ECOM sneaker website

Repository: https://github.com/ViktorVelizarov/simple-ecom

Vercel deployment: https://simple-ecom-fx5ouffgk-viktorvelizarovs-projects.vercel.app/
(The deployed version of the app can't receive the product data from the Database so it doesn't really work. That's something I couldn't fix in time)

Technologies/Tools used:

Front-end: React(as part of NextJS), TailwindCSS
Back-end: Node(as part of NextJS), MongoDB, Mongoose ORM, Postman

Why?

NextJS - I chose to use NextJS over plain React and Node/Express because it offers very good built-in features and a simplified development process. Some examples of these features are Server-Side Rendering (SSR) and Static Site Generation (SSG), File-based Routing, API Routes, and TypeScript Support.
TailwindCSS - I chose to use Tailwind for styling because It simplifies the styling process and ensures consistency across the application by providing pre-designed components and utility classes.
MongoDB - I chose to use MongoDB for a Database because the document-based model allows for easy integration with JavaScript-based stacks like Node.js.Also, the Mongoose ORM makes it so much easier to work with the data from your app by using prebuild functions, instead of SQL queries.
Postman - I used Postman to test my API endpoints during development.

Back-End

Database:
My Database structure consists of 2 tables - sneakers and carts
The sneakers table holds all the products and has the following fields:

QUERY RESULTS: 1-20 OF MANY

```
_id: ObjectId('65fc8363a640d9500ea26e06')
title: "Nike Air Max Dn"
description: "Say hello to the next generation of Air technology. The Air Max Dn is …"
picture: "https://static.nike.com/a/images/t_PDP_1728_v1/f_auto,q_auto:eco/6b1eb…"
price: "169.99"
color: "Black"
type: "Men's shoes"
category: "Lifestyle"
sale: "no"
```

And this is how the Sneakers table translates into a Model in my app using Mongoose ORM:

```
models > JS sneaker.js > ...
1  import mongoose, { Schema } from "mongoose";
2
3  const sneakerSchema = new Schema(
4    {
5      title: String,
6      description: String,
7      picture: String,
8      price: String,
9      color: String,
10     type: String,
11     category: String,
12     sale: String,
13   },
14 );
15 const Sneaker = mongoose.models.Sneaker || mongoose.model("Sneaker", sneakerSchema);
16 export default Sneaker;
17
```

The cart table holds all the carts, which in this case is only 1 cart since the app doesn't have a user system and doesn't require a different cart object for each user(but can be a good update for the future). The cart table holds only the IDs of the sneakers inside of the cart:

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId('65fdd1772fe948123ec2fc75')
__v: 203
▾ sneakers: Array (3)
    0: ObjectId('65fc8b12a640d9500ea26e08')
    1: ObjectId('65fc8bbda640d9500ea26e09')
    2: ObjectId('65fc8363a640d9500ea26e06')
```

And here is the Mongoose model for the cart:

```
models > JS cart.js > ...
1  import mongoose, { Schema } from "mongoose";
2
3  const cartSchema = new Schema(
4    {
5      sneakers: [
6        {
7          type: Schema.Types.ObjectId,
8          ref: "Sneaker"
9        }
10     ]
11   },
12 );
13 const Cart = mongoose.models.Cart || mongoose.model("Cart", cartSchema);
14 export default Cart;
15
```

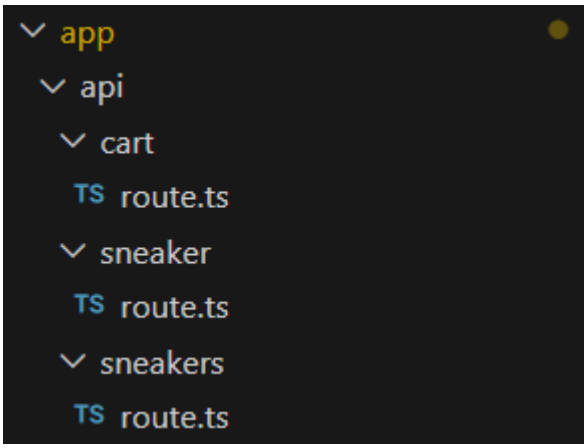
Here is the code I use to connect to my database from my app. It’s a bad practice to leave my connection string unsecured in the app but I'm doing it because it’s a small project. Otherwise, i would put the connection string in a .env file and not commit it to the repo.

libs > JS mongodb.js > ...

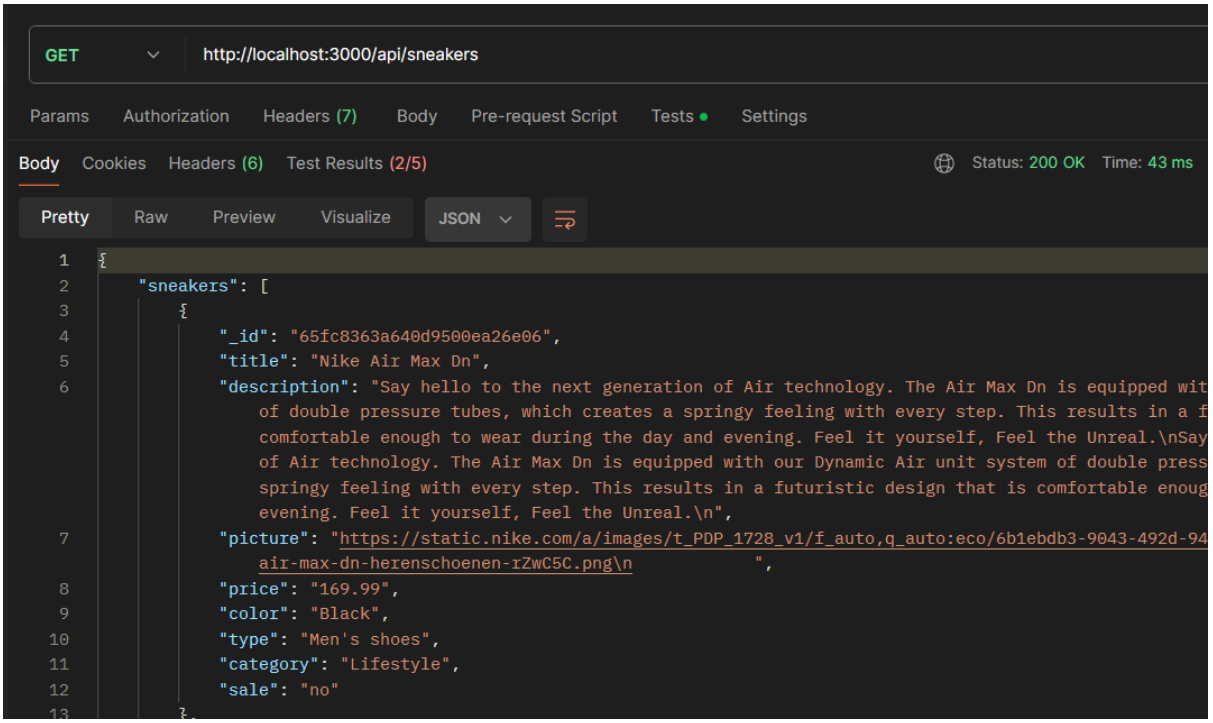
```
1  //connecting to MongoDB
2
3  import mongoose from "mongoose";
4
5  const connectMongoDB = async () => {
6    try {
7      await mongoose.connect("mongodb+srv://viktorvelizarov1:mC9ATnzc0CaviHLW@cluster0.dv6rv5z.mongodb.net/crud_db");
8      console.log("Connected to MongoDB.");
9    } catch (error) {
10     console.log(error);
11   }
12 };
13
14 export default connectMongoDB;
15
```

REST APIs

The backend of my application has a couple of REST API routes that serve the product data to the app and manage the cart object.



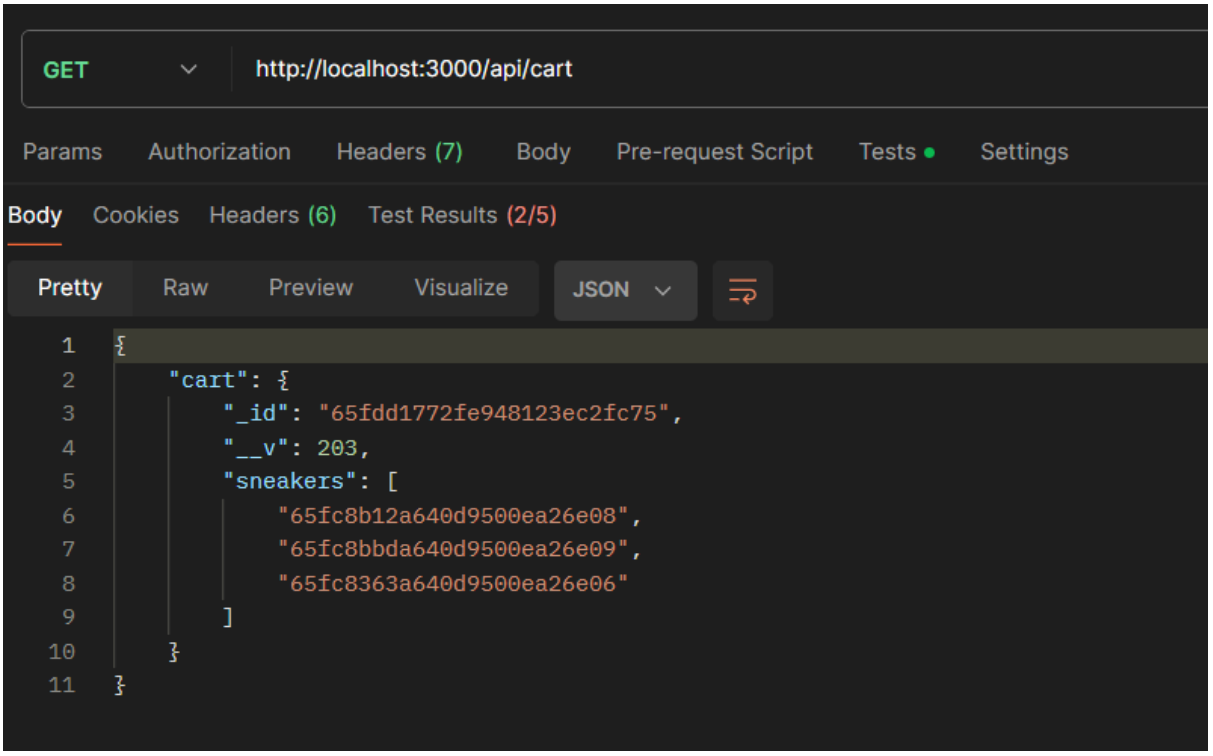
The sneakers route has only a GET endpoint which is used to return all the available sneakers(products) saved in the Database:



The sneaker route has a POST endpoint that takes a sneaker ID as a request body and returns the whole sneaker object corresponding to the given ID as a response

```
// Return a Sneaker Object by a given ID
export async function POST(request: Request) {
  const requestData = await request.json();
  const sneakerId = requestData.sneakerId;
  const sneaker = await Sneaker.findById(sneakerId );
  return NextResponse.json({ sneaker });
}
```

The cart route has a GET, POST and DELETE endpoints which are used to manage the sneakers in the cart. The GET endpoint returns all the sneakers that are currently in the cart:



The POST endpoint adds a sneaker to the cart (I am putting a photo of the function because I can't test the output in Postman like with GET):

```
// adds a sneaker to the cart
export async function POST(request: Request) {
    await connectMongoDB();
```

And the DELETE endpoint removes a sneaker from the cart:

```
// removes a sneaker from the cart
export async function DELETE(request: Request) {
    await connectMongoDB();
```

Front-End:

The front end consists of just 3 components which is very badly organized but I just didn't have enough time to break it down to multiple smaller components and make it cleaner. These components are the Navbar, the main page with the products, and the cart page.

AI Integration (Optional):

I didn't have time to do this optional step but if I did here is how I would've completed it.
I will make an AI-powered search tool where the user can write what product he is looking for in a message. Then an AI Chatbot model will process the message, connect to the Database product data, and return back products matching the user's preferences.

To do this I will use a web API like OpenAI API (https://openai.com/blog/openai-api) to connect a chatbot model to my app via requests and responses. So basically:

- user writes a message on what he is looking for
- message is sent as a request to a chatbot model connected to the app via web API
- chatbot processes the message and also has access to the product data
- chatbot returns matching products via API response