TDD

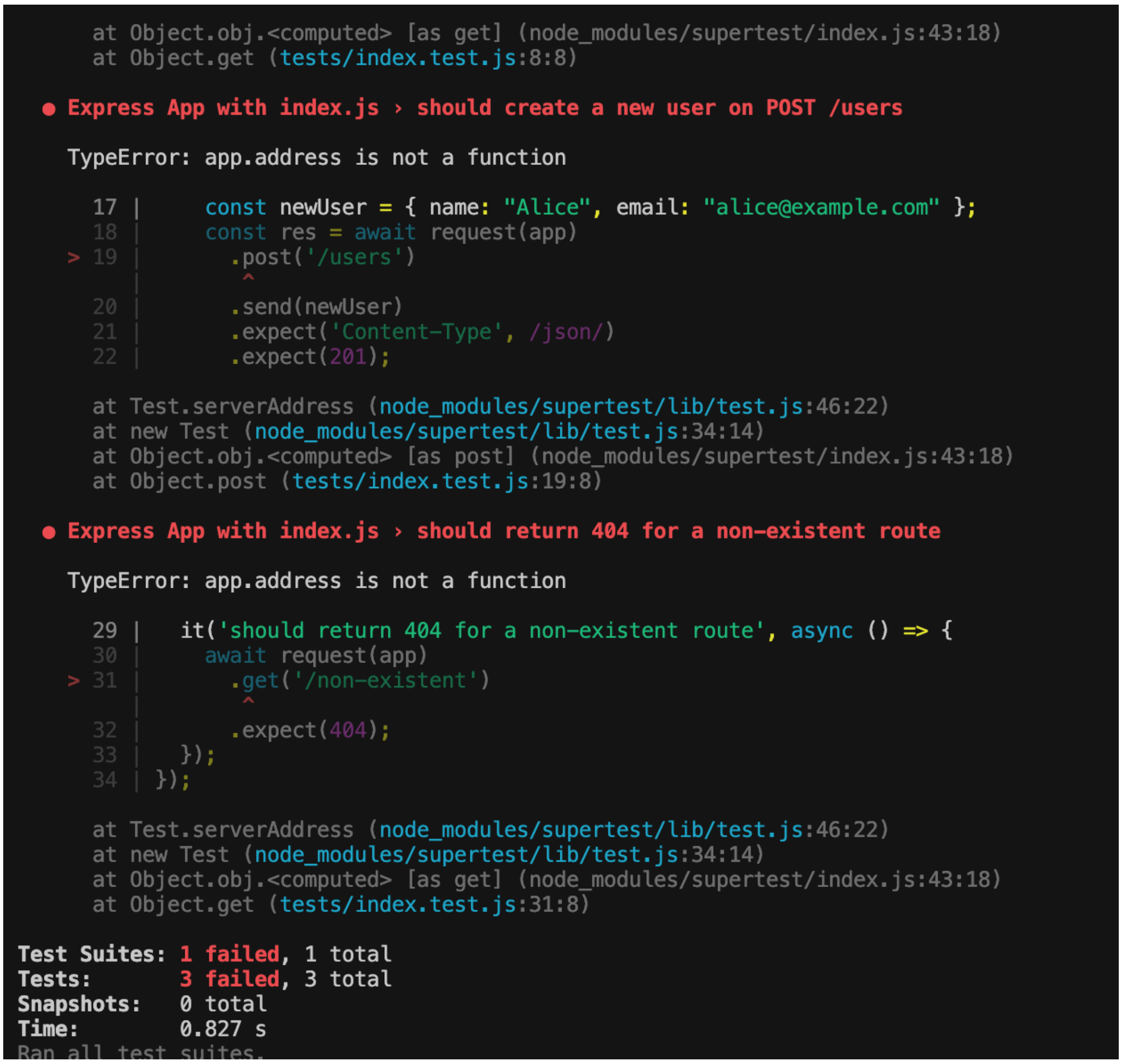
Test-Driven Development is a software development methodology that involves the creation of tests prior to the actual coding process, which serves as a guide for the coding process. It is a cycle that involves the creation of a test that fails, the implementation of code to pass the test, and the subsequent refactoring of the code for optimisation. This method simplifies debugging, guarantees code reliability, and enables future refactoring.

BDD

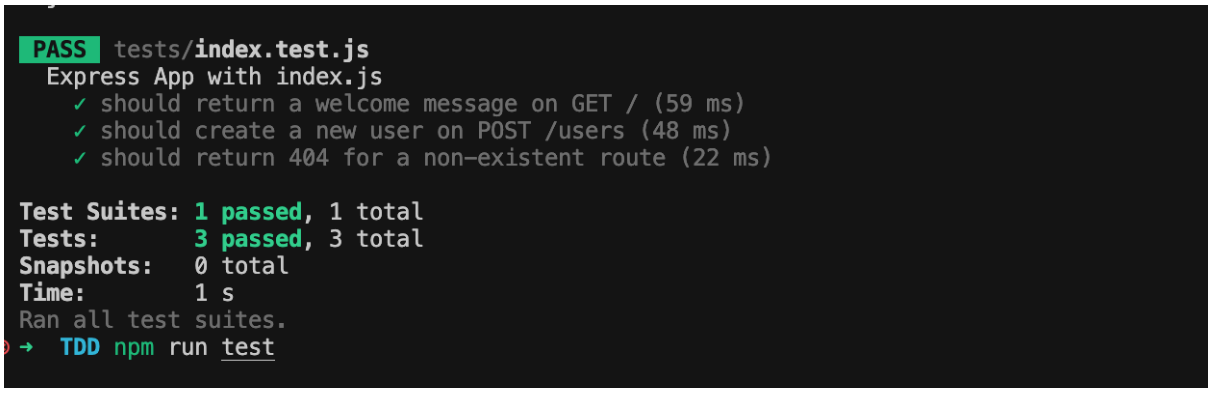
Behavior-Driven Development is an agile software development methodology that prioritises collaboration among developers, testers, and business stakeholders to define the behaviour of an application in a manner that is easily comprehensible. It augments Test-Driven Development by employing natural language constructs to communicate the anticipated outcomes and behaviour of software, thereby guaranteeing that all stakeholders possess a mutual comprehension. BDD utilises a domain-specific language with a "Given-When-Then" syntax to generate scenarios that function as both executable tests and documentation. This methodology promotes enhanced communication, unambiguous acceptability criteria, and an emphasis on the provision of business value.

TDD implementation

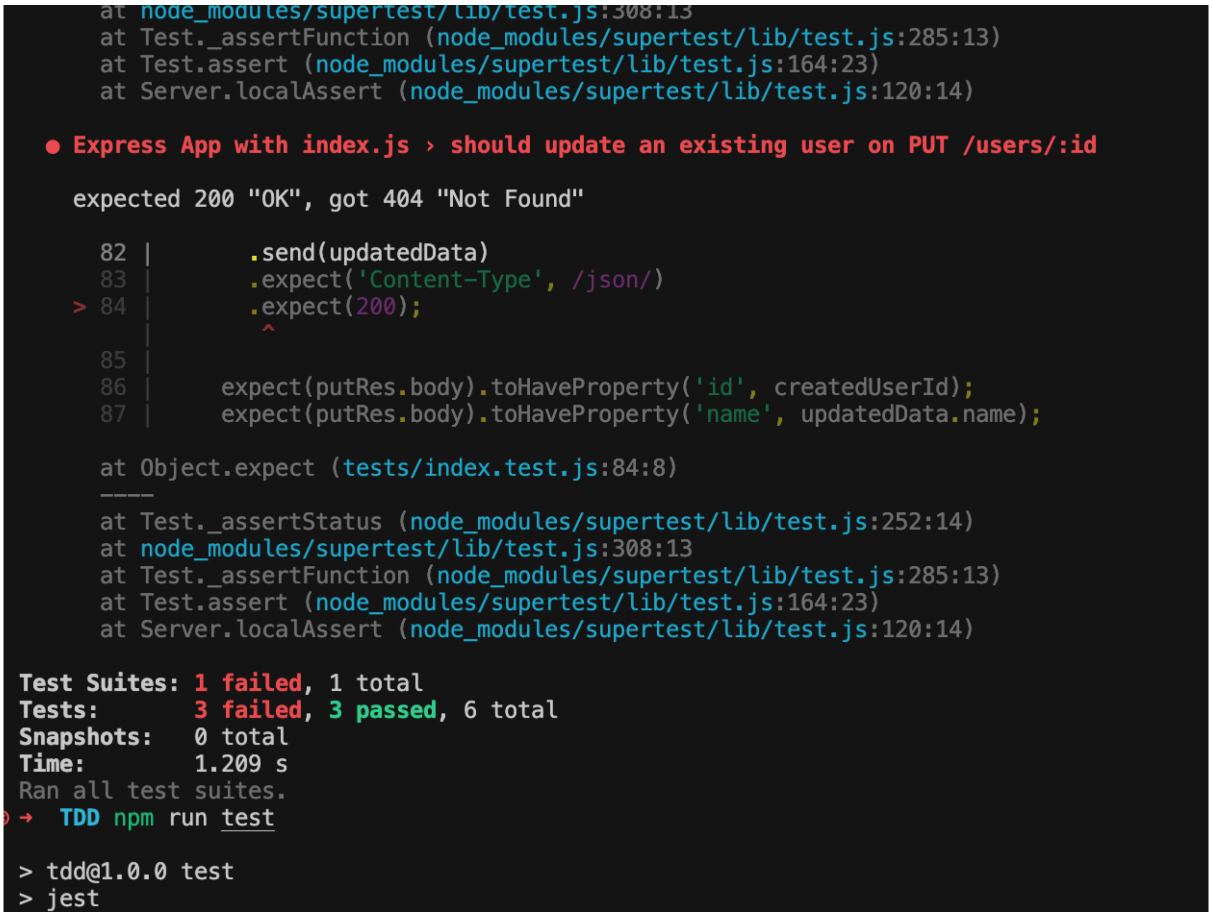
I initiated my development process by adopting Test-Driven Development. Before writing any implementation code, I utilised Jest and Supertest to develop tests that were designed to specify the anticipated behaviour of each endpoint. I developed tests for the following: a GET request at the root endpoint to return a welcome message, a POST request to create a new user, a GET request to enumerate all users, a GET request to retrieve a specific user by ID, and a PUT request to update user information. Initially, these tests failed, confirming that I had not yet implemented the required functionality and providing a crystal-clear roadmap for the necessary development.



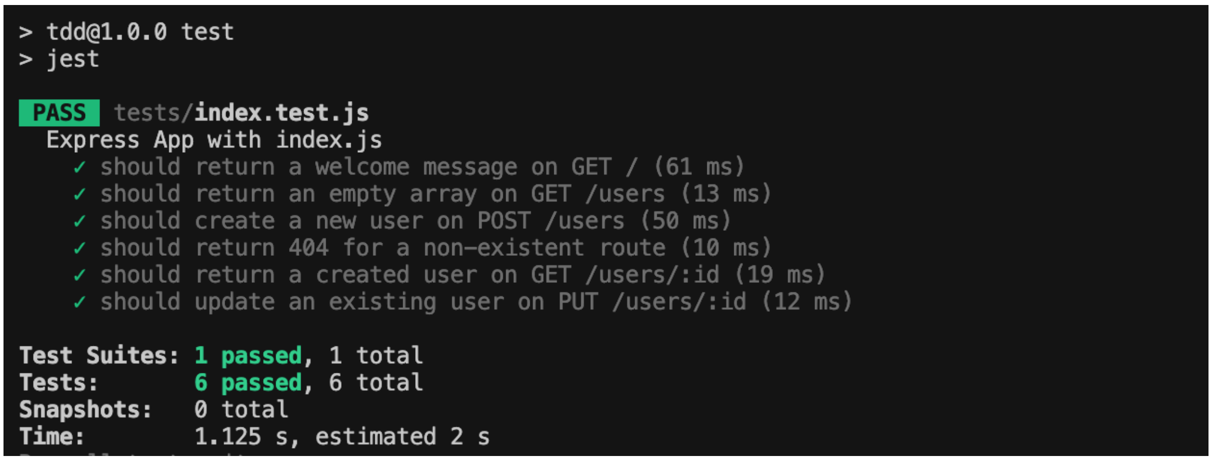
```
      at Object.obj.<computed> [as get] (node_modules/supertest/index.js:43:18)
      at Object.get (tests/index.test.js:8:8)

  ● Express App with index.js › should create a new user on POST /users

    TypeError: app.address is not a function

      17 |     const newUser = { name: "Alice", email: "alice@example.com" };
      18 |     const res = await request(app)
    > 19 |       .post('/users')
         |        ^
      20 |       .send(newUser)
      21 |       .expect('Content-Type', /json/)
      22 |       .expect(201);

      at Test.serverAddress (node_modules/supertest/lib/test.js:46:22)
      at new Test (node_modules/supertest/lib/test.js:34:14)
      at Object.obj.<computed> [as post] (node_modules/supertest/index.js:43:18)
      at Object.post (tests/index.test.js:19:8)

  ● Express App with index.js › should return 404 for a non-existent route

    TypeError: app.address is not a function

      29 |   it('should return 404 for a non-existent route', async () => {
      30 |     await request(app)
    > 31 |       .get('/non-existent')
         |        ^
      32 |       .expect(404);
      33 |   });
      34 | });

      at Test.serverAddress (node_modules/supertest/lib/test.js:46:22)
      at new Test (node_modules/supertest/lib/test.js:34:14)
      at Object.obj.<computed> [as get] (node_modules/supertest/index.js:43:18)
      at Object.get (tests/index.test.js:31:8)

Test Suites: 1 failed, 1 total
Tests:       3 failed, 3 total
Snapshots:   0 total
Time:        0.827 s
Ran all test suites.
```



```
 PASS  tests/index.test.js
  Express App with index.js
    ✓ should return a welcome message on GET / (59 ms)
    ✓ should create a new user on POST /users (48 ms)
    ✓ should return 404 for a non-existent route (22 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1 s
Ran all test suites.
➜  TDD npm run test
```



```
    at node_modules/supertest/lib/test.js:308:13
    at Test._assertFunction (node_modules/supertest/lib/test.js:285:13)
    at Test.assert (node_modules/supertest/lib/test.js:164:23)
    at Server.localAssert (node_modules/supertest/lib/test.js:120:14)

  ● Express App with index.js › should update an existing user on PUT /users/:id

    expected 200 "OK", got 404 "Not Found"

      82 |         .send(updatedData)
      83 |         .expect('Content-Type', /json/)
    > 84 |         .expect(200);
         |          ^
      85 |
      86 |       expect(putRes.body).toHaveProperty('id', createdUserId);
      87 |       expect(putRes.body).toHaveProperty('name', updatedData.name);

      at Object.expect (tests/index.test.js:84:8)
      ----
      at Test._assertStatus (node_modules/supertest/lib/test.js:252:14)
      at node_modules/supertest/lib/test.js:308:13
      at Test._assertFunction (node_modules/supertest/lib/test.js:285:13)
      at Test.assert (node_modules/supertest/lib/test.js:164:23)
      at Server.localAssert (node_modules/supertest/lib/test.js:120:14)

Test Suites: 1 failed, 1 total
Tests:       3 failed, 3 passed, 6 total
Snapshots:   0 total
Time:        1.209 s
Ran all test suites.
➜  TDD npm run test

> tdd@1.0.0 test
> jest
```



```
> tdd@1.0.0 test
> jest

 PASS  tests/index.test.js
  Express App with index.js
    ✓ should return a welcome message on GET / (61 ms)
    ✓ should return an empty array on GET /users (13 ms)
    ✓ should create a new user on POST /users (50 ms)
    ✓ should return 404 for a non-existent route (10 ms)
    ✓ should return a created user on GET /users/:id (19 ms)
    ✓ should update an existing user on PUT /users/:id (12 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        1.125 s, estimated 2 s
```

After completing my tests, I proceeded to integrate the application endpoints into my index.js file. I established an Express server and configured it to parse JSON payloads. To return a welcome message in JSON format, I initiated the process by implementing the most basic endpoint, the GET / route. Moving on to the POST /users endpoint, I was satisfied that it met the test's expectations. I have composed the code to receive user data, generate a unique identifier, and store the user in an in-memory array. I then implemented the GET /users endpoint to retrieve the entire list of users, followed by the GET /users/:id endpoint, which retrieves a specific user based on the provided ID and returns an error if the user is not discovered. Lastly, I implemented the PUT /users/:id endpoint to facilitate the integration of new data with the existing record in order to update a user's information.
I conducted my tests at each stage of this process to ensure that my implementation aligned with the criteria outlined in my test suite. The application was constructed in a methodical

and reliable manner as a result of the iterative cycle of writing a failure test, coding the requisite functionality to pass the test, and refactoring the code as necessary.

TDD code

index.test.js

```
const request = require('supertest');
const app = require('../index'); // Import your Express app

describe('Express App with index.js', () => {
 // Test 1: Check that GET / returns a welcome message.
 it('should return a welcome message on GET /', async () => {
   const res = await request(app)
     .get('/')
     .expect('Content-Type', /json/)
     .expect(200);

   expect(res.body).toHaveProperty('message', 'Welcome to my Express app!');
 });

 // Test 2: GET /users should return an empty array initially.
 it('should return an empty array on GET /users', async () => {

   const res = await request(app)
     .get('/users')
     .expect('Content-Type', /json/)
     .expect(200);

   expect(Array.isArray(res.body)).toBe(true);
   expect(res.body.length).toBe(0);
 });

 // Test 3: Verify that POST /users creates a new user.
 it('should create a new user on POST /users', async () => {
   const newUser = { name: "Alice", email: "alice@example.com" };
   const res = await request(app)
     .post('/users')
     .send(newUser)
     .expect('Content-Type', /json/)
     .expect(201);

   expect(res.body).toHaveProperty('id');
   expect(res.body.name).toBe(newUser.name);
 });
```

```javascript
// Test 4: Ensure that an undefined route returns a 404 error.
it('should return 404 for a non-existent route', async () => {
  await request(app)
    .get('/non-existent')
    .expect(404);
});

// Test 5: GET /users/:id returns a created user.
it('should return a created user on GET /users/:id', async () => {
  const newUser = { name: "Bob", email: "bob@example.com" };
  const postRes = await request(app)
    .post('/users')
    .send(newUser)
    .expect('Content-Type', /json/)
    .expect(201);

  const createdUserId = postRes.body.id;
  const getRes = await request(app)
    .get(`/users/${createdUserId}`)
    .expect('Content-Type', /json/)
    .expect(200);

  expect(getRes.body).toHaveProperty('id', createdUserId);
  expect(getRes.body).toHaveProperty('name', newUser.name);
});

// Test 6: PUT /users/:id updates the user and returns the updated information.
it('should update an existing user on PUT /users/:id', async () => {
  const newUser = { name: "Charlie", email: "charlie@example.com" };
  const postRes = await request(app)
    .post('/users')
    .send(newUser)
    .expect(201);

  const createdUserId = postRes.body.id;

  // Now, update the user.
  const updatedData = { name: "Charles", email: "charles@example.com" };
  const putRes = await request(app)
    .put(`/users/${createdUserId}`)
    .send(updatedData)
    .expect('Content-Type', /json/)
    .expect(200);

  expect(putRes.body).toHaveProperty('id', createdUserId);
```

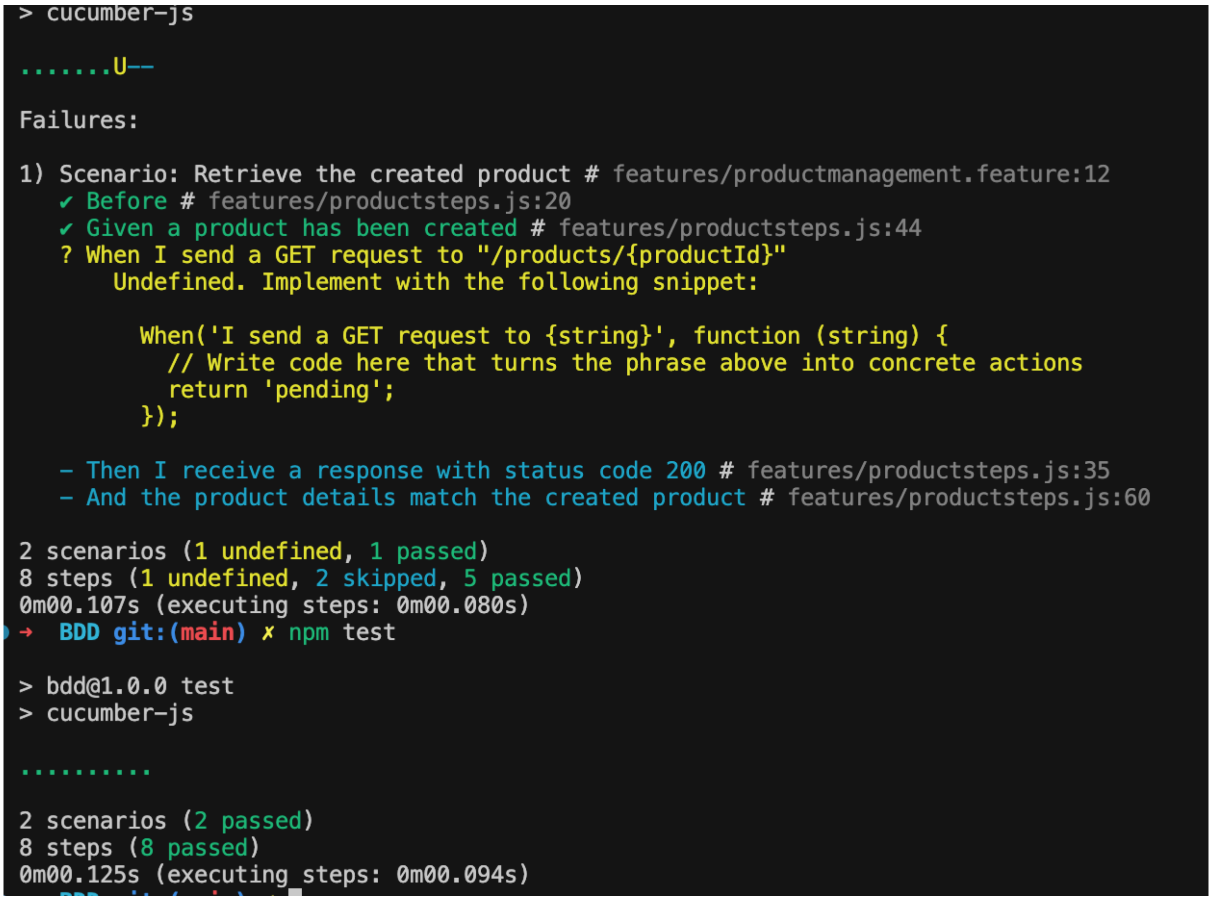
```
    expect(putRes.body).toHaveProperty('name', updatedData.name);
  });
});
```

index.js

```
const express = require('express');
const app = express();

app.use(express.json());

let users = []; // In-memory users array

app.get('/', (req, res) => {
  res.status(200).json({ message: 'Welcome to my Express app!' });
});

app.post('/users', (req, res) => {
  const user = { id: Date.now(), ...req.body };
  users.push(user);
  res.status(201).json(user);
});

// New endpoint: GET /users returns all users
app.get('/users', (req, res) => {
  res.status(200).json(users);
});

// New endpoint: GET /users/:id returns a specific user
app.get('/users/:id', (req, res) => {
  const user = users.find(u => u.id == req.params.id);
  if (user) {
    res.status(200).json(user);
  } else {
    res.status(404).json({ error: 'User not found' });
  }
});

// New endpoint: PUT /users/:id updates an existing user
app.put('/users/:id', (req, res) => {
  const index = users.findIndex(u => u.id == req.params.id);
  if (index !== -1) {
    users[index] = { ...users[index], ...req.body };
    res.status(200).json(users[index]);
  } else {
   res.status(404).json({ error: 'User not found' });
 }
});

// Catch-all for undefined routes
app.use((req, res) => {
 res.status(404).json({ error: 'Not Found' });
});

module.exports = app;
```

BDD implementation

I began by writing a feature file in plain language to describe the desired behaviour of the product management functionality. The feature file outlined scenarios such as creating a new product and retrieving a product by its ID, using a natural language format to specify the expected outcomes. Next, I implemented the step definitions in a JavaScript file using Cucumber.js and Supertest to map those scenarios to concrete HTTP requests against my Express application. One challenge I encountered was with the Cucumber expression syntax; specifically, using curly braces in the step expression for product IDs led to parser errors. The parser complained about empty alternatives when it encountered the placeholder, so I had to adjust my approach by dynamically replacing the placeholder with the actual product ID in the step definition. Additionally, I faced issues with undefined steps because not all the steps described in the feature file had corresponding implementations at first. This required me to iteratively update and refine both the feature file and the step definitions to ensure complete coverage. Overall, the process highlighted the need for clear communication between the behaviour specifications and the code, and it reinforced the benefits of using BDD to align technical implementation with business requirements, even though it required careful attention to detail in mapping natural language expressions to code.



```
> cucumber-js

.......U--

Failures:

1) Scenario: Retrieve the created product # features/productmanagement.feature:12
   ✔ Before # features/productsteps.js:20
   ✔ Given a product has been created # features/productsteps.js:44
   ? When I send a GET request to "/products/{productId}"
       Undefined. Implement with the following snippet:

         When('I send a GET request to {string}', function (string) {
           // Write code here that turns the phrase above into concrete actions
           return 'pending';
         });

   - Then I receive a response with status code 200 # features/productsteps.js:35
   - And the product details match the created product # features/productsteps.js:60

2 scenarios (1 undefined, 1 passed)
8 steps (1 undefined, 2 skipped, 5 passed)
0m00.107s (executing steps: 0m00.080s)
➜  BDD git:(main) ✗ npm test

> bdd@1.0.0 test
> cucumber-js

..........

2 scenarios (2 passed)
8 steps (8 passed)
0m00.125s (executing steps: 0m00.094s)
```

BDD code

productmanagement.feature

```
Feature: Product Management
  As a system user
  I want to create and retrieve products
  So that I can manage product information

  Scenario: Create a new product
    Given I have a valid product payload
    When I send a POST request to "/products"
    Then I receive a response with status code 201
    And the response contains a product ID

  Scenario: Retrieve the created product
    Given a product has been created
    When I send a GET request to "/products/{productId}"
    Then I receive a response with status code 200
    And the product details match the created product
```

productsteps.js

```
const { Given, When, Then, Before } = require('@cucumber/cucumber');
const request = require('supertest');
const assert = require('assert');
const app = require('../index');

let response;
let productPayload;
let createdProductId;

Before(() => {
 // Initialize a valid product payload before each scenario
 productPayload = { name: 'Gaming PC', price: 1500 };
});

Given('I have a valid product payload', function () {
 // productPayload is already set up in the Before hook
});

When('I send a POST request to {string}', async function (endpoint) {
 response = await request(app)
   .post(endpoint)
   .send(productPayload);
});

Then('I receive a response with status code {int}', function (statusCode) {
 assert.strictEqual(response.status, statusCode);
});

Then('the response contains a product ID', function () {
 assert.ok(response.body.id, 'Expected response to contain a product ID');
 createdProductId = response.body.id;
});

Given('a product has been created', async function () {
 // Create a product and save its ID for later retrieval
 const postResponse = await request(app)
   .post('/products')
   .send(productPayload);
 assert.strictEqual(postResponse.status, 201);
 assert.ok(postResponse.body.id, 'Expected product to have an ID');
 createdProductId = postResponse.body.id;
});

When('I send a GET request to {string}', async function (endpoint) {
 // Replace the placeholder {productId} in the endpoint with the actual product ID
 if (endpoint.includes('{productId}')) {
   endpoint = endpoint.replace('{productId}', createdProductId);
 }
 response = await request(app).get(endpoint);
});

Then('the product details match the created product', function () {
 assert.strictEqual(response.status, 200);
 assert.strictEqual(response.body.id, createdProductId);
 assert.strictEqual(response.body.name, productPayload.name);
 assert.strictEqual(response.body.price, productPayload.price);
});
```

index.js

```
const express = require('express');
const app = express();

app.use(express.json());

let products = [];

// Endpoint to create a new product
app.post('/products', (req, res) => {
 const product = { id: Date.now(), ...req.body };
 products.push(product);
 res.status(201).json(product);
});

// Endpoint to retrieve a product by ID
app.get('/products/:id', (req, res) => {
 const product = products.find(p => p.id == req.params.id);
 if (product) {
   res.status(200).json(product);
 } else {
   res.status(404).json({ error: 'Product not found' });
 }
});

// Catch-all for undefined routes
app.use((req, res) => {
 res.status(404).json({ error: 'Not Found' });
});

module.exports = app;
```

Conclusion

From what I've seen, TDD and BDD both have unique benefits and also present their own set of challenges. Using TDD really makes me think about how my code should behave before I even start writing it, which is super helpful for spotting problems early on. Writing a failing test, then adding just enough code to make it pass, and finally refactoring has really helped me debug more easily and boosted my confidence in how reliable the code is. Sometimes, I feel like TDD really emphasises the technical side of things, which can result in tests that check if something works but might not fully reflect the overall business goals for a feature.

On the other hand, BDD has really helped me connect my work with what non-technical stakeholders expect. By using natural language scenarios and the Given-When-Then format, I can explain requirements more clearly, making sure that everyone from developers to business users understands what the system is supposed to do. This method has really boosted teamwork and allowed me to create features that provide genuine business value. One downside is that BDD can add extra overhead. Maintaining both feature files and step definitions can sometimes feel cumbersome, and it can be challenging to get the natural language expressions just right.

I think TDD is really great for making sure the code is high quality and works well internally, while BDD is awesome for helping with communication and checking that the application actually meets what users need in the real world. For different projects, I usually mix parts of both methods to take advantage of what each one does best.