Prepared by: Kulsoom Adnan
Date:18 january 2025

Hackathon Day 3

API Integration and Data Migration Report

Introduction:

In Day 3 of our hackathon, the focus is on integrating APIs and migrating data into Sanity CMS to build a functional marketplace backend. This report outlines the steps taken for API integration and data migration in the marketplace project. The goal of this task was to integrate the provided API, validate and adjust the schema, and migrate data to Sanity CMS. Additionally, the frontend was integrated with the APIs to display product listings, categories, and other relevant data.

API Integration Process:

1. **Understanding the Provided API:**

   We began by reviewing the API documentation to understand the available endpoints. Key endpoints identified include:

   - **Product Listings**: /products – This endpoint provides product data, including titles, descriptions, prices, and images.discountedPrices , discountedPercentanfe,IsNew etc.

2. **API Integration in Next.js :**

   This section outlines the step-by-step process of integrating the API into the frontend application to fetch and display data dynamically.

   1. **Backend API Setup:**
      - Created a Next.js API route **(/api/products**) to query data from Sanity CMS using a Sanity client.

- Returned the fetched product data as a JSON response.

2. **Frontend API Call:**
   - Used React's useEffect hook to call the /api/products endpoint via fetch.
   - Stored the fetched data in state (products) using useState.
3. **State Management and Rendering:**
   - Managed loading state (isLoading) for a smooth user experience.
   - Rendered ProductCard components dynamically with the fetched data.
4. **Error Handling:**
   - Implemented error handling in both backend and frontend to manage API failures and connection issues.

Summary

By combining the backend API route with dynamic frontend components, the API integration provides a seamless experience for fetching and displaying product data from Sanity CMS in real-time. The structured approach ensures scalability and maintainability.

3. **Testing the API Integration:** To ensure data consistency, we used tools like Postman and browser developer tools to test the API endpoints. We logged the responses and checked for issues related to missing or incorrect data.

Schema Validation and Adjustments

- **Comparing the API Data with the Existing Sanity Schema:** We reviewed the Sanity CMS schema and compared it with the structure of the data provided by the API. A key adjustment involved renaming fields to match the existing schema:
  - **API Field**: imagUrl → **Schema Field**: productImages

This adjustment ensured that data could be mapped correctly when imported into Sanity.

Scheme Code ;

```
import { defineType } from "sanity";

export const product = defineType({
  name: "product",
  title: "Product",
  type: "document",
  fields: [
    {
      name: "title",
      title: "Title",
      validation: (rule) => rule.required(),
      type: "string",
    },
    {
      name: "description",
      type: "text",
      validation: (rule) => rule.required(),
      title: "Description",
    },
    {
      name: "productImage",
      type: "image",
      options: {
        hotspot: true,
      },
      validation: (rule) => rule.required(),
      title: "Product Image",
    },
    {
      name: "price",
      type: "number",
      validation: (rule) => rule.required(),
      title: "Price",
    },
    {
      name: "tags",
      type: "array",
      title: "Tags",
      of: [{ type: "string" }],
    },
    {
      name: "dicountPercentage",
      type: "number",
      title: "Discount Percentage",
    },
    {
      name: "isNew",
      type: "boolean",
      title: "New Badge",
    },

    {
      name: "productDetails",
      type: "string",
      validation: (rule) => rule.required(),
    },
    {
      name: "discountedPrice",
      type: "number",
      // validation: (rule) => rule.required(),
    },
  ],
});

```

Data Migration Script :

The core of the data migration process involved creating a script called importData.js. This script was responsible for fetching data from the API, transforming it to match the Sanity schema, and then importing it into the Sanity CMS. Below is an overview of the migration steps:

Script Overview

Environment Variables Setup

To securely manage sensitive data such as API tokens, I set up environment variables in my project. This ensures that sensitive information is not hard-coded into the script or source code, which is a best practice for security.

1. **Setting up the .env File:**
    - In the root directory of the project, I created a **.env** file where I stored the Sanity API token and other relevant environment variables.

      Example .env file:

      NEXT_PUBLIC_SANITY_PROJECT_ID

      NEXT_PUBLIC_SANITY_DATASET

      NEXT_PUBLIC_TOKEN_API

```
async function uploadImageToSanity(imageUrl : string) {
  try {
    console.log(`Uploading image: ${imageUrl}`);

    const response = await fetch(imageUrl);
    if (!response.ok) {
      throw new Error(`Failed to fetch image: ${imageUrl}`);
    }

    const buffer = await response.arrayBuffer();
    const bufferImage = Buffer.from(buffer);

    const asset = await client.assets.upload('image', bufferImage, {
      filename: imageUrl.split('/').pop(),
    });

    console.log(`Image uploaded successfully: ${asset._id}`);
    return asset._id;
  } catch (error) {
    console.error('Failed to upload image:', imageUrl, error);
    return null;
  }
}

async function uploadProduct(product : any ) {
  try {
    const imageId = await uploadImageToSanity(product.imageUrl);

    if (imageId) {
      const document = {
        _type: 'product',
        title: product.title,
        price: product.price,
        productImage: {
          _type: 'image',
          asset: {
            _ref: imageId,
          },
        },
        tags: product.tags,
        dicountPercentage: product.dicountPercentage, // Typo in field name: dicountPercentage -> discountPercentage
        description: product.description,
        isNew: product.isNew,
      };

      const createdProduct = await client.create(document);
      console.log(`Product ${product.title} uploaded successfully:`, createdProduct);
    } else {
      console.log(`Product ${product.title} skipped due to image upload failure.`);
    }
  } catch (error) {
    console.error('Error uploading product:', error);
  }
}

async function importProducts() {
  try {
    const response = await fetch('https://template6-six.vercel.app/api/products');

    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }

    const products = await response.json();

    for (const product of products) {
      await uploadProduct(product);
    }
  } catch (error) {
    console.error('Error fetching products:', error);
  }
}

importProducts();

```

The script migrates product data from an external API to Sanity CMS. It fetches product details, uploads images, and stores the products in Sanity, ensuring the data matches the Sanity schema.

Key Functions:

1. **uploadImageToSanity(imageUrl)**
   Uploads the product image to Sanity and returns the image asset ID.
2. **uploadProduct(product)**
   Transforms the product data to match the Sanity schema and creates a product document in Sanity with the image reference.
3. **importProducts()**
   Fetches product data from the API, processes each product, and calls uploadProduct() to upload the product data to Sanity.

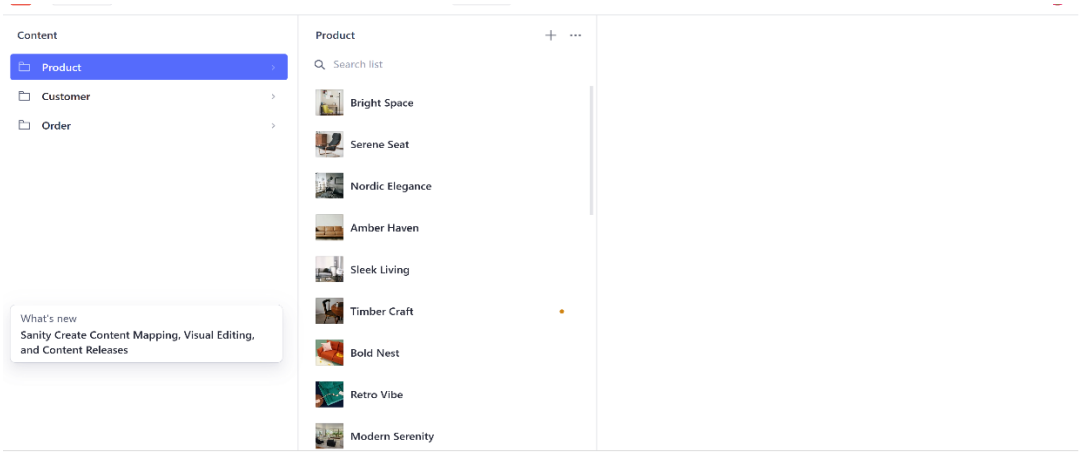
Importing Data into Sanity:

- The script used Sanity's JavaScript client (@sanity/client) to insert the transformed data into the Sanity CMS.
- The transformed product listings and categories were added to their respective datasets in Sanity.

Process Flow:

- Fetch data from the external API.
- For each product, upload the image to Sanity and create a product document with the transformed data.
- Ensure all product information, including title, price, description, tags, and image, is stored correctly in Sanity.

Post-Migration: After running the script, I checked the Sanity CMS to ensure that the data had been successfully migrated.Here Is the Response



API Integration in Next.js:

In this section, we explain how the API was integrated into the Next.js frontend using API routes. Here's how you handled the API responses and ensured the data was displayed properly:

API Route (api/products.js):

```
import { client } from "@/sanity/lib/client";
import { NextRequest, NextResponse } from "next/server";

export async function GET(req: NextRequest) {
  try {
    // Query for fetching products
    const query = `*[_type == "product"]`;

    const products = await client.fetch(query);

    // Return the data as JSON
    return NextResponse.json({ success: true, products }, { status: 200 });
  } catch (error) {
    console.error("Error fetching products:", error);
    return NextResponse.json(
      { success: false, message: "Failed to fetch products." },
      { status: 500 }
    );
  }
}

```

- The **GET** method fetches product data from Sanity using a query that retrieves all documents of type "product."
- The products are returned as a JSON response. If an error occurs, a failure response is sent.

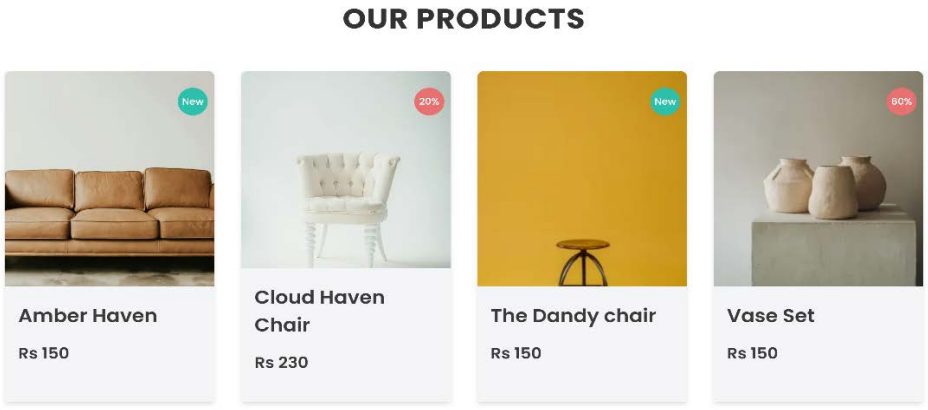
Product Page.tsx Code ;

```tsx
"use client";
import Filter from "@/components/Filter";
import RouteHero from "@/components/RouteHero";
import Services from "@/components/Services";
import React, { useEffect, useState } from "react";
import Skeleton from "@/components/Skeleton";
import ProductCard from "@/components/ProductCard";
import PaginationUi from "@/components/Pagination";
import { ProductInterface } from "@/components/Types";

function page() {
  const [products, setProducts] = useState<ProductInterface[]>(); // State to store products
  const [isLoading, setIsLoading] = useState(true); // State for loading indicator

  useEffect(() => {
    async function getData() {
      try {
        let response = await fetch("http://localhost:3000/api/products");
        const data = await response.json();
        if (data.success) {
          setProducts(data.products);
        } else {
          console.error("Failed to fetch products:", data.message);
        }
      } catch (error) {
        console.error("Error fetching products:", error);
      } finally {
        setIsLoading(false); // Stop loading
      }
    }

    getData();
  }, []); // Empty dependency array ensures this runs once when the component mounts

  if (isLoading) {
    return <Skeleton />;
  }
  console.log(products);
  return (
    <div className="h-auto overflow-hidden">
      <RouteHero prop="Shop" />
      <Filter />

      {/* Product Grid */}
      <div className="w-full py-20 px-10">
        <ul className="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 lg:grid-cols-4 gap-8">
          {products &&
            products.map((product) => (
              <ProductCard key={product._id} product={product} />
            ))}
        </ul>
      </div>
      <div className="pb-10">
        <PaginationUi />
      </div>
      <Services />
    </div>
  );
}

export default page;

```

Key Steps:

1. **Fetching Data from the API:**
    - The useEffect hook is used to fetch product data from the backend API when the component loads.
    - The fetch function makes a request to the /api/products endpoint to retrieve product data.
    - The data is fetched as JSON, and if successful, it is stored in the state variable products using setProducts.
2. **Handling Loading State:**
    - While data is being fetched, a loading state (isLoading) is set to true to show a skeleton loader (Skeleton) until the data is ready.
    - Once the data is fetched, isLoading is set to false, and the actual product data is rendered.
3. **Displaying Products:**
    - The products are displayed in a grid using the ProductCard component.
    - Each product is passed as a prop to the ProductCard component, where its details (such as title, price, and image) are shown.

Display Product To UI



Self-Validation Checklist 🌟

🌐 API Understanding:

- ✅

📜 Schema Validation:

- ✅

📦 Data Migration:

- ✅

💻 API Integration in Next.js:

- ✅

📄 Submission Preparation:

- ✅

Conclusion:

This document captures the successful completion of **Day 3 - API Integration and Data Migration** for Marketplace **Furniro**. All required tasks, including API integration, schema adjustments, data migration, and frontend display verification, have been thoroughly executed and documented. Screenshots, code snippets, and validation processes ensure a comprehensive understanding of the work done.