Day 3 - API Integration Report

[Hekto: Building the Future of Furniture Shopping]

Overview

This report provides a comprehensive walkthrough of the API integration, schema adjustments, data migration, and frontend rendering for the Hekto marketplace project. Key steps are illustrated with detailed explanations, code snippets, and screenshots to ensure clarity. The goal is to showcase how backend data is integrated and displayed dynamically on the frontend for a seamless user experience.

API Integration Process

Step 1: Fetching Data from the API

- **API URL**: https://hekto-abdulrehman.vercel.app
- **Objective**: Retrieve product data, including:
  - Name, image, price, description, category, discount percentage, stock level, and featured product status.
- **Method**: Axios GET request.

Code Snippet:

```
const response = await axios.get
("https://hekto-abdulrehman.vercel.app ");
const products = response.data.products;
```

Step 2: Setting Up Sanity Client

- **Objective**: Establish a connection to the Sanity CMS for data storage.
- **Environment Variables**:
  - `NEXT_PUBLIC_SANITY_PROJECT_ID`
  - `NEXT_PUBLIC_SANITY_DATASET`
  - `SANITY_API_TOKEN`

Sanity Client Initialization:

```
const client = createClient({
  projectId: process.env.NEXT_PUBLIC_SANITY_PROJECT_ID,
  dataset: process.env.NEXT_PUBLIC_SANITY_DATASET,
  token: process.env.SANITY_API_TOKEN,
  apiVersion: '2025-01-15',
  useCdn: false,
});
```

Step 3: Schema Design in Sanity

- **Schema Name**: `productData`
- **Fields**:
  - **Name**: String, required.
  - **Image**: Image, hotspot enabled.
  - **Price**: String, required.
  - **Description**: Text, max 150 characters.
  - **Discount Percentage**: Number, range 0-100.
  - **Is Featured Product**: Boolean.
  - **Stock Level**: Number, non-negative.
  - **Category**: String, predefined options.

Code Snippet:

```
export default {
  name: 'productData',
  type: 'document',
  title: 'Product Data',
  fields: [
    { name: 'name', type: 'string', title: 'Name', validation: (Rule) => Rule.required()
},
    { name: 'image', type: 'image', title: 'Image', options: { hotspot: true } },
    { name: 'price', type: 'string', title: 'Price', validation: (Rule) =>
Rule.required() },
    { name: 'description', type: 'text', title: 'Description', validation: (Rule) =>
Rule.max(150) },
    { name: 'discountPercentage', type: 'number', title: 'Discount Percentage',
validation: (Rule) => Rule.min(0).max(100) },
    { name: 'isFeaturedProduct', type: 'boolean', title: 'Is Featured Product' },
    { name: 'stockLevel', type: 'number', title: 'Stock Level', validation: (Rule) =>
Rule.min(0) },
    {
      name: 'category',
      type: 'string',
      title: 'Category',
      options: { list: [{ title: 'Chair', value: 'Chair' }, { title: 'Sofa', value:
'Sofa' }] },
      validation: (Rule) => Rule.required(),
    },
  ],
};
```

Data Migration Steps

Step 4: Image Upload to Sanity

- **Objective**: Upload product images from the API to the Sanity CMS.
- **Method**:
  1. Fetch image data using Axios.
  2. Upload image data as a Sanity asset.

Code Snippet:

```
async function uploadImageToSanity(imageUrl) {
  const response = await axios.get(imageUrl, { responseType: 'arraybuffer' });
  const buffer = Buffer.from(response.data);
  const asset = await client.assets.upload('image', buffer, { filename: imageUrl.split('/').pop() });
  return asset._id;
}
```

Step 5: Uploading Data to Sanity

- **Objective**: Store product data in the `productData` schema.
- **Process**:
  - Loop through each product from the API.
  - Map API data to Sanity schema fields.
  - Create new entries in Sanity.

Code Snippet:

```
async function importData() {
  for (const item of products) {
    // Upload image and get reference
    const imageRef = item.imagePath ? await uploadImageToSanity(item.imagePath) : null;
    // Map fields to Sanity schema
    const sanityItem = {
      _type: 'productData',
      name: item.name,
      category: item.category || null,
      price: item.price,
      description: item.description || '',
      discountPercentage: item.discountPercentage || 0,
      stockLevel: item.stockLevel || 0,
      isFeaturedProduct: item.isFeaturedProduct,
      image: imageRef ? { _type: 'image', asset: { _type: 'reference', _ref: imageRef } } : undefined,
    };
    // Log data and create entry
    console.log("Products: ", sanityItem);
    await client.create(sanityItem);
  }
}
```

Frontend Rendering

Step 6: Fetching Data from Sanity

- **Objective**: Retrieve stored data for rendering.
- **Method**: Use **GROQ queries**.

Example Query:

```
const query = `*[_type == "productData"]{ name, price, description, image,
category }`;
const products = await client.fetch(query);
```

Step 7: Rendering Data on the Frontend

- **Objective**: Dynamically display product details.
- **Method**: Map fetched data to React components.

Code Example:

```
<div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-3 gap-6">
  {products.map((product) => (
    <div
      key={product.name}
      className="bg-white shadow-lg rounded-lg overflow-hidden border border-
gray-200 hover:shadow-xl transition-shadow duration-300"
    >
      <img
        src={product.image.asset.url}
        alt={product.name}
        className="w-full h-48 object-cover"
      />
      <div className="p-4">
        <h2 className="text-lg font-semibold text-gray-800">{product.name}</h2>
        <p className="text-gray-600 mt-2">${product.price}</p>
      </div>
    </div>
  ))}
</div>
```

Tools Used

- **Sanity CMS**: Content management system for storing product data.
- **Axios**: HTTP client for API calls.
- **dotenv**: Environment variable management.
- **Node.js**: For running server-side code.

Visual Representation of Process

1. **API Calls**: Screenshot of API response fetching product data.
2. **Populated Sanity CMS Fields**: Screenshot of Sanity showing product data entries.
3. **Frontend Rendering**: Screenshot of products dynamically displayed on the website.

Conclusion

This report outlines the API integration, schema creation, data migration, and frontend rendering processes in detail. Future improvements include:

- Enhanced error handling.
- Optimizing performance.
- Adding advanced features like search and filtering.