By Muhammad Abdullah

Furniro: API Integration & Sanity Migration

Introduction

Furniro is a modern furniture marketplace designed to offer seamless product listings, category management, and a user-friendly shopping experience. To ensure efficient data management and scalability, we have integrated APIs and migrated data into Sanity CMS.

API Integration

1. Understanding the Provided API

- The API provides structured data, including product listings, categories, and order details.
- We used endpoints such as:
    - **/products** –This endpoint fetches all available furniture products, making it easy to display multiple items on the marketplace.
    - **/product[title]** –Retrieves details of a specific product based on its title, allowing users to view individual product information

I have gained a deeper understanding of how structured API endpoints facilitate efficient data retrieval and management. This enables me to integrate dynamic content seamlessly while ensuring optimized performance and scalability.

Sanity CMS Migration

2. Schema Validation & Adjustments

This section explains how the schema is validated and adjusted to match API data fields for seamless integration with Sanity CMS.

- **Schema Matching:** The schema is structured to align with Sanity data, ensuring consistency between stored and retrieved information.
- **Sanity Schema Definition:** It defines a `product` document with key fields such as `title`, `price`, and `image`, allowing structured data management.
- **Example Schema:** The provided code snippet demonstrates a simplified schema format, highlighting how product details are structured within Sanity CMS.

```
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",
      validation: (rule) => rule.required(),
      title: "Product Image",
    },
    {
      name: "price",
      type: "number",
      validation: (rule) => rule.required(),
      title: "Price",
    },
    {
      name: "tags",
      type: "array",
```

```
{
  name: "dicountPercentage",
  type: "number",
  title: "Discount Percentage",
},
{
  name: "isNew",
  type: "boolean",
  title: "New Badge",
},
{
  name: "slug",
  type: "slug",
  title: "Slug",
  options: {
    source: "title",
    maxLength: 96,
  },
},
{
  name: "inventoryInStock",
  type: "number",
  title: "Inventory In Stock",
  initialValue: 0,
  validation: (rule) => rule.required().min(0),
},
{
  name: "featured",
  type: "string",
  title: "Featured Name",
```

---

3. Data Migration Process

The data migration process involves fetching product data from the API, transforming it to match Sanity's schema, and efficiently storing it in the CMS.

1. **Fetching Data:**
    - The `fetchProducts` function retrieves product data from the API.
2. **Transforming Data:**
    - The fetched data is reformatted to align with Sanity's schema fields (`title`, `price`, `image`).
3. **Migrating to Sanity CMS:**

- A script iterates over the products and creates new entries in Sanity CMS using the `sanityClient.create()` method.

```
async function uploadProduct(product) {
  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();
```

---

4. API Integration in Next.js - Day 3

Step 1: Create Utility Functions

- **Objective:** Build functions to handle API requests.
- **Key Points:**
  - Use `fetch` or libraries like Axios.
  - Centralize API calls for better maintainability.
  - Create reusable functions for different endpoints.

Step 2: Render Data in Components

- **Objective:** Display the data fetched from the API within your components.
- **Key Points:**
  - Use React's `useEffect` to fetch data on component load.
  - Store the data in `useState` and pass it to the components.
  - Ensure proper error handling if the API request fails.

Step 3: Test API Integration

- **Objective:** Verify that the API integration works as expected.
- **Key Points:**
  - Test API calls to ensure correct responses.
  - Handle errors gracefully (e.g., loading states, error messages).
  - Test on multiple environments (development/production).

```ts
export async function fetchProducts(): Promise<ProductCardData[]> {
  const query = `*[_type == "product"]{
    _id,
    title,
    description,
    "productImage": productImage.asset->url,
    price,
    tags,
    featured,
    dicountPercentage,
    isNew,
    inventoryInStock,
    slug
  }`;
```

```ts
return products.map((product: FetchedProduct) => {
  // console.log("Fetched Product:", product);
  return {
    _id: product._id,
    image: product.productImage,
    sideImages: [
      product.productImage,
      product.productImage,
      product.productImage,
      product.productImage,
    ],
    dicountPercentage: product.dicountPercentage
      ? { text: `-${product.dicountPercentage}%`, color: "#E97171" }
      : undefined,
    title: product.title,
    tags: product.tags.join(", "),
    price: product.price.toString(),
    oldPrice: product.dicountPercentage
      ? (
          parseFloat(product.price) *
          (1 + parseFloat(product.dicountPercentage) / 100)
        ).toFixed(2)
      : undefined,
    stars: ["★", "★", "★", "★"],
    reviews: "0 Customer Review",
    description: product.description,
    featured: product.featured,
    isNew: product.isNew,
    inventoryInStock: product.inventoryInStock,
    slug: product.slug
  };
```

```tsx
const ProductCard = ({ showProducts }: { showProducts?: number }) => {
  const { addToCart } = useCart();
  const [products, setProducts] = useState<ProductCardData[]>([]);
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    async function loadProducts() {
      try {
        const fetchedProducts = await fetchProducts();
        setProducts(fetchedProducts);
        setError(null);
      } catch (err) {
        setError(
          err instanceof Error ? err.message : "Failed to load products"
        );
      } finally {
        setIsLoading(false);
      }
    }

    loadProducts();
  }, []);
```

```
if (error) {
  return (
    <div className="text-center py-10">
      <p className="text-red-500 mb-4">{error}</p>
      <button
        onClick={() => window.location.reload()}
        className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600"
      >
        Try Again
      </button>
    </div>
  );
}

return (
  <div className="flex flex-col justify-center items-center sm:grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 lg:grid-cols-4 gap-10 sm:gap-8 my-6
  mx-4 lg:mx-6">
    {isLoading
? Array.from({ length: 4 }).map((_, index) => (
    <div
      key={index}
      className="relative flex flex-col w-[250px] sm:w-auto h-[300px] lg:h-[330px] bg-gray-100 rounded-sm shadow-md overflow-hidden sm:mx-10
      md:mx-2 lg:mx-0 animate-pulse"
    >
      <div className="relative w-full h-0 pb-[75%] bg-gray-300"></div>
      <div className="flex flex-col gap-2 p-4">
      <div className="h-6 bg-gray-300 rounded w-3/4 mb-2">
          <span className="text-gray-600 text-xs">Loading product...</span>
        </div>
        <div className="h-4 bg-gray-300 rounded w-1/2"></div>
        <div className="h-4 bg-gray-300 rounded w-1/3"></div>
      </div>
    </div>
  ))
      : products.slice(0, showProducts).map((card) => (
          <div
            key={card._id}
            className="relative flex flex-col w-[250px] sm:w-auto h-[350px] lg:h-[370px] bg-[#F4F5F7] rounded-sm shadow-md overflow-hidden
```

```
const ProductDetail = () => {
  const { addToCart } = useCart();
  const [quantity, setQuantity] = useState(1);
  const { id } = useParams<{ id: string }>();
  const [products, setProducts] = useState<ProductCardData[] | null>(null);

  useEffect(() => {
    async function loadProducts() {
      const fetchedProducts = await fetchProducts();
      setProducts(fetchedProducts);
    }
    loadProducts();
  }, []);

  if (!products) {…
  }

  const product = products.find(
    (item) => item._id === id || item.slug?.current === id
  );

  if (!product) {
    return (
      <div className="text-center my-16">
        <p>Product not found. Please check the URL.</p>
        <a href="/" className="text-blue-500 underline">…
        </a>
      </div>
    );
  }
  const handleIncrement = () => {…
  };
  return (
    <div className="w-full mx-auto py-10 mb-12 px-6 sm:px-10 lg:px-24 font-poppins flex flex-col lg:flex-row gap-10 lg:gap-6">
      <div className="basis-[40%] flex flex-col lg:flex-row gap-6">
        <div className="flex flex-row lg:flex-col gap-3 sm:gap-4">
          {product.sideImages?.map((img, idx) => (
            <Image
              key={idx}
              src={img}
```

5. Error Handling & Debugging

I implemented logging for debugging API responses

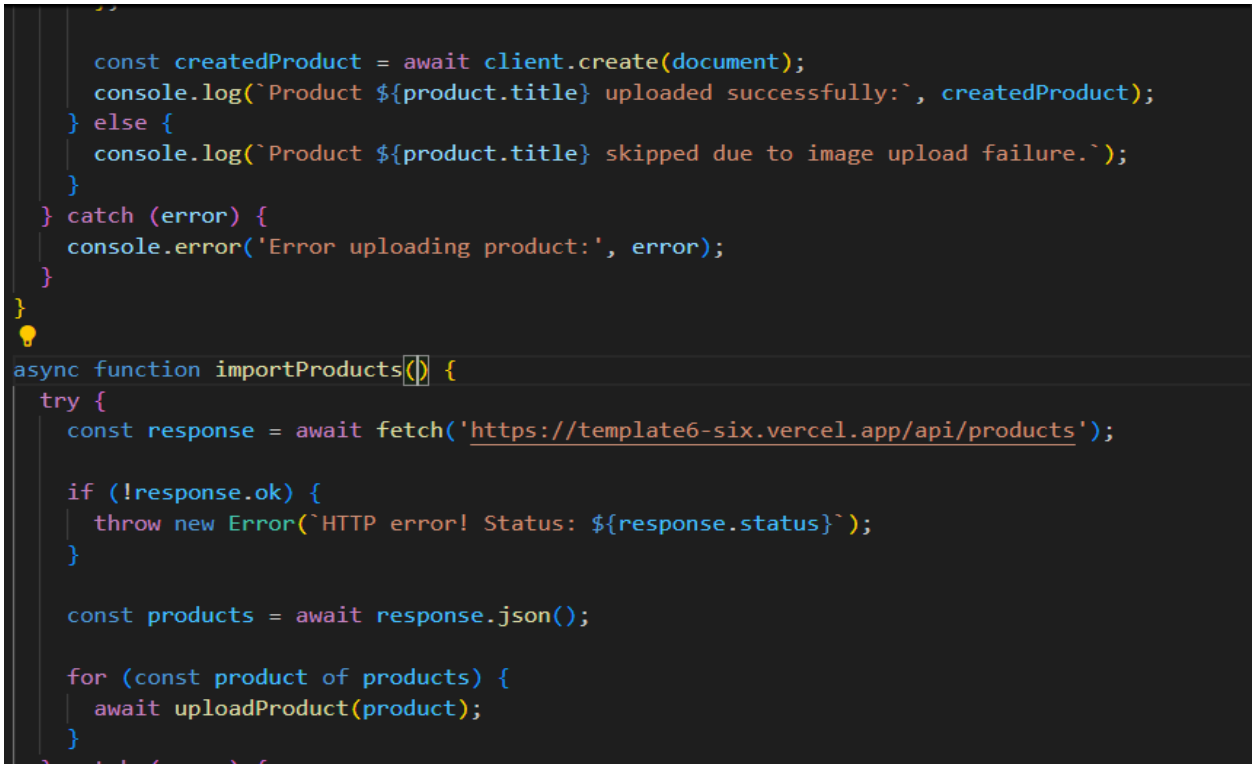
- To ensure everything was running smoothly, I set up logging for the API requests and responses. This helped me track request details like URL, headers, and data, and allowed me to log the API response status and data.

I used Postman to test API endpoints before migration

- Before integrating the APIs into the Next.js project, I thoroughly tested them using Postman. This allowed me to verify that the endpoints were returning the correct data and to catch any potential issues like incorrect responses, bad requests, or authentication errors. Postman's environment variables were super helpful for testing different endpoints quickly.

I added fallback UI states for improved user experience

- **I** made sure that even if the API failed or took time to respond, the user experience would remain smooth. I implemented loading indicators (loading) and error messages to keep users informed. Additionally, I created fallback UI elements to ensure that the page wouldn't break if something went wrong.



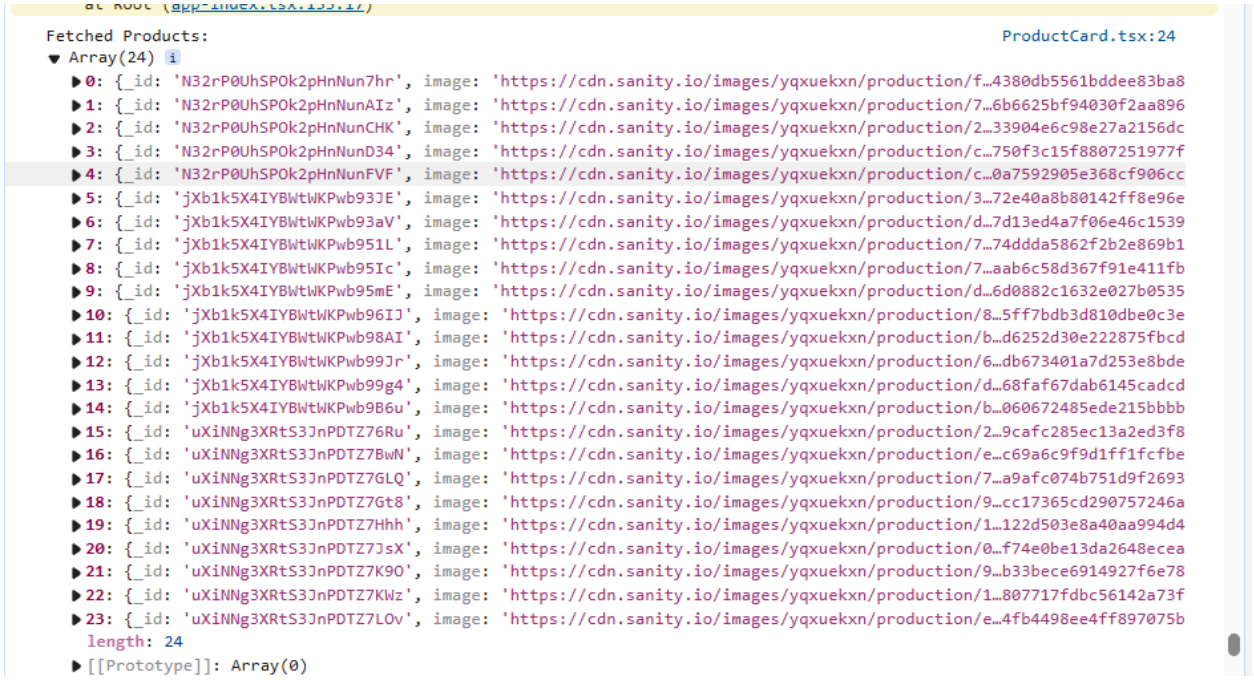
```
      const createdProduct = await client.create(document);
      console.log(`Product ${product.title} uploaded successfully:`, createdProduct);
    } else {
      console.log(`Product ${product.title} skipped due to image upload failure.`);
    }
  } catch (error) {
    console.error('Error uploading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

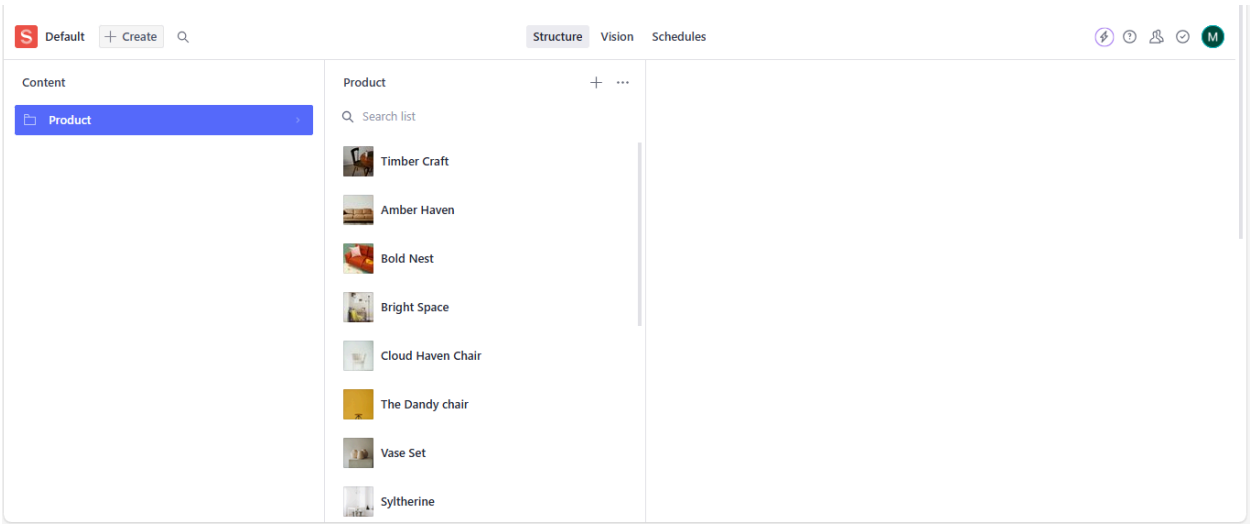



```
useEffect(() => {
  async function loadProducts() {
    try {
      const fetchedProducts = await fetchProducts();
      setProducts(fetchedProducts);
      console.log("Fetched Products:", fetchedProducts);
      setError(null);
    } catch (err) {
      setError(
        err instanceof Error ? err.message : "Failed to load products"
      );
    } finally {
      setIsLoading(false);
    }
  }

  loadProducts();
}, []);
```

Conclusion

- Successfully integrated API data into Next.js for real-time updates.
- Migrated and structured data in Sanity CMS for better management.
- Ensured schema compatibility and efficient data handling for a scalable furniture marketplace.



Day 3 Checklist:

Self-Validation Checklist:

API Understanding: ✔

Schema Validation: ✔

Data Migration: ✔

API Integration in Next.js: ✔

Submission Preparation: ✔