GUIDE A02

Create GET Endpoints to Fetch Data

Arranged By
Omar Al-Maktary

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

Contents

Create GET Endpoints to Fetch Data

Objectives

1. Students understand how to create a GET endpoint to return a list of products.
2. Students understand how to create a GET endpoint to return a specific product using a slug.
3. Students understand how to create a web interface to display a table of all products and create a page to view details about each product.

The purpose of guide A02 is for students to learn the method GET and its use cases in a NodeJS RESTful API application. The previous application established in guide A01 will be used in this session to build new files to facilitate the usage of the GET method.

Students should learn how to develop a GET method to retrieve data. Students should also grasp how data flows utilizing object models and controllers. Students will also learn how to display all the products in a table for the web interface.

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 A02
- Tests: api/testA02.test.js, web/testA02.test.js
- Supplements: initial_data.json

Task Description

Students can create two GET endpoints to fetch product data from the database. The first function is to retrieve all data from a schema or a collection in MongoDB Atlas using a mongoose schema object. This endpoint can be useful for retrieving a comprehensive list of all products in the database. The second method is to retrieve data based on its slug value. This endpoint can be used to fetch a specific product by its unique slug, which is a user-friendly version of the product name that can be used in URLs or search queries.

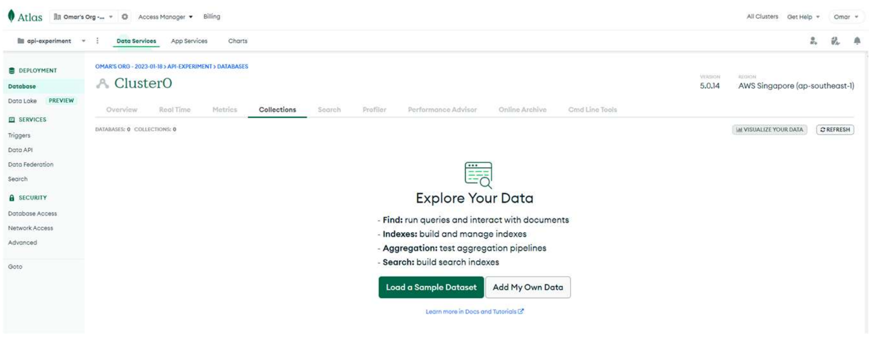
By creating these two endpoints, students can provide a basic but essential set of functionalities for accessing product data from the database. Additionally, students can build upon these endpoints by adding filtering or sorting options based on product attributes such as price.

Students can learn how to create a web interface that shows all the products fetched from the database and display them in a table. This table will contain data about each product and a button to display each product's details alongside buttons for actions to be performed on the product's data.

Adding The Initial Data to MongoDB Atlas

In this section, new data will be added to MongoDB to act as initial data that can be retrieved. To add these initial data, the file "initial_data.json" will be used by following these steps:

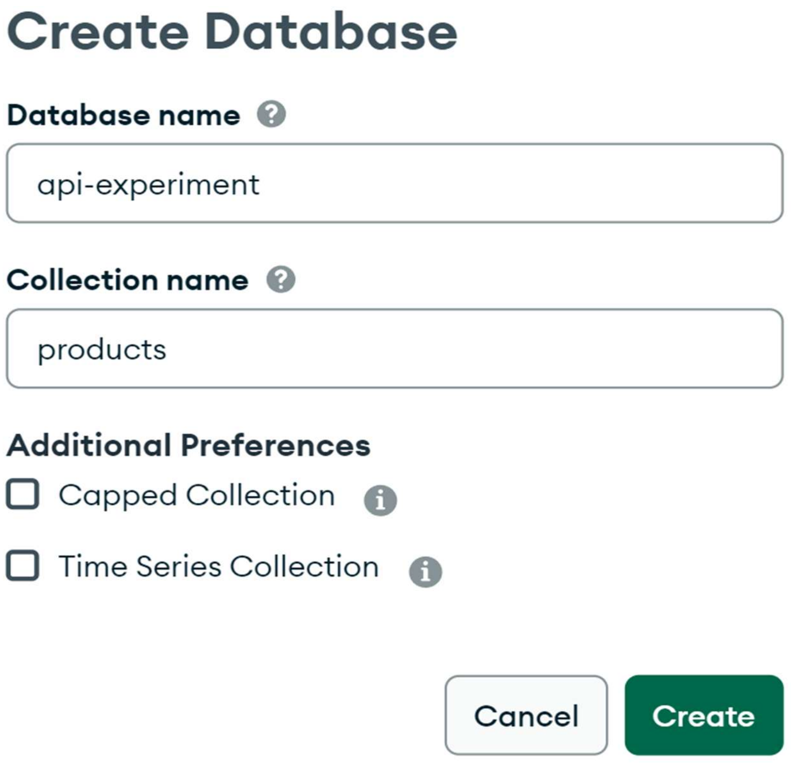
1. Copy the "initial_data.json" file into the base directory of the project "api-experiment".
2. Go to the MongoDB Atlas website and access the cluster that has been created in the previous guide. Go to the "Collections" tab and click on the "Add My Own Data" button.



- **Find:** run queries and interact with documents
- **Indexes:** build and manage indexes
- **Aggregation:** test aggregation pipelines
- **Search:** build search indexes

Figure 1 MongoDB Atlas Collections Panel

3. Create a new database with the name "api-experiment" and a collection with the name "products". Leave the "Additional Preferences" empty and click the "Create" button.



Database name: api-experiment

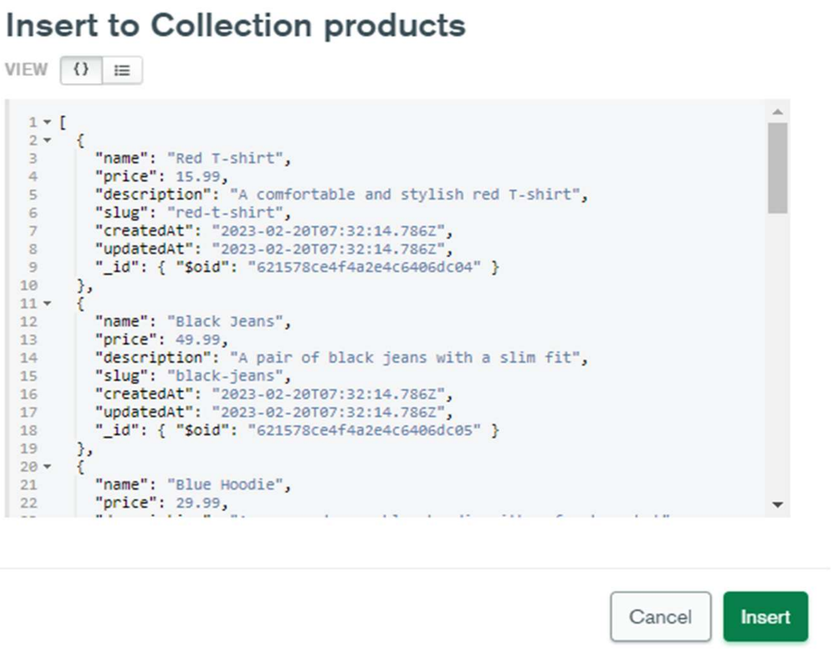
Collection name: products

☐ Capped Collection

☐ Time Series Collection

Figure 2 Create a New Database in MongoDB Atlas

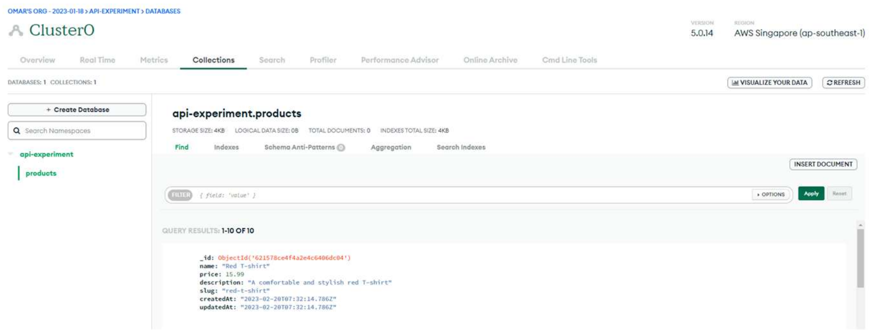
4. After the database has been created, click on "INSERT DOCUMENT". Choose the raw format and then copy all the content of the "initial_data.json" file and paste it into the data box. Click "Insert".



```
[
  {
    "name": "Red T-shirt",
    "price": 15.99,
    "description": "A comfortable and stylish red T-shirt",
    "slug": "red-t-shirt",
    "createdAt": "2023-02-20T07:32:14.786Z",
    "updatedAt": "2023-02-20T07:32:14.786Z",
    "_id": { "$oid": "621578ce4f4a2e4c6406dc04" }
  },
  {
    "name": "Black Jeans",
    "price": 49.99,
    "description": "A pair of black jeans with a slim fit",
    "slug": "black-jeans",
    "createdAt": "2023-02-20T07:32:14.786Z",
    "updatedAt": "2023-02-20T07:32:14.786Z",
    "_id": { "$oid": "621578ce4f4a2e4c6406dc05" }
  },
  {
    "name": "Blue Hoodie",
    "price": 29.99,
```

Figure 3 Insert New Data into Products Collection

5. There should be 10 new data which should be added to the database as the following:



```
_id: ObjectId('621578ce4f4a2e4c6406dc04')
name: "Red T-shirt"
price: 15.99
description: "A comfortable and stylish red T-shirt"
slug: "red-t-shirt"
createdAt: "2023-02-20T07:32:14.786Z"
updatedAt: "2023-02-20T07:32:14.786Z"
```

Figure 4 Products Collection Data

RESTful API Theory

In this section, an explanation of concepts and references will be used in this guide document related to RESTful API. RESTful API stands for Representational State Transfer API. It is an architectural style that is commonly used for creating web-based APIs. RESTful APIs provide a standardized way of communicating between client applications and servers. Here are some of the concepts used in this guide.

1. Client-server architecture is a model for computing in which a client, such as a web browser, requests resources or services from a server, such as a web server, over a network. The server processes the request and sends the response back

to the client, which can then display the requested information or execute the requested service.

2. A URL (Uniform Resource Locator) is a string of characters that identifies a specific resource on the web. It contains the following parts in the example http://localhost:8080/api/v1/products?search=product:
    - "http" is the protocol used to communicate with the server.
    - "localhost:8080" is the server address and port number. This specifies the domain name or IP address of the server where the resource is located. In this case, it is localhost, which refers to the local computer. The port specifies the port number that the server listens to for incoming requests. In this case, it is 8080.
    - "/api/v1" is the base path or endpoint that identifies the API version.
    - "/products" is the resource being requested from the API.
    - "?search=product" is a query parameter that filters the results to only include products that match the search term "product".
3. An API Endpoint Path is a URI that identifies a specific resource on the server. It is used by the client to access a specific operation or function exposed by the API. The endpoint path consists of a base path followed by a resource identifier. In the example, the endpoint path is /api/v1/products.
    - "api": This is the name of the API.
    - "v1": This is the version of the API.
    - "products": The name of the resource being accessed through the API.
4. The Request Method is a way of telling the server what action the client wants to perform on the resource identified by the endpoint path. The most commonly used request methods are:
    - GET: It is used to retrieve data from the server.
    - POST: It is used to submit data to the server to create a new resource.
    - PATCH: It is used to update a specific part of an existing resource.
    - DELETE: It is used to delete a specific resource.
5. Response Format: The Response Format is the format in which the server sends the response to the client. The most commonly used response formats are:
    - JSON (JavaScript Object Notation): It is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate.
    - XML (Extensible Markup Language): It is a markup language that is designed to store and transport data.

    - HTML (Hypertext Markup Language): It is the standard markup language for creating web pages and web applications.

6. The Request Parameters or Payload are the data that the client sends to the server in the request. They are used to perform specific actions on the resource identified by the endpoint path. In the example, the search parameter is passed as a request parameter.
7. The Response Parameters are the data that the server sends back to the client in the response. They are used to provide information about the resource identified by the endpoint path. In the example, the product is returned as a response parameter.
8. The Success Responses are the HTTP status codes that the server sends back to the client when the request is successfully processed. The most commonly used success responses are:
    - 200 OK: It indicates that the request was successful and the server has returned the requested data.
    - 201 Created: It indicates that the request was successful and a new resource has been created.
    - 204 No Content: It indicates that the request was successful, but there is no data to return.
9. Error Responses are the HTTP status codes that the server sends back to the client when an error occurs. They are used to provide information about the type of error that occurred. The most commonly used error responses are:
    - 400 Bad Request: It indicates that the request was malformed and could not be understood by the server.
    - 401 Unauthorized: It indicates that the client is not authorized to access the requested resource. This may occur if the client fails to authenticate itself, or if the client lacks the necessary permissions to access the resource.
    - 403 Forbidden: It indicates that the server understood the request, but refuses to authorize it. This may occur if the client lacks the necessary permissions to access the resource.
    - 404 Not Found: It indicates that the server could not find the requested resource. This may occur if the requested resource does not exist or if the URL is incorrect.

- 409 Conflict: It indicates that there was a conflict when processing the request. This may occur if the requested operation cannot be performed due to a conflict with the current state of the resource.
- 500 Internal Server Error: It indicates that an unexpected error occurred on the server while processing the request. This may occur if there is a bug in the server code or if there is a problem with the server's infrastructure.
- 503 Service Unavailable: It indicates that the server is currently unable to handle the request. This may occur if the server is undergoing maintenance or if it is overloaded with requests.

API Architecture Design Example

To create an API, developers often design tables outlining the necessary points of an API endpoint. The design usually includes the structure, request payload, response parameters, success response, and error response.

Here is an explanation of the endpoint created in the previous guide but with more details and structure:

GET “/api/v1/test” ENDPOINT STRUCTURE			
API Endpoint Path	*Request Method*	*Response Format*	*Description*
“/api/v1/test”	GET	JSON	Returns a response with the "message" field if present else returns the response with "alive" set to "True".
Request Parameters			
Parameter		*Type*	*Description*
“message”		String	A string containing a message from the client to the server.
Response Parameters			
Parameter		*Type*	*Description*
“message”		String	Will only return this parameter if the client requested a “message” property.
“alive”		String	If there is no message from the client the server will return a “True” value.
Success Responses			
HTTP Status Code			*Response*
200			{ "message": message }
200			{ "alive": "true" }
Error Responses			

HTTP Status Code	*Response*
500	Internal Server Error

Table 1 Endpoint GET "/api/v1/test" API Architecture Design

Start Coding

1. The concept of API Architecture Design should be understood to continue this guide's instructions. Please read the previous section in case the references in the coming steps are unclear.
2. Once the installation has been completed, create a folder named “models” in the base directory. In this folder, create a file named “product.model.js”. This file will contain the schema of the product object. Copy the following code to the file and read the explanation below.

```
const mongoose = require("mongoose");
const mongoose_slug_generator = require("mongoose-slug-generator");
const options = {
  separator: "-",
  lang: "en",
  truncate: 120,
};
mongoose.plugin(mongoose_slug_generator, options);
const Schema = mongoose.Schema;
const productSchema = new Schema(
  {
    name: {
      type: String,
      required: [true, "Name is required"],
    },
    price: {
      type: Number,
      required: [true, "Price is required"],
      min: [0, "Price must be greater than 0"],
    },
    description: {
      type: String,
      required: [true, "Description is required"],
    },
    slug: {
      type: String,
      slug: "name",
      unique: true,
    },
  },
  {
    timestamps: true,
    collection: "products",
  }
);
productSchema.index({ name: "text", description: "text" });
module.exports = mongoose.model("Product", productSchema);
```

Figure 5 "models/product.model.js" Code

The code begins by importing the **mongoose** and **mongoose-slug-generator** modules. The options object is used to configure the **mongoose-slug-generator** plugin. It sets the separator for the slug to be an underscore, the language to be English, and the maximum length of the slug to be 120 characters.

The productSchema object defines the structure of the product object using Mongoose's **Schema** constructor. It has four fields: **name**, **price**, **description**, and **slug**. The **name**, **price**, and **description** fields are required and have validation rules set for them. The **slug** field is generated automatically by the **mongoose-slug-generator** plugin using the **name** field as the source of the slug. The **timestamps** option adds **createdAt** and **updatedAt** fields to the schema to track the creation and modification times of the product object. The **collection** option specifies the name of the MongoDB collection that will be used to store the product documents.

Finally, the **productSchema** is indexed to enable text search on the **name** and **description** fields of the product document. The schema is exported as a Mongoose model named "Product" that can be used to create, read, update, and delete product objects in the database.

3. Create a folder named "controllers" and inside it, create two folders namely "api" and "web" in the "api" folder, and create a file named "product.controller.js". Copy the following code into the file.

```
const Product = require("../../models/product.model");
const getProducts = async (req, res) => {
  try {
  } catch (error) {
    return res.status(500).json(error);
  }
};
const getProduct = async (req, res) => {
 try {
 } catch (error) {
  res.status(500).json(error);
 }
};

module.exports = {
 getProducts,
 getProduct,
};
```
Figure 6 "controllers/api/product.controller.js" Code

The **require()** function imports the **Product** model from the **../../models/product.model** module. This model provides an interface to query the database and interact with product documents.

The **getProducts** function handles an HTTP GET request to retrieve a list of products. It is declared as an asynchronous function and wrapped in a try-catch block to handle any errors that may occur during the database operation.

The **getProduct** function handles an HTTP GET request to retrieve a single product by its slug. It is similar to the **getProducts** function, but it takes a slug parameter from the request and uses it to query the database for the corresponding product document.

The **module.exports** statement exports the **getProducts** and **getProduct** functions as an object, which can be imported and used in other parts of the application. The two functions will be filled in later steps according to their API Architecture Design.

4. Create a folder named "routes" and inside it, create two folders namely "api" and "web" in the "api" folder, and create a file named "product.routes.js". Copy the following code into the new file.

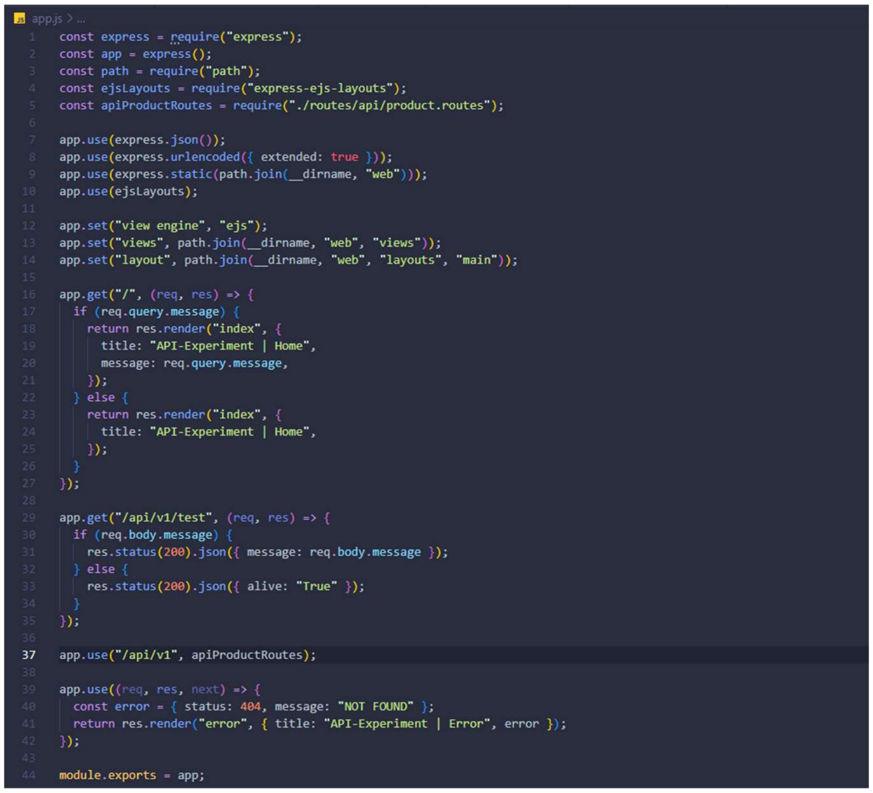
```
const express = require("express");
const {
 getProducts,
 getProduct,
} = require("../../controllers/api/product.controller");
const router = express.Router();
router.get("/products", getProducts);
router.get("/product/:slug", getProduct);
module.exports = router;
```
Figure 7 "routes/api/product.routes.js" Code

This code imports the express module and functions from a product controller module. It creates a router instance to handle product requests. Two routes are defined using router.get(): one to retrieve all products and one to retrieve a single product by its slug. The router object is exported using module.exports, allowing it to be used in the main Express app with app.use().

5. Open the file "app.js" and update the code to allow the use of api endpoint by specifying the version to be "v1". To define the endpoint, copy the following lines of code to the "app.js" file.

```
const apiProductRoutes= require("./routes/api/product.routes");
app.use("/api/v1", apiProductRoutes);
```

The "app.js" file should look like the following:



```
const express = require("express");
const app = express();
const path = require("path");
const ejsLayouts = require("express-ejs-layouts");
const apiProductRoutes = require("./routes/api/product.routes");

app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, "web")));
app.use(ejsLayouts);

app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "web", "views"));
app.set("layout", path.join(__dirname, "web", "layouts", "main"));

app.get("/", (req, res) => {
  if (req.query.message) {
    return res.render("index", {
      title: "API-Experiment | Home",
      message: req.query.message,
    });
  } else {
    return res.render("index", {
      title: "API-Experiment | Home",
    });
  }
});

app.get("/api/v1/test", (req, res) => {
  if (req.body.message) {
    res.status(200).json({ message: req.body.message });
  } else {
    res.status(200).json({ alive: "True" });
  }
});

app.use("/api/v1", apiProductRoutes);

app.use((req, res, next) => {
  const error = { status: 404, message: "NOT FOUND" };
  return res.render("error", { title: "API-Experiment | Error", error });
});

module.exports = app;
```

Figure 8 app.js Code

The file structure of the project should be similar to the following:

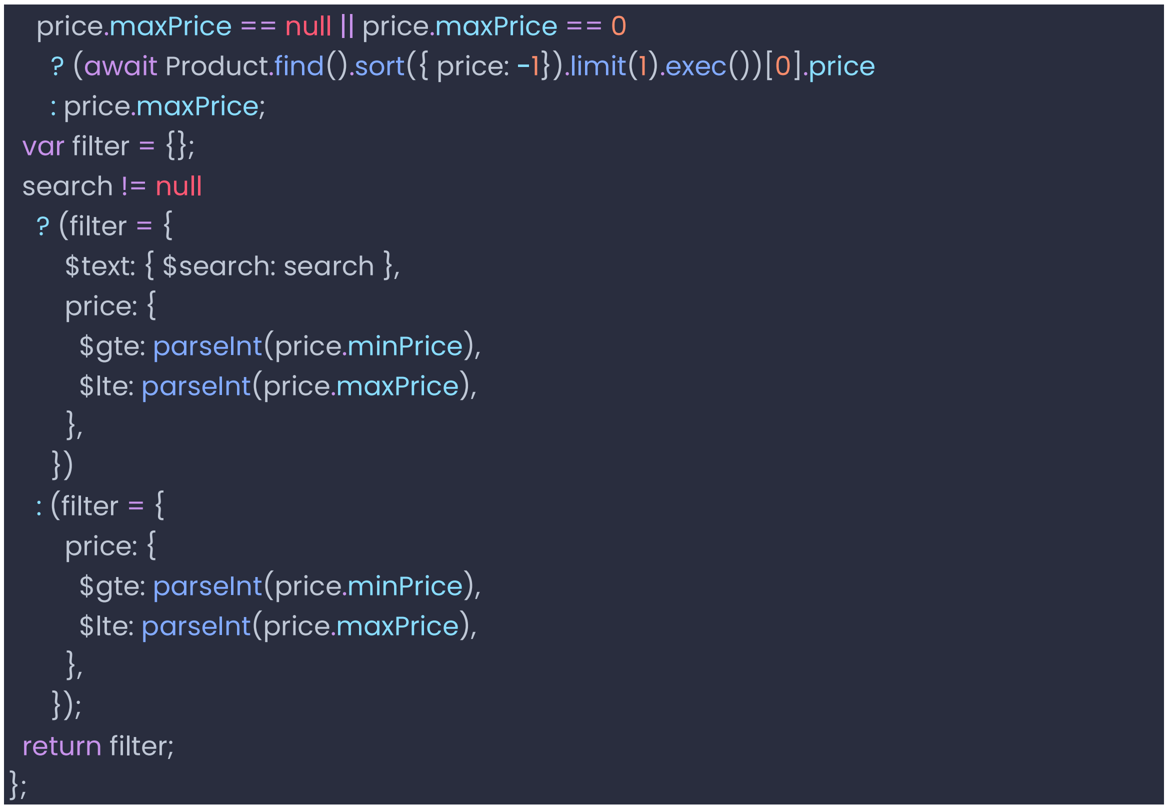
api-experiment (folder)
|_ controllers (folder)
|___ api (folder)
|______ product.controller.js (file)
|___ web (folder)
|_ models (folder)

```
|___ product.model.js (file)
|_ node_modules (folder)
|_ routes (folder)
|___ api (folder)
|______ product.routes.js (file)
|___ web (folder)
|_ tests (folder)
|___ api (folder)
|___ web (folder)
|______ images (folder)
|_ web (folder)
|___ layouts (folder)
|______ main.ejs (file)
|___ styles (folder)
|______ main.ejs (file)
|___ views (folder)
|______ error.ejs (file)
|______ index.ejs (file)
|_ .env (file)
|_ .env.example (file)
|_ .gitignore (file)
|_ app.js (file)
|_ initial_data.json (file)
|_ package-lock.json (file)
|_ package.json (file)
|_ server.js (file)
```

Figure 9 Project A File Structure

6. Go back to the file "controllers/api/product.controller.js" and copy the following code. This code is a function that will allow for creating filters for the request in the function **getProducts.** No need to export this function, it will be used to handle the request for search and price range filters to fetch product data. This code defines an asynchronous function called **getFilters** that takes two parameters, **search** and **price**.

```
const getFilters = async (search, price) => {
 price.minPrice = price.minPrice ?? 0;
 price.maxPrice =
```



```
  price.maxPrice == null || price.maxPrice == 0
    ? (await Product.find().sort({ price: -1}).limit(1).exec())[0].price
    : price.maxPrice;
 var filter = {};
 search != null
  ? (filter = {
      $text: { $search: search },
      price: {
        $gte: parseInt(price.minPrice),
        $lte: parseInt(price.maxPrice),
      },
    })
  : (filter = {
      price: {
        $gte: parseInt(price.minPrice),
        $lte: parseInt(price.maxPrice),
      },
    });
 return filter;
};
```

Figure 10 getFilters Function Code

The function first sets **price.minPrice** to 0 if it is null or undefined. It then sets **price.maxPrice** to the highest price of a product in the database if it is null or zero. If **price.maxPrice** is already set, it remains unchanged.

The function then creates a variable called **filter**, which is an object that will be used to filter a database query. If the **search** is not null, the filter includes a **$text** property that searches for **search** in a text index of the database. Additionally, the filter includes a **price** property that specifies that the price of the products returned by the query must be greater than or equal to **price.minPrice** and less than or equal to **price.maxPrice**. If the **search** is null, the filter only includes the **price** property.

Finally, the function returns the **filter** object.

7. The next step is to code the functions **getProducts** (/api/v1/products) and **getProduct** (/api/v1/product/:slug). Start coding the functions according to the following tables, knowing that the function getProducts is the function used for retrieving a list of products meanwhile the function getProduct is to fetch one product using its slug value in the request parameter.

GET "/api/v1/products" ENDPOINT STRUCTURE			
API Endpoint Path	*Request Method*	*Response Format*	*Description*
"/api/v1/products"	GET	JSON	Get a list of products.
Request Parameters			
Parameter	*Type*		*Description*
"search"	String		The search query for the product name or description.
"price"	Object		The price range filter for the products. Should be an object with **minPrice** and **maxPrice** fields.
Response Parameters			
Parameter	*Type*		*Description*
"products"	Array		An array of product objects.
"message"	String		A message indicating the status of the response.
Success Responses			
HTTP Status Code	*Response*		
200	{ "products": [{ "_id": "621578ce4f4a2e4c6406dc05", "name": "Black Jeans", "price": 49.99, "description": "A pair of black jeans with a slim fit", "slug": "black-jeans", "createdAt": "2023-02-20T07:32:14.786Z", "updatedAt": "2023-02-20T07:32:14.786Z" }], "message": "Products found" }		
Error Responses			
HTTP Status Code	*Response*		
404	{ "message": "No products found" }		
500	{ error object }		

Table 2 Endpoint GET "/api/v1/products" API Architecture Design

Note that the successful response could have more data than shown in table 2. The function **getFilters** is used in the **getProducts** function in case there is a request with the parameters of search and price. The following table explains the endpoint which retrieves one product using its slug value.

GET “/api/v1/product/:slug” ENDPOINT STRUCTURE			
API Endpoint Path	*Request Method*	*Response Format*	*Description*
“/api/v1/product/:slug”	GET	JSON	Get a product with a specific slug.
Request Parameters			
Parameter	*Type*		*Description*
“slug”	String		The slug of the product to retrieve.
Response Parameters			
Parameter	*Type*		*Description*
“product”	Object		The product object.
“message”	String		A message indicating the status of the response.
Success Responses			
HTTP Status Code	*Response*		
200	{ "product": { "_id": "621578ce4f4a2e4c6406dc05", "name": "Black Jeans", "price": 49.99, "description": "A pair of black jeans with a slim fit", "slug": "black-jeans", "createdAt": "2023-02-20T07:32:14.786Z", "updatedAt": "2023-02-20T07:32:14.786Z" }, "message": "Product found" }		
Error Responses			
HTTP Status Code	*Response*		
404	{ "message": “No product found” }		
500	{ error object }		

Table 3 Endpoint GET "/api/v1/product/:slug" API Architecture Design

Note that the response contains one product and the message is in a singular form meanwhile the previous function uses the plural form to describe the message String.

8. The following are hints that can be used to complete the two functions:
    a. The request is using the property query to access the filter search and price. The slug can be obtained from the params object in the request.

b. "const products = await Product.find().lean().exec();"

This code retrieves all products from the database using the Mongoose find() function. The lean() function is used to retrieve a plain JavaScript object instead of a Mongoose document, which can improve performance for read-heavy use cases. Finally, the exec() function is called to execute the query and return a promise containing the results.

c. "const products = await Product.find({ $and: [await getFilters(search, price)] }.lean().exec();"

This code retrieves products from the database based on the search and price filters returned by the getFilters() function. The $and operator is used to combine multiple search conditions. Similar to the first code snippet.

d. "const product = await Product.findOne({ slug: req.params.slug }).lean().exec();"

This code retrieves a single product from the database using the findOne() function, with a search condition matching the product's slug.

Running The API Application

To run the application, use the following command:

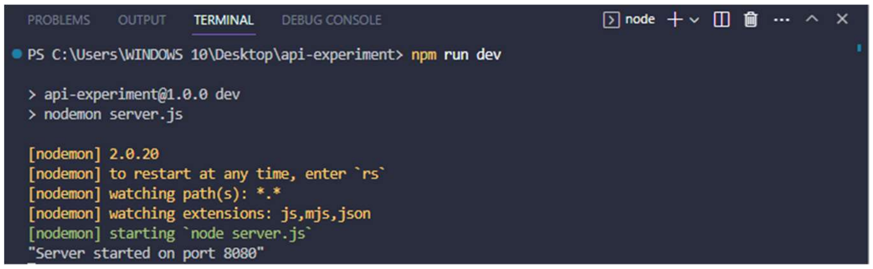
For this guide and development purposes the command "npm run dev" is used to execute the command "nodemon server.js" which will run the "server.js" using the nodemon package. This package allows the server to reload if any changes occur in the code of the application. Run the development command "npm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

Via Console and Browser

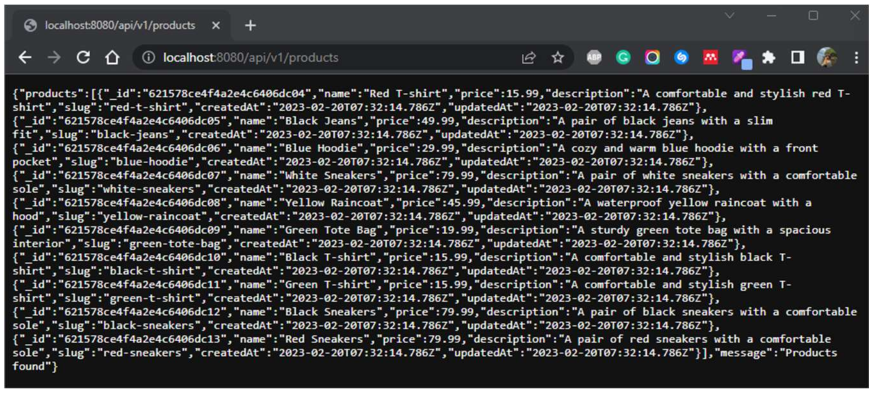
After running the application, the console in VSCode integrated terminal will show a message indicating that the server has started on port 8080 as the following:



```
PS C:\Users\WINDOWS 10\Desktop\api-experiment> npm run dev

> api-experiment@1.0.0 dev
> nodemon server.js

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
"Server started on port 8080"
```

Figure 11 Console Message After Running the Application

In the browser, open the following link http://localhost:8080/api/v1/products. The link will show the list of the products imported in this step.

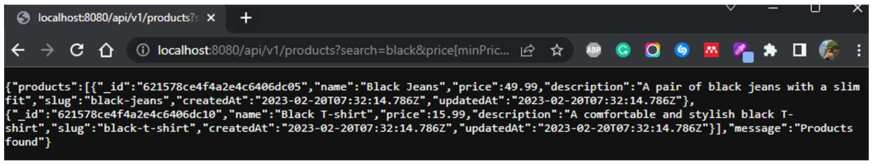

localhost:8080/api/v1/products
localhost:8080/api/v1/products

```
{"products":[{"_id":"621578ce4f4a2e4c6406dc04","name":"Red T-shirt","price":15.99,"description":"A comfortable and stylish red T-
shirt","slug":"red-t-shirt","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc05","name":"Black Jeans","price":49.99,"description":"A pair of black jeans with a slim
fit","slug":"black-jeans","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc06","name":"Blue Hoodie","price":29.99,"description":"A cozy and warm blue hoodie with a front
pocket","slug":"blue-hoodie","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc07","name":"White Sneakers","price":79.99,"description":"A pair of white sneakers with a comfortable
sole","slug":"white-sneakers","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc08","name":"Yellow Raincoat","price":45.99,"description":"A waterproof yellow raincoat with a
hood","slug":"yellow-raincoat","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc09","name":"Green Tote Bag","price":19.99,"description":"A sturdy green tote bag with a spacious
interior","slug":"green-tote-bag","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc10","name":"Black T-shirt","price":15.99,"description":"A comfortable and stylish black T-
shirt","slug":"black-t-shirt","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc11","name":"Green T-shirt","price":15.99,"description":"A comfortable and stylish green T-
shirt","slug":"green-t-shirt","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc12","name":"Black Sneakers","price":79.99,"description":"A pair of black sneakers with a comfortable
sole","slug":"black-sneakers","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc13","name":"Red Sneakers","price":79.99,"description":"A pair of red sneakers with a comfortable
sole","slug":"red-sneakers","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"}],"message":"Products
found"}
```

Figure 12 Results of Endpoint /api/v1/products List

Try also to access the following link which will return a list of products with the name black and a minimum price of 10:

http://localhost:8080/api/v1/products?search=black&price[minPrice]=10

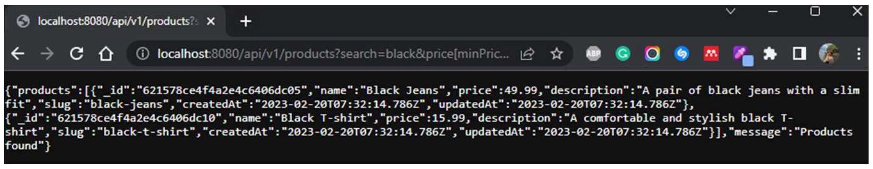

localhost:8080/api/v1/products?s
localhost:8080/api/v1/products?search=black&price[minPric...

```
{"products":[{"_id":"621578ce4f4a2e4c6406dc05","name":"Black Jeans","price":49.99,"description":"A pair of black jeans with a slim
fit","slug":"black-jeans","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc10","name":"Black T-shirt","price":15.99,"description":"A comfortable and stylish black T-
shirt","slug":"black-t-shirt","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"}],"message":"Products
found"}
```

Figure 13 Results of endpoint /api/v1/products List with filters

For the endpoint "/api/v1/product/:slug", try accessing the following link which will return one product.

http://localhost:8080/api/v1/product/red-t-shirt



localhost:8080/api/v1/products?s
localhost:8080/api/v1/products?search=black&price[minPric...

```
{"products":[{"_id":"621578ce4f4a2e4c6406dc05","name":"Black Jeans","price":49.99,"description":"A pair of black jeans with a slim
fit","slug":"black-jeans","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"},
{"_id":"621578ce4f4a2e4c6406dc10","name":"Black T-shirt","price":15.99,"description":"A comfortable and stylish black T-
shirt","slug":"black-t-shirt","createdAt":"2023-02-20T07:32:14.786Z","updatedAt":"2023-02-20T07:32:14.786Z"}],"message":"Products
found"}
```

Figure 14 Results of Endpoint /api/v1/prdouct/:slug

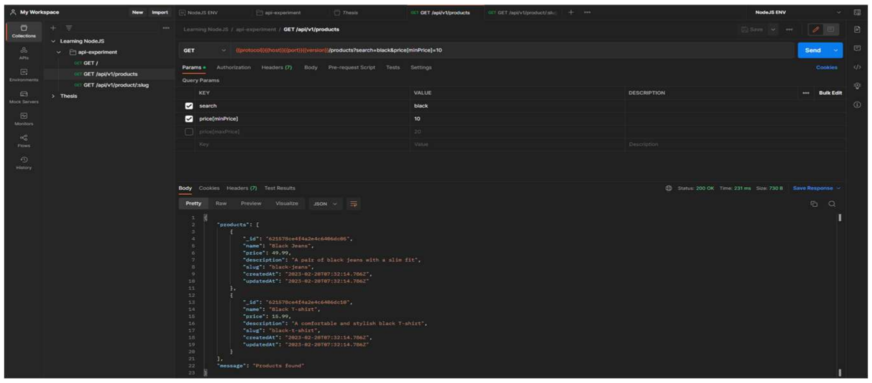
Try to access the following links, it should return a message indicating that products are not found:

1. http://localhost:8080/api/v1/products?search=black&price[minPrice]=1000
2. http://localhost:8080/api/v1/product/red-t-shirt-1

Using Postman

In the Postman app create two new requests in the folder "api-experiment" which was created in the previous guide, A01. Both requests use "GET" but with different URLs.

1. The first URL: {{protocol}}{{host}}{{port}}{{version}}/products
2. The second URL: {{protocol}}{{host}}{{port}}{{version}}/product/red-t-shirt

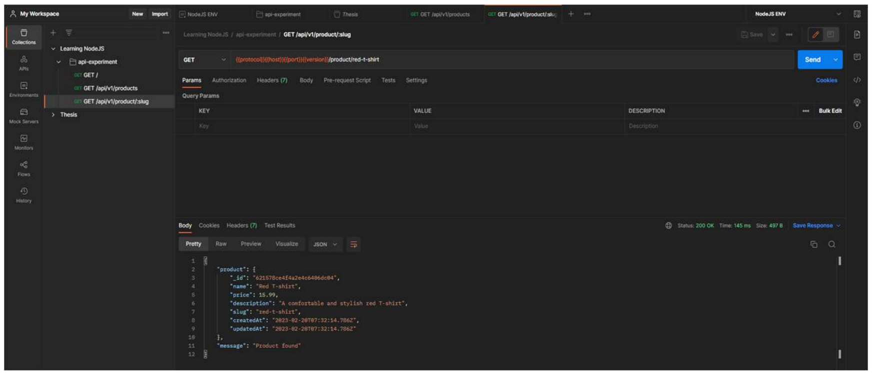


GET /api/v1/products
GET
{{protocol}}{{host}}{{port}}{{version}}/products?search=black&price[minPrice]=10
Send
Params
Query Params
KEY
VALUE
DESCRIPTION
Body
Pretty
JSON

KEY	VALUE
search	black
price[minPrice]	10
price[maxPrice]	20

```
{
    "products": [
        {
            "_id": "621578ce4f4a2e4c6406dc05",
            "name": "Black Jeans",
            "price": 49.99,
            "description": "A pair of black jeans with a slim fit",
            "slug": "black-jeans",
            "createdAt": "2023-02-20T07:32:14.786Z",
            "updatedAt": "2023-02-20T07:32:14.786Z"
        },
        {
            "_id": "621578ce4f4a2e4c6406dc10",
            "name": "Black T-shirt",
            "price": 15.99,
            "description": "A comfortable and stylish black T-shirt",
            "slug": "black-t-shirt",
            "createdAt": "2023-02-20T07:32:14.786Z",
            "updatedAt": "2023-02-20T07:32:14.786Z"
        }
    ],
    "message": "Products found"
}
```

Figure 15 Postman Results for Endpoint /api/v1/products



```
{
    "product": {
        "_id": "621578ce4f4a2e4c6406dc04",
        "name": "Red T-shirt",
        "price": 15.99,
        "description": "A comfortable and stylish red T-shirt",
        "slug": "red-t-shirt",
        "createdAt": "2023-02-20T07:32:14.786Z",
        "updatedAt": "2023-02-20T07:32:14.786Z"
    },
    "message": "Product found"
}
```

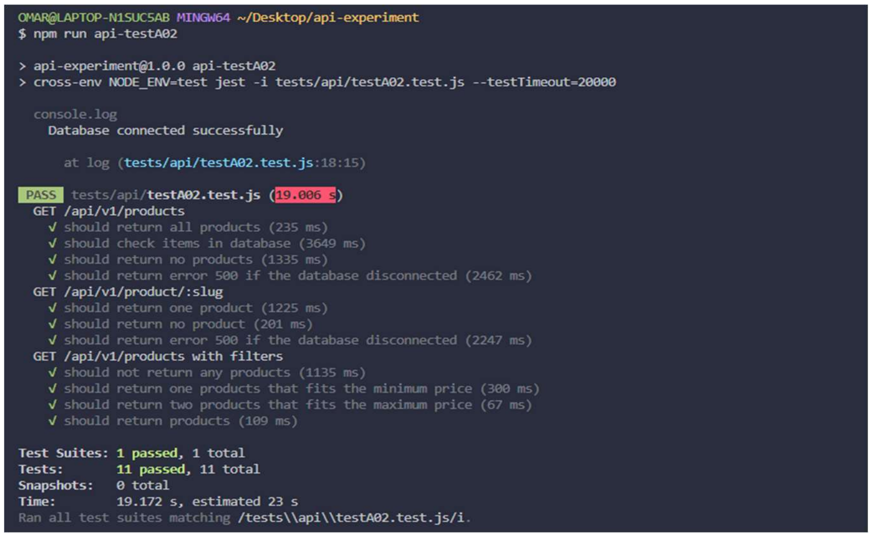
Figure 16 Postman Results for Endpoint /api/v1/product/:slug

Note that the filters can be added in the params tab for the request.

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.

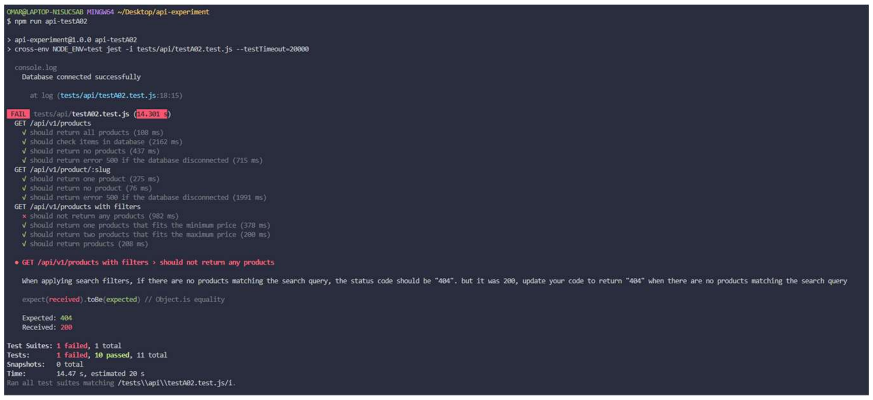
1. Copy the file "testA02.test.js" from the "api" folder within the "tests" folder for this material to the "tests/api" folder of your project base directory.
2. In the terminal run the command “npm run api-testA02” and notice the results.
3. If encounter any errors regarding mongoose doesn’t allow callbacks, 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-testA02

> api-experiment@1.0.0 api-testA02
> cross-env NODE_ENV=test jest -i tests/api/testA02.test.js --testTimeout=20000

  console.log
    Database connected successfully

      at log (tests/api/testA02.test.js:18:15)

 PASS  tests/api/testA02.test.js (19.006 s)
  GET /api/v1/products
    √ should return all products (235 ms)
    √ should check items in database (3649 ms)
    √ should return no products (1335 ms)
    √ should return error 500 if the database disconnected (2462 ms)
  GET /api/v1/product/:slug
    √ should return one product (1225 ms)
    √ should return no product (201 ms)
    √ should return error 500 if the database disconnected (2247 ms)
  GET /api/v1/products with filters
    √ should not return any products (1135 ms)
    √ should return one products that fits the minimum price (300 ms)
    √ should return two products that fits the maximum price (67 ms)
    √ should return products (109 ms)

Test Suites: 1 passed, 1 total
Tests:       11 passed, 11 total
Snapshots:   0 total
Time:        19.172 s, estimated 23 s
Ran all test suites matching /tests\\api\\testA02.test.js/i.
```

Figure 17 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-testA02

> api-experiment@1.0.0 api-testA02
> cross-env NODE_ENV=test jest -i tests/api/testA02.test.js --testTimeout=20000

  console.log
    Database connected successfully

      at log (tests/api/testA02.test.js:18:15)

 FAIL  tests/api/testA02.test.js (14.301 s)
  GET /api/v1/products
    √ should return all products (108 ms)
    √ should check items in database (2162 ms)
    √ should return no products (437 ms)
    √ should return error 500 if the database disconnected (715 ms)
  GET /api/v1/product/:slug
    √ should return one product (275 ms)
    √ should return no product (76 ms)
    √ should return error 500 if the database disconnected (1991 ms)
  GET /api/v1/products with filters
    × should not return any products (982 ms)
    √ should return one products that fits the minimum price (378 ms)
    √ should return two products that fits the maximum price (200 ms)
    √ should return products (208 ms)

  ● GET /api/v1/products with filters › should not return any products

    When applying search filters, if there are no products matching the search query, the status code should be "404". but it was 200, update your code to return "404" when there are no products matching the search query

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

    Expected: 404
    Received: 200

Test Suites: 1 failed, 1 total
Tests:       1 failed, 10 passed, 11 total
Snapshots:   0 total
Time:        14.47 s, estimated 20 s
Ran all test suites matching /tests\\api\\testA02.test.js/i.
```

Figure 18 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 new pages added to it. The same basic endpoints explained in the previous sections will be implemented in a web page that can show all the products in a table and another page to display all the details of one product.

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

1. In the "controllers/web" folder, create a file named "product.controller.js". This file will have the following functions for this material. Upon the next lessons, it will be updated.
    a. **getProducts**: similar to the GET "/api/v1/products" endpoint function but will have a different syntax for the return statement.
        i. If there are no products it should return the following statement

```
res.render("products/index", {
    title: "API-Experiment | Products", products: [],
  });
```

        ii. If the products exist, then this is what the function should return.

```
res.render("products/index", {
   title: "API-Experiment | Products",
   products: products,
   message: req.query.message,
 });
```

        iii. The "title" refers to the page tab title, "products" refers to the product data, and "message" is used for the next lesson.
    b. **getProduct**: this function will be a similar function to the GET "/api/v1/product/:slug" function but different in the return syntax.
        i. If the product was not found meaning the slug was not in the database then the function should return the following syntax.

```
const error = { status: 404, message: "No product found" };
return res.render("error", { title: "API-Experiment | Error", error });
```

        ii. If the product was found then the function should render the product details page.

```
res.render("products/details", {
  title: "API-Experiment | Product",
  product,
  message: req.query.message,
});
```

c. **getFilters**: this function is similar to the function in the api controller but it has its differences because when working with HTML inputs they return an empty string rather than just a null value. As long as the input exists with the right name, it will return an empty string. To avoid the error when filtering data, the function needs to be edited as the following to overcome this issue:

```
const getFilters = async (search, price) => {
  price.minPrice = price.minPrice == "" ? 0 : price.minPrice;
  price.maxPrice =
    price.maxPrice == "" || price.maxPrice == 0
      ? (await Product.find().sort({ price: -1 }).limit(1).exec())[0].price
      : price.maxPrice;
  var filter = {};
  search != ""
    ? (filter = {
        $text: { $search: search },
        price: {
          $gte: parseInt(price.minPrice),
          $lte: parseInt(price.maxPrice),
        },
      })
    : (filter = {
        price: {
          $gte: parseInt(price.minPrice),
          $lte: parseInt(price.maxPrice),
        },
      });
  return filter;
};
```

Figure 19 getFilters Code for The Web

d. Finally, export the two functions, **getProducts**, and **getProduct.**

2. After finishing the code in the “controllers/web/product.controller.js” file. Create a new file called "products.routes.js" in the "routes/web" folder.
3. In "products.routes.js," import the two functions from "controllers/web/product.controller.js."

4. Create a constant called "express" that requires the "express" library.
5. Create another constant called "router" that uses the "express.Router()" function.
6. Use the "router" constant to create two GET routes:
    a. The first route uses the "/" path and the getProducts function.
    b. The second route uses the path "/show/:slug" and the getProduct function.
7. Next, export the “router” constant.
8. In the “web/views” folder create a new folder called “products”. In this folder create two new files namely “index.ejs” and “details.ejs”. Create the EJS code by understanding the following table to the “web/views/products/index.ejs” file: The following is an explanation for some EJS syntax used in the code:
    - <% %> - Encloses code to be executed by EJS, such as conditional statements and loops
    - <%= %> - Used to output the value of a variable or expression in the HTML, such as the message variable or the properties of a product object
    - <% if (condition) { %> ... <% } %> - Used to create conditional statements, where the code inside the if statement is executed if the condition is true
    - <% for (let i = 0; i < products.length; i++) { %> ... <% } %> - Used to loop over an array, where the code inside the for loop is executed for each item in the array

    The following table explains the structure of the index web page for the product data using EJS code.

Element	HTML Tag	Attribute(s) / Inner Text
Alert message	<div class="alert">	If the message is defined, it is displayed using the EJS syntax <% if (typeof message !== 'undefined') { %><p class="message"><%= message %></p><% } %>
Title and Description	<div class="container flex-col-container">	<h1 class="title">Products</h1> and <p class="description">Here you can see all the products</p>
The "Create new product" button	<div class="container">	<a href="/products/create" class="btn btn-primary">Create a new product</a>
Search form	<div class="container">	<form action="/products" method="GET" class="flex-row-container"> followed by 3 input fields and a submit button. The first input is for

		text with the name "search", and the other two are for numbers with the names "price[minPrice]" and "price[maxPrice]". Each input has a label: "Search", "Minimum Price", and "Maximum Price". All the inputs have the "form-control" class, and each input with its label is wrapped in a div with a "form-group" class
Products table	<div class="container">	If the products array is not empty, it is displayed using the EJS syntax <% if (products.length > 0) { %> ... <% } %> to loop over the products array and display the details of each product in a table format
"No products" message	<div class="alert alert-warning">	If the products array is empty, it is displayed using the EJS syntax <% }else{ %><p class="message">There are no products in the database</p><% } %>

Table 4 "web/views/products/index.ejs" EJS Code Structure

9. The following is the code for the "web/views/products/details.ejs" file alongside the explanation for each line. Make sure you understand the code and then copy it to the file "web/views/products/details.ejs".

No	HTML/JS Code	Description
1.	<% if (typeof message !== 'undefined') { %>	Check if the message variable is defined or not
2.	<div class="alert">	Open a div element with a class of alert
3.	<p class="message"><%= message %></p>	Add a paragraph element inside the div with the value of the message variable
4.	</div>	Close the div element
5.	<% } %>	Close the if condition
6.	<div class="container">	Open a div element with a class of container
7.	<h1 class="title">Product details</h1>	Add a heading with a class of title
8.	<p class="description">Here you can see the details of this product</p>	Add a paragraph element with a class of description
9.	</div>	Close the div element
10.	<div class="container">	Open another div element with a class of container
11.	<div class="card">	Open a div element with a class of card
12.	<div class="card-body">	Open a div element with a class of card-body

13.	<h5 class="card-title"><%= product.name %></h5>	Add a heading element with a class of card-title and a value of product.name
14.	<h6 class="card-subtitle">$<%= product.price %></h6>	Add a subheading element with a class of card-subtitle and a value of product.price
15.	<p class="card-text"><%= product.description %></p>	Add a paragraph element with a class of card-text and a value of product.description
16.	<div class="action">	Open a div element with a class of action
17.	<a href="/products/update/<%= product.slug %>" class="btn btn-primary">Edit this product</a>	Add a hyperlink element with a class of btn btn-primary and a URL value of /products/update/ followed by the product.slug value
18.	<form action="/products/delete/<% = product.slug %>" method="POST">	Add a form element with a POST method and a URL value of /products/delete/ followed by the product.slug value
19.	<button type="submit" class="btn btn-danger">Delete this product</button>	Add a button element with a class of btn btn-danger and a text value of Delete this product
20.	</form>	Close the form element
21.	</div>	Close the div element
22.	</div>	Close the div element
23.	</div>	Close the div element
24.	</div>	Close the div element

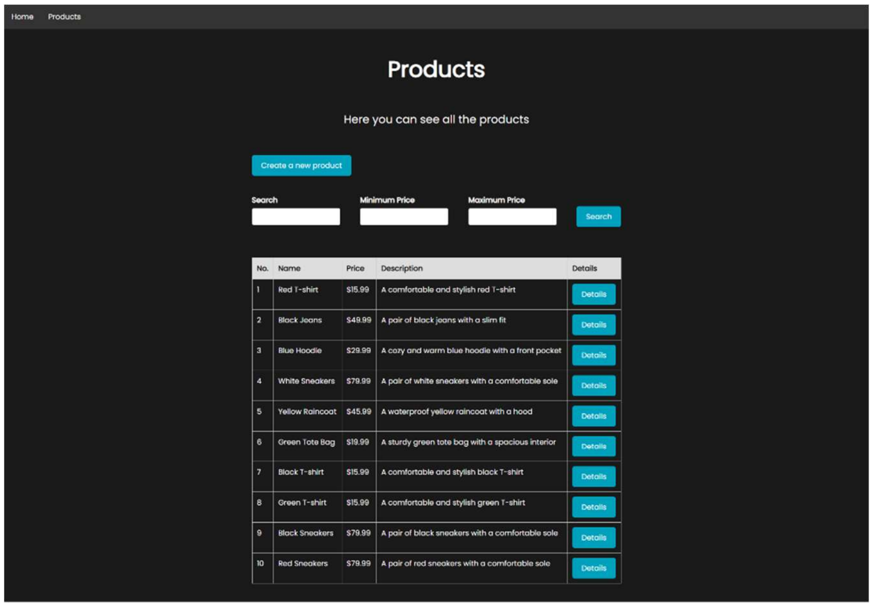
Table 5 web/views/products/details.ejs" EJS Code Structure

10. Once all the views have been set up, Open the "app.js" file.
11. Create a constant named "webProductRoutes".
12. Assign the value of the router from the "routes/web/product.routes.js" file to "webProductRoutes".
13. Add a new line to "app" const to use "webProductRoutes" at the "/products" path.
14. This will ensure that the web application is properly set up to use the views created in the previous steps.

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, the page should look similar to the following image:

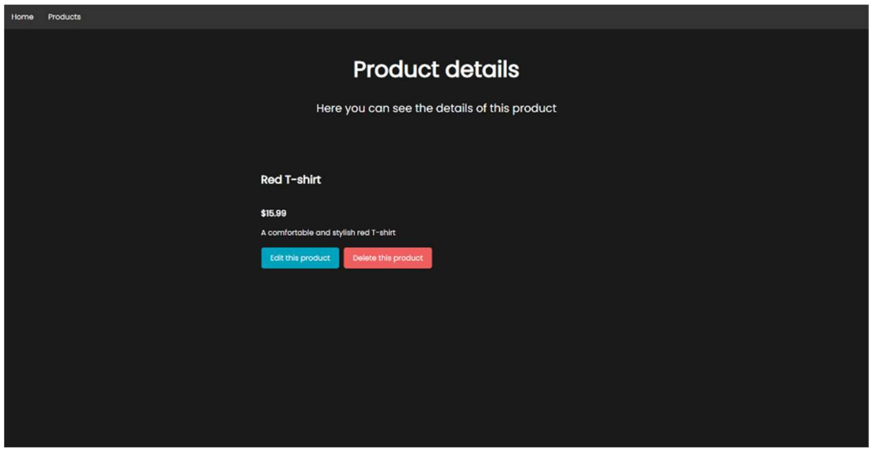


Products

Here you can see all the products

No.	Name	Price	Description	Details
1	Red T-shirt	$15.99	A comfortable and stylish red T-shirt	Details
2	Black Jeans	$49.99	A pair of black jeans with a slim fit	Details
3	Blue Hoodie	$29.99	A cozy and warm blue hoodie with a front pocket	Details
4	White Sneakers	$79.99	A pair of white sneakers with a comfortable sole	Details
5	Yellow Raincoat	$45.99	A waterproof yellow raincoat with a hood	Details
6	Green Tote Bag	$19.99	A sturdy green tote bag with a spacious interior	Details
7	Black T-shirt	$15.99	A comfortable and stylish black T-shirt	Details
8	Green T-shirt	$15.99	A comfortable and stylish green T-shirt	Details
9	Black Sneakers	$79.99	A pair of black sneakers with a comfortable sole	Details
10	Red Sneakers	$79.99	A pair of red sneakers with a comfortable sole	Details

Figure 20 Products Page



Product details

Here you can see the details of this product

Red T-shirt

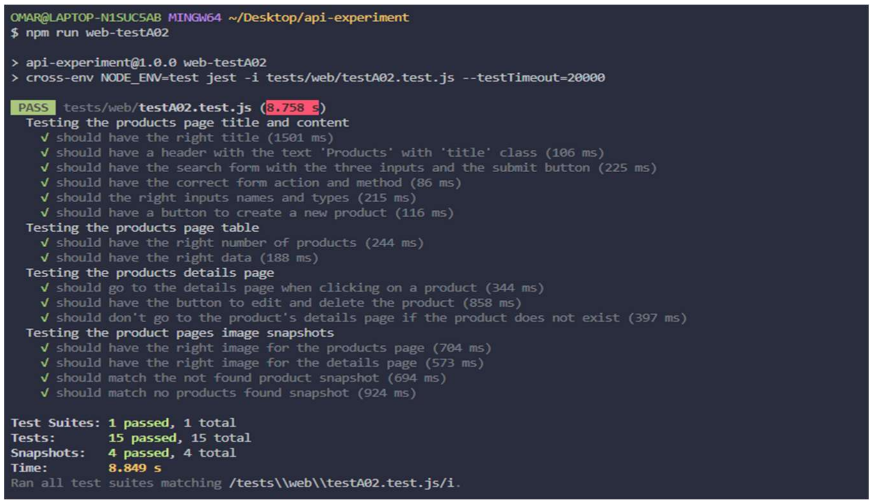
$15.99

A comfortable and stylish red T-shirt

Figure 21 Product Details Page

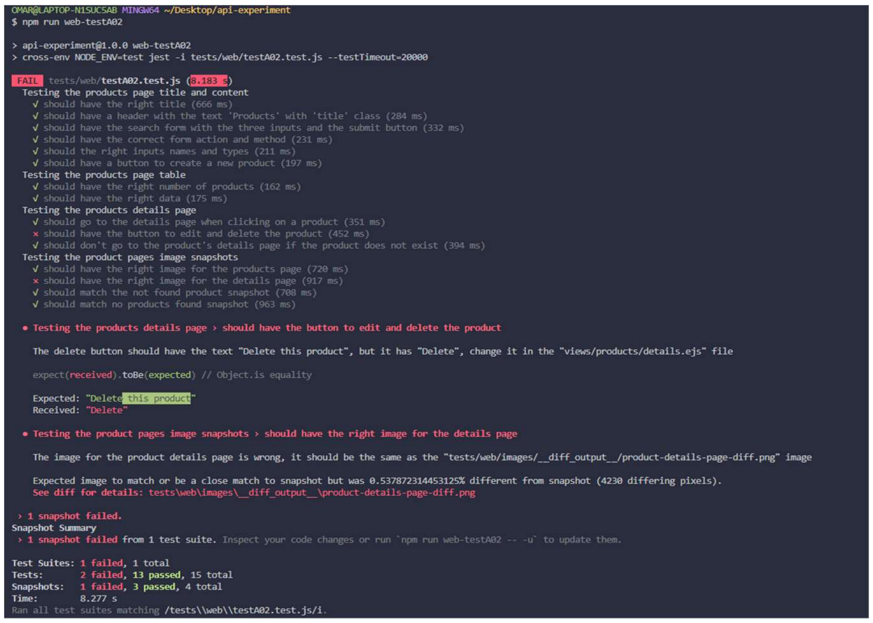
To test the application, copy the file “testA02.test.js” from the “/tests/web” folder and paste it to the folder “/tests/web” in your project directory. After that, run the command “npm run web-testA02” and notice the results.

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-testA02

> api-experiment@1.0.0 web-testA02
> cross-env NODE_ENV=test jest -i tests/web/testA02.test.js --testTimeout=20000

 PASS  tests/web/testA02.test.js (8.758 s)
  Testing the products page title and content
    √ should have the right title (1501 ms)
    √ should have a header with the text 'Products' with 'title' class (106 ms)
    √ should have the search form with the three inputs and the submit button (225 ms)
    √ should have the correct form action and method (86 ms)
    √ should the right inputs names and types (215 ms)
    √ should have a button to create a new product (116 ms)
  Testing the products page table
    √ should have the right number of products (244 ms)
    √ should have the right data (188 ms)
  Testing the products details page
    √ should go to the details page when clicking on a product (344 ms)
    √ should have the button to edit and delete the product (858 ms)
    √ should don't go to the product's details page if the product does not exist (397 ms)
  Testing the product pages image snapshots
    √ should have the right image for the products page (704 ms)
    √ should have the right image for the details page (573 ms)
    √ should match the not found product snapshot (694 ms)
    √ should match no products found snapshot (924 ms)

Test Suites: 1 passed, 1 total
Tests:       15 passed, 15 total
Snapshots:   4 passed, 4 total
Time:        8.849 s
Ran all test suites matching /tests\\web\\testA02.test.js/i.
```

Figure 22 Successful Web Test Results



```
OMAR@LAPTOP-N1SUC5AB MINGW64 ~/Desktop/api-experiment
$ npm run web-testA02

> api-experiment@1.0.0 web-testA02
> cross-env NODE_ENV=test jest -i tests/web/testA02.test.js --testTimeout=20000

 FAIL  tests/web/testA02.test.js (8.183 s)
  Testing the products page title and content
    √ should have the right title (666 ms)
    √ should have a header with the text 'Products' with 'title' class (284 ms)
    √ should have the search form with the three inputs and the submit button (332 ms)
    √ should have the correct form action and method (231 ms)
    √ should the right inputs names and types (211 ms)
    √ should have a button to create a new product (197 ms)
  Testing the products page table
    √ should have the right number of products (162 ms)
    √ should have the right data (175 ms)
  Testing the products details page
    √ should go to the details page when clicking on a product (351 ms)
    × should have the button to edit and delete the product (452 ms)
    √ should don't go to the product's details page if the product does not exist (394 ms)
  Testing the product pages image snapshots
    √ should have the right image for the products page (720 ms)
    × should have the right image for the details page (917 ms)
    √ should match the not found product snapshot (708 ms)
    √ should match no products found snapshot (963 ms)

  ● Testing the products details page › should have the button to edit and delete the product

    The delete button should have the text "Delete this product", but it has "Delete", change it in the "views/products/details.ejs" file

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

    Expected: "Delete this product"
    Received: "Delete"

  ● Testing the product pages image snapshots › should have the right image for the details page

    The image for the product details page is wrong, it should be the same as the "tests/web/images/__diff_output__/product-details-page-diff.png" image

    Expected image to match or be a close match to snapshot but was 0.537872314453125% different from snapshot (4230 differing pixels).
    See diff for details: tests\web\images\__diff_output__\product-details-page-diff.png

 › 1 snapshot failed.
Snapshot Summary
 › 1 snapshot failed from 1 test suite. Inspect your code changes or run `npm run web-testA02 -- -u` to update them.

Test Suites: 1 failed, 1 total
Tests:       2 failed, 13 passed, 15 total
Snapshots:   1 failed, 3 passed, 4 total
Time:        8.277 s
Ran all test suites matching /tests\\web\\testA02.test.js/i.
```

Figure 23 Failed Web Test Results

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

Results

After completing this learning material, students will have acquired the skills to comprehend the basics of RESTful API and implement them in practical applications using NodeJS. Students will have a firm grasp of the GET method and its role in fetching data from a MongoDB collection. Additionally, students will be able to create a data model, object, or blueprint that can be used to manage different types of data, complete with a range of attributes.