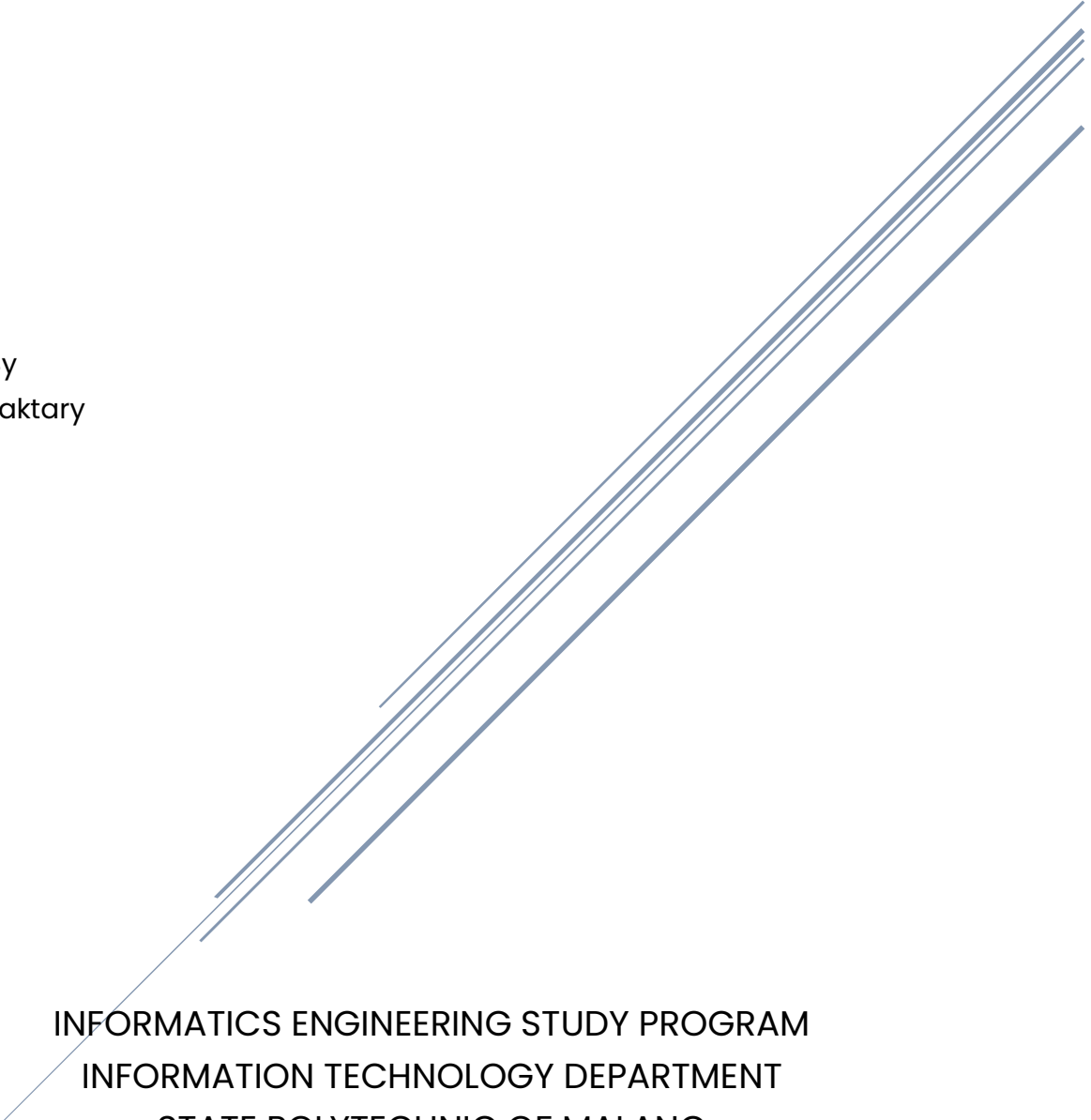


# GUIDE A04

## Create a PATCH Endpoint to Update a Resource

Arranged By  
Omar Al-Maktary



INFORMATICS ENGINEERING STUDY PROGRAM  
INFORMATION TECHNOLOGY DEPARTMENT  
STATE POLYTECHNIC OF MALANG

2023

## Contents

Objectives.....	1
Requirements .....	1
Hardware Specifications .....	1
Minimum Requirements.....	1
Recommended Requirements .....	2
Software required .....	2
NPM Packages .....	2
Resource.....	3
Task Description.....	3
Start Coding.....	3
Run The API Application.....	5
Testing The API Application .....	5
Using Postman .....	5
Running The API Test file .....	7
Creating The Web Application.....	8
Running and Testing The Web Interface.....	11
Results.....	12

# Create a PATCH Endpoint to Update a Resource

## Objectives

1. Students understand how to create a PATCH endpoint to update a product resource in the database using a slug as the product identifier.
2. Students understand how to create a web page to update product resources in the database.

Learning the method PATCH and how to create an endpoint to update an existing resource in a MongoDB collection are the goals of guide A04. The practice of this guide will be used in the same application established in guides A01, A02, and A03.

Students should learn the routes needed to create a web page that can update a resource for the database. The web page has inputs that are required to update the existing 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 A04
  - Tests: api/testA04.test.js, web/testA04.test.js
- 

## Task Description

To allow updates to a product's information, students can create a new PATCH endpoint in their application. When a client sends a request to this endpoint, the function responsible for handling it should be able to receive the product's unique identifier (slug) along with the new data attributes (name, price, and description) that the client wants to update. If the product was not found the server should return an error informing that the product was not found and the data sent should not be saved. If the slug was found the product resource will be updated according to the request body attributes. Students will also create a web page to update 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 updateProduct = async (req, res) => {  
  try {  
  } catch (error) {  
    return res.status(500).json(error);  
  }  
};
```

*Figure 1 updateProduct Function Initial Code*

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

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

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

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

```
router.patch("/product/:slug", updateProduct);
```

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

The function **updateProduct** needs to be required from the controller to handle the requests to the PATCH endpoint `"/api/v1/product/:slug"` similar to the following code:

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

Figure 4 *product.controller.js Required Functions*

- Go back to the file `"product.controller.js"` and update the function **updateProduct** to fit the requirements of the following table. This table contains information needed for the endpoint details.

PATCH <code>"/api/v1/product/:slug"</code> ENDPOINT STRUCTURE			
<i>API Endpoint Path</i>	<i>Request Method</i>	<i>Response Format</i>	<i>Description</i>
<code>"/api/v1/product/:slug"</code>	PATCH	JSON	Updates an existing product with the information provided in the request body.
Request Parameters			
<i>Parameter</i>	<i>Type</i>	<i>Description</i>	
<code>"slug"</code>	String	The slug of the product to update.	
<code>"name"</code>	String	String The updated name of the product.	
<code>"price"</code>	Number	The updated price of the product.	
<code>"description"</code>	String	The updated description of the product.	
Response Parameters			
<i>Parameter</i>	<i>Type</i>	<i>Description</i>	
<code>"product"</code>	Object	An object containing information about the updated product.	
<code>"message"</code>	String	A message indicating the status of the response	
Success Responses			
<i>HTTP Status Code</i>	<i>Response</i>		
200	{ "product": product, "message": "Product updated" }		
Error Responses			
<i>HTTP Status Code</i>	<i>Response</i>		
404	{ "message": "No product found" }		
500	{ error object }		

Table 1 *Endpoint PATCH `"/api/v1/product/:slug"` API Architecture Design*

Note that the **product** in the Success Responses section is an instance of the product object after being updated.

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

```
const filter = { slug: req.params.slug };
const update = {
  name: req.body.name,
  price: parseInt(req.body.price),
  description: req.body.description,
};
await Product.findOneAndUpdate(filter, update, {
  new: true,
});
```

*Figure 5 Update a Product Code Hint*

The first two objects define a **filter** object with a property **slug** whose value is obtained from the request parameters, and an **update** object with the new values for the **name**, **price**, and **description** properties, which are obtained from the request body.

The **findOneAndUpdate** method is provided by Mongoose to update the product that matches the **filter** object with the new values in the **update** object. The **new** option is set to **true**, which means that the method will return the updated version of the document after the update.

## Run 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 PATCH as its method.
2. Use the following link as the URL.

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

Change the “:slug”, to an existing product slug in the database.

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

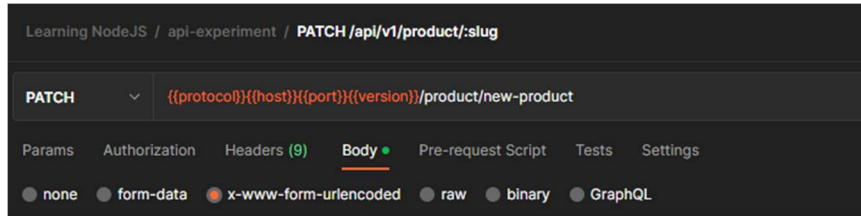


Figure 6 Postman Request Body Options

4. Fill in the name, price, or description to update the product.

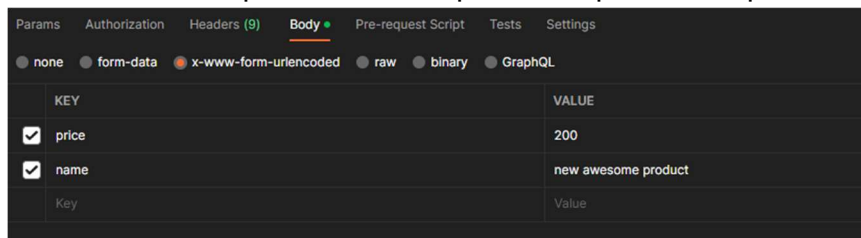


Figure 7 Postman Request x-www-form-urlencoded Data

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

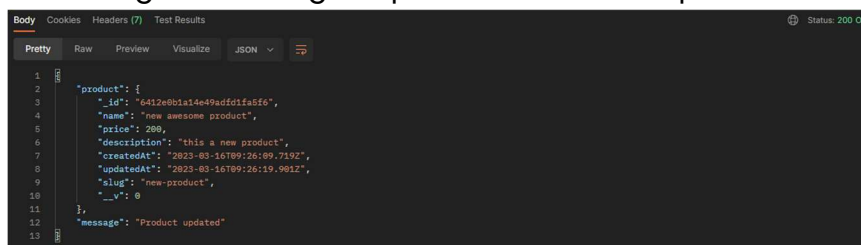


Figure 8 Postman Response Results Updated

Note that the status is 200 meaning the request is OK. Additionally, notice the message indicating the product data was updated. This also can be verified by viewing the data on the MongoDB Atlas panel, it should be updated.

6. Try to send the same request but with a non-existing slug, it should return an error that the product is not found.



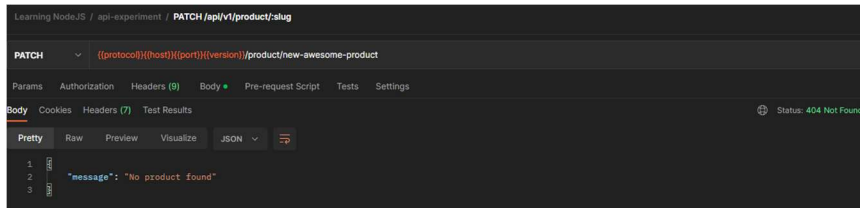


Figure 9 Postman Response Results Not Found

Note that the status is 404 indicating content not found. The response message shows that the product was not found.

## 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 "testA04.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-testA04" and notice the results.
3. If everything is correct and working well the results in the terminal should look as the following:

```
OMAR@LAPTOP-M1SUC5AB MINGW64 ~/Desktop/api-experiment
$ npm run api-testA04

> api-experiment@1.0.0 api-testA04
> cross-env NODE_ENV=test jest -i tests/api/testA04.test.js --testTimeout=20000

console.log
  Database connected successfully

    at log (tests/api/testA04.test.js:18:15)

PASS tests/api/testA04.test.js
  PATCH /api/v1/product/:slug
    ✓ should update a product (110 ms)
    ✓ should not update a product because it does not exist (45 ms)
    ✓ should return error 500 (732 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:  0 total
Time:        3.428 s, estimated 4 s
Ran all test suites matching /tests\\api\\testA04.test.js/i.
```

Figure 10 Successful Test Results

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

```

OWAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/api-experiment
$ npm run api-testA04

> api-experiment@1.0.0 api-testA04
> cross-env NODE_ENV=test jest -i tests/api/testA04.test.js --testTimeout=20000

console.log
  Database connected successfully

    at log (tests/api/testA04.test.js:18:15)

FAIL tests/api/testA04.test.js
  PATCH /api/v1/product/:slug
    ✓ should update a product (161 ms)
    ✗ should not update a product because it does not exist (73 ms)
    ✓ should return error 500 (976 ms)

    • PATCH /api/v1/product/:slug › should not update a product because it does not exist

      Expected status code 404, but got 405, the status 404 means that the server can not find the requested resource. Change it in the file
      "controllers/api/product.controller.js"

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

      Expected: 404
      Received: 405

Test Suites: 1 failed, 1 total
Tests:       1 failed, 2 passed, 3 total
Snapshots:  0 total
Time:        4.905 s
Ran all test suites matching /tests\\api\\testA04.test.js/i.

```

Figure 11 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 Application

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 update an existing 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 "updateProduct". Read the explanation below then fill in the rest of the code.

```

const updateProduct = async (req, res) => {
  if (req.method === "GET") {
    // Find the product using the req.params.slug
    // If the product is not found render the error page
    // with status 404 and message "No product found"
    // Render the update view with the product found and
    // title should be "API-Experiment | Update Product"
    // write your code here ...
  } else {
    if (
      req.body.name === "" ||
      req.body.price === "" ||
      req.body.description === ""
    ) {
      // If any of the fields are empty
    }
  }
}

```

```

    // Find the product using the req.params.slug
    // Render the update view with the product found and
    // title should be "API-Experiment / Update Product" and
    // message should be "Please fill all fields"
    // write your code here ...
  }
  // Update the product using the method findOneAndUpdate
  // pass the new: true option to get the updated product
  // write your code here ...
  // If the product is not found
  // Render the error view with the error object
  // error should be { status: 404, message: "No product found" }
  if (product === null) {
    // write your code here ...
  }
  // Render the details view with the product found and
  // message should be "Product updated"
  // write your code here ...
  return await getProduct(req, res);
}
};

```

Figure 12 "updateProduct" 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 update page in the "web/views/products" folder with the product found and the title "API-Experiment / Update 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 product data will be updated and render the details page with the message "Product Updated". If the product was not found the function should render the error page with an error object.

2. In the "routes/web/product.routes.js" file add two routes with the "/update/:slug" 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 “update.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 "Update this product", and a p element with class description and inner text "Fill the form below to update this 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/update/<%= product.slug %>" 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 update. The value is set to <%= product.name %>.
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 update. The value is set to <%= product.price %>.
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 update. The value is set to <%= product.description %>.
Update button	<button>	A button element with type submit, class btn, and class btn-primary. The button text reads "Update".

Figure 13 "web/views/products/update.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 “Details” for any of the products then clicking the button “Edit this product”, the page should look similar to the following image:

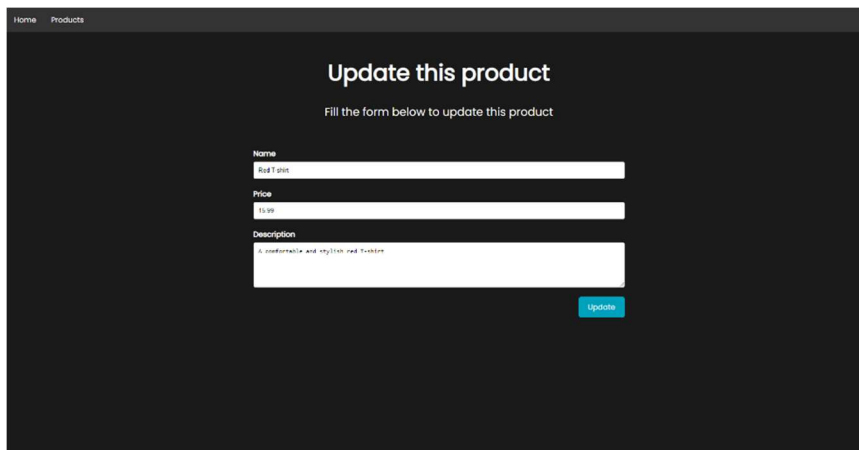
A screenshot of a web application interface. At the top, there's a navigation bar with 'Home' and 'Products' links. The main heading is 'Update this product'. Below it, a subtitle says 'Fill the form below to update this product'. The form has three input fields: 'Name' with the value 'Red T-shirt', 'Price' with the value '15.99', and 'Description' with the value 'A comfortable and optimized T-shirt'. There is an 'Update' button at the bottom right of the form.

Figure 14 Update Product Page

To test the application, copy the file “testA04.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-testA04” 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 ~/Desktop/api-experiment
$ npm run web-testA04

PASS tests/web/testA04.test.js (17.445 s)
  Testing the update page title and content
    ✓ should have the correct title (4697 ms)
    ✓ should have the correct content title and description (516 ms)
  Testing the create product page form
    ✓ should have the correct form fields (635 ms)
    ✓ should the right inputs names and types (1005 ms)
    ✓ should have the correct form action and method (1103 ms)
  Testing the create product page form submission
    ✓ should update the product (1111 ms)
    ✓ should not update the product if the name is empty (718 ms)
    ✓ should don't update the product if the product does not exist (611 ms)
  Testing the update product page image snapshot
    ✓ should match the update product page image snapshot (1526 ms)

Test Suites: 1 passed, 1 total
Tests: 9 passed, 9 total
Snapshots: 1 passed, 1 total
Time: 17.556 s
Ran all test suites matching /tests\\web\\testA04.test.js/i.
```

Figure 15 Successful Web Test Results

```
OMAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/api-experiment
$ npm run web-testA04

> api-experiment@1.0.0 web-testA04
> cross-env NODE_ENV=test jest -i tests/web/testA04.test.js --testTimeout=20000

FAIL tests/web/testA04.test.js (10.806 s)
  Testing the update page title and content
    ✓ should have the correct title (1044 ms)
    ✓ should have the correct content title and description (623 ms)
  Testing the create product page form
    ✓ should have the correct form fields (414 ms)
    ✓ should have the right inputs names and types (496 ms)
    ✓ should have the correct form action and method (591 ms)
  Testing the create product page form submission
    ✗ should update the product (940 ms)
    ✓ should not update the product if the name is empty (710 ms)
    ✓ should don't update the product if the product does not exist (614 ms)
  Testing the update product page image snapshot
    ✓ should match the update product page image snapshot (1006 ms)

  ● Testing the create product page form submission > should update the product

  The message should be "Product updated", but it has "Product updated successfully". You can change it in the "controllers/web/products.controller.js" file.

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

    Expected: "Product updated"
    Received: "Product updated successfully"

Test Suites: 1 failed, 1 total
Tests: 1 failed, 8 passed, 9 total
Snapshots: 1 passed, 1 total
Time: 10.893 s, estimated 14 s
Ran all test suites matching /tests/web/testA04.test.js/i.
```

Figure 16 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 or updated 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

After completing this material, students will have the ability to comprehend the basic concepts of RESTful API and apply them by using NodeJS. Students will have a thorough knowledge of the PATCH function and its role in updating data in a MongoDB collection.