GUIDE A03

Create a POST Endpoint to Add New Data

Arranged By
Omar Al-Maktary

INFORMATICS ENGINEERING STUDY PROGRAM
INFORMATION TECHNOLOGY DEPARTMENT
STATE POLYTECHNIC OF MALANG
2023

Contents

Create a POST Endpoint to Add New Data

Objectives

1. Students understand how to create a POST endpoint to add a product resource to the database.
2. Students understand how to create a web page to add new product resources to the database.

The guide A03 aims for learning the method POST to add new data to MongoDB using NodeJS for a RESTful API application. The practice of this guide will be used in the same application established in guides A01 and A02.

Students should learn the routes needed to create a web page that can create a new resource for the database. The web page has inputs that are required to create a new product resource.

Requirements

Having the correct hardware and software components is essential for ensuring the successful execution of the tasks outlined in this guide. The hardware configuration and software required for completing this guide tasks are as the following:

Hardware Specifications

The minimum hardware specifications for running a Node.js API application on the Windows operating system and using software such as Postman and Visual Studio Code are the following:

Minimum Requirements

- Processor: Intel Core i3 or equivalent.
- RAM: 4 GB.
- Storage: 500 GB HDD with at least 20 GB of available storage.
- Graphics: Integrated graphics card.
- Connectivity: Ethernet and Wi-Fi capabilities.

Recommended Requirements

- Processor: Intel Core i5 or equivalent.
- RAM: 8 GB or more.
- Storage: 256 GB SSD with at least 20 GB of available storage.
- Graphics: Integrated graphics card.
- Connectivity: Ethernet and Wi-Fi capabilities.

Software required

It is important to have the correct software installed on your system to ensure that the application runs smoothly and meets performance expectations. The software required is as follows:

- Operating System: Windows 10 or later.
- NodeJS: Latest stable version installed.
- Visual Studio Code: Latest stable version installed.
- Postman: Latest stable version is installed.

Note: NodeJS, Visual Studio Code, and Postman installation have been explained in the previous guide, A01.

NPM Packages

- nodemon: Automatically restarts Node application on file changes.
- cross-env: Sets environment variables in a cross-platform way.
- jest: Creates and executes tests.
- jest-expect-message: Enhances Jest assertions with custom error messages.
- jest-image-snapshot: Adds image snapshot testing to Jest.
- puppeteer: Node library to control a headless Chrome or Chromium browser.
- supertest: Makes HTTP queries to the application and checks results.
- dotenv: Simplifies management of environment variables.
- express: NodeJS framework for creating apps with routing and middleware.
- ejs: Embedded JavaScript templating.
- express-ejs-layouts: Layout support for EJS in Express.
- mongoose: MongoDB object modeling library for NodeJS.
- mongoose-slug-generator: Automatically generates slugs based on a Mongoose schema field.

Resource

- Documents: Guide A03
- Tests: api/testA03.test.js, web/testA03.test.js

Task Description

Students can create a POST endpoint to add product data to the database. Students will create a new route in the application which will receive requests from clients to add new data to the database. The function which will handle the requests of the endpoint should be able to receive 3 parameters of a new product namely the name, price, and description. If a product with the same data as the request body data exists in the database, the server should return an error that the product exists. If the data is new the server will save it in the database. Students will also create a web page to create product resources. The web page should have three inputs namely the name, price, and description.

Start Coding

1. Open the file "controllers/api/product.controller.js". and copy the following code to it:

```
const createProduct = async (req, res) => {
  try {
  } catch (error) {
    return res.status(500).json(error);
  }
};
```

Figure 1 createProduct Function Initial Code

Don't forget to export the function **createProduct** as the following:

```
module.exports = {
  getProducts,
  getProduct,
  createProduct,
};
```

Figure 2 "controllers/api/product.controller.js" Exports Functions Code

2. Go to the file "routes/api/product.routes.js", and define a new route as the following:

```
router.post("/product", createProduct);
```

Figure 3 "routes/api/product.routes.js" New Route

The function **createProduct** needs to be required from the controller to handle the requests to the POST endpoint “/api/v1/product” similar to the following code:

```
const {
  getProducts,
  getProduct,
  createProduct,
} = require("../../controllers/api/product.controller");
```

Figure 4 product.controller.js Required Functions

3. Go back to the file “product.controller.js” and update the function **createProduct** to fit the requirements of the following table. This table contains information needed for the endpoint details.

POST “/api/v1/product” ENDPOINT STRUCTURE			
API Endpoint Path	***Request Method***	***Response Format***	***Description***
“/api/v1/product”	POST	JSON	Creates a new product using the information provided in the request body.
Request Parameters			
Parameter	***Type***		***Description***
“name”	String		The name of the product. Required.
“price”	Number		The price of the product. Required. The minimum value is 0.
“description”	String		A description of the product. Required.
Response Parameters			
Parameter	***Type***		***Description***
“product”	Object		An object containing information about the created product.
“message”	String		A message indicating the status of the response
Success Responses			
HTTP Status Code	***Response***		
201	{ "product": product, "message": "Product created" }		
Error Responses			
HTTP Status Code	***Response***		
409	{ "product": FoundProduct, "message": "Product already exists" }		

500	{ error object }

Table 1 Endpoint POST "/api/v1/product" API Architecture Design

Note that the **product** and **FoundProduct** are instances of the product object after being created or in the case FoundProduct it indicates that the resource already exists in the database. The server will send the existing resource with a status code of 409 indicating conflict.

4. Hints: use the following code to help in completing the task:

```
const product = await Product.create({
    name: req.body.name,
    price: parseInt(req.body.price),
    description: req.body.description,
  });
```

Figure 5 Create New Product Code Hint

This code creates a new product object by calling the **create** method on a **Product** model (presumably defined elsewhere in the codebase).

The method takes an object as an argument, which specifies the new product's properties. In this case, the name, price, and description are set based on data from the HTTP request (**req**).

The **await** keyword is used to wait for the asynchronous **create** method to finish executing before moving on to the next line of code.

Once the new **Product** object is created, it is assigned to the **product** constant for further use in the code.

Running The API Application

To run the application, use the following command:

For this guide and development purposes the command “npm run dev” is used to execute the command “nodemon server.js” which will run the “server.js” using the nodemon package. This package allows the server to reload if any changes occur in the code of the application. Run the development command “npm run dev” in the terminal and notice the console message.

Testing The API Application

In this section, several tests in different ways will be explored to verify the results of the student's work on this document.

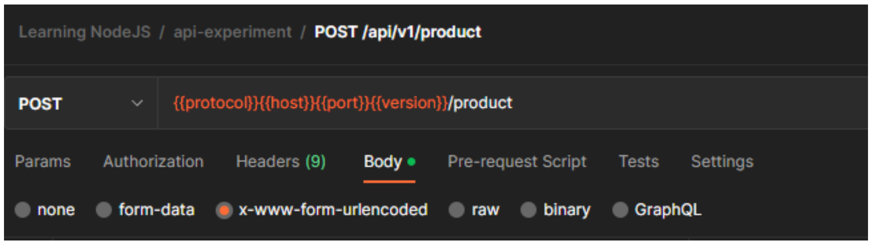
Using Postman

To verify results using Postman, follow these steps:

1. Create a new request in the folder “api-experiment” with the option POST as its method.
2. Use the following link as the URL.

```
{{protocol}}{{host}}{{port}}{{version}}/product
```

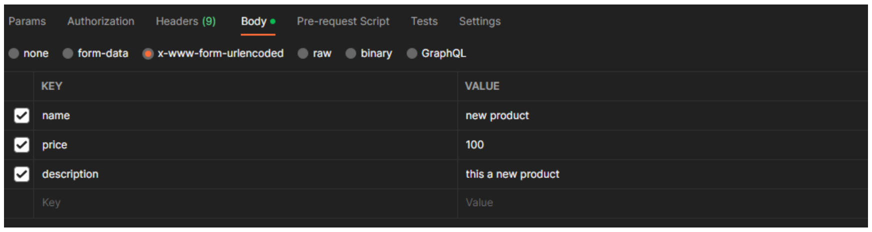
3. In the **Body** tab, choose the **x-www-form-urlencoded** option to send data attributes.



{{protocol}}{{host}}{{port}}{{version}}/product

Figure 6 Postman Request Body Options

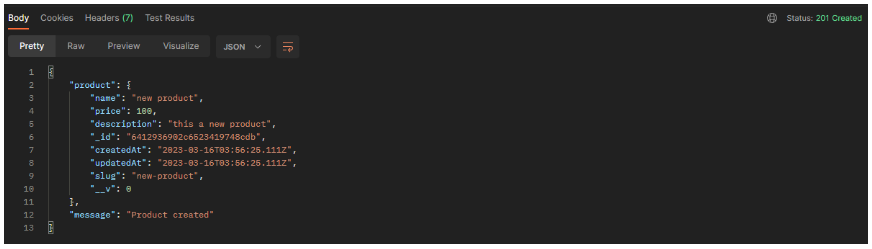
4. Fill in the name, price, and description of the new product.



	KEY	VALUE
✓	name	new product
✓	price	100
✓	description	this a new product
	Key	Value

Figure 7 Postman Request x-www-form-urlencoded Data

5. Send the request, it should return the new object as a MongoDB document and a message indicating the product has been created.



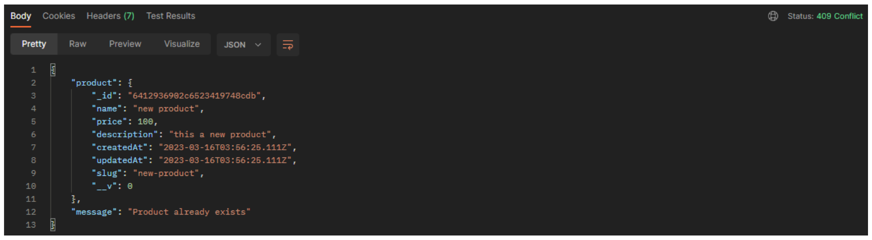
Status: 201 Created

```
{
    "product": {
        "name": "new product",
        "price": 100,
        "description": "this a new product",
        "_id": "6412936902c6523419748cdb",
        "createdAt": "2023-03-16T03:56:25.111Z",
        "updatedAt": "2023-03-16T03:56:25.111Z",
        "slug": "new-product",
        "__v": 0
    },
    "message": "Product created"
}
```

Figure 8 Postman Response Results Created

Note that the status is 201 meaning the resource has been created. Additionally, notice the message indicating the product data was created. Check the database, the new product should be found.

6. Try to send the same request, it should return an error that the product already exists.

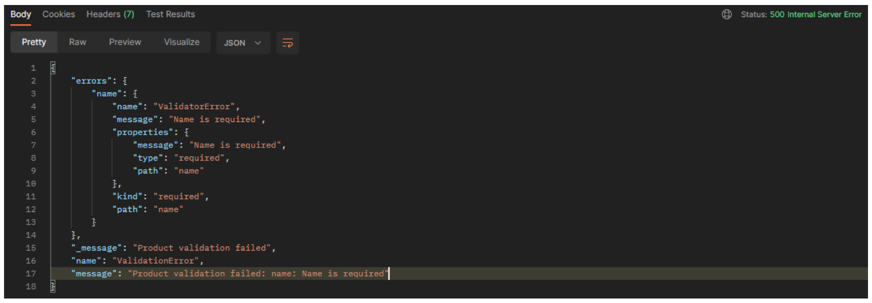


```
{
    "product": {
        "_id": "6412936902c6523419748cdb",
        "name": "new product",
        "price": 100,
        "description": "this a new product",
        "createdAt": "2023-03-16T03:56:25.111Z",
        "updatedAt": "2023-03-16T03:56:25.111Z",
        "slug": "new-product",
        "__v": 0
    },
    "message": "Product already exists"
}
```

Figure 9 Postman Response Results Not Created

Note that the status is 409 indicating server conflict. The response message shows that the product already exists.

7. Try to remove the name attribute, it should return an error indicating that the product requires a name. this is due to the configuration of the model file.



```
{
    "errors": {
        "name": {
            "name": "ValidatorError",
            "message": "Name is required",
            "properties": {
                "message": "Name is required",
                "type": "required",
                "path": "name"
            },
            "kind": "required",
            "path": "name"
        }
    },
    "_message": "Product validation failed",
    "name": "ValidationError",
    "message": "Product validation failed: name: Name is required"
}
```

Figure 10 Validation Error in Postman

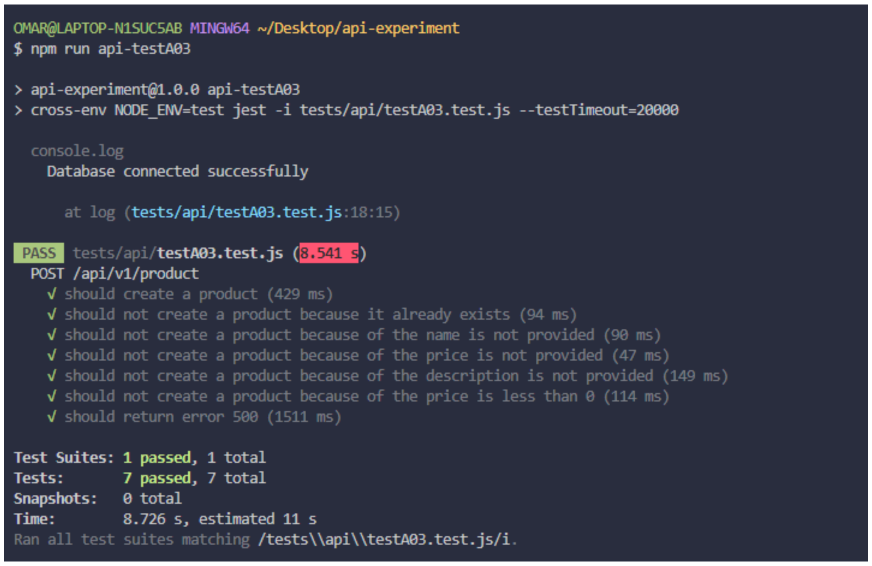
The request status code is 500 indicating an internal server error that is thrown by the mongoose schema. The message contains the reason for the error object which is a validation failure because the name data is required.

Running The API Test file

Note: Sometimes the test will have an error of time limit, try to re-run the test or increase the testTimeout in the next step.

1. Copy the file "testA03.test.js" from the "api" folder within the "tests" folder for this material to the "tests/api" folder of your project base directory.
2. In the terminal run the command “npm run api-testA03” and notice the results.
3. If encounter any errors regarding mongoose doesn’t allow callbacks, try downgrading the mongoose library by editing the “package.json” file. Change the version of the mongoose library to “^6.10.0” and then try to rerun the test file.

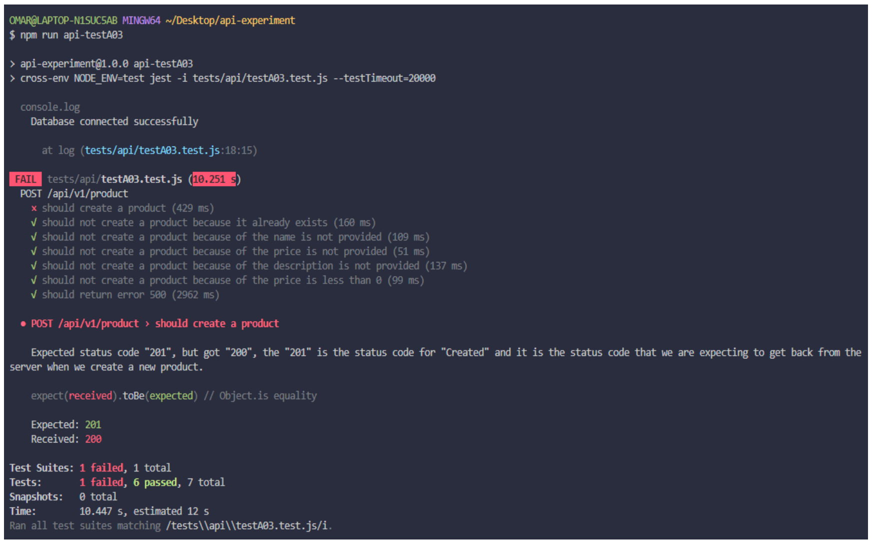
4. If everything is correct and working well the results in the terminal should look as the following:



```
OMAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/api-experiment
$ npm run api-testA03

> api-experiment@1.0.0 api-testA03
> cross-env NODE_ENV=test jest -i tests/api/testA03.test.js --testTimeout=20000

  console.log
    Database connected successfully

      at log (tests/api/testA03.test.js:18:15)

 PASS  tests/api/testA03.test.js (8.541 s)
  POST /api/v1/product
    √ should create a product (429 ms)
    √ should not create a product because it already exists (94 ms)
    √ should not create a product because of the name is not provided (90 ms)
    √ should not create a product because of the price is not provided (47 ms)
    √ should not create a product because of the description is not provided (149 ms)
    √ should not create a product because of the price is less than 0 (114 ms)
    √ should return error 500 (1511 ms)

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        8.726 s, estimated 11 s
Ran all test suites matching /tests\\api\\testA03.test.js/i.
```

Figure 11 Successful Test Results

5. If the test failed and it shows an error similar to the following figure:



```
OMAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/api-experiment
$ npm run api-testA03

> api-experiment@1.0.0 api-testA03
> cross-env NODE_ENV=test jest -i tests/api/testA03.test.js --testTimeout=20000

  console.log
    Database connected successfully

      at log (tests/api/testA03.test.js:18:15)

 FAIL  tests/api/testA03.test.js (10.251 s)
  POST /api/v1/product
    × should create a product (429 ms)
    √ should not create a product because it already exists (160 ms)
    √ should not create a product because of the name is not provided (109 ms)
    √ should not create a product because of the price is not provided (51 ms)
    √ should not create a product because of the description is not provided (137 ms)
    √ should not create a product because of the price is less than 0 (99 ms)
    √ should return error 500 (2962 ms)

  ● POST /api/v1/product › should create a product

    Expected status code "201", but got "200", the "201" is the status code for "Created" and it is the status code that we are expecting to get back from the
server when we create a new product.

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

    Expected: 201
    Received: 200

Test Suites: 1 failed, 1 total
Tests:       1 failed, 6 passed, 7 total
Snapshots:   0 total
Time:        10.447 s, estimated 12 s
Ran all test suites matching /tests\\api\\testA03.test.js/i.
```

Figure 12 Failed Test Result

Notice the feedback indicating what might be the issue. Try to find out why the test failed and fix it until the test result shows successful results.

Creating The Web Interface

In this section, the web interface for this application will have a new page added to it. The same basic endpoints explained in the previous sections will be implemented in a web page that can create a new product.

To start working on the web interface, follow these steps:

1. In the “controllers/web/product.controller.js” file, create a new function named “**createProduct**”. Read the explanation below then fill in the rest of the code.

```
const createProduct = async (req, res) => {
 if (req.method === "GET") {
  // Render the create product page with the
  // title “API - Experiment | Create Product”
  // write your code here ...
 } else {
  if (
   req.body.name === "" ||
   req.body.price === "" ||
   req.body.description === ""
  ) {
   // Render the create product page with the
   // error message “Please fill all fields”
   // write your code here ...
  }
  // Check if the product already exists
  // write your code here ...
  // Render the create product page with the
  // error message “Product already exists”
  if (FindProduct !== null) {
   // write your code here ...
  }
  // Create a product using the req.body attributes
  // write your code here ...
  // Return to the getProducts function with the
  // success message “Product created”
  req.query.message = "Product created";
  // write your code here ...
 }
};
```

Figure 13 "createProduct" Code Structure for The Web Interface

Make sure to export the function at the end of the file. This function checks the HTTP method used in the request. If the method is GET, it will render the create page in the "web/views/products" folder with the title *"API-Experiment | Create Product"*. If the method is not GET, the function will validate the data submitted in the request body. If the data is incomplete, it will render the create page again with a message "Please fill all fields", indicating that all fields must be filled before submitting the form.

If all the fields are filled, the function will check if a similar product already exists in the database. If a product with the same data is found, the function should render the create page again with the message "Product already exists", informing the user that the product already exists. If the product is not found in the database, the function will create new product data and return to the products list page with the message "Product created".

2. In the "routes/web/product.routes.js" file add two routes with the "/create" path. Both routes use the same function created in the previous step.
3. The first route uses the GET method and the second route uses the POST method.
4. Create a new file in the "web/views/products" folder named "create.ejs". Write the EJS code for this file, and use the following table.

Element	**HTML Tag**	**Attribute(s) / Inner Text**
Alert message	<div class="alert">	If the message is defined, it is displayed as a paragraph with a class message inside a div with a class alert. The display is controlled using EJS syntax: <% if (typeof message !== 'undefined') { %><p class="message"><%= message %></p><% } %>
Title and Description	<div class="container">	A div with a class container containing an h1 element with the class title and inner text "Create a new product", and a p element with class description and inner text "Fill the form below to create a new product".
Form container	<div class="container">	A div with a class container is used to contain the form elements.
Form	<form>	A form element with action "/products/create" and method "POST".
Name field	<input>	An input element with type text, name name, id name, and class form control. This field is for the user to input the name of the product they want to create.

Price field	<input>	An input element with type number, name price, id price, and class form control. This field is for the user to input the price of the product they want to create.
Description field	<textarea>	A textarea element with name description, id description, class form-control, and rows="5". This field is for the user to input the description of the product they want to create.
Create button	<button>	A button element with type submits, class btn, and class btn-primary. The button text reads "Create".

Figure 14 "web/views/products/create.ejs" EJS Code Structure

Running and Testing The Web Interface

If the application still running from the previous exercise then try to visit the following link http://localhost:8080/. If the application is not running in the terminal then use the command "npm run dev" to start the app.

Upon visiting the web interface and clicking on the "Products" button or the "Products" navigation bar link then clicking the button "Create a new product", the page should look similar to the following image:

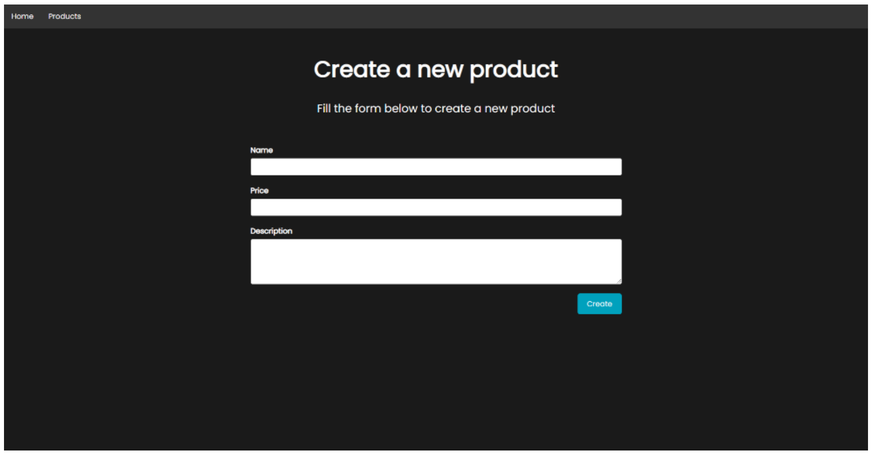
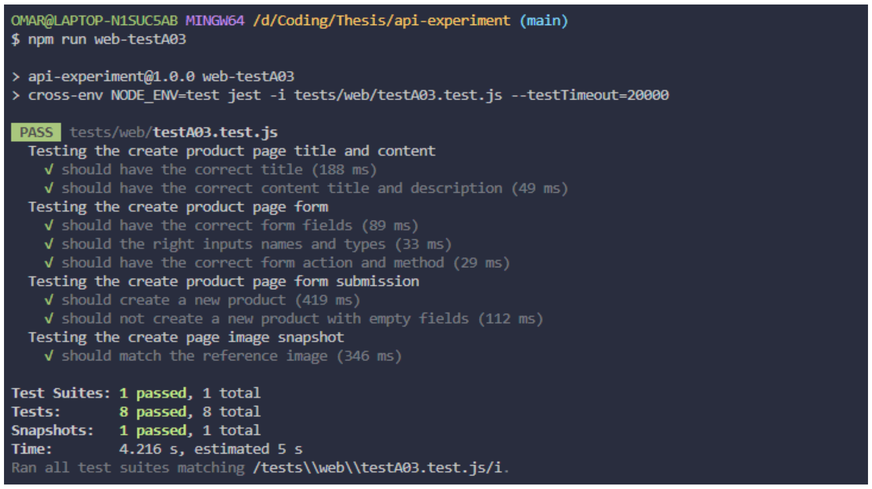


Figure 15 Create Product Page

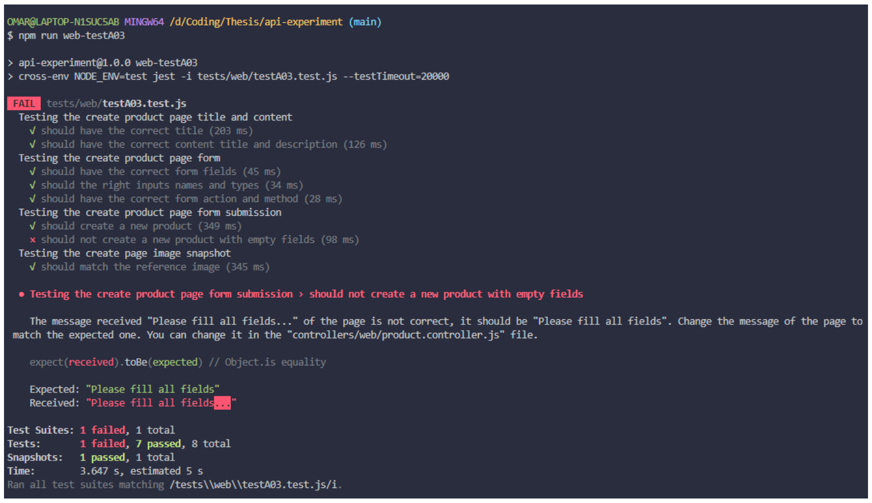
To test the application, copy the file "testA03.test.js" from the "/tests/web" folder and paste it to the folder "/tests/web" in your project directory. After that, run the command "npm run web-testA03" and notice the results.

If everything is correct and working well the results in the terminal should look as the following:



```
OMAR@LAPTOP-N1SUC5AB MINGW64 /d/Coding/Thesis/api-experiment (main)
$ npm run web-testA03

> api-experiment@1.0.0 web-testA03
> cross-env NODE_ENV=test jest -i tests/web/testA03.test.js --testTimeout=20000

 PASS  tests/web/testA03.test.js
  Testing the create product page title and content
    √ should have the correct title (188 ms)
    √ should have the correct content title and description (49 ms)
  Testing the create product page form
    √ should have the correct form fields (89 ms)
    √ should the right inputs names and types (33 ms)
    √ should have the correct form action and method (29 ms)
  Testing the create product page form submission
    √ should create a new product (419 ms)
    √ should not create a new product with empty fields (112 ms)
  Testing the create page image snapshot
    √ should match the reference image (346 ms)

Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   1 passed, 1 total
Time:        4.216 s, estimated 5 s
Ran all test suites matching /tests\\web\\testA03.test.js/i.
```

Figure 16 Successful Web Test Results



```
OMAR@LAPTOP-N1SUC5AB MINGW64 /d/Coding/Thesis/api-experiment (main)
$ npm run web-testA03

> api-experiment@1.0.0 web-testA03
> cross-env NODE_ENV=test jest -i tests/web/testA03.test.js --testTimeout=20000

 FAIL  tests/web/testA03.test.js
  Testing the create product page title and content
    √ should have the correct title (203 ms)
    √ should have the correct content title and description (126 ms)
  Testing the create product page form
    √ should have the correct form fields (45 ms)
    √ should the right inputs names and types (34 ms)
    √ should have the correct form action and method (28 ms)
  Testing the create product page form submission
    √ should create a new product (349 ms)
    × should not create a new product with empty fields (98 ms)
  Testing the create page image snapshot
    √ should match the reference image (345 ms)

  ● Testing the create product page form submission › should not create a new product with empty fields

    The message received "Please fill all fields..." of the page is not correct, it should be "Please fill all fields". Change the message of the page to
 match the expected one. You can change it in the "controllers/web/product.controller.js" file.

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

    Expected: "Please fill all fields"
    Received: "Please fill all fields..."

Test Suites: 1 failed, 1 total
Tests:       1 failed, 7 passed, 8 total
Snapshots:   1 passed, 1 total
Time:        3.647 s, estimated 5 s
Ran all test suites matching /tests\\web\\testA03.test.js/i.
```

Figure 17 Failed Web Test Results

Try to figure out the reason for the problem until the test shows successful results.

After running the test, the new products created through Postman or the web interface will be deleted because the test will delete all the product data and insert the initial data products in the base directory. It is important to keep the file "initial_data.json" in the base directory of the project.

Results

Upon finishing this learning material, students will be able to understand the fundamental concepts of RESTful API and put them into practice by utilizing NodeJS. The students will have a comprehensive understanding of the POST method and its function in adding new data to a MongoDB collection.