Building a Simple Data Dashboard with HTML, JS (Express), and MongoDB

Setup and Sample Data Preparation

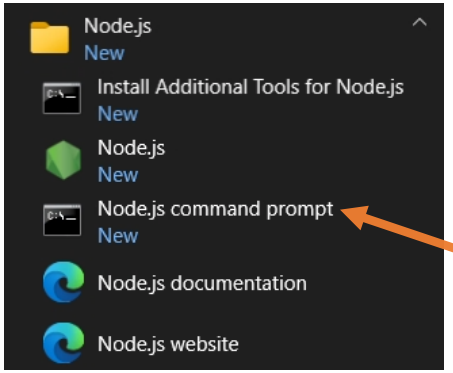
In either your local MongoDB server or MongoDB Atlas (cloud):

- **Create** a database named **Shop**.
- **Set up** a collection named **products** (or use an existing one).
- **Download** the file **`products.json`** from Canvas.
  - It contains data for **100 random products** with fields such as:
    `_id, product_name, category, price, stock, and sales_count`.
- **Import** the JSON file into your products collection.
  - If successful, you will have **100 product documents** added to your collection.

Building a Simple Data Dashboard with HTML, JS (Express), and MongoDB

Setting up Node JS project

1. Create a folder named LAB 7.



2. Open **Node.js Command Prompt** from your Start Menu.

3. Navigate to your project folder (i.e., LAB 7), e.g.:

```
cd C:\Users\student\Documents\LAB 7
```

4. Next, run the following commands one after another:

```
npm init -y
```

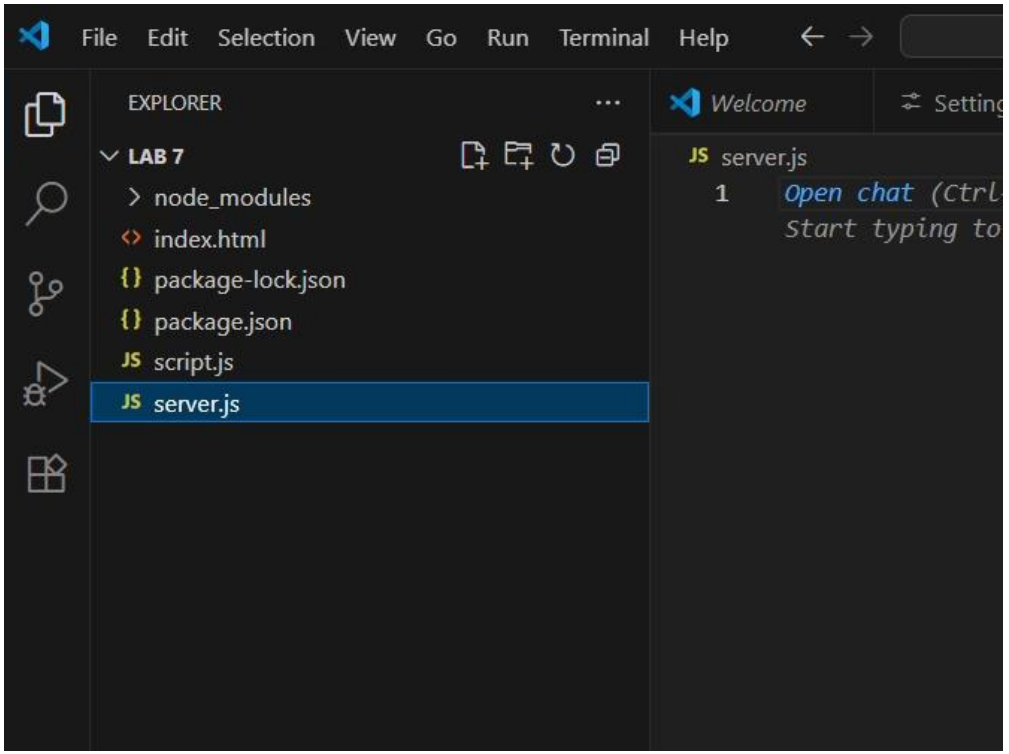
```
npm install express mongoose cors nodemon
```

Connecting to MongoDB using VS code

1. Open the **LAB 7** folder using **Visual Studio Code**.

2. Inside the Lab 7 folder, create the following files:
   - **`server.js`**
   - **`index.html`**
   - **`script.js`**

After creating the files, your VS Code Explorer should look similar to this:



Connecting to MongoDB using VS code

Open the **`server.js`** file and type the following code.

```
JS server.js > ...
1   const express = require("express");
2   const mongoose = require("mongoose");
3   const cors = require("cors");
4
5   const app = express();
6   app.use(express.json());
7   app.use(cors());
8   app.use(express.static(__dirname)); // serve current directory
9
10  const connectionString="mongodb+srv://<USER_NAME>:<PASSWORD>@cluster0.zvdyokj.mongodb.net/Shop?retryWrites=true&w=majority";
11  // Connect to MongoDB Atlas
12  mongoose.connect(connectionString, {
13    family: 4,
14    tls: true
15  })
16  .then(() => console.log(" Connected to MongoDB database"))
17  .catch(err => console.error(" MongoDB connection error:", err));
18
19  // Start server
20  const PORT = 3000;
21  app.listen(PORT, () => console.log(`Server running at http://localhost:${PORT}`));
```

- For future projects, you can reuse lines 1–7 — these are standard setup lines for most Node.js + Express applications.
  - Each line serves an important purpose — do some quick research to understand why they are needed!

- **Line 8**: This line tells the program that the **`index.html`** file is located in the current folder and should be served from here.
  - If your **`index.html`** is inside a subfolder, replace **`__dirname`** with the path to that folder.

Connecting to MongoDB using VS code

Open the **`server.js`** file and type the following code.

```
JS server.js > ...
1   const express = require("express");
2   const mongoose = require("mongoose");
3   const cors = require("cors");
4
5   const app = express();
6   app.use(express.json());
7   app.use(cors());
8   app.use(express.static(__dirname)); // serve current directory
9
10  const connectionString="mongodb+srv://<USER_NAME>:<PASSWORD>@cluster0.zvdyokj.mongodb.net/Shop?retryWrites=true&w=majority";
11  // Connect to MongoDB Atlas
12  mongoose.connect(connectionString, {
13    family: 4,
14    tls: true
15  })
16  .then(() => console.log(" Connected to MongoDB database"))
17  .catch(err => console.error(" MongoDB connection error:", err));
18
19  // Start server
20  const PORT = 3000;
21  app.listen(PORT, () => console.log(`Server running at http://localhost:${PORT}`));
```

- **Line 10:** very important. Here, you are defining the **connection string** (or link) to your **MongoDB database**. In this example, the connection string points to a MongoDB Atlas cluster.
  - You must replace **<USER_NAME>** and **<PASSWORD>** with your own Atlas database username and password.
  - Notice that the database name **Shop** is included at the end of the connection string (after cluster0.zvdyokj.mongodb.net/Shop)
- If using a local MongoDB server, change the connection string to:**"mongodb://localhost:27017/Shop"** and comment out line 14 **`//tls:true`**

To run your code, in the Node.js Command Prompt, type: `node server`

This runs the code once. After any change, you must rerun this command.

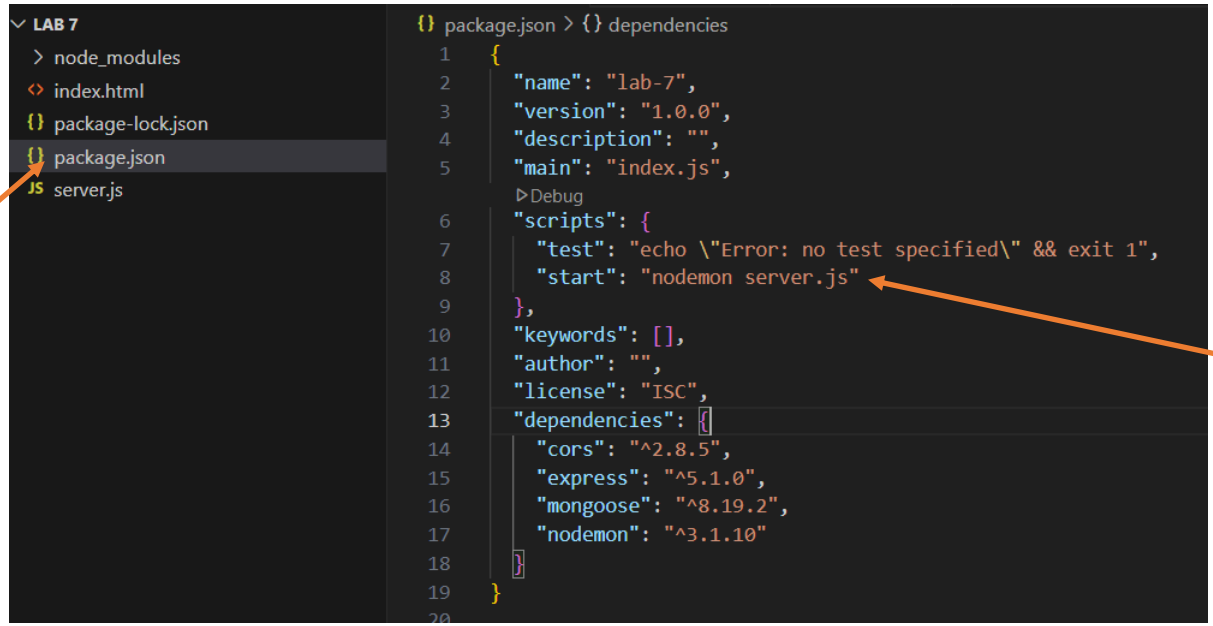
To **auto-run** the server after every save:

- Open **package.json**
- Edit the **"start"** script as shown below
- Save the file

Then start your server with: `npm start`

Now, your server will automatically restart whenever you save changes in **server.js**.

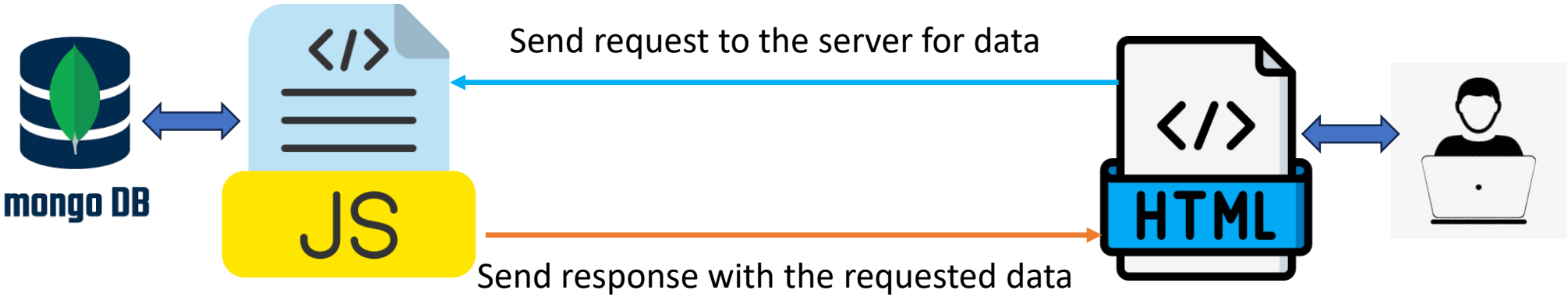
package.json file



Add this line **"start": "nodemon server.js"**

Reading from MongoDB and Displaying on a Webpage

To stay organised, we'll separate the **Front-End** and **Back-End** tasks.
Once the concept becomes clear, this approach will help us write code in a more **structured and efficient** way.



Back-End
(**server.js** in this example)

Front-End
(**index.html** in this example)

✅ Communicate with the database
✅ Execute queries to perform CRUD operation

❌ No direct interaction with the users
❌ Don't deal with how the data will be presented to the user

✅ Provide user interface to interact with the users
✅ Deal with the representation of data on the website

❌ No direct interaction with the database

Reading from MongoDB and Displaying on a Webpage

First, we'll write the necessary code in **server.js (Backend)** to read data from the connected MongoDB database.

Before we begin coding, let's **revisit the MongoDB document structure** to understand the data we're working with.

```
{
  "_id": "PID001",
  "product_name": "Keyboard",
  "category": "Home Appliances",
  "price": 1584.4,
  "stock": 120,
  "sales_count": 804
},
```

1. **Create a document schema** that matches the exact structure of your MongoDB documents.

```
// Define Product schema
const productSchema = new mongoose.Schema({
  _id:String,
  product_name: String,
  category: String,
  price: Number,
  stock: Number,
  sales_count: Number
});

// Create model
const Product = mongoose.model('Product', productSchema);
```

2. Create the document model class called **Product** for the earlier defined document schema

By default, Mongoose looks for (or creates) a collection named products (lowercase, plural form of "Product").

This model acts as your **gateway** to the collection — allowing you to **create, read, update, and delete** documents.

If your MongoDB collection is named **`product`** (singular), Mongoose won't find it and might even create a new collection called **`products`**. To explicitly specify the collection name, pass it as a third argument when defining the model (see below).

```
const Product = mongoose.model('Product', productSchema, 'product');
```

Reading from MongoDB and Displaying on a Webpage

```
29   // Create model
30   const Product = mongoose.model('Product', productSchema);
```

Given that we have created document model class called **Product,** this model class lets you perform Create, Read, Update, and Delete (CRUD) operations on the MongoDB collection**.**

To read all products

```
const products = await Product.find();
console.log(products);//to print the output
```

The keyword `await` is used to **pause execution** until a **Promise** is resolved — do some research to understand what this means!

Note: `Await` only works inside an `async` function. So, the above code must be inside an `async` function like:

```
async function getProducts() {
  const products = await Product.find();
  console.log(products);
}

getProducts();
```

If you prefer **not to use** `await` and `async`, you can use the `.then()` syntax instead:

```
Product.find()
  .then(products => console.log(products))
  .catch(err => console.error(err));
```

Reading from MongoDB and Displaying on a Webpage

Assuming that we are writing code inside an async function,

To read all products where category is **"Electronics"**

```
const products = await Product.find({ category: "Electronics" });
```

Read all products where category is **"Electronics"** and sort by **price ascending**

```
const products = await Product.find({ category: "Electronics" }).sort({
price: 1 });
```

Read all products where category is **"Accessories"** and show only **product_name** and **price**

```
const products = await Product.find(
  { category: "Accessories" },  { product_name: 1, price: 1, _id: 0 }
);
```

Same as above, but show only **first 5 results**

```
const products = await Product.find(
  { category: "Accessories" },  { product_name: 1, price: 1, _id:0}
).limit(5);
```

Read all products where **price < 200** and **stock > 100**

```
const products = await Product.find({price: { $lt: 200 },stock: { $gt: 100 }}
);
```

Reading from MongoDB and Displaying on a Webpage

Now, type the following lines of code in your **`server.js`** file.

```
33  // API route to get all products
34  app.get('/products', async (req, res) => {
35    try {
36      const products = await Product.find();
37      res.json(products); // send JSON to frontend
38    } catch (err) {
39      res.status(500).send(err);
40    }
41  });
```

`req` means request
`res` means response

Purpose: Exposes an API route `/products` to fetch all products from the database.
Usage: Frontend can call this API to dynamically display product data (e.g., in a table).

Line 34: Defines an HTTP GET route `/products` using Express. When this endpoint is called, the server executes the function.
Line 36: Uses the Mongoose model **Product** to query all records (`Product.find()`). You can modify this query to get different results.
Line 37: Sends the retrieved data back to the client (frontend) in JSON format, which is easy for the frontend (e.g., HTML/JS) to process.
Lines 38-39: Catches any errors (e.g., database failure) and responds with HTTP 500 — indicating a server-side error.

Reading from MongoDB and Displaying on a Webpage

Copy content from **`index.html`** (available on canvas) to your project's **`index.html`** file.
Run **`server.js`** and open your browser and go to the following project URL

http://localhost:3000/

What You Should See:
A table displaying **all products** from the MongoDB **`products`** collection.

Understanding the Code:

- In **index.html**, check the <script> tag (lines 36–53).
- **Line 36:** Calls the backend API **`/products`**.
- **Line 40:** Iterates through each product document.
- **Lines 43–48**: Accesses product attributes → populates table rows.

```
34  <script>
35    // Fetch products from server and display in table
36    fetch('/products')
37      .then(res => res.json())
38      .then(products => {
39        const tbody = document.querySelector('#productTable tbody');
40        products.forEach(p => {
41          const row = document.createElement('tr');
42          row.innerHTML = `
43            <td>${p._id}</td>
44            <td>${p.product_name}</td>
45            <td>${p.category}</td>
46            <td>${p.price}</td>
47            <td>${p.stock}</td>
48            <td>${p.sales_count}</td>
49          `;
50          tbody.appendChild(row);
51        });
52      })
53      .catch(err => console.error(err));
54  </script>
```

In **`server.js`**, replace **`Product.find()`** with other queries (see page 10). Observe how results change in the table. It should give you good understanding of how you can build and run queries to get data from the MongoDB using Javasrcipt.

Building the Dashboard Structure in HTML

- You have learned how to:
  - Connect to a MongoDB database using JavaScript
  - Run **hard-coded queries** in the database
  - Display retrieved data in a table

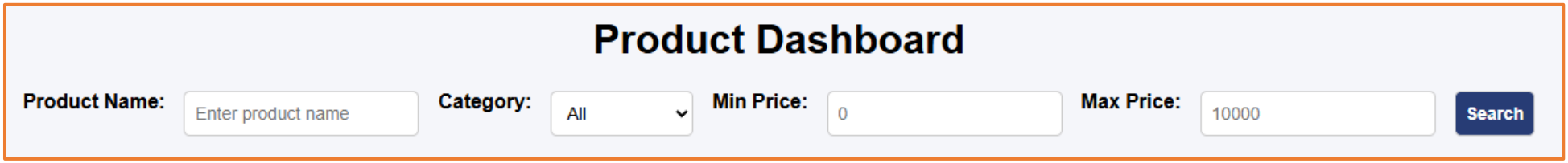
Limitations so far:

- **No user inputs** are handled

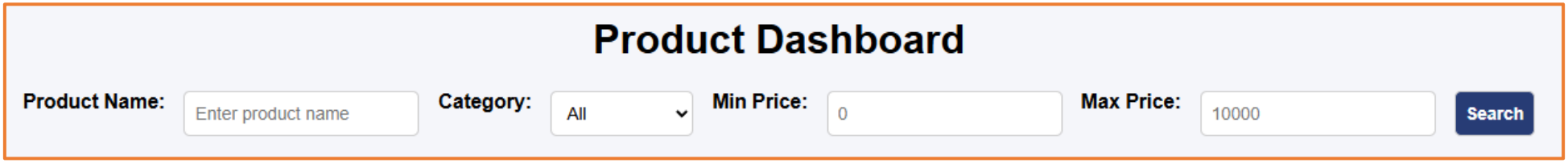
- Now, you will learn how to:
  - Capture **user input** from different input types: textbox, dropdown list, radio button, etc.
  - **Create dynamic queries** based on user input
  - **Execute dynamic queries** to retrieve desired data from MongoDB
  - **Dynamically update** the table with retrieved data

Within the **index.html** file, design an HTML form for querying product data with the following fields:

- **Product name**: Text input
- **Category**: Dropdown list (e.g., Electronics, Accessories).
- **Price Range:** Two number inputs → Minimum Price and Maximum Price
- **Search Button:** Triggers the query when clicked



Building the Dashboard Structure in HTML



- When the Search Button is Clicked, it triggers a JavaScript function called **`loadProducts()`**. This behaviour can be achieved by modelling the button in the following way.

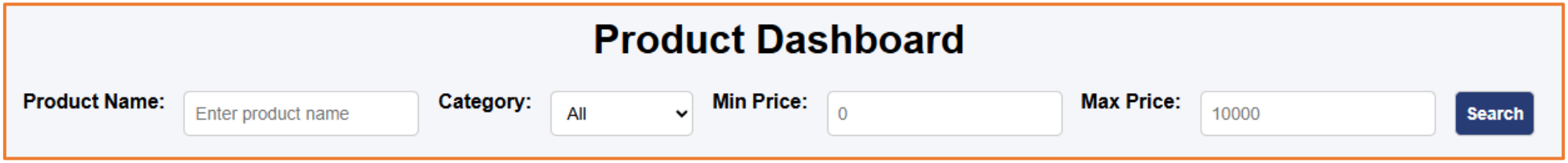
```
<button onclick="loadProducts()">Search</button>
```

- The **`loadProducts()`** function can be written as below:
  - Remember these are JavaScript functions, so in the HTML file, you need to write them within the <script> tag

```
function loadProducts() {
    const queryString = buildQueryString();//build the query string based on user input
    fetchProducts(queryString);//read the data based on the query string and show in a table
};
```

- As seen above, the `loadProducts()` function calls two other functions:
  - `buildQueryString()` → Builds the query string, `queryString`, based on user input
  - `fetchProducts(queryString)` → Sends the query to the backend API and retrieves data from MongoDB.

Building the Query String Based on User Input



- Complete the partially written `buildQueryString()` function as shown in the following script.
- Part 1 (Before `const params = new URLSearchParams()`): Retrieve **user inputs** from the HTML form and store them in variables. Example: product name, category, min price, max price.
- Part 2 (After `const params = new URLSearchParams()`): **Append** the user input values to the `params` object using `params.append()`.

```
function buildQueryString() {

      const name = document.getElementById("productName").value.trim(); //getting
value from the Product Name field
      //write code here to get values form other fields (follow the above pattern)

      const params = new URLSearchParams();

      if (name) params.append("product_name", name); //if the user provided input
in the name field, it is added to the URL

      //Write code here to append the remaining input to the URL (follow the above
pattern)


      return params.toString() ? `?${params.toString()}` : "";
    }
```

Fetching data dynamically from MongoDB and displaying in a Table:

Previously the frontend called the backend **`/products`** endpoint with no query string (see page 12). Now the frontend builds a query string to send to the backend, so update that part of code with the following code. You should notice that now we have written the code within the `fetchProducts(queryString = "")` function. The code is not much different from before. Carefully analyse the code in lines 87-96.

```
85  // Function to fetch and display products
86  async function fetchProducts(queryString = "") {
87    const res = await fetch(`/products${queryString}`);
88    const products = await res.json();
89
90    const tbody = document.querySelector("#productTable tbody");
91    tbody.innerHTML = ""; // clear old rows
92
93    if (products.length === 0) {
94      tbody.innerHTML = `<tr><td colspan="6">No products found</td></tr>`;
95      return;
96    }
97
98    products.forEach(p => {
99      const row = document.createElement("tr");
100     row.innerHTML = `
101       <td>${p._id}</td>
102       <td>${p.product_name}</td>
103       <td>${p.category}</td>
104       <td>${p.price}</td>
105       <td>${p.stock}</td>
106       <td>${p.sales_count}</td>
107     `;
108     tbody.appendChild(row);
109   });
110 }
```

Building query based on the query string and fetching data dynamically from MongoDB:

Now at the backend (in **server.js**), the query string sent from the frontend must be processed to build a query to run. To do that we need to update the code shown in page 11. The updated code is given below.

Line 36: initiated the query variable with an empty document; **Line 38**: extracts the query parameters (product_name, category, minPrice, and maxPrice) sent from the client's request URL so they can be used to build a database query. **Line 39**: adds a case-insensitive regular expression filter to the MongoDB query so that product names containing the given text (partial match) are retrieved; **Line 40**: If a category is provided in the request, this line filters products to include only those matching that category exactly; **Lines 41-44**: After checking whether either a minimum or maximum price filter has been provided by the user, appropriate filters are added to price parameter of the query. Finally, **line 47** runs the dynamically generated query.

```
33  // API route to get all products
34  app.get('/products', async (req, res) => {
35      try {
36          const query = {};
37
38          const { product_name, category, minPrice, maxPrice } = req.query;
39          if (product_name) query.product_name = { $regex: product_name, $options: 'i' };
40          if (category) query.category = category;
41          if (minPrice || maxPrice) {
42            query.price = {};
43            if (minPrice) query.price.$gte = Number(minPrice);
44            if (maxPrice) query.price.$lte = Number(maxPrice);
45          }
46
47          const products = await Product.find(query);
48          res.json(products);
49        } catch (err) {
50          res.status(500).send(err);
51        }
52  });
```

Sample output 1:

Category was selected as '**Home Appliances**'

Product Dashboard

Product Name: Enter product name **Category:** Home Appliances **Min Price:** 0 **Max Price:** 10000 **Search**

_id	Product Name	Category	Price	Stock	Sales Count
PID012	Printer	Home Appliances	1371.6	159	549
PID013	Speaker	Home Appliances	951.01	45	152
PID039	Tablet	Home Appliances	1607.1	200	504
PID042	Headphones	Home Appliances	1196.27	154	2
PID049	Laptop	Home Appliances	1791.1	181	350
PID069	Headphones	Home Appliances	399.97	99	577
PID072	Tablet	Home Appliances	1740.03	150	758
PID079	Monitor	Home Appliances	1406.61	111	856
PID085	Laptop	Home Appliances	534.65	95	356
PID099	Laptop	Home Appliances	628.34	142	3

Sample output 2:

Category was selected as '**Electronics**' and Min Price was set to **800**

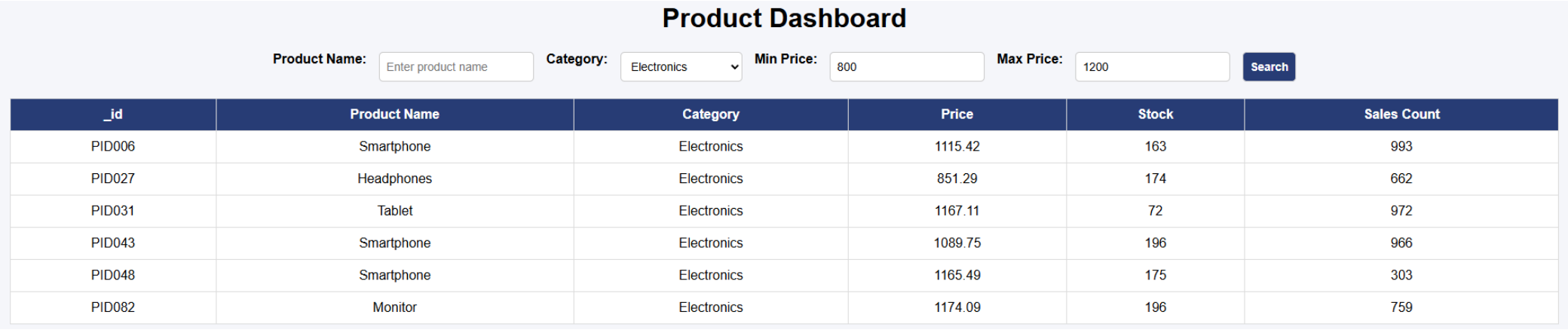
Product Dashboard

Product Name: Enter product name **Category:** Electronics **Min Price:** 800 **Max Price:** 10000 **Search**

_id	Product Name	Category	Price	Stock	Sales Count
PID006	Smartphone	Electronics	1115.42	163	993
PID016	Laptop	Electronics	1522.96	77	727
PID027	Headphones	Electronics	851.29	174	662
PID031	Tablet	Electronics	1167.11	72	972
PID043	Smartphone	Electronics	1089.75	196	966
PID048	Smartphone	Electronics	1165.49	175	303
PID050	Monitor	Electronics	1742.93	127	787
PID051	Monitor	Electronics	1896.79	124	164
PID064	Headphones	Electronics	1767.42	26	404
PID068	Printer	Electronics	1496.66	11	284
PID082	Monitor	Electronics	1174.09	196	759

Sample output 3:

Category was selected as '**Electronics**' and Min and Max Price were set to **800** and **1200** respectively



Product Dashboard

Product Name: Enter product name Category: Electronics Min Price: 800 Max Price: 1200 Search

_id	Product Name	Category	Price	Stock	Sales Count
PID006	Smartphone	Electronics	1115.42	163	993
PID027	Headphones	Electronics	851.29	174	662
PID031	Tablet	Electronics	1167.11	72	972
PID043	Smartphone	Electronics	1089.75	196	966
PID048	Smartphone	Electronics	1165.49	175	303
PID082	Monitor	Electronics	1174.09	196	759

Sample output 4:

Product name was set to '**Laptop**' and Category was set to '**All**'

Product Dashboard

Product Name: Laptop **Category:** All **Min Price:** 0 **Max Price:** 10000 **Search**

_id	Product Name	Category	Price	Stock	Sales Count
PID016	Laptop	Electronics	1522.96	77	727
PID021	Laptop	Mobile Devices	1769.19	37	339
PID033	Laptop	Accessories	600.82	182	508
PID037	Laptop	Mobile Devices	1143.55	69	756
PID040	Laptop	Accessories	1690.75	124	559
PID049	Laptop	Home Appliances	1791.1	181	350
PID054	Laptop	Accessories	1484.56	45	872
PID057	Laptop	Computers	1926.23	43	178
PID062	Laptop	Accessories	567.45	190	685
PID075	Laptop	Mobile Devices	281.08	81	164
PID085	Laptop	Home Appliances	534.65	95	356
PID087	Laptop	Accessories	671.81	76	821
PID096	Laptop	Electronics	323.95	130	925
PID099	Laptop	Home Appliances	628.34	142	3

Can you add a **drop-down list** like the following to allow the user to choose the sorting order (ascending or descending) based on **price** for the query results?