Enhancing a Node.js API with Validation and Error Handling

Objective

This hands-on lab will guide you through enhancing your existing Products API built with Node.js and Express.js by adding robust input validation and centralized custom error handling middleware. You will learn how to validate incoming data for product creation and update routes, ensuring that only well-formed and complete data enters your system. Additionally, you will implement a centralized error handling middleware to manage and respond to errors consistently across your API.

These improvements will not only increase the robustness of your API by reducing invalid data and unexpected failures but also make the codebase more maintainable and user-friendly. By the end of this session, you will have a cleaner, more reliable API that provides clear feedback to clients and simplifies debugging and future enhancements.

Prerequisites

- A working Products API with full CRUD routes created from the **previous module**.
- Familiarity with Express.js request handling, including middleware and routing concepts.
- Understanding of JavaScript object destructuring and basic error handling using try-catch blocks.
- Node.js installed on your local machine, along with a text editor like VS Code or another preferred IDE.

Having these prerequisites will ensure you can follow the step-by-step instructions smoothly and focus on enhancing your API with validation and error handling.

Validating Incoming Product Data

To improve the reliability and integrity of your Products API, it is essential to validate the data clients send to your server. In this section, you will create a custom validation middleware that checks incoming requests for required product fields and verifies their types before allowing further processing. This helps prevent invalid data from entering your database and reduces unexpected errors.

Step 1: Create the Validation Middleware File

We will be using the previous project directory only for this project. Start by creating a new file named `validateProduct.js`. It's a good practice to keep middleware functions separate, so place this file inside a `middleware` or `utils` folder within your project structure:

- `middleware/validateProduct.js` or `utils/validateProduct.js`

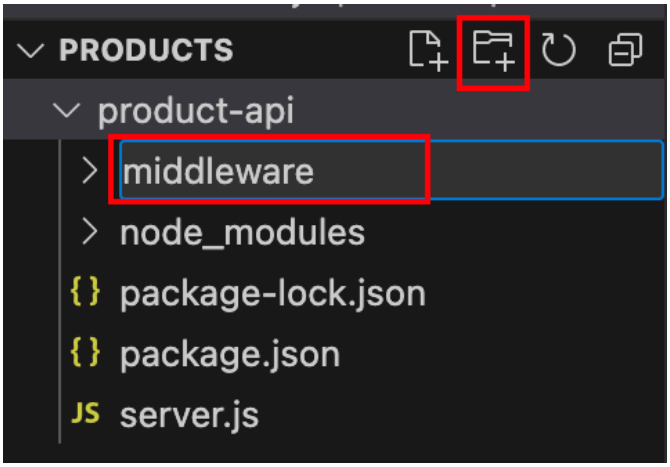

***Figure 1**: middleware directory created*

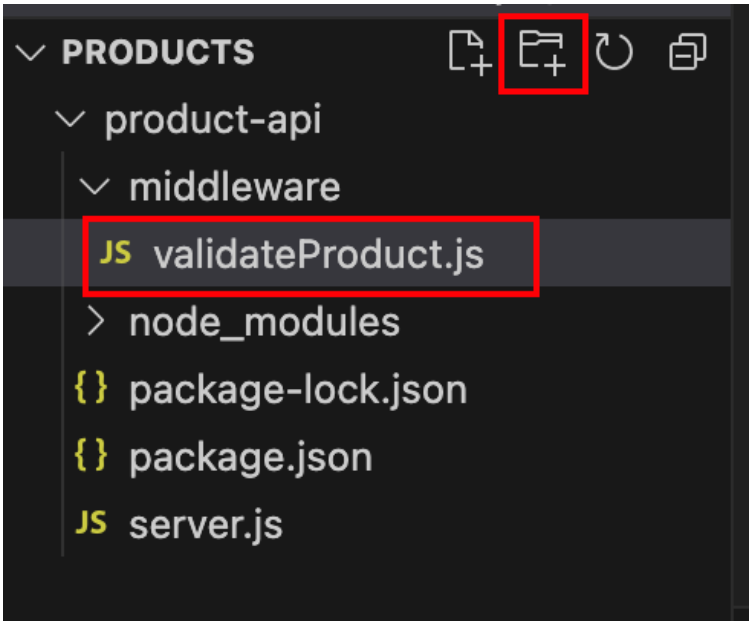

***Figure 2**: validateProduct.js file created*

This middleware will examine the `req.body` object to check for the presence and type of the key product fields: **name**, **price**, and **description**.

Step 2: Write the Validation Middleware Function

Inside `validateProduct.js`, export a function that Express can use as middleware. This function should:

- Check if `req.body.name`, `req.body.price`, and `req.body.description` are present.

- Verify that `name` and `description` are strings.
- Verify that `price` is a number.
- If any checks fail, respond immediately with HTTP status `400 Bad Request` and a clear error message.
- If validation passes, call `next()` to pass control to the next middleware or route handler.

```
function validateProduct(req, res, next) {
   const { name, price, description } = req.body;
    // Check for missing fields (note: 0 is valid for price)
   if (!name || price === undefined || !description) {
     return res.status(400).json({
       error: 'Missing required fields: name, price, and description are
all required.'
     });
   }
    // Trim and validate empty strings
   if (typeof name !== 'string' || !name.trim()) {
     return res.status(400).json({ error: 'Invalid or empty name.' });
   }
    if (typeof description !== 'string' || !description.trim()) {
     return res.status(400).json({ error: 'Invalid or empty description.'
});
   }
    // Validate price
   if (typeof price !== 'number' || isNaN(price)) {
     return res.status(400).json({ error: 'Price must be a valid number.'
});
   }
    if (price <= 0) {
     return res.status(400).json({ error: 'Price must be a positive
number.' });
   }
    // All good
   next();
 }
  module.exports = validateProduct;
```

Explanation: This middleware destructures the product fields from `req.body` and performs a sequence of checks. Early returns ensure that if any validation fails, the client immediately receives a descriptive 400 response. Only when all checks pass, `next()` is called, allowing your route handler to continue processing the request.

Step 3: Apply the Validation Middleware to Routes

Now, integrate this middleware into your `POST /products` and `PUT /products/:id` routes where products are created or updated. Import the middleware at the top of your route or server file, then include it as part of the route handler chain.

1. Import the middleware in `server.js`

Add this line near the top of your `server.js`:

```
const validateProduct = require('./middleware/validateProduct');
```

Place it after the `require('express')` line for organization.

2. Apply the middleware in the POST route

Replace this part of your `/products` POST route:

```
app.post('/products', (req, res) => {
```

with:

```
app.post('/products', validateProduct, (req, res) => {
```

Replace this part of your `/products` PUT route:

```
app.put('/products/:id', (req, res) => {
```

with

```
app.put('/products/:id', validateProduct, (req, res) => {
```

This makes `validateProduct` run **before** your main route logic.

Also, since `validateProduct` already handles the input validation, you can **remove this block** inside the route:

```
// Basic validation: Check if required fields are present
if (!name || price === undefined || !description) {
  return res.status(400).json({ message: 'Name, price, and description are
```

```
required.' });
}
```

Summary

You have now created a reusable validation middleware that ensures any product data sent to your API for creation or update includes the necessary fields with correct types. This simple validation step catches data mistakes early, returning informative error messages to clients. Integrating this validation middleware into your `POST` route strengthens your API's robustness without cluttering the core business logic. In the next section, you will learn how to centralize your error handling to keep your API organized and maintainable.

Creating Custom Error Handling Middleware

Centralized error handling is a best practice in Express.js APIs because it provides a single place to manage all errors occurring throughout the application. Instead of scattering error response logic across multiple route handlers and middleware, centralized error handling simplifies maintenance, improves code readability, and ensures consistent, user-friendly error responses.

In this section, you will create a custom error handling middleware function. This middleware will catch any errors passed via `next(err)` and respond appropriately to the client while safely logging useful debugging details on the server side.

Step 1: Create the Error Handling Middleware File

Create a new file called `errorHandler.js` in your `middleware` folder (or wherever you organize middleware):

- `middleware/errorHandler.js`

This module will export a function following Express's special 4-argument signature `(err, req, res, next)` which identifies it as an error-handling middleware.

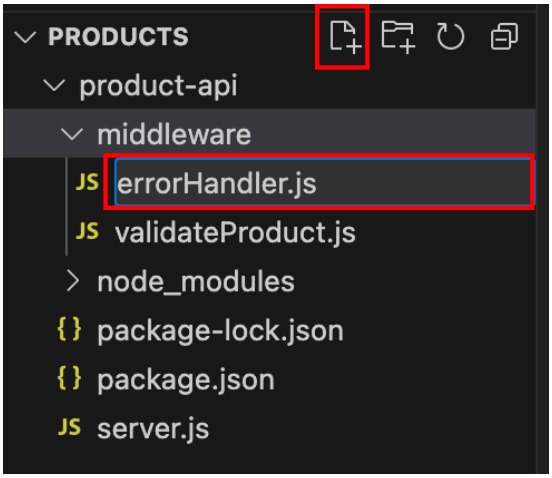

***Figure 3**: errorHandler.js file created*

Step 2: Write the Error Handler Function

Add the following code to `errorHandler.js`:

```
// middleware/errorHandler.js

// Centralized error handling middleware for Express
function errorHandler(err, req, res, next) {
   console.error('Error caught by middleware:', err); // Log full error for
debugging
    // Set default status and message for unexpected errors
   const status = err.status || 500;
   let message = err.message || 'Internal Server Error';
    // For unexpected errors, avoid exposing sensitive details
   if (status === 500) {
     message = 'Something went wrong. Please try again later.';
   }
    // Send JSON error response with status and message
   res.status(status).json({ error: message });
 }
  module.exports = errorHandler;
```

***Explanation**:* This function logs the error details to the console, which helps developers trace issues during development or debugging. It then responds with a JSON object containing an `error` message and HTTP status code. For known errors, you can set `err.status` and `err.message`, but if not provided, the middleware defaults to a 500 status and a generic message to avoid leaking sensitive information.

Step 3: Mount the Middleware in Your Server

To ensure Express uses this error handler for all endpoints, add it as the last middleware in your `server.js` (or main application file), after all routes:

```
// server.js or index.js
const express = require('express');
const app = express();
const errorHandler = require('./middleware/errorHandler');

// ... your other middleware and routes here ...
// Mount the error handler last
app.use(errorHandler);

// rest of the code
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

***Important**:* Mounting the error handler last ensures it catches any errors passed down from previous middleware or routes.

Step 4: Triggering Errors to Test the Handler

You can test that your error handler works by:

- Accessing an undefined route like `GET /nonexistent`, which Express will forward as a 404 error if you handle it by calling `next(err)`.
- Manually throwing errors inside route handlers using `throw new Error('Some error')` or `next(new Error('Some error'))`.
- Passing custom error objects with `status` and `message` for more fine-grained control, for example:

```
// Example in a route handler:
app.get('/products/:id', (req, res, next) => {
  const product = products.find(p => p.id === req.params.id);
  if (!product) {
    const error = new Error('Product not found');
    error.status = 404;
    return next(error); // Forward error to centralized handler
  }
  res.json(product);
```

```
});
```

This approach removes the need to manually send error responses inside every route, delegating error response handling to your centralized middleware.

By following these steps, you establish a clear and maintainable error-handling strategy that boosts the reliability of your API and improves client-side error communication.

Update Routes to Use Next for Error Propagation

To fully leverage centralized error handling, your route handlers must forward errors to the error handling middleware instead of responding directly. This means wrapping your route logic within `try-catch` blocks and calling `next(err)` when an error occurs. Doing so ensures all errors funnel through one place, promoting consistency and easier maintenance.

Before Refactoring

Here is your typical route handler that directly sends error responses:

```
app.get('/products/:id', (req, res) => {
  const id = parseInt(req.params.id); // Get the ID from the URL parameters
and convert to a number
   // Find the product in the array by ID
  const product = products.find(p => p.id === id);
   // Handle the edge case where the product is not found
  if (!product) {
    // Send a 404 Not Found status if the product doesn't exist
    return res.status(404).json({ message: 'Product not found.' });
  }
   // Send back the found product with a 200 OK status
  res.json(product); // Status 200 is default
});
```

After Refactoring with Try-Catch and Next

Modify your existing code as shown below, to catch synchronous or asynchronous errors and pass them to `next(err)`:

```
app.get('/products/:id', async (req, res, next) => {
   try {
     const id = parseInt(req.params.id);
     const product = products.find(p => p.id === id); // Compare number ===
number
```

```
      if (!product) {
       const error = new Error('Product not found');
       error.status = 404;
       throw error;
     }
      res.json(product);
   } catch (err) {
     next(err);
   }
 });
```

Repeat this same pattern to the other route: `PUT /products/:id`:

```
// PUT /products/:id with validation and error forwarding
app.put('/products/:id', validateProduct, async (req, res, next) => {
   try {
       const id = parseInt(req.params.id);
       const existingProductIndex = products.findIndex(p => p.id === id);

     if (existingProductIndex === -1) {
       const error = new Error('Product not found');
       error.status = 404;
       throw error;
     }
     products[existingProductIndex] = { id: id, ...req.body };
     res.json({ message: 'Product updated successfully', product:
products[existingProductIndex] });
   } catch (err) {
     next(err); // Pass any unexpected errors here also
   }
});
```

Final `server.js`

```
const express = require('express');
const app = express();
const PORT = 3000;
const validateProduct = require('./middleware/validateProduct');
const errorHandler = require('./middleware/errorHandler');

// Middleware to parse JSON request bodies
```

```
app.use(express.json());

// In-memory storage for products
const products = [];
let nextProductId = 1; // Counter for generating unique IDs

// Basic route to check if server is running
app.get('/', (req, res) => {
res.send('Welcome to the Product API!');
});

app.use(errorHandler);

// Start the server and listen on PORT
app.listen(PORT, () => {
console.log(`Server is running on http://localhost:${PORT}`);
});

app.post('/products',validateProduct, (req, res) => {
  const { name, price, description } = req.body;
   const newProduct = {
    id: nextProductId++, // Assign unique ID and increment
counter
    name,
    price,
    description,
  };
   products.push(newProduct); // Add the new product to the
array
   // Send back the created product with a 201 Created status
  res.status(201).json(newProduct);
});

// GET /products - Get all products
app.get('/products', (req, res) => {
  // Simply send the entire products array as a JSON response
```

```
  res.json(products);
});

// GET /products/:id - Get a single product by ID
app.get('/products/:id', async (req, res, next) => {
 try {
   const id = parseInt(req.params.id);
   const product = products.find(p => p.id === id); // Compare
number === number
    if (!product) {
     const error = new Error('Product not found');
     error.status = 404;
     throw error;
   }
    res.json(product);
 } catch (err) {
   next(err);
 }
});

// PUT /products/:id - Update a product by ID
// PUT /products/:id with validation and error forwarding
app.put('/products/:id', validateProduct, async (req, res, next)
=> {
 try {
     const id = parseInt(req.params.id);
     const existingProductIndex = products.findIndex(p => p.id
=== id);
   if (existingProductIndex === -1) {
     const error = new Error('Product not found');
     error.status = 404;
     throw error;
   }
   products[existingProductIndex] = { id: id, ...req.body };
   res.json({ message: 'Product updated successfully', product:
products[existingProductIndex] });
```

```
 } catch (err) {
   next(err); // Pass any unexpected errors here also
 }
});

// DELETE /products/:id - Delete a product by ID
app.delete('/products/:id', (req, res) => {
  const id = parseInt(req.params.id); // Get ID from URL and
convert to number
   // Find the index of the product in the array
  const index = products.findIndex(p => p.id === id);
   // Handle the edge case where the product is not found
  if (index === -1) {
    // Send a 404 Not Found status
    return res.status(404).json({ message: 'Product not found.'
});
  }
   // Remove the product from the array using its index
  // splice(startIndex, deleteCount)
  products.splice(index, 1);
   // Send a success message with a 200 OK status
  res.json({ message: 'Product deleted successfully.' });
});
```

Why use next(err)? By passing errors to `next`, you delegate error responses to your centralized error handler middleware. This eliminates duplicated error handling code in routes, keeps route logic focused and readable, and ensures consistent error responses across the API.

Using async functions with try-catch is especially important if your routes involve asynchronous operations like database calls. Any errors thrown or rejected Promises caught will be forwarded properly, avoiding unhandled rejections and unexpected crashes.

This refactoring enhances your API's robustness and maintainability, making debugging easier and the client experience more consistent.

Testing the API

After adding validation and centralized error handling, it is important to verify these features work as expected. You can use tools like **Postman** or **curl** to send HTTP requests to your API and observe responses.

Start the server with the following command:

```
node server.js
```

1. Testing Valid Requests

- Send a `POST /products` or `PUT /products/:id` request with a JSON body including `name` (string), `price` (number), and `description` (string).

In **Postman** select POST, enter the URL `http://localhost:3000/products`, and in the Body tab, choose *raw* and set type to `JSON`. Include JSON data like:

```
{
  "name": "Laptop1",
  "price": 1200,
  "description": "High performance laptop"
}
```

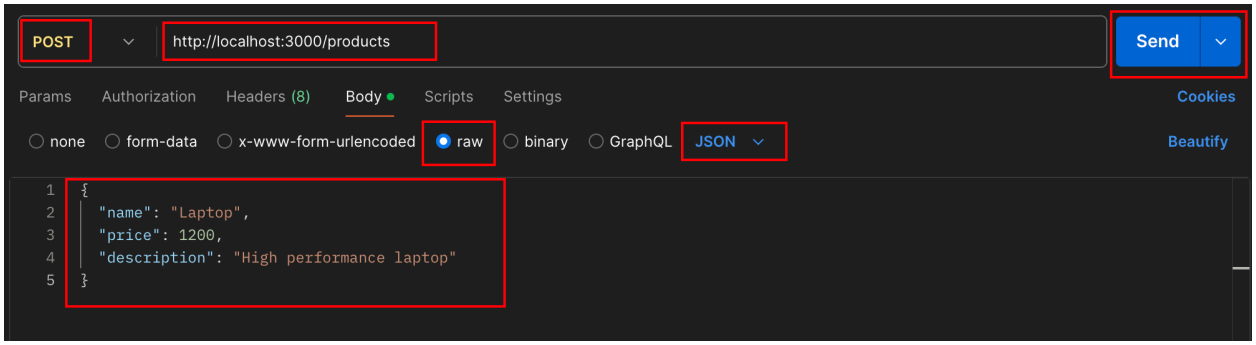

***Figure 4**: POST request ready to test*

- Hit send button

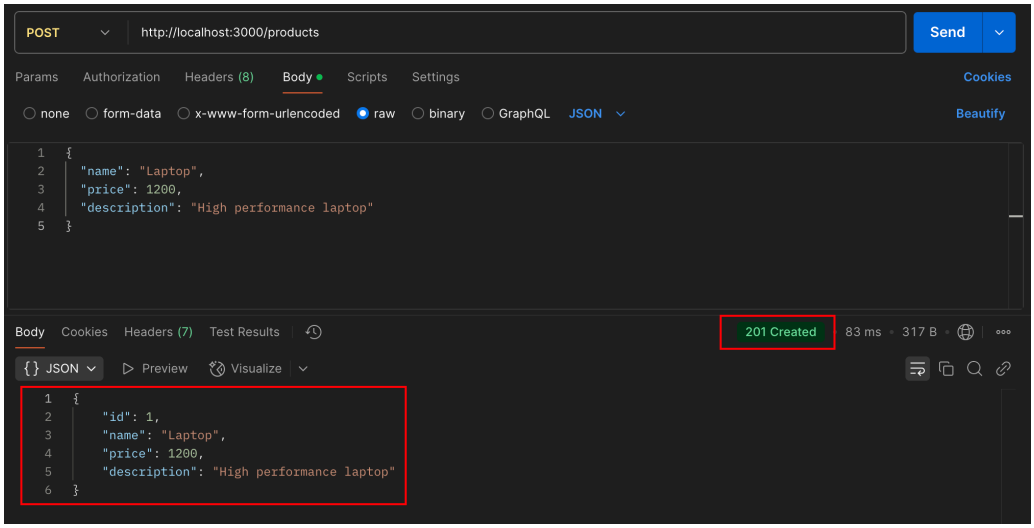

***Figure 5**: POST request successful*

Send updated fields in JSON format.

To update the price:

```
{
  "name": "Laptop1",
  "price": 1100,
  "description": "High performance laptop"
}
```

Postman: PUT `http://localhost:3000/products/1` with the JSON body.

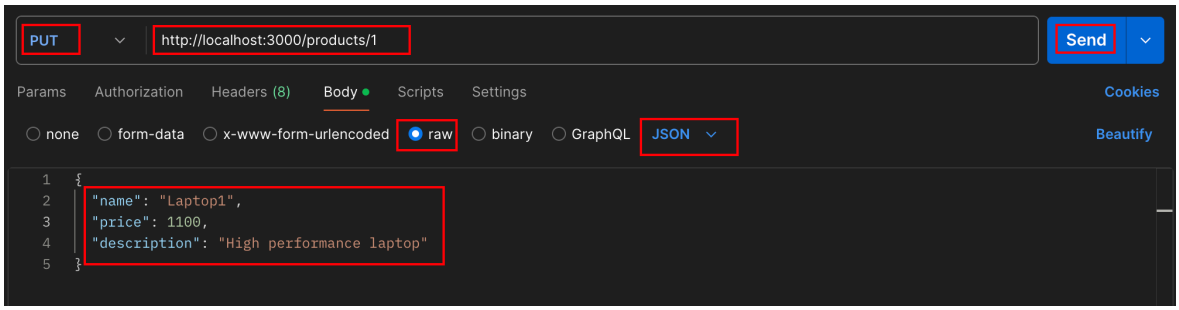

***Figure 6**: PUT request config*

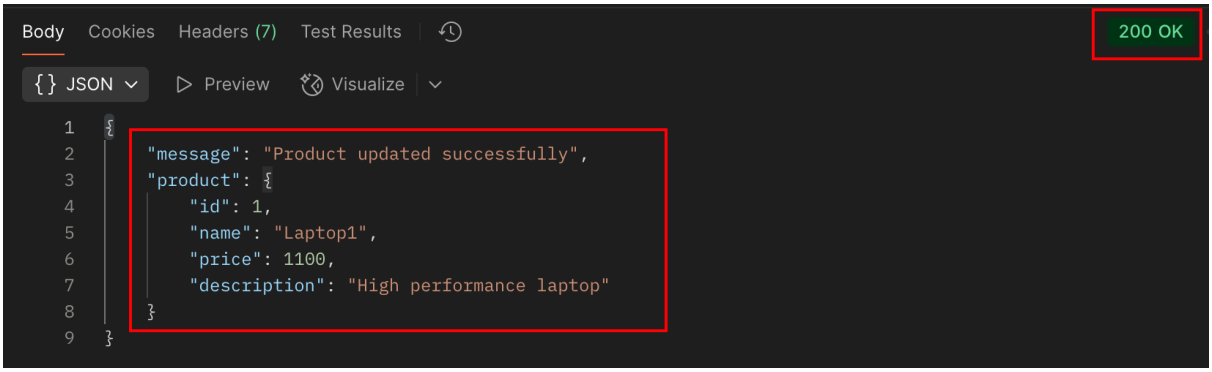

***Figure 7**: PUT request successful*

- Expect a `201 Created` (POST) or `200 OK` (PUT) response with a success message and the product data.

2. Testing Validation Failures

- Try sending a request missing one or more required fields or with incorrect data types (e.g., `price` as a string).
    - The API should respond with `400 Bad Request` and a JSON error message describing the validation issue

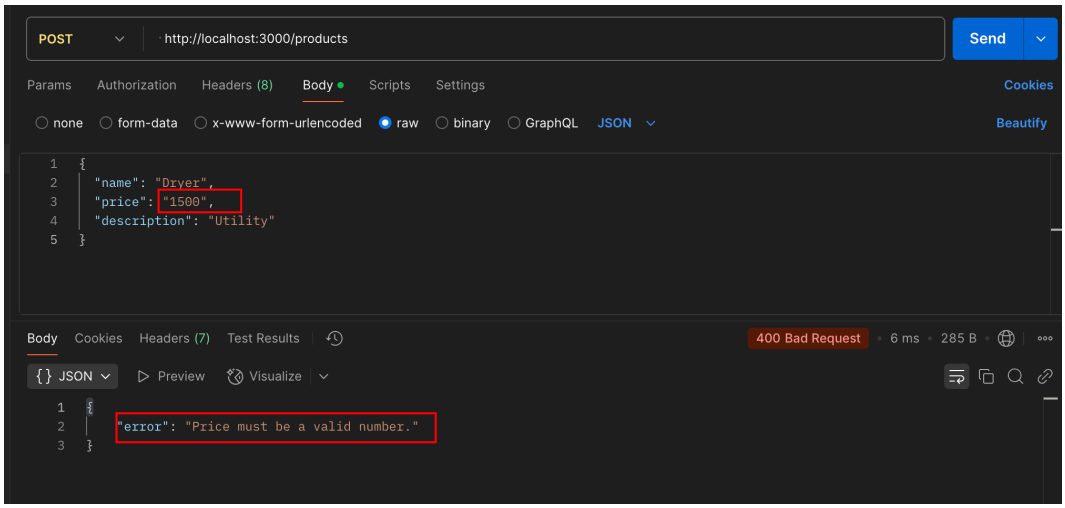

***Figure 8**: POST request failed*

3. Testing Error Handling

- Request a non-existent product with `GET /products/:id` (where the ID does not exist). You should receive a `404 Not Found` status with a relevant JSON error message.

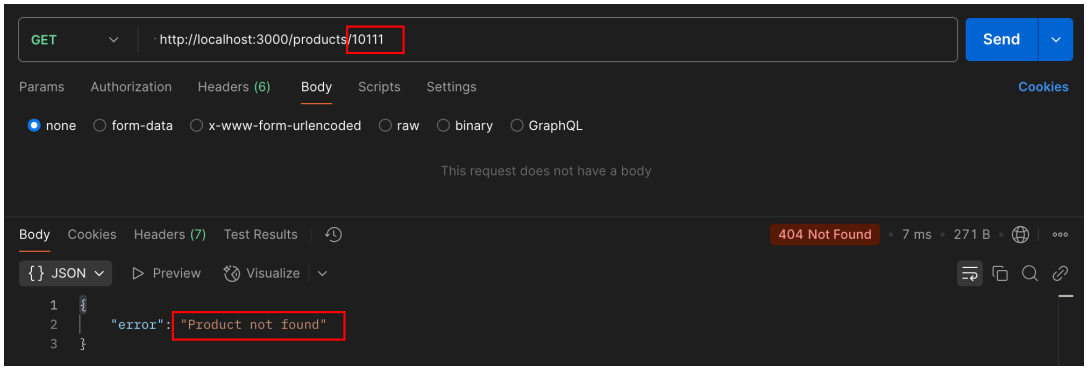

***Figure 9**: GET request failed: "**PRODUCT NOT FOUND**"*

- To simulate a server error, you can temporarily throw an error in a route handler (e.g., `throw new Error('Test error')`). The centralized error handler should catch this and respond with a `500 Internal Server Error` and a generic error message.

By following these tests, you ensure your API gracefully handles both expected and unexpected input while providing clear, consistent feedback. This improves client experience and simplifies debugging.

Conclusion

Throughout this hands-on lab, you enhanced your existing Products API by adding robust input validation and centralized error handling middleware. Implementing reusable validation middleware ensures that only well-formed product data with the required fields and correct types enter your system, significantly reducing the risk of bad data affecting your API's behavior. By refactoring route handlers to use `next(err)` and adopting a centralized custom error handler, you streamlined error management, making your codebase cleaner, more maintainable, and consistent in responding to client errors. These improvements not only boost API reliability but also improve the developer and user experience.

*To further enhance your API, consider exploring popular validation libraries such as **Joi** or **Zod**, which offer advanced schema definitions and richer validation capabilities. Additionally, adopting standardized error response formats using structured error objects or codes will make your API more professional and easier to integrate with diverse clients.*