

# Day 3 - API Integration and Data Migration Report

## E-Commerce Marketplace Development

---

### Objective

The goal for Day 3 was to align our **Sanity CMS schema** with the provided API structure, migrate product data using a custom script, and integrate the data into our Next.js application. These steps ensured the backend was functional and ready to support our E-Commerce Marketplace.

---

### Schema Comparison: Day 2 vs. Day 3

#### Day 2 Schema

During Day 2, our schema was initially designed for a **Q-Commerce Marketplace**, where the primary focus was on rapid delivery. The fields reflected basic product details such as **name**, **price**, **category**, and **tags**. It did not include some E-Commerce-specific attributes such as product descriptions, images, discounts, or new product badges.

#### Day 3 Schema Adjustments

To align with the **E-Commerce Marketplace** requirements and the API structure, we updated the schema as follows:

- New Fields Added:**
  - title:** Replaced the **name** field for better representation.
  - description:** Detailed product descriptions for users to make informed decisions.
  - productImage:** Enabled image uploads to enhance the user experience.
  - discountPercentage:** Captured discount information for promotional campaigns.
  - isNew:** Added a boolean field to highlight new arrivals.
- Validation Rules:** Added required field validation to ensure data integrity during migration.

These changes ensured our schema was fully compatible with the API data structure while meeting E-Commerce business needs.

---

## Data Migration into Sanity CMS

To populate data into Sanity CMS:

1. **Folder Setup:** We created a `scripts` folder within our project to organize migration-related files.
  2. **Migration Script:** Inside the `scripts` folder, we created a file named `importData.mjs`. This script fetched data from the API (<https://template6-six.vercel.app/api/products>) and imported it into Sanity.  
The script handled:
    - Fetching product details from the API.
    - Mapping the API fields to the schema fields in Sanity.
    - Uploading the data to populate Sanity CMS seamlessly.
  3. **Schema File Setup:**
    - Under the `sanity` folder, we created a file named `product.ts` for our schema.
    - The schema file contained the definitions required to handle product data.
    - To ensure this schema is reflected in the Sanity portal, we registered it in the `index.ts` file within the folder of `schemaTypes` located under the folder of Sanity.
  4. **Validation:** After running the script, we verified the imported data on the Sanity dashboard to ensure accuracy. The fields for `title`, `description`, `productImage`, `price`, `tags`, `discountPercentage`, and `isNew` were all successfully populated.
- 

## API Integration Steps

To enable dynamic data fetching for the frontend, we integrated APIs into our Next.js project:

1. **Folder Structure:**
  - Created an `api` folder within the `app router` of the Next.js project.
  - Inside the `api` folder, created a subfolder named `products`.
2. **Route File:**
  - Inside the `products` folder, added a `route.ts` file.
  - The `route.ts` file fetched product data from Sanity CMS using a query and returned it as a JSON response.

## Frontend Rendering

To display the product data on the browser:

1. Created a `page.tsx` file under the app router.
2. Used the API to fetch product details dynamically.

3. Displayed the fetched product **title** and **description** on the webpage.
- 

## Results and Reflection

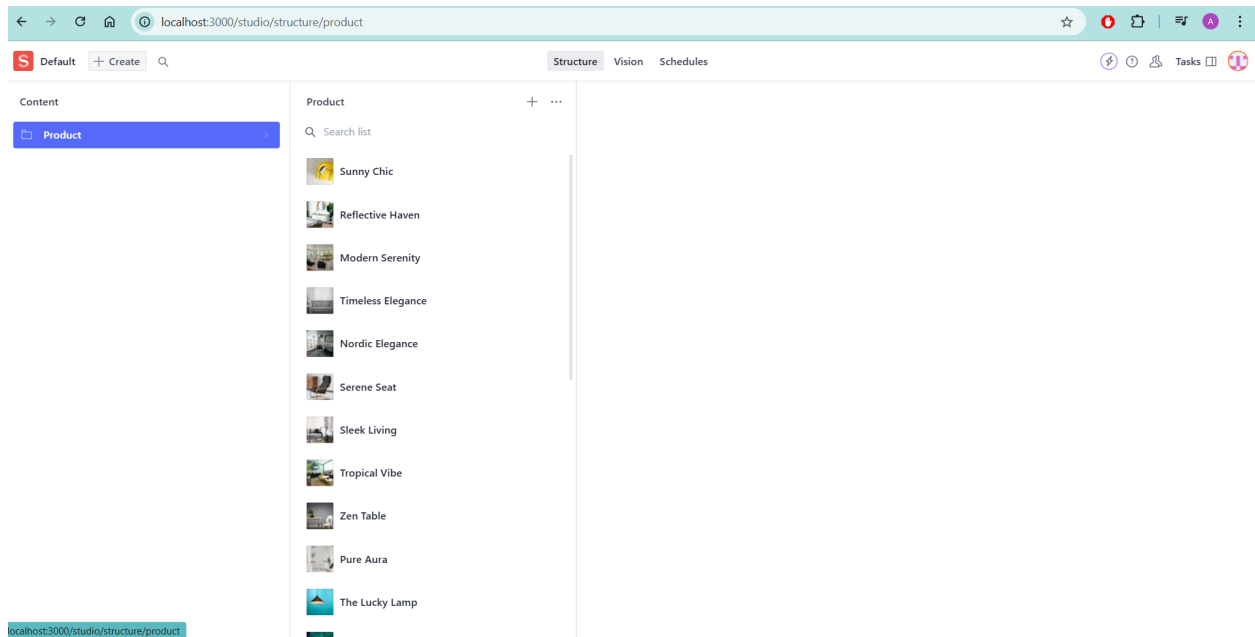
1. **Sanity CMS:**
    - Successfully populated with product data from the API using the migration script.
    - Key fields like **title**, **description**, and **productImage** were accurately reflected in the CMS.
  2. **Frontend Integration:**
    - Displayed dynamic product details (title and description) fetched via the API.
    - This setup ensured the data pipeline from Sanity CMS to the Next.js frontend was functional.
  3. **Schema Adjustments:**
    - The schema modifications ensured compatibility with the E-Commerce template and API structure.
- 

## Conclusion

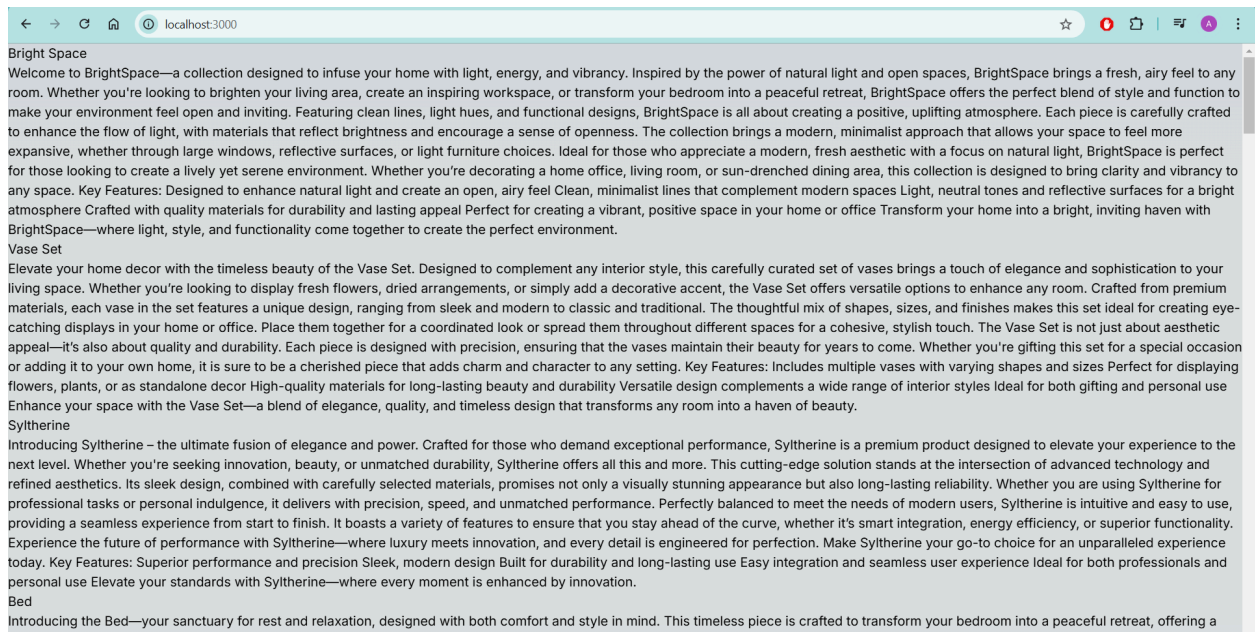
Day 3 was pivotal in transitioning from planning to implementation. Adjusting the schema, populating data into Sanity CMS using a migration script, and integrating APIs into the Next.js frontend gave us a robust backend. This process mimics real-world practices, preparing us for handling diverse client requirements in the future.

Would you like this formatted as a PDF for submission?

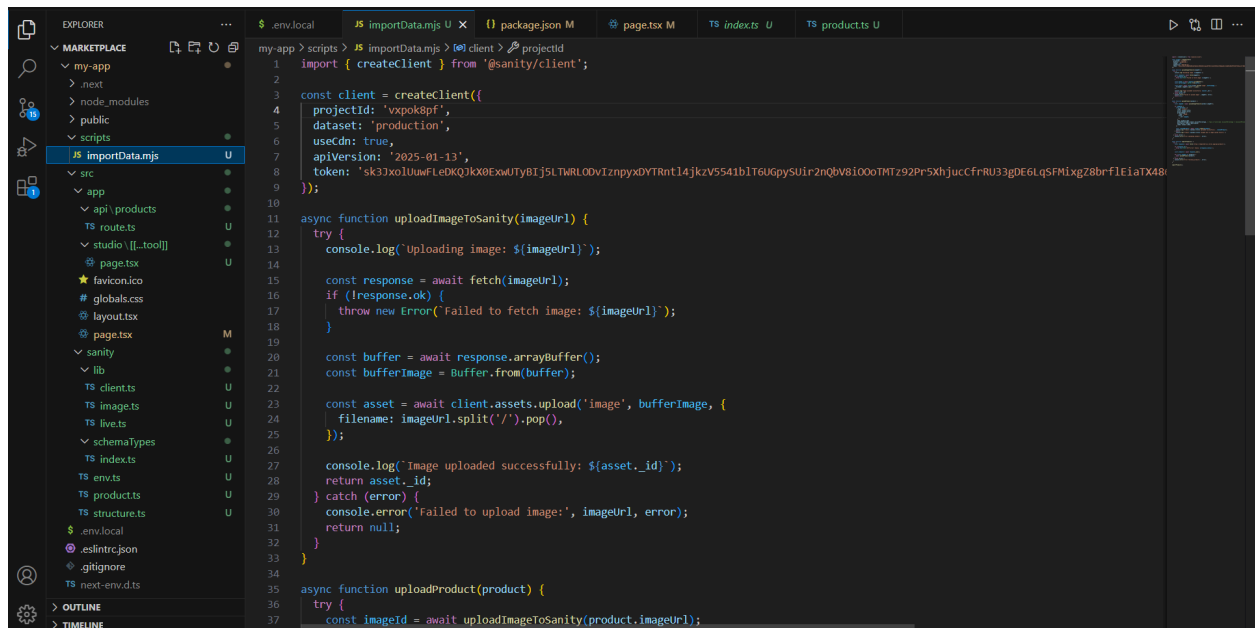
## Appendix 1 - Sanity CMS :



## Appendix 2 - Frontend (Displaying API's Data):



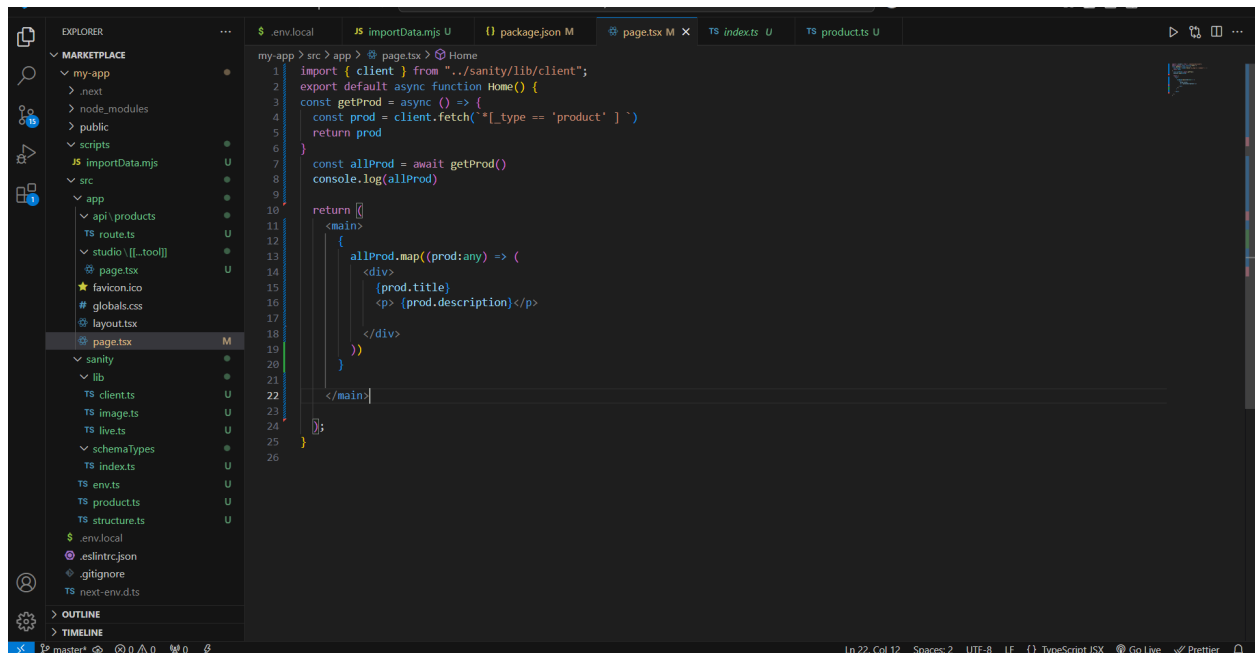
## Appendix 3 - Data Migration:



The screenshot shows a VS Code editor with a project structure on the left and a code editor on the right. The project structure includes a 'my-app' directory with 'scripts', 'src', and 'public' subdirectories. The 'src' directory contains 'api/products', 'route.ts', 'studio', 'page.tsx', 'sanity', 'lib', 'clients.ts', 'images.ts', 'livets', 'schemaTypes', 'index.ts', 'env.ts', 'products.ts', and 'structure.ts'. The 'scripts' directory contains 'importData.mjs'. The code editor shows the implementation of the 'importData.mjs' script, which uses the 'sanity/client' package to create a client and upload images and products to a Sanity.io dataset.

```
1 import { createClient } from '@sanity/client';
2
3 const client = createClient({
4   projectId: 'vxpok8pf',
5   dataset: 'production',
6   useCdn: true,
7   apiVersion: '2025-01-13',
8   token: 'sk33xo1UuwFleDKQJkX0ExwUTyBTj5LTmRL00vIznpysXYTRnT14JkzV5541b1T6UgpySUIr2nQbV81000TMT292Pr5XhjucCfrRU33gpE6Lq5FMixgZ8brf1EiaTX48';
9 });
10
11 async function uploadImageToSanity(imageUrl) {
12   try {
13     console.log('Uploading image: ${imageUrl}');
14
15     const response = await fetch(imageUrl);
16     if (!response.ok) {
17       throw new Error('Failed to fetch image: ${imageUrl}');
18     }
19
20     const buffer = await response.arrayBuffer();
21     const bufferImage = Buffer.from(buffer);
22
23     const asset = await client.assets.upload('image', bufferImage, {
24       filename: imageUrl.split('/').pop(),
25     });
26
27     console.log('Image uploaded successfully: ${asset._id}');
28     return asset._id;
29   } catch (error) {
30     console.error('Failed to upload image:', imageUrl, error);
31     return null;
32   }
33 }
34
35 async function uploadProduct(product) {
36   try {
37     const imageId = await uploadImageToSanity(product.imageUrl);
```

## Appendix 4 - Code For Displaying API Data on Frontend:



The screenshot shows a VS Code editor with a project structure on the left and a code editor on the right. The project structure is the same as in Appendix 3. The code editor shows the implementation of the 'Home' component in 'page.tsx', which uses the 'sanity/client' package to fetch product data from the Sanity.io dataset and display it in a list.

```
1 import { client } from '../sanity/lib/client';
2 export default async function Home() {
3   const getProd = async () => {
4     const prod = client.fetch('*[_type == 'product']')
5     return prod
6   }
7   const allProd = await getProd()
8   console.log(allProd)
9
10   return (
11     <main>
12       {
13         allProd.map((prod:any) => (
14           <div>
15             {prod.title}
16             <p> {prod.description}</p>
17           </div>
18         ))
19       }
20     </main>
21   );
22 }
```