Documentation for Testing Login and Signup with Jest and Supertest

Overview

This document outlines the testing strategy and implementation details for the authentication endpoints in our Node.js application. We use Jest as our testing framework and Supertest for HTTP assertions to validate the functionality of our login and signup routes.

Prerequisites

- Node.js installed (preferably the latest LTS version)
- Familiarity with Jest testing framework and Supertest library
- Application must have the `server.js` setup exporting an Express app and authentication routes configured in `authRoutes.js`.

Installing Dependencies

Before running the tests, ensure the following packages are installed:

```
npm install jest supertest --save-dev
```

If using ES6 import/export statements, install Babel dependencies:

```
npm install @babel/preset-env @babel/core babel-jest --save-dev
```

Set up Babel by creating a `.babelrc` file at the root with the following configuration:

```
{
  "presets": ["@babel/preset-env"]
}
```

Configuration

Update the `package.json` to include a test script:

```
"scripts": {
 "test": "jest"
}
```

Ensure your application uses environment-specific configurations, especially for connecting to the database:

```
if (process.env.NODE_ENV === 'test') {
  require('dotenv').config({ path: '.env.test' });
} else {
  require('dotenv').config();
}
```

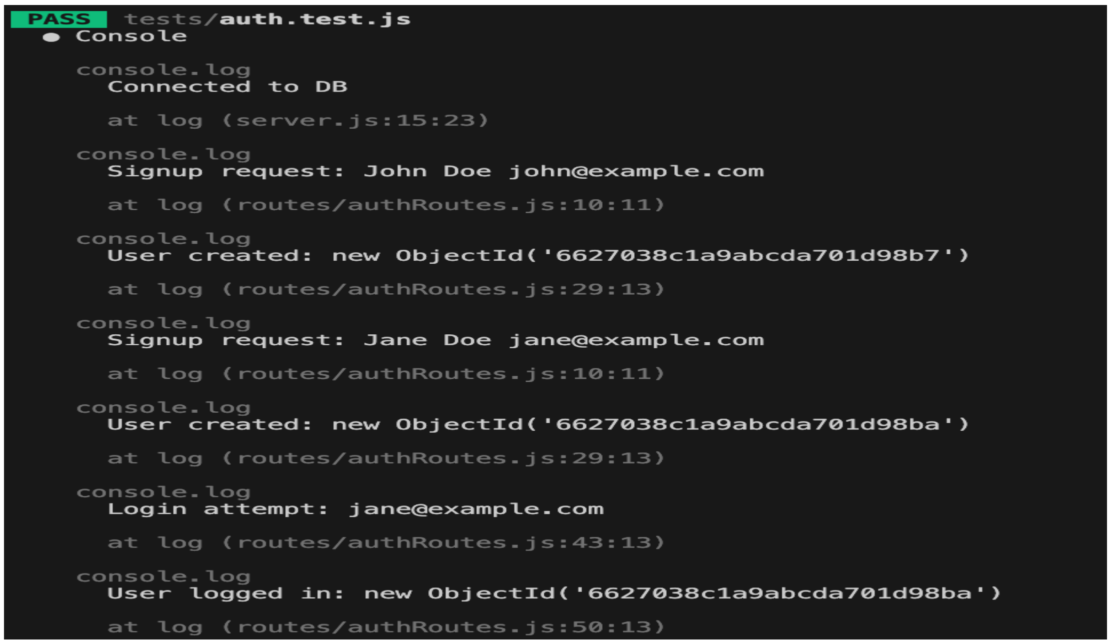
Test File Structure

Place your test files in a `tests` directory under the root:

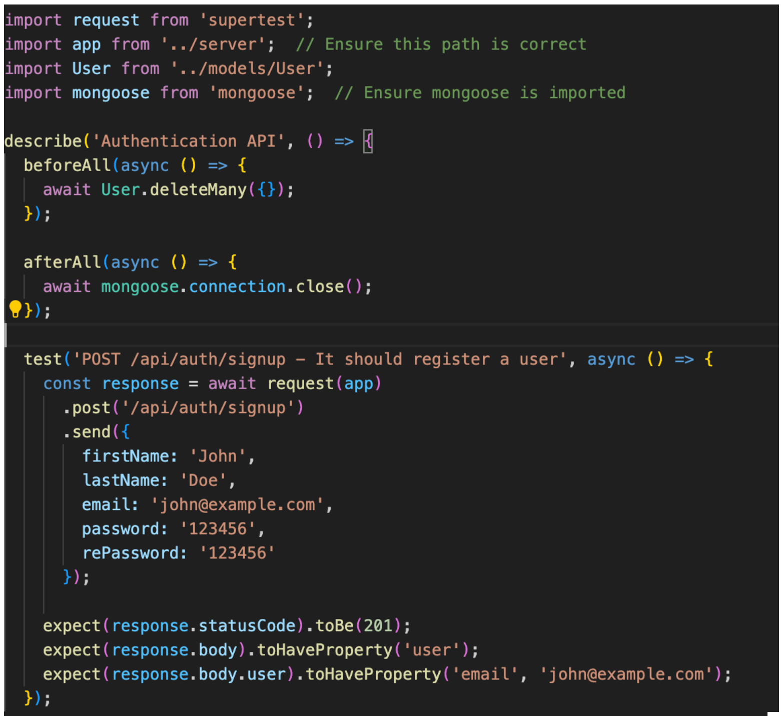
```
/backend
  /tests
    - auth.test.js
  - server.js
  - package.json
```

Sample Test Cases

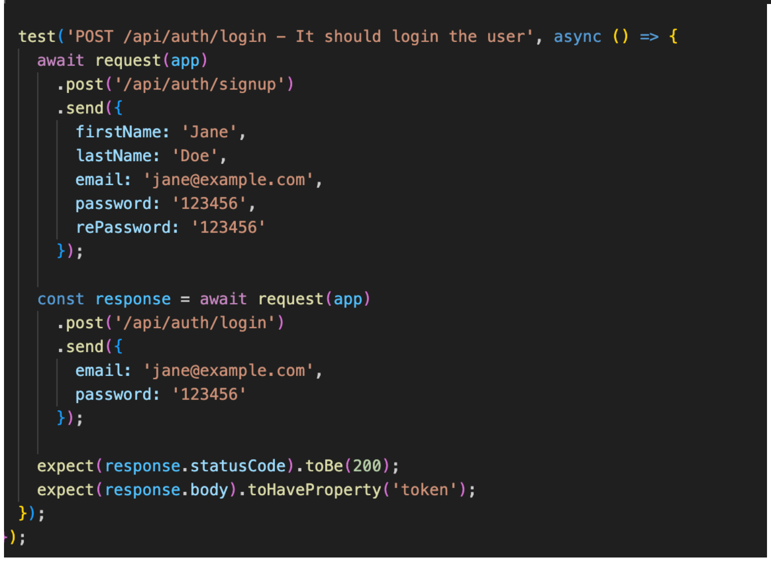
auth.test.js



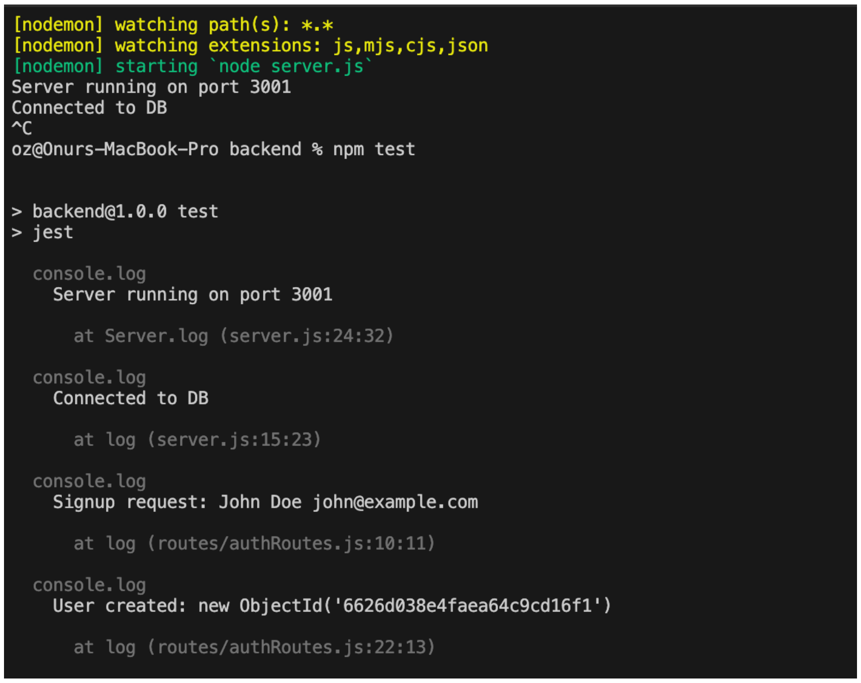
```
 PASS  tests/auth.test.js
  ● Console

    console.log
      Connected to DB

      at log (server.js:15:23)

    console.log
      Signup request: John Doe john@example.com

      at log (routes/authRoutes.js:10:11)

    console.log
      User created: new ObjectId('6627038c1a9abcda701d98b7')

      at log (routes/authRoutes.js:29:13)

    console.log
      Signup request: Jane Doe jane@example.com

      at log (routes/authRoutes.js:10:11)

    console.log
      User created: new ObjectId('6627038c1a9abcda701d98ba')

      at log (routes/authRoutes.js:29:13)

    console.log
      Login attempt: jane@example.com

      at log (routes/authRoutes.js:43:13)

    console.log
      User logged in: new ObjectId('6627038c1a9abcda701d98ba')

      at log (routes/authRoutes.js:50:13)
```



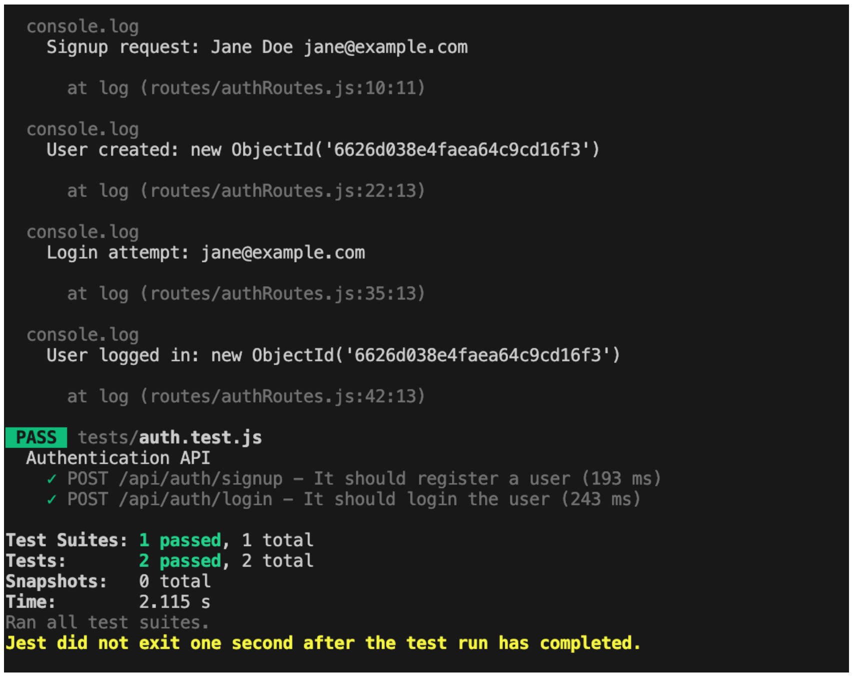
```javascript
import request from 'supertest';
import app from '../server';  // Ensure this path is correct
import User from '../models/User';
import mongoose from 'mongoose';  // Ensure mongoose is imported

describe('Authentication API', () => {
  beforeAll(async () => {
    await User.deleteMany({});
  });

  afterAll(async () => {
    await mongoose.connection.close();
  });

  test('POST /api/auth/signup - It should register a user', async () => {
    const response = await request(app)
      .post('/api/auth/signup')
      .send({
        firstName: 'John',
        lastName: 'Doe',
        email: 'john@example.com',
        password: '123456',
        rePassword: '123456'
      });

    expect(response.statusCode).toBe(201);
    expect(response.body).toHaveProperty('user');
    expect(response.body.user).toHaveProperty('email', 'john@example.com');
  });
```



```javascript
  test('POST /api/auth/login - It should login the user', async () => {
    await request(app)
      .post('/api/auth/signup')
      .send({
        firstName: 'Jane',
        lastName: 'Doe',
        email: 'jane@example.com',
        password: '123456',
        rePassword: '123456'
      });

    const response = await request(app)
      .post('/api/auth/login')
      .send({
        email: 'jane@example.com',
        password: '123456'
      });

    expect(response.statusCode).toBe(200);
    expect(response.body).toHaveProperty('token');
  });
});
```



```
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node server.js`
Server running on port 3001
Connected to DB
^C
oz@Onurs-MacBook-Pro backend % npm test


> backend@1.0.0 test
> jest

  console.log
    Server running on port 3001

      at Server.log (server.js:24:32)

  console.log
    Connected to DB

      at log (server.js:15:23)

  console.log
    Signup request: John Doe john@example.com

      at log (routes/authRoutes.js:10:11)

  console.log
    User created: new ObjectId('6626d038e4faea64c9cd16f1')

      at log (routes/authRoutes.js:22:13)
```



```
  console.log
    Signup request: Jane Doe jane@example.com

      at log (routes/authRoutes.js:10:11)

  console.log
    User created: new ObjectId('6626d038e4faea64c9cd16f3')

      at log (routes/authRoutes.js:22:13)

  console.log
    Login attempt: jane@example.com

      at log (routes/authRoutes.js:35:13)

  console.log
    User logged in: new ObjectId('6626d038e4faea64c9cd16f3')

      at log (routes/authRoutes.js:42:13)

 PASS  tests/auth.test.js
  Authentication API
    ✓ POST /api/auth/signup - It should register a user (193 ms)
    ✓ POST /api/auth/login - It should login the user (243 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        2.115 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

Product Testing Documentation

Overview

This document outlines the approach and setup for automated testing of product-related API endpoints in an Express.js application. The testing focuses on the CRUD operations associated with products, including fetching all products, adding a new product, updating an existing product, and deleting a product.

Tools and Libraries Used

- **Jest**: Testing framework for running the tests.
- **supertest:** Library to make HTTP assertions.
- **Express.js**: Web application framework.
- **Mongoose**: MongoDB object modeling tool.

Setup

The testing environment is configured to mock Mongoose model methods using Jest to simulate database operations without affecting a real database.

```
javascript
import request from 'supertest';
import app from '../server';  // Path to the Express application
import Product from '../models/Product';  // Mongoose model for products
import { generateToken } from '../middleware/jwt';  // Utility to generate
JWT for auth

// Mock the Product model
jest.mock('../models/Product', () => ({
  find: jest.fn().mockResolvedValue([...]),
  findByIdAndUpdate: jest.fn().mockResolvedValue({...}),
  findByIdAndDelete: jest.fn().mockResolvedValue({...}),
  // Add more mocked methods as needed
}));
```

Test Cases

GET /api/products/getall

- **Objective**: Ensure that all products can be retrieved successfully.
- **Method**: `GET`
- **Path**: `/api/products/getall`
- **Mock Setup**: Return a fixed list of products when the `find` method is called.
- **Expectations**:
  - Status code should be 200.
  - Response body should match the mocked product list.

POST /api/products/add

- **Objective**: Test the addition of a new product.
- **Method**: `POST`
- **Path**: `/api/products/add`
- **Mock Setup**: Mock the `save` method of the Product model to simulate database insertion.
- **Expectations**:

    - Status code should be 201 if the product is added successfully.
    - The response should match the product data sent in the request.

PUT /api/products/:productId

- **Objective**: Validate the updating of a product.
- **Method**: `PUT`
- **Path**: `/api/products/:productId`
- **Mock Setup**: Simulate updating a product by returning the updated product data.
- **Expectations**:
    - Status code should be 200.
    - The response should reflect the updated product data.

DELETE /api/products/:productId

- **Objective**: Confirm that a product can be deleted.
- **Method**: `DELETE`
- **Path**: `/api/products/:productId`
- **Mock Setup**: Simulate product deletion and return a success message or status.
- **Expectations**:
    - Status code should be 200.
    - Response should confirm successful deletion.

Running Tests

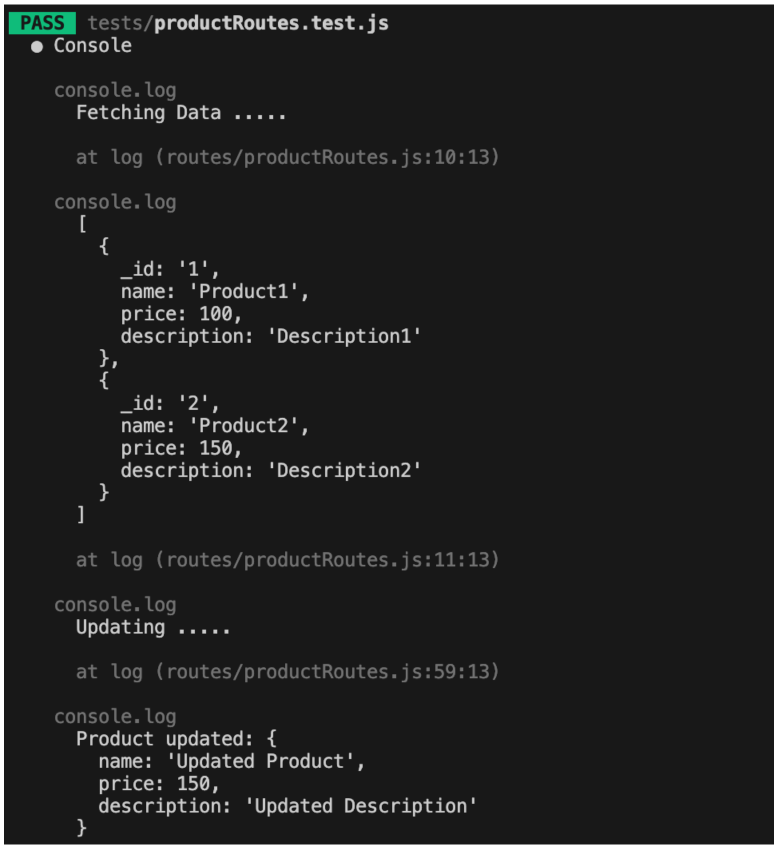
To execute the tests, run the following command in your terminal:

```
bash
npm test
```

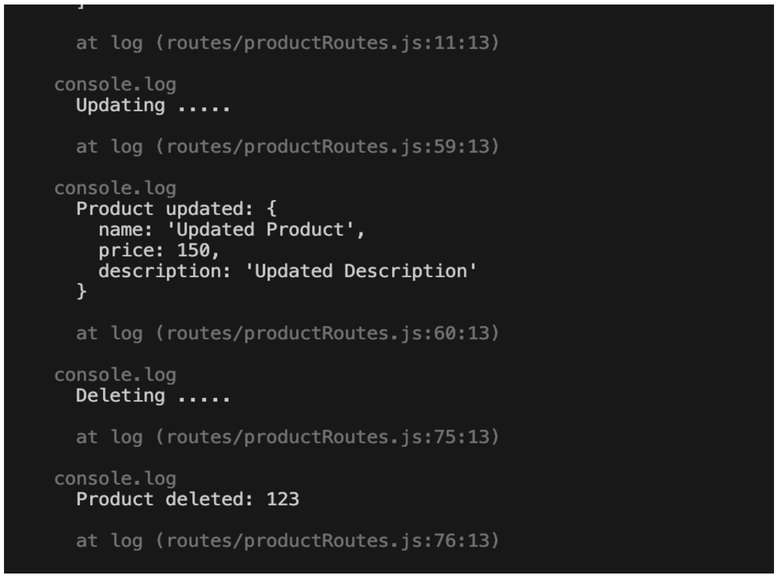
This command triggers the Jest test runner configured in the `package.json` file and outputs the test results.

Conclusion

This setup and documentation provide a structured approach to testing API endpoints related to product management in an Express.js application, ensuring that each component functions correctly before deployment.



```
PASS  tests/productRoutes.test.js
  ● Console

    console.log
      Fetching Data .....

      at log (routes/productRoutes.js:10:13)

    console.log
      [
        {
          _id: '1',
          name: 'Product1',
          price: 100,
          description: 'Description1'
        },
        {
          _id: '2',
          name: 'Product2',
          price: 150,
          description: 'Description2'
        }
      ]

      at log (routes/productRoutes.js:11:13)

    console.log
      Updating .....

      at log (routes/productRoutes.js:59:13)

    console.log
      Product updated: {
        name: 'Updated Product',
        price: 150,
        description: 'Updated Description'
      }
```



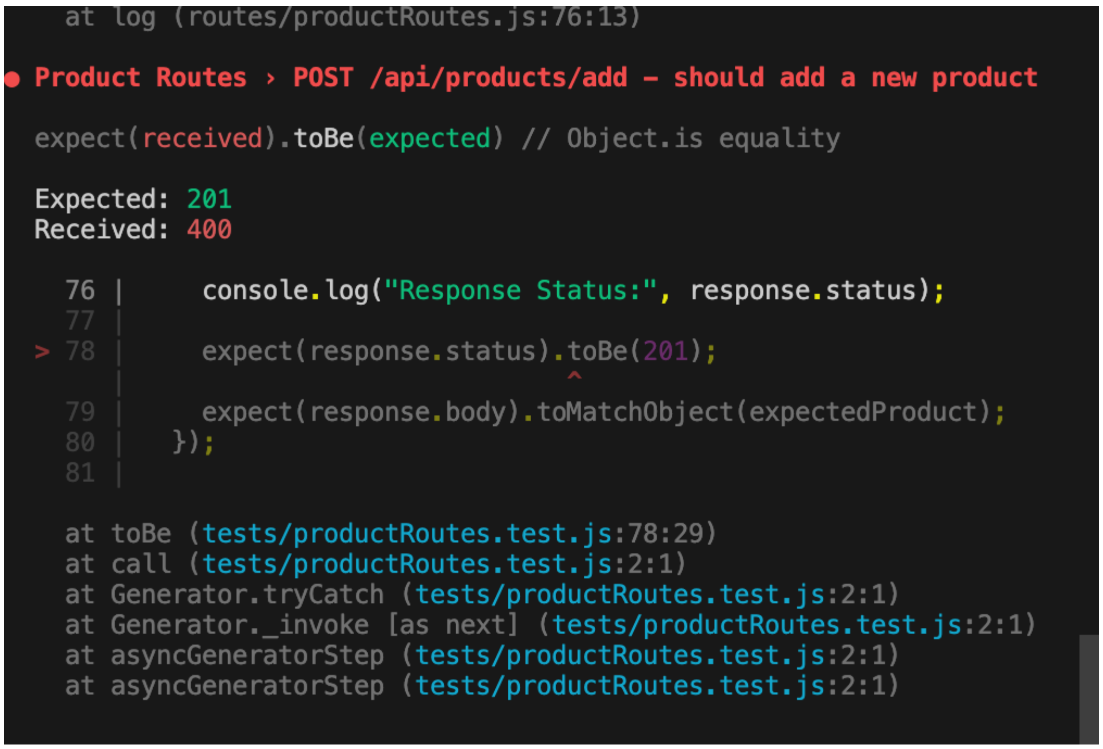
```
      at log (routes/productRoutes.js:11:13)

    console.log
      Updating .....

      at log (routes/productRoutes.js:59:13)

    console.log
      Product updated: {
        name: 'Updated Product',
        price: 150,
        description: 'Updated Description'
      }

      at log (routes/productRoutes.js:60:13)

    console.log
      Deleting .....

      at log (routes/productRoutes.js:75:13)

    console.log
      Product deleted: 123

      at log (routes/productRoutes.js:76:13)
```

Reasons for "TypeError: Product is not a constructor" Error

1. **Incorrect Mocking of the Product Model**:
   - If the `Product` model is not correctly mocked, attempting to create a new instance with `new Product(...)` will fail because the mock may not be returning a constructor function but rather an object or other non-constructor type.

   - Ensure that your jest mock for `Product` correctly simulates a constructor. If `Product` is a class, the mock should return an object that can be instantiated.

2. **Inconsistent Exports/Imports**:
   - This error can also occur if the actual `Product` model is not exported as a class but is being imported and used as one.
   - Verify that your real `Product` model is exported as a class or constructor function and that it matches how you're trying to use it in your routes.
3. **Configuration Issues in Jest**:
   - Your Jest setup might not be correctly configured to handle ES modules or the specific export style of your `Product` model. This can lead to the imported module not behaving as expected.
   - Ensure that your Babel and Jest configurations support the syntax and module types you're using in your project.



```
    at log (routes/productRoutes.js:76:13)

● Product Routes › POST /api/products/add - should add a new product

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

  Expected: 201
  Received: 400

    76 |     console.log("Response Status:", response.status);
    77 |
  > 78 |     expect(response.status).toBe(201);
       |                             ^
    79 |     expect(response.body).toMatchObject(expectedProduct);
    80 |   });
    81 |

    at toBe (tests/productRoutes.test.js:78:29)
    at call (tests/productRoutes.test.js:2:1)
    at Generator.tryCatch (tests/productRoutes.test.js:2:1)
    at Generator._invoke [as next] (tests/productRoutes.test.js:2:1)
    at asyncGeneratorStep (tests/productRoutes.test.js:2:1)
    at asyncGeneratorStep (tests/productRoutes.test.js:2:1)
```