TD SR10

Qualité du code

Exercice 1 : tests unitaires

Dans cette partie, nous allons utiliser le framework Jest pour implémenter des tests unitaires :

1. Lancer le terminal (ou utiliser l'onglet terminal de vscode)
2. Aller dans le dossier de votre projet : par exemple, cd myapp
3. Lancer la commande suivante : npm install -- save-dev jest
4. Modifier package.json en indiquant la commande pour lancer les tests et activer le calcul de la couverture de code (modifications surlignées en jaune):

```
{
 "name": "myapp",
 "version": "0.0.0",
 "private": true,
 "scripts": {
  "start": "node ./bin/www",
  "test": "jest"
 },
 "jest": {
  "collectCoverage":true
  },
 "dependencies": {
  "cookie-parser": "~1.4.4",
  "debug": "~2.6.9",
  "ejs": "^3.1.9",
  "express": "^4.18.2",
  "http-errors": "~1.6.3",
  "jest": "^29.5.0",
  "morgan": "~1.9.1",
  "save-dev": "^0.0.1-security"
 }
}
```

5. Créer un test basique :
   - Ajouter dans votre projet un dossier test :
   - Créer un fichier « hello.test.js » :

```
test("equality matchers", () => {
  expect(3-2).toBe(1);
  expect(4/2).not.toBe(0);
})
```

- ***Voir l'annexe1 pour plus d'information sur la création des tests et les différentes options proposées par Jest.***

6. Lancer le test via la commande « run » : > npm run test

```
> myapp@0.0.0 test
> jest

 PASS  test/hello.test.js
  √ equality matchers (4 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:        1.117 s
Ran all test suites.
```

7. Dans le résultat, on constate que la couverture de code est à 0% car le test qu'on a lancé ne teste aucun code développé dans notre projet. Nous allons maintenant créer un fichier « model.test.js » (= une unité de test) pour tester les fonctions CRUD (model/) :

```
const DB= require ("../model/db.js");
const model= require ("../model/user.js");
describe("Model Tests", () => {

    beforeAll(() => {
        // des instructions à exécuter avant le lancement des tests
    });

    afterAll((done) => {
        function callback (err){
            if (err) done (err);
            else done();

        }
        DB.end(callback);

    });

    test ("read user",()=>{
       nom=null;
       function cbRead(resultat){
        nom = resultat[0].nom;
        expect(nom).toBe("test");

       }

         model.read("test@test.fr",  cbRead);

    });

})
```

Question :

1. Identifier vos fonctions à tester et les scénarios de tests.
2. Implémenter ces tests.

Exercice 2 : tester des routes

Dans cette partie, nous allons utiliser le module « **supertest »** pour pouvoir tester la partie http d'une application web comme notre projet : npm install -- save-dev supertest

Le code ci-dessous qui va tester la méthode get de la route racine « / » définie dans le script « route/index.js » de notre projet.

```
// test/route.test.js file

const request = require("supertest");
const app = require("../app");

describe("Test the root path", () => {
  test("It should response the GET method", done => {
    request(app)
      .get("/")
      .then(response => {
        expect(response.statusCode).toBe(200);
        done();
      });
  });
});
```

* Argument « done » : Jest attendra que le callback done soit appelé avant de terminer le test.

Question :

1. Développer les tests automatiques de vos routes et méthodes http du projet SR10 en utilisant différents types de vérification :
    - Le code http retourné (200, 404, …)
    - Le type de contenu retourné
    - Le contenu
    - Autres (les entêtes http, …)

    Voir : https://dev.to/franciscomendes10866/testing-express-api-with-jest-and-supertest-3gf

Annexe 1 - Documentation Framework Jest

1. Structure d'un fichier de test

Il existe trois méthodes principales dans un fichier de test :

- describe() - C'est une suite de scripts de test qui donne une description externe pour la suite de tests.
- test() - C'est le plus petit cas de test unitaire écrit pour être exécuté. La chaîne entre guillemets représente le nom du test.

- expect() - C'est une assertion. Chaque instruction test() a une fonction expect() qui prend une valeur et attend un retour sous forme vraie.

2. Des exemples sur les matchers les plus couramment utilisés avec les tests Jest

1.1 Egalité

```
test("equality matchers", () => {

  expect(2*2).toBe(4);

  expect(4-2).not.toBe(1);

 })
```

1.2 Comparaison booléenne

```
test("truthy operators", () => {

  var name="Software testing help"

  var n = null

  expect(n).toBeNull()

  expect(name).not.toBeNull


  // name has a valid value

  expect(name).toBeTruthy()



  //fail - as null is non success

  expect(n).toBeTruthy()
```

```
  // pass - null treated as false or negative
  expect(n).toBeFalsy()
  // 0 - treated as false
  expect(0).toBeFalsy()
})
```

1.3 Comparaison entre nombres

```
test("numeric operators", () => {

  var num1 = 100;
  var num2 = -20;
  var num3 = 0;

  // greater than
  expect(num1).toBeGreaterThan(10)

  // less than or equal
  expect(num2).toBeLessThanOrEqual(0)

  // greater than or equal
  expect(num3).toBeGreaterThanOrEqual(0)
})
```

1.4 Expressions régulières sur les chaines de caractères

```
test("string matchers",() => {
  var string1 = "software testing help - a great resource for testers"
```

```
  // test for success match
  expect(string1).toMatch(/test/);

  // test for failure match
  expect(string1).not.toMatch(/abc/)
})
```

1.5 Autres : voir la documentation officielle sur https://jestjs.io/docs/expect