METTRE A DISPOSITION DES UTILISATEURS UN SERVICE INFORMATIQUE

JEST

16 MARS 2023

IPSSI

Saint-Quentin-en-Yvelines

Sommaire :

1. JestJS

JestJS est un Framework de test JavaScript open source populaire pour les applications web, développé par Facebook. Il offre une suite complète d'outils pour écrire, exécuter et gérer des tests unitaires, des tests d'intégration et des tests de bout en bout pour des applications développées en JavaScript, TypeScript, React, Angular et d'autres frameworks JavaScript.

Jest est conçu pour être facile à utiliser, rapide et fiable. Il offre des fonctionnalités telles que la mise en cache intelligente pour accélérer les tests, des outils de débogage intégrés, des fonctions d'assertion avancées pour les tests unitaires, des tests parallèles pour une exécution plus rapide des tests, et bien plus encore.

Jest est également extensible grâce à son architecture modulaire, qui permet aux développeurs d'ajouter des plugins et des extensions pour étendre ses fonctionnalités de base (superTest dans mon cas). Cela en fait un choix populaire pour les développeurs qui cherchent à automatiser leurs tests et améliorer la qualité de leur code.

2. SuperTest

Supertest est une bibliothèque JavaScript open source utilisée pour tester les API HTTP en simulant les requêtes HTTP. Elle est couramment utilisée pour les tests d'API pour les applications web basées sur Node.js.

Supertest fournit une API simple mais puissante pour simuler des requêtes HTTP, envoyer des données de formulaire, télécharger des fichiers, définir des en-têtes de requête et gérer les cookies. Elle peut être utilisée avec des Framework web populaires tels que Express, Koa, Hapi et bien d'autres encore.

La bibliothèque est conçue pour être facile à utiliser et offre des fonctionnalités telles que des assertions intégrées pour vérifier les réponses HTTP, la gestion de sessions pour tester les sessions d'utilisateur et la prise en charge des requêtes avec des jetons d'authentification.

En utilisant Supertest, les développeurs peuvent automatiser les tests d'API pour s'assurer que leur code fonctionne correctement et répond aux spécifications. Cela permet de détecter les erreurs de manière précoce et de les corriger avant qu'elles ne deviennent des problèmes plus graves dans le développement.

3. Situation

Le but de ce projet est de réaliser les tests d'intégration et d'acceptation d'un service.

Nous allons faire une api de recette de cuisine qui aura une page de connexion, une page utilisateurs avec sa classe et son CRUD(pareil pour les recettes) et une page serveurs pour les différentes requêtes.

4. Création de la base de donnée

Avant de commencer à développer le back-end de l'application afin d'utiliser JestJS, il m'a fallu commencé à créé une base de donnée du nom de 'sitecuisine' qui recensera une table utilisateur qui contiendra seulement un nom et un mail ainsi que table recette constituer d'un nom et d'un temps en seconde.

```
MariaDB [(none)]> use sitecuisine
Database changed
MariaDB [sitecuisine]> select * from utilisateur;
+----+-------+-----------------+
| id | nom   | mail            |
+----+-------+-----------------+
|  3 | laury | laury@gmail.com |
|  4 | bee   | bee@gmail.com   |
+----+-------+-----------------+
2 rows in set (0.001 sec)

MariaDB [sitecuisine]> select * from recette;
+----+-----------------+---------+
| id | nom             | temps   |
+----+-----------------+---------+
|  1 | tarte aux pommes | 225sec  |
|  2 | Tajine          | 9999sec |
+----+-----------------+---------+
2 rows in set (0.001 sec)
```

Voilà à quoi ressemblera la base de donnée.

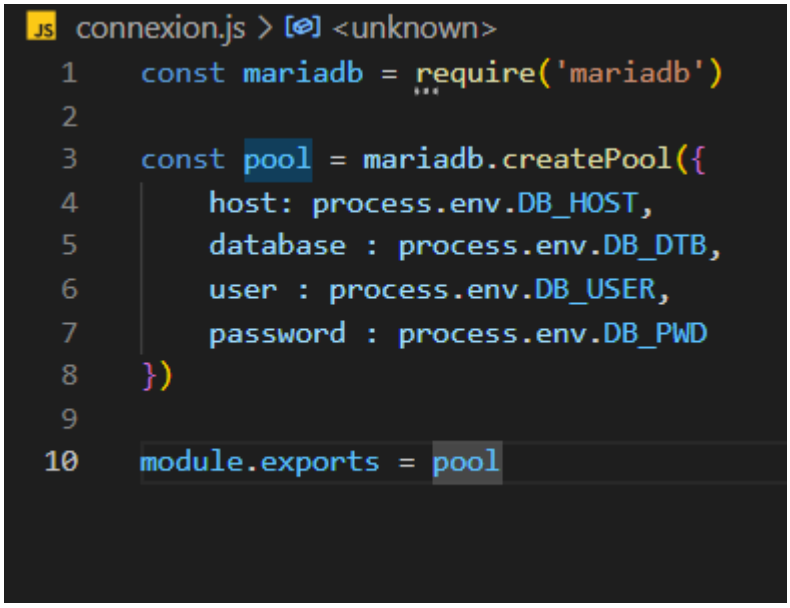
Elle sera largement suffisante pour pouvoir faire nos test.

5. Connexion à la base de donnée

Afin de pouvoir communiquer avec notre base de donnée dans le code, il faut s'y connecter en y spécifiant notre nom d'utilisateur (mariadb par exemple), notre mot de passe, le nom de notre base de donnée et le host utilisé.

Nous stockerons toutes ces informations dans une constante appelé « pool » que nous exporterons afin de l'utiliser dans nos classes.

La page de connexion à la base de donnée ressemble à ceci :



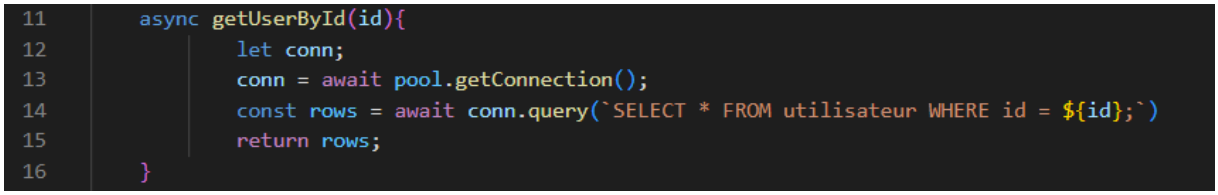
Les variable DB_HOST,DB_DTB, DB_USER et DB_PWD sont stocké dans un .env (variable d'environnement) qui n'est pas mis en ligne lors de la publication du projet car il contient des informations sensible tel que les mot de passe.

6. Classe Utilisateur

Le fichier utilisateur.js exporte une classe Utilisateur qui fournit des fonctions pour récupérer, ajouter, modifier et supprimer des utilisateurs dans la base de données. La classe Utilisateur utilise l'objet de pool de connexion exporté par connexion.js pour se connecter à la base de données.

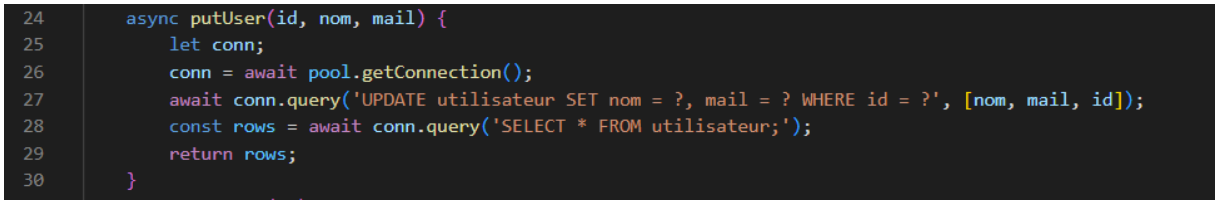
getUser() - récupère tous les utilisateurs dans la base de données.

getUserById() - récupère un utilisateur dans la base de données en utilisant l'ID.



addUser() - ajoute un utilisateur dans la base de données avec un nom et une adresse e-mail.

putUser() - modifie un utilisateur dans la base de données en utilisant l'ID.



delUser() - supprime un utilisateur dans la base de données en utilisant l'ID.

```
31      async delUser(id){
32              let conn;
33              conn = await pool.getConnection();
34              await conn.query(`DELETE FROM utilisateur WHERE id = ?;`,[id])
35              const rows = await conn.query(`SELECT * FROM utilisateur;`)
36              return rows
37      }
38
39  }
40
41  module.exports = Utilisateur
```

6. Classe Recette

Tout comme utilisateur.js le fichier recette.js exporte une classe Recette qui fournit des fonctions pour récupérer, ajouter, modifier et supprimer des recettes dans la base de données. La classe Recette utilise l'objet de pool de connexion exporté par connexion.js pour se connecter à la base de données.

getRecette() - récupère toutes les recettes dans la base de données.

recette.js > Recette > getRecetteById

```
1   require('dotenv').config()
2   const pool = require('./connexion')
3
4   class Recette{
5     async getRecette(){
6         let conn;
7         conn = await pool.getConnection();
8         const rows = await conn.query('SELECT * FROM recette;')
9         return rows;
10    }
```

getRecetteById() - récupère une recette dans la base de données en utilisant l'ID.

```
11  async getRecetteById(id){
12      let conn;
13      conn = await pool.getConnection();
14      const rows = await conn.query(`SELECT * FROM recette WHERE id = ${id};`)
15      return rows;
16  }
```

addRecette() - ajoute une recette dans la base de données avec un nom et un temps de préparation.

```
17  async addRecette(nom, temps){
18      let conn;
19      conn = await pool.getConnection();
20      await conn.query('INSERT INTO recette(nom, temps) VALUES (?,?)',[nom, temps])
21      const rows = await conn.query('SELECT * FROM recette;')
22      return rows;
23  }
```

putRecette() - modifie une recette dans la base de données en utilisant l'ID.

```
24  async putRecette(id, nom, temps){
25      let conn;
26      conn = await pool.getConnection();
27      await conn.query(`UPDATE recette SET nom = ?, temps = ? WHERE id = ${id}`,[nom, temps])
28      const rows = await conn.query('SELECT * FROM recette;')
29      return rows;
30  }
```

delRecette() - supprime une recette dans la base de données en utilisant l'ID.

```
31    async delRecette(id){
32        let conn;
33        conn = await pool.getConnection();
34        await conn.query(`DELETE FROM recette WHERE id = ?;`,[id])
35        const rows = await conn.query(`SELECT * FROM recette;`)
36        return rows;
37    }
38
39  }
40
41  module.exports = Recette;
42
```

7. Serveur

Le fichier server.js crée une application Express, définit des routes et écoute les demandes HTTP entrantes.

L'application Express utilise les classes Utilisateur et Recette exportées par utilisateur.js et recette.js pour fournir des routes pour interagir avec la base de données.

Les routes disponibles dans l'application Express sont les suivantes:

GET /utilisateur - récupère tous les utilisateurs dans la base de données.

server.js > ...

```
1   const express = require('express')
2   const app = express()
3   var cors = require('cors')
4
5   const Utilisateur = require('./utilisateur')
6   user = new Utilisateur()
7   const Recette = require('./recette')
8   recette = new Recette()
9
10  app.use(express.json())
11  app.use(cors())
12
13  app.get('/utilisateur', async(req, res) =>{
14      let personne = {nom: req.body.nom, mail: req.body.mail}
15      let result = await user.getUser(personne)
16      res.status(200).json(result)
17  })
18
```

GET /utilisateur/:id - récupère un utilisateur dans la base de données en utilisant l'ID.

```
19  app.get('/utilisateur/:id', async(req, res) =>{
20      let id = parseInt(req.params.id)
21      let result = await user.getUserById(id)
22      res.status(200).json(result)
23  })
24
```

POST /utilisateur - ajoute un utilisateur dans la base de données avec un nom et une adresse e-mail.

```
25  app.post('/utilisateur',async(req,res)=>{
26      let personne = {nom: req.body.nom, mail: req.body.mail}
27      let result = await user.addUser(personne)
28      res.status(200).json(result)
29  })
```

PUT /utilisateur/:id - modifie un utilisateur dans la base de données en utilisant l'ID.

```
30
31    app.put('/utilisateur/:id',async(req,res)=>{
32        let id = parseInt(req.params.id)
33        let result = await user.putUser(id)
34        res.status(200).json(result)
35    })
36
```

DELETE /utilisateur/:id - supprime un utilisateur dans la base de données en utilisant l'ID.

```
37    app.delete('/utilisateur/:id', async(req, res) =>{
38        let id = parseInt(req.params.id)
39        let result = await user.delUser(id)
40        res.status(200).json(result)
41    })
42
```

GET /recette - récupère toutes les recettes dans la base de données.

```
43  app.get('/recette', async(req, res) =>{
44        let laRecette = {nom: req.body.nom, temps: req.body.temps}
45        let result = await recette.getRecette(laRecette)
46        res.status(200).json(result)
47    })
48
```

GET /recette/:id : récupère la recette correspondant à l'id passé en paramètre.

```
49    app.get('/recette/:id', async(req, res) =>{
50        let id = parseInt(req.params.id)
51        let result = await recette.getRecetteById(id)
52        res.status(200).json(result)
53    })
```

POST /recette : ajoute une nouvelle recette avec les informations passées en paramètre.

```
55    app.post('/recette',async(req,res)=>{
56        let laRecette = {nom: req.body.nom, temps: req.body.temps}
57        let result = await recette.addRecette(laRecette)
58        res.status(200).json(result)
59    })
```

PUT /recette/:id : modifie la recette correspondant à l'id passé en paramètre avec les nouvelles informations passées en paramètre.

```
61  app.put('/recette/:id',async(req,res)=>{
62      let id = parseInt(req.params.id)
63      let result = await recette.putRecette(id)
64      res.status(200).json(result)
65  })
66
```

DELETE /recette/:id : supprime la recette correspondant à l'id passé en paramètre.

```
67  app.delete('/recette/:id', async(req, res) =>{
68      let id = parseInt(req.params.id)
69      let result = await recette.delRecette(id)
70      res.status(200).json(result)
71  })
72
73  app.listen(8000, () =>{
74      console.log("Serveur à l'écoute");
75  })
76
77  module.exports = app
```

8. Inclusion de JestJS

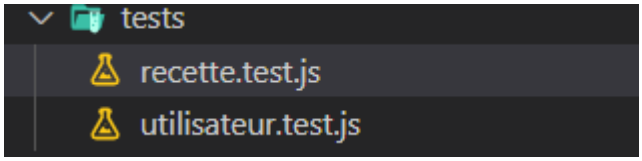
Tout d'abord, il on va installer jest tet supertest avec npm la commande :

npm i jest supertest

puis dans votre packages.json nous allons effectuer cette modification:

```
"scripts": {
  "test": "jest",
  "start": "node server.js"
},
```

Pour mettre en place ses tests, il va falloir créer un dossier à part dans notre projet qui s'appellera « tests » qui regroupera tous les tests de notre applications (tests unitaires, intégrations etc...). Tous les fichiers de ce dossier doivent finir par l'extension ,".test.js" comme ceci.



(Je ne vais montrer que les testes fait pour utilisateur pour éviter la redondance)

Nous allons commencer par importer notre classe Utilisateur, notre serveur et superTest comme ceci :

```
tests > utilisateur.test.js > user
1   const request = require('supertest');
2   const app = require ('../server');
3   const user = require('../utilisateur')
4
```

Puis dans une fonction « describe() » qui est une fonction Jest dans la quel nous allons regrouper nos tests, elle prend deux arguments, une chaîne de caractères qui décrit le comportement que l'on veut tester et une fonction qui contient les tests correspondants.

Voici le code nécessaire pour tester toutes nos requête :

```
const request = require('supertest');
const app = require ('../server');
const utilisateur = require('../utilisateur');

describe('Test des routes', () => {
  it('Retourne la liste des utilisateurs', async () => {
    const response = await request(app).get('/utilisateur');
    expect(response.status).toBe(200);
    expect(response.body).toEqual(await new utilisateur().getUser());
  });
```

Ce premier block vérifie si ont peu récupérer la liste des utilisateurs

```
  it('Retourne un utilisateur grâce à son ID', async () => {
    const response = await request(app).get('/utilisateur/3');
    expect(response.status).toBe(200);
    expect(response.body).toEqual(await new utilisateur().getUserById(3));
  });
```

Puis ce deuxième block vérifie si ont peu récupérer un utilisateur par son id

```
  it('Ajoute un utilisateur', async () => {
    const response = await request(app)
      .post('/utilisateur')
      .send({ nom: 'Test', mail: 'test@test.com' });
    expect(response.status).toBe(200);
    expect(response.body).toEqual(await new utilisateur().addUser('Test',
'test@test.com'));
  });
```

Celui-ci vérifie si ont peu ajouter un utilisateur

```
  it('Met à jour un utilisateur', async () => {
    const response = await request(app)
      .put('/utilisateur/4')
      .send({ nom: 'Test Modifié', mail: 'test@test.com' });
    expect(response.status).toBe(200);
    expect(response.body).toEqual(await new utilisateur().putUser(4, 'Test
Modifié', 'test@test.com'));
  });
```

Puis celui là vérifie si ont peu modifier un utilisateur

```
  it('Supprime un utilisateur', async () => {
    const response = await request(app).delete('/utilisateur/6');
    expect(response.status).toBe(200);
    expect(response.body).toEqual(await new utilisateur().delUser(6));
  });
});
```

Et enfin on vérifie si ont peu supprimer un utilisateur

Pour la suppression, je rajoute un utilisateur avec le bonne id afin de ne pas avoir de problème lors de la suppression.

```
MariaDB [sitecuisine]> insert into utilisateur (nom, mail) values ('deleteUeser', 'user@delete.com');
Query OK, 1 row affected (0.005 sec)

MariaDB [sitecuisine]> select * from utilisateur
    -> ;
+----+------------+----------------+
| id | nom        | mail           |
+----+------------+----------------+
|  3 | laury      | laury@gmail.com |
|  4 | bee        | bee@gmail.com   |
|  6 | deleteUeser | user@delete.com |
+----+------------+----------------+
3 rows in set (0.001 sec)
```

Et je remarque que cela à bien fonctionner

```
MariaDB [sitecuisine]> select * from utilisateur;
+----+-------+-----------------+
| id | nom   | mail            |
+----+-------+-----------------+
|  3 | laury | laury@gmail.com |
|  4 | bee   | bee@gmail.com   |
+----+-------+-----------------+
2 rows in set (0.000 sec)

MariaDB [sitecuisine]>
```