Supertest kasutamine

```
const request = require('supertest');
const app = require('../app');
const { expect } = require('chai');
const { describe, it } = require('mocha');

describe('GET /ping', function() {
  it('responds with json', function(done) {
    request(app)
      .get('/')
      .set('Accept', 'application/json')
      .expect('Content-Type', /json/)
      .expect(200, done);
  });
});
```

Eeltöö

Enne, kui saame hakata oma rakendusele test kirjutama, peame oma rakendust veel natukene struktureerima. Nimelt tuleb selleks, et me saaksime API-t automaatselt testida, eraldada API loomise kood ja API käivitamise kood, kuna testimine eeldab, et API on eraldi moodulina kasutatav.

Struktureerimine

Loome oma projekti juurkausta eraldi faili `server.js`, kuhu tõstame `app.js` failist selle osa, mis käivitab serveri.

```
const app = require('./app');

app.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

See tähendab, et `app.js` failis on ainult API loomise kood ja see tuleb sealt ka eraldi eksportida.

Supertesti testide kirjutamine

```
const request = require('supertest');
const { expect } = require('chai');
const { describe, it } = require('mocha');

const app = require('../app');

const okResponse = {
  success: true,
  message: 'pong',
};

describe('GET /ping', () => {
  it('should return 200 OK', async () => {
    const response = await request(app).get('/ping');
    expect(response.status).to.equal(200);
    expect(response.body).to.deep.equal(okResponse);
  });
});
```

Sisselogimist nõudvate päringute testimine

Nagu ka manuaalselt testides, on automaattestides vaja sisselogimist nõudvate päringute testimiseks saata sisselogimiseks vajalikud andmed.

```
const request = require('supertest');
const { expect } = require('chai');
const { describe, it, before } = require('mocha');
let token;

// Sisselogimine

before(async () => {
  const response = await request(app)
    .post('/login')
    .send(user);
  token = response.body.token;
});

// Sisselogimist nõudvate päringute testimine

it('should return 200 OK', async () => {
  const response = await request(app)
    .get('/protected')
    .set('Authorization', `Bearer ${token}`);
  expect(response.status).to.equal(200);
});
```

Testid ja andmebaas

Praegu on meil olukord, kus meil tehakse teste kasutades päris andmebaasi. See on halb praktika, kuna testid võivad andmebaasi muuta ja andmebaas võib muuta testide tulemusi.

Mõistlik oleks teha testide jaoks eraldi andmebaas, mida kasutatakse ainult testide tegemiseks. Selline andmebaas tuleks luua iga kord, kui testid käivitatakse ja kustutada pärast testide lõppu. See annab meile võimaluse teste kirjutada selliselt, et me teame väga täpselt, mis andmed andmebaasis on ja millised on siis eeldavalt päringute vastused.

Keskkonnamuutujad

Me peame tekitama kuidagi võimaluse rakendusel vahet teha, kas ta töötab testide käivitamise ajal või mitte. Selleks saame kasutada keskkonnamuutujat `NODE_ENV`, mille väärtuseks saab olla `test` või `development`.

Keskkonnamuutuja on muutuja, mille väärtus on seadistatud operatsioonisüsteemi keskkonnas ja mida saab kasutada rakenduse käitumise mõjutamiseks.

Keskkonnamuutujaid saab rakendustes kasutada mitmel viisil:

- `.env` failid
- käsurea argumentidega
- operatsioonisüsteemi keskkonnamuutujad
- ...

Keskkonnamuutujate kasutamine

Selleks, et saaksime keskkonnamuutujaid lugeda, paigaldame Node.js mooduli `dotenv`.

```
npm install dotenv
```

Andmebaasi seadistamine

Muudame `config.js` faili nii, et seadistused, mida rakendus kasutab, sõltuvad keskkonnamuutujast.

```
require('dotenv').config();

const config = {
  development: {
    port: 3000,
    jwtSecret: 'my-secret-key',
    saltRounds: 10,
    db: {
      host: 'localhost',
      ...
    },
  },
  test: {
    port: 3000,
    jwtSecret: 'my-secret-key',
    saltRounds: 10,
    db: {
      host: 'localhost',
      ...
    },
  },
};

const env = process.env.NODE_ENV || 'development'

module.exports = config[env];
```

`package.json` skripti täiendamine

Nüüd, kui meil on võimalus keskkonnamuutujaid kasutada, saame täiendada testide käivitamise skripti nii, et enne testide käivitamist seatakse keskkonnamuutuja `NODE_ENV` väärtuseks `test`. Siinkohal peame meeles pidama, et erinevad operatsioonisüsteemid kasutavad erinevaid süntakse keskkonnamuutujate seadistamiseks. Selleks, et me ei peaks muretsema, kuidas seda teha, kasutame Node.js moodulit `cross-env`.

```
npm install cross-env
```

Ja täiendame testide käivitamise skripti järgmiselt:

```
{
  "scripts": {
    "test": "cross-env NODE_ENV=test mocha --exit"
  }
}
```

Test andmebaasi lisamine

Nüüd on meil vaja luua testide jaoks eraldi andmebaas, sinna andmed lisada ja testide käivitamisel andmebaas luua ja pärast testide lõppu kustutada.

Kõige lihtsam on teha selle jaoks eraldi `docker`-i konteiner, nagu me oleme varem teinud - nüüd peab lihtsalt silmas pidama, et selleks, et test ja `päris` andmebaas saaksid koos eksisteerida, peavad nad kasutama erinevaid porte.

Test andmebaasi seadistamine

Lisaks loome testide jaoks eraldi ka andmebaasi ühenduse faili koos andmete lisamisega andmebaasi.

Faili leiate repositooriumist `testDbSetup.js`.

Selles failis imporditakse testandmebaasi jaoks loodud `sql` fail, jupitatakse see eraldi päringuteks ja lisatakse andmebaasi.

Testide täiendamine

Nüüd lisame testidesse enne teste andmebaasi loomise:

```
const { setupTestDatabase } = require('../testDbSetup');

before(async () => {
  await setupTestDatabase();
});
```

> Pärast testide lõppu me hetkel andmebaasi ära ei kustuta, kuna seda tehakse iga testi käivitamisel juba niikuinii.

Testidega kaetus

Ilmselt oleks meil vaja teada ka seda, kui suur osa meie rakenduse koodist on testidega kaetud. Selleks on olemas mitmeid tööriistu, millest üks populaarsemaid on `nyc`.

```
npm install nyc
```

Täienda testide käivitamise skripti järgmiselt:

```
{
  "scripts": {
    "test": "cross-env NODE_ENV=test nyc mocha --exit"
  }
}
```

Nüüd saab testide käivitamisel näha, kui suur osa koodist on testidega kaetud.

Kodune töö

Püüa lisada oma API-le nii palju teste, kui võimalik