

🌟 Project Documentation Day 3: Integrating Sanity API with Next.js 🌟

Step 1: Installing Next.js

First, I installed Next.js in my project folder by running the following command

```
npx create-next-app@latest
```

This set up a new Next.js project in my chosen folder, where I could start building the application.

Step 2: Installing Sanity

Next, I installed Sanity CMS to manage my content by running:

```
npm install @sanity/client
```

Sanity CMS is a headless CMS that allows me to manage and fetch my data. I used this to manage products, categories, and other relevant information for the project.

Step 3: Setting Up Environment Variables

I created a `.env` file in the root of my project to store my environment variables securely.

Inside this file, I defined the following variables:

```
NEXT_PUBLIC_SANITY_PROJECT_ID=your-project-id  
NEXT_PUBLIC_SANITY_DATASET=your-dataset-name SANITY_API_TOKEN=your-  
api-token
```

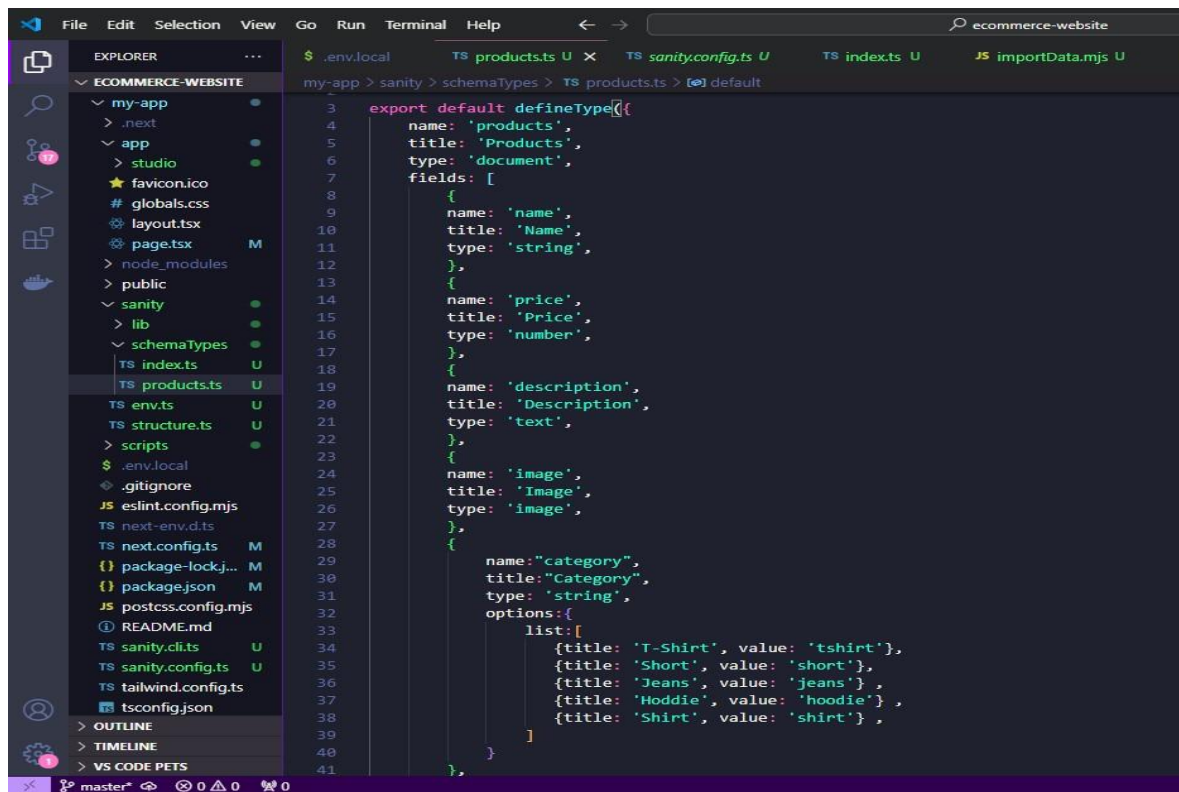
I used these environment variables to configure my connection with the Sanity API.

Step 4: Generating API Token

To access the Sanity API, I generated an API token from the Sanity dashboard. Once generated, I added this token into the `.env` file under the `SANITY_API_TOKEN` variable.

Step 5: Defining Schema Types

I then defined the schema types in the Sanity dashboard to set up the structure of my data. This involved creating schema definitions for `product` , `category` , `discount` , etc. Each schema had fields like `name` , `price` , `description` , `image` , and other necessary fields.



```
File Edit Selection View Go Run Terminal Help
$ .env.local TS products.ts U TS sanity.config.ts U TS index.ts U JS importData.mjs U
EXPLORER
ECOMMERCE-WEBSITE
  my-app
    .next
    app
      studio
        favicon.ico
        # globals.css
        layout.tsx
        page.tsx
      node_modules
      public
    sanity
      lib
      schemaTypes
        TS index.ts
        TS products.ts
      TS env.ts
      TS structure.ts
      scripts
    $ .env.local
    .gitignore
    JS eslint.config.mjs
    TS next-env.d.ts
    TS next.config.ts
    {} package-lock.json
    {} package.json
    JS postcss.config.mjs
    README.md
    TS sanity.cli.ts
    TS sanity.config.ts
    TS tailwind.config.ts
    tsconfig.json
  OUTLINE
  TIMELINE
  VS CODE PETS
  master
  0 0 0 0

my-app > sanity > schemaTypes > TS products.ts > default
3 export default defineType({
4   name: 'products',
5   title: 'Products',
6   type: 'document',
7   fields: [
8     {
9       name: 'name',
10      title: 'Name',
11      type: 'string',
12    },
13    {
14      name: 'price',
15      title: 'Price',
16      type: 'number',
17    },
18    {
19      name: 'description',
20      title: 'Description',
21      type: 'text',
22    },
23    {
24      name: 'image',
25      title: 'Image',
26      type: 'image',
27    },
28    {
29      name: 'category',
30      title: 'Category',
31      type: 'string',
32      options: {
33        list: [
34          {title: 'T-Shirt', value: 'tshirt'},
35          {title: 'Short', value: 'short'},
36          {title: 'Jeans', value: 'jeans'},
37          {title: 'Hoddie', value: 'hoodie'},
38          {title: 'Shirt', value: 'shirt'},
39        ],
40      },
41    },
42  ],
43 })
```

Note : I made sure to check the given API and schemas to ensure they matched properly with the expected structure for the project.

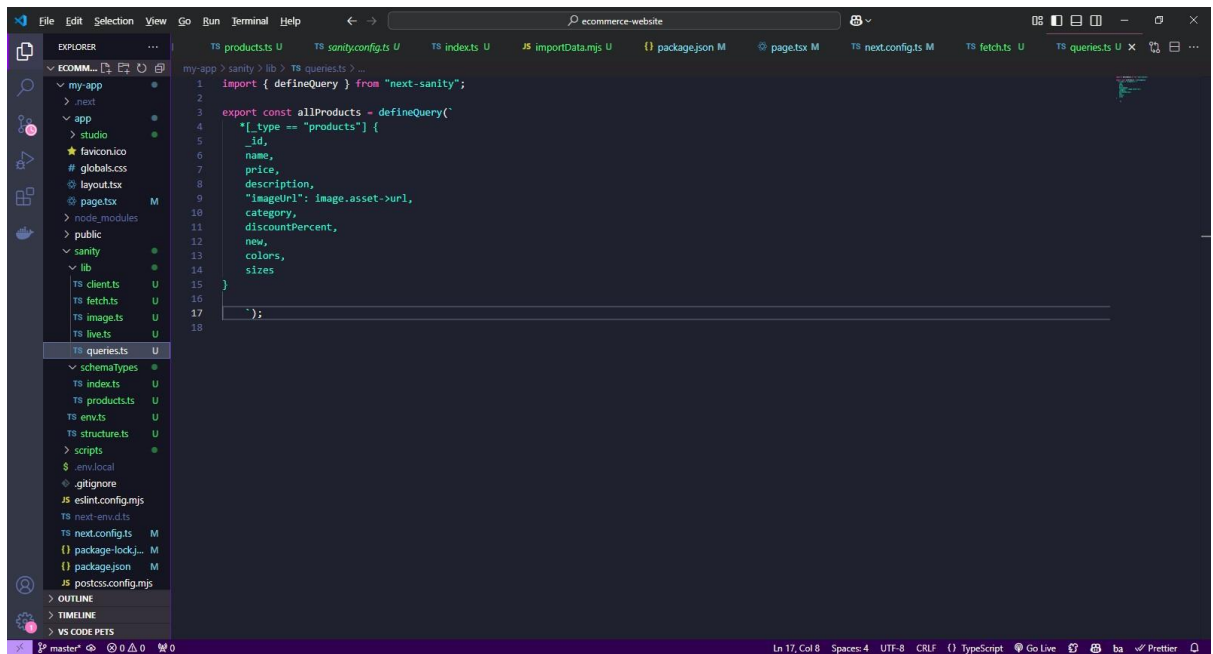
Step 6: Verifying the API Endpoint

Since I couldn't check the data during the upload process, I used tools like Postman to verify the Sanity API endpoint. I sent a GET request to the Sanity API to check if the data was returned as expected. This ensured the API was working correctly and the data was available for use in my project.

Step 7: Setting Up Fetch and Queries Files

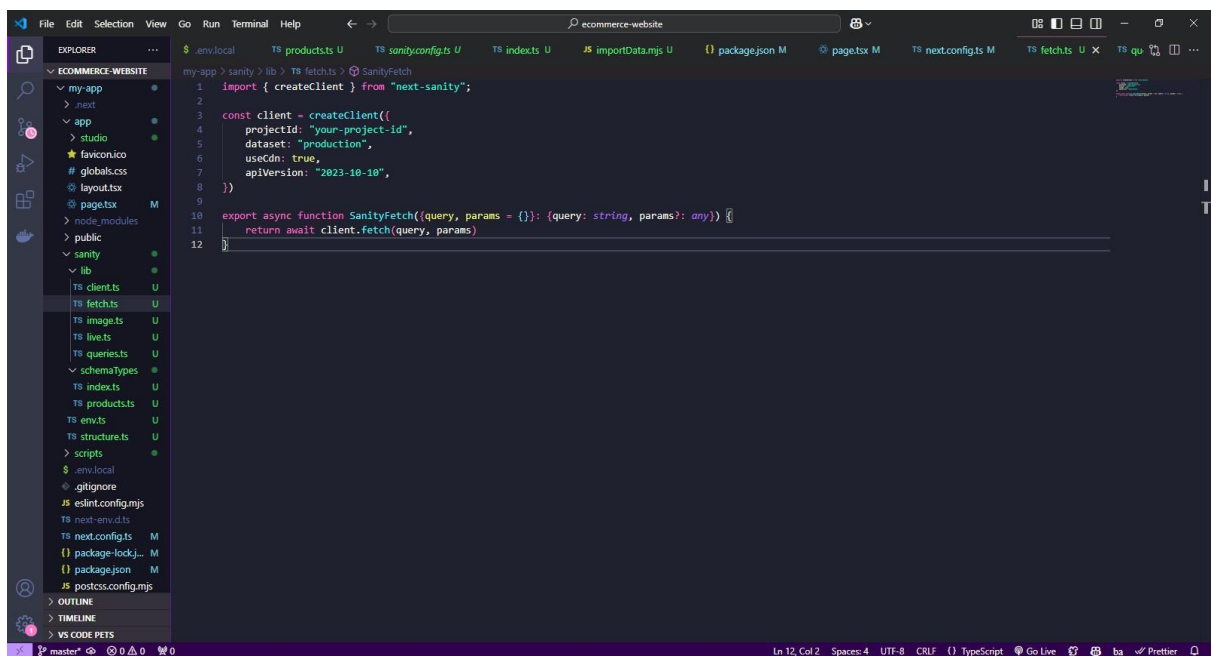
For a clean code structure, I created two files: `fetch.ts` and `queries.ts`.

- **fetch.ts**: This file contained the function to fetch data from the Sanity API.
- **queries.ts**: I defined the necessary queries for fetching data, such as fetching products and categories.



The screenshot shows the VS Code editor with the `queries.ts` file open. The file is located in the `lib` directory of the `my-app` project. The code defines a `defineQuery` function and an `allProducts` query.

```
1 import { defineQuery } from "next-sanity";
2
3 export const allProducts = defineQuery(
4   "[*[_type == 'products'] {
5     _id,
6     name,
7     price,
8     description,
9     'imageUrl': image.asset->url,
10    category,
11    discountPercent,
12    new,
13    colors,
14    sizes
15   }
16 ];
```

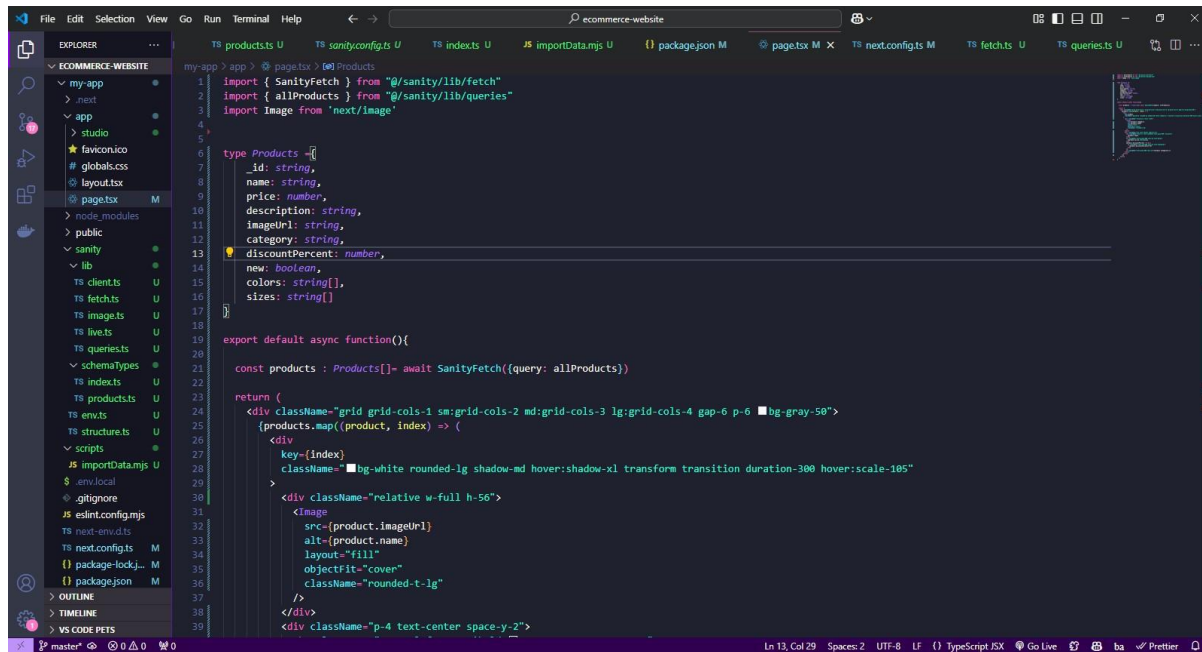


The screenshot shows the VS Code editor with the `fetch.ts` file open. The file is located in the `lib` directory of the `my-app` project. The code defines a `SanityFetch` function that uses the `createClient` function from `next-sanity` to fetch data from the Sanity API.

```
1 import { createClient } from "next-sanity";
2
3 const client = createClient({
4   projectId: "your-project-id",
5   dataset: "production",
6   useCdn: true,
7   apiVersion: "2023-10-18",
8 });
9
10 export async function SanityFetch(query, params = {}): {query: string, params?: any} {
11   return await client.fetch(query, params)
12 }
```

Step 8: Creating the Scripts Folder

I created a `scripts` folder in my project. Inside this folder, I created the `importSanityData.mjs` file, where I set up the migration data provided. This file helped me import all the data into Sanity.



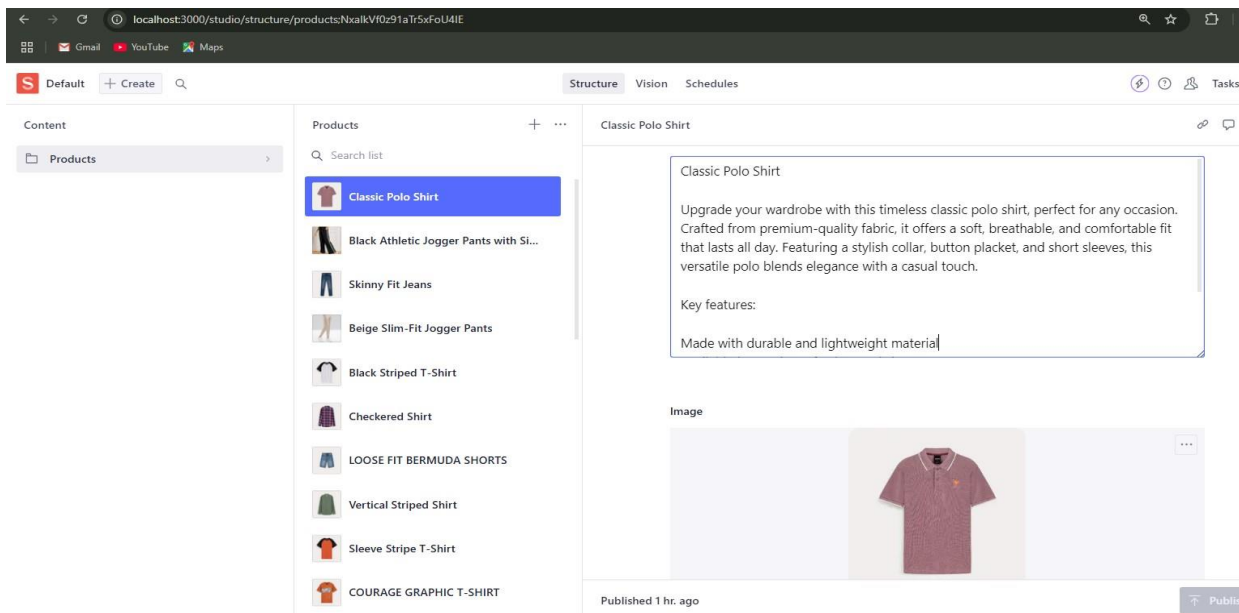
I also defined the required commands in `package.json` to run the scripts easily:

```
"scripts": {
  "import-data": "node scripts/importSanityData.mjs"
}
```

Step 9: Importing Data into Sanity

I used the script defined in the `importSanityData.mjs` file to import all the provided migration data into Sanity. This ensured that the data was available for use in my Next.js application.

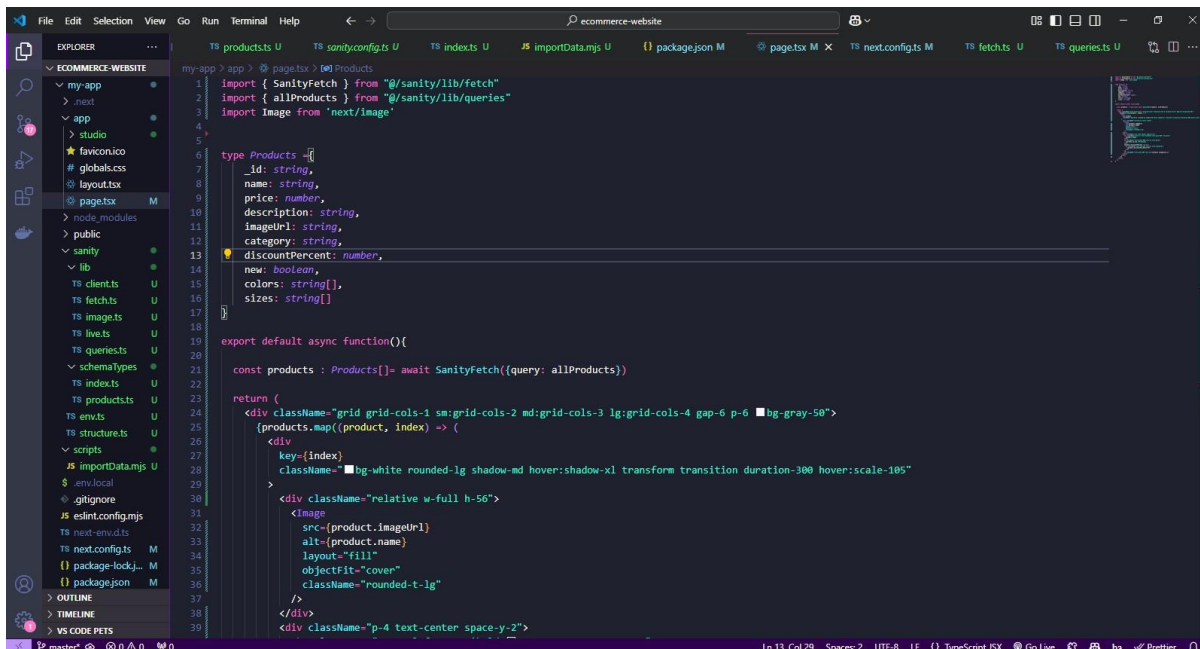
The data was successfully imported into sanity



Step 10: Fetching Data in Next.js

After successfully importing the data into Sanity, I moved on to displaying it in the UI. Instead of using `getServerSideProps`, I defined the types for `products` and other data models. I then imported the `fetch.ts` and `queries.ts` files into my component.

Using the `sanityFetch` function (defined in the `fetch.ts` file), I fetched the data and displayed it on the UI. The data was passed as props to the component, allowing me to display product information like the name, price, description, and image in a clean and structured layout.



Step 11: Displaying Data on the UI

Finally, I used Next.js to create a UI that dynamically rendered the fetched product data. Each product was displayed in a card layout with the name, price, description, and image.

