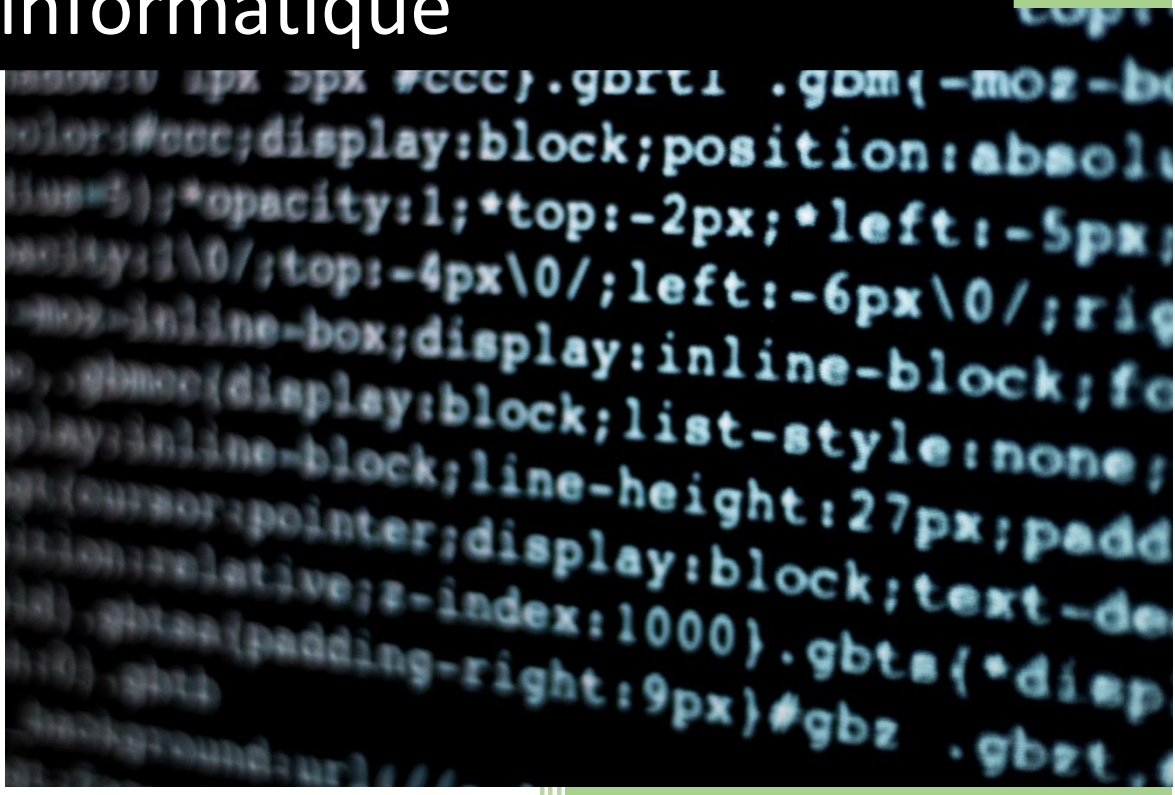
Mettre à disposition des utilisateurs un service informatique



Arthur BESNARD

Nous avons dû, dans un TP réaliser une API Rest et par la suite effectuer des tests unitaires afin de voir si les fonctions de nos classes fonctionnent.

Voici tout d'abord des captures d'écrans du projet en NodeJs.

On a ici la page de connexion 'connexion.js' permettant la connexion à la base de données du projet.

connexion.js > ...

```
1   const mysql = require('mysql2/promise');
2
3   async function getConnection() {
4     const connection = await mysql.createConnection({
5       host: 'localhost:8000',
6       user: 'root',
7       password: 'respons11',
8       database: 'cuisine'
9     });
10    return connection;
11  }
12
13  module.exports = { getConnection };
14
```

On a ici la page contenant les utilisateurs se nommant 'utilisateur.js' contenant une classe utilisateur avec le CRUD de l'utilisateur.

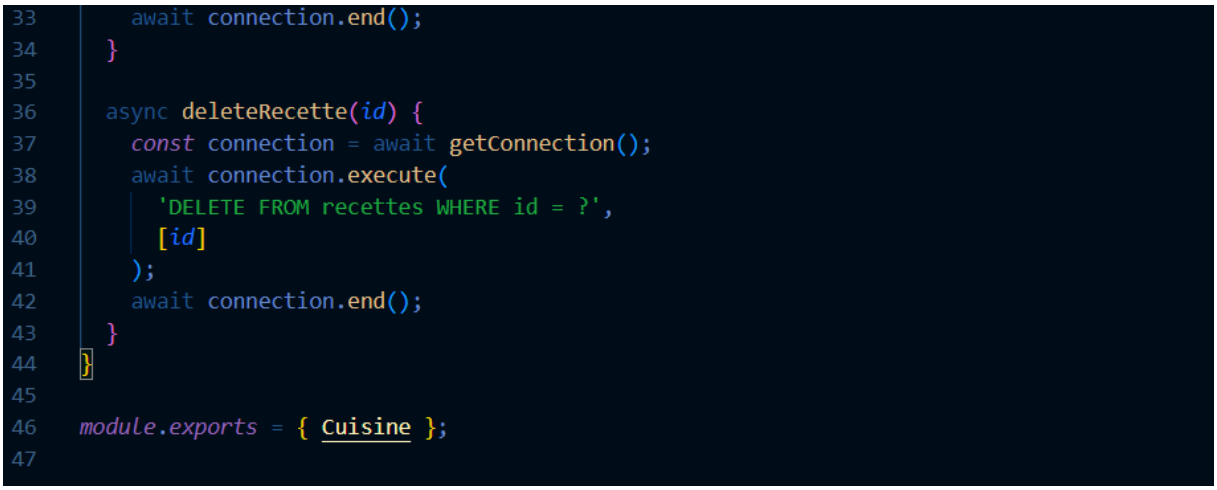
```
const mysql = require('mysql2/promise');

class Utilisateur {
  async createUtilisateur(utilisateur) {
    const connection = await mysql.createConnection({
      host: 'localhost',
      user: 'root',
      password: 'password',
      database: 'utilisateur'
    });
    const [result] = await connection.execute(
      'INSERT INTO utilisateur (nom, id, mail) VALUES (?, ?, ?)',
      [utilisateur.nom, utilisateur.id, utilisateur.mail]
    );
    await connection.end();
    return result.insertId;
  }

  async getUtilisateurById(id) {
    const connection = await mysql.createConnection({
      host: 'localhost',
      user: 'root',
      password: 'password',
      database: 'utilisateur'
    });
```

```
    const [rows, fields] = await connection.execute(
      'SELECT * FROM utilisateur WHERE id = ?',
      [id]
    );
    await connection.end();
    if (rows.length === 0) {
      return null;
    }
    return rows[0];
  }

  async updateUtilisateur(id, utilisateur) {
    const connection = await mysql.createConnection({
      host: 'localhost',
      user: 'root',
      password: 'password',
      database: 'utilisateur'
    });
    await connection.execute(
      'UPDATE utilisateur SET nom = ?,id = ?, mail = ? WHERE id = ?',
      [utilisateur.nom, utilisateur.id, utilisateur.mail]
    );
    await connection.end();
  }

  async deleteUtilisateur(id) {
    const connection = await mysql.createConnection({
      host: 'localhost',
      user: 'root',
      password: 'password',
      database: 'utilisateur'
    });
    await connection.execute(
      'DELETE FROM utilisateur WHERE id = ?',
      [id]
    );
```

```
61        );
62        await connection.end();
63      }
64    }
65
66    module.exports = Utilisateur;
67
```

On a ici la page recette ‘recette.js’ contenant la classe cuisine avec le CRUD de l’utilisateur.

```
1   const { getConnection } = require('./connexion.js');
2
3   class Cuisine {
4     async createRecette(recette) {
5       const connection = await getConnection();
6       const [result] = await connection.execute(
7         'INSERT INTO recettes (nom, description, ingredients) VALUES (?, ?, ?)',
8         [recette.nom, recette.description, recette.ingredients]
9       );
10      await connection.end();
11      return result.insertId;
12    }
13
14    async readRecette(id) {
15      const connection = await getConnection();
16      const [rows, fields] = await connection.execute(
17        'SELECT * FROM recettes WHERE id = ?',
18        [id]
19      );
20      await connection.end();
21      if (rows.length === 0) {
22        throw new Error(`Recette avec l'ID ${id} non trouvée`);
23      }
24      return rows[0];
25    }
26
27    async updateRecette(id, recette) {
28      const connection = await getConnection();
29      await connection.execute(
30        'UPDATE recettes SET nom = ?, description = ?, ingredients = ? WHERE id = ?',
31        [recette.nom, recette.description, recette.ingredients, id]
32      );
33      await connection.end();
34    }
35
36    async deleteRecette(id) {
37      const connection = await getConnection();
```



```
    await connection.end();
  }

  async deleteRecette(id) {
    const connection = await getConnection();
    await connection.execute(
      'DELETE FROM recettes WHERE id = ?',
      [id]
    );
    await connection.end();
  }
}

module.exports = { Cuisine };

```

On a ici une page ‘route.js’ qui contient les routes de notre application qui appelle la connexion à la base de données, les classes utilisateur et recette à utiliser dans vos routes.

route.js > ...

```
const express = require('express');
const { getConnection } = require('./connexion.js');
const { Utilisateur } = require('./utilisateur.js');
const { Cuisine } = require('./recette.js');

const app = express();
const port = 3000;

const connection = getConnection();

app.use(express.json());


app.post('/utilisateur', async (req, res) => {
  const utilisateur = new Utilisateur();
  const id = await utilisateur.createUtilisateur(req.body);
  res.status(201).json({ id });
});


app.get('/utilisateur/:id', async (req, res) => {
  const utilisateur = new Utilisateur();
  const resultat = await utilisateur.readUtilisateur(req.params.id);
  res.json(resultat);
});


app.put('/utilisateur/:id', async (req, res) => {
  const utilisateur = new Utilisateur();
  await utilisateur.updateUtilisateur(req.params.id, req.body);
  res.sendStatus(204);
});


app.delete('/utilisateur/:id', async (req, res) => {
  const utilisateur = new Utilisateur();
  await utilisateur.deleteUtilisateur(req.params.id);
```

```
    res.sendStatus(204);
});


app.post('/recette', async (req, res) => {
    const cuisine = new Cuisine();
    const id = await cuisine.createRecette(req.body);
    res.status(201).json({ id });
});


app.get('/recette/:id', async (req, res) => {
    const cuisine = new Cuisine();
    const resultat = await cuisine.readRecette(req.params.id);
    res.json(resultat);
});


app.put('/recette/:id', async (req, res) => {
    const cuisine = new Cuisine();
    await cuisine.updateRecette(req.params.id, req.body);
    res.sendStatus(204);
});


app.delete('/recette/:id', async (req, res) => {
    const cuisine = new Cuisine();
    await cuisine.deleteRecette(req.params.id);
    res.sendStatus(204);
});
```

Et voila notre base de données.

```
MariaDB [cuisine]> select * from utilisateur
    -> ;
+----+--------------+--------------------------+
| id | nom          | mail                     |
+----+--------------+--------------------------+
|  1 | Alice Dupont | alice.dupont@example.com |
|  2 | Bob Martin   | bob.martin@example.com   |
|  3 | Claire Garcia | claire.garcia@example.com |
+----+--------------+--------------------------+
3 rows in set (0.009 sec)

MariaDB [cuisine]> select * from recette;
+----+-----------------------+-----------------------------------------+-----------------------------------------------------------------+
| id | nom                   | description                             | ingredients                                                     |
+----+-----------------------+-----------------------------------------+-----------------------------------------------------------------+
|  1 | Tarte aux pommes      | Une délicieuse tarte aux pommes faite maison | Pâte feuilletée, pommes, sucre, cannelle                        |
|  2 | Poulet rôti           | Un plat simple et savoureux             | Poulet, sel, poivre, herbes de Provence                         |
|  3 | Risotto aux champignons | Un plat crémeux et parfumé              | Riz Arborio, champignons, bouillon de volaille, parmesan, vin blanc |
+----+-----------------------+-----------------------------------------+-----------------------------------------------------------------+
3 rows in set (0.001 sec)
```

Jest et Supertest sont deux outils couramment utilisés dans le développement de tests pour les applications web.

Jest est un framework de test pour les applications JavaScript qui fournit une plateforme complète pour l'exécution des tests.

Supertest est une bibliothèque Node.js qui permet de tester les applications web en simulant des requêtes HTTP.

En combinant ces deux outils, les développeurs peuvent écrire des tests complets pour les applications web, garantissant ainsi que l'application fonctionne correctement dans toutes les situations et en évitant les erreurs et les bugs dans le code.

Dans mon TP j'ai décidé de réaliser les tests avec les utilisateurs. J'ai réalisé des tests pour les fonctions suivantes : « PUT, POST, DELETE, GET ».

Les voici :

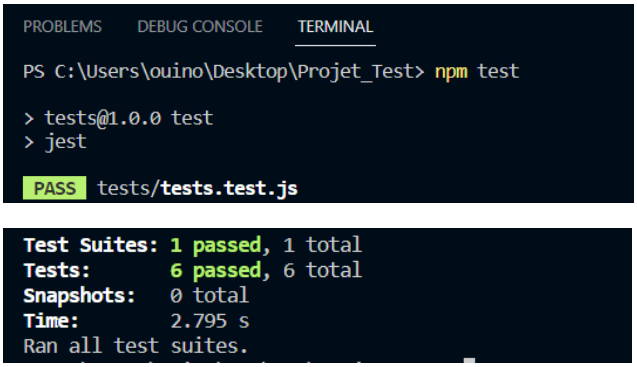
```
describe('PUT /utilisateur/:id', () => {
  it('Met a jour l-utilisateur', async () => {
    const utilisateur = new Utilisateur();
    const newUser = { nom: 'Charlie', mail: 'charlie@example.com' };
    const id = await utilisateur.createUtilisateur(newUser);
    const updatedUser = { nom: 'Charlie Smith', mail: 'charlie.smith@example.com' };
    const response = await request(app)
      .put(`/utilisateur/${id}`)
      .send(updatedUser);
    expect(response.status).toBe(204);
    const resultat = await utilisateur.readUtilisateur(id);
    expect(resultat).toHaveProperty('nom', 'Charlie Smith');
    expect(resultat).toHaveProperty('mail', 'charlie.smith@example.com');
  });

  it('', async () => {
    const response = await request(app).put('/utilisateur/99').send({ nom: 'David' });
    expect(response.status).toBe(404);
  });
});
```

```
describe('POST /utilisateur', () => {
  it('creation d-utilisateur', async () => {
    const response = await request(app)
      .post('/utilisateur')
      .send({ nom: 'Alice', mail: 'alice@example.com' });
    expect(response.status).toBe(201);
    expect(response.body).toHaveProperty('id');
    const utilisateur = new Utilisateur();
    const resultat = await utilisateur.readUtilisateur(response.body.id);
    expect(resultat).toHaveProperty('nom', 'Alice');
    expect(resultat).toHaveProperty('mail', 'alice@example.com');
  });
});
```

```
describe('DELETE /utilisateur/:id', () => {
  it('Supprime l-utilisateur', async () => {
    const utilisateur = new Utilisateur();
    const newUser = { nom: 'Emma', mail: 'emma@example.com' };
    const id = await utilisateur.createUtilisateur(newUser);
    const response = await request(app).delete(`/utilisateur/${id}`);
    expect(response.status).toBe(204);
    const resultat = await utilisateur.readUtilisateur(id);
    expect(resultat).toBeNull();
  });

  it('', async () => {
    const response = await request(app).delete('/utilisateur/99');
    expect(response.status).toBe(404);
  });
});
```

```
describe('GET /utilisateur/:id', () => {
  it('Retourne l-utilisateur', async () => {
    const utilisateur = new Utilisateur();
    const newUser = { nom: 'Bob', mail: 'bob@example.com' };
    const id = await utilisateur.createUtilisateur(newUser);
    const response = await request(app).get(`/utilisateur/${id}`);
    expect(response.status).toBe(200);
    expect(response.body).toHaveProperty('nom', 'Bob');
    expect(response.body).toHaveProperty('mail', 'bob@example.com');
  });

  it('', async () => {
    const response = await request(app).get('/utilisateur/99');
    expect(response.status).toBe(404);
  });
});
```

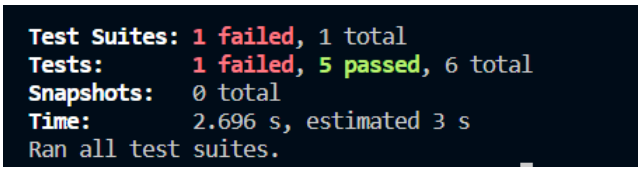
Voila le résultat lorsque tous les tests fonctionnent :



J’ai ici fait exprès de changer quelque chose dans mon code afin d’exécuter a nouveau la commande et voir si l’erreur va bien s’afficher dans le terminal. On a bien une erreur. La raison pour laquelle cette erreur se produit est que j’ai utilisé ‘utilisateur’ au lieu de ‘resultat’ dans les deux assertions qui suivent la création d'un nouvel utilisateur :

```
expect(utilisateur).toHaveProperty('nom', 'Alice');
expect(utilisateur).toHaveProperty('mail', 'alice@example.com');
```

On a donc bien l’erreur dans le terminal :



Pour corriger cette erreur, il suffit de remplacer toutes les ‘utilisateur’ par ‘resultat’ :

```
expect(resultat).toHaveProperty('nom', 'Alice');
expect(resultat).toHaveProperty('mail', 'alice@example.com');
```

Et si on refait la commande ‘npm test’, tout fonctionne parfaitement :