

Introduction

The goal of this report is to document the process of integrating external APIs into the marketplace platform and migrating data into the Sanity CMS for a seamless user experience. The integration involves fetching product data for the shop page and ensuring that all dynamic content is handled efficiently. This report covers the steps taken to set up API integrations, adjustments made to schemas, and migration procedures, as well as screenshots illustrating the successful display of data in the frontend and Sanity CMS.

API Integration Process

Overview of the API Integration

The marketplace requires real-time data fetching to display various products from the store and associated chefs. This was achieved through the integration of two APIs: one to fetch shop data (`fetchShopData()`) and another to fetch chef-related data (`fetchChefData()`). These APIs ensure that the frontend dynamically updates with product and chef data whenever users interact with the platform.

API Setup and Call Structure

1. Fetching Shop Data:

- The `fetchShopData()` utility function is responsible for fetching the shop data, such as product names, prices, and images.
- This data is fetched asynchronously using the `useEffect` hook within the `ShopPage` component, ensuring that data is loaded only once the component is mounted.

2. Fetching Chef Data:

- Similar to the shop data fetching mechanism, `fetchChefData()` is used to pull data related to chefs and their profiles. This can be used in a separate component for displaying chef-specific details.

API Call Implementation: The following code snippet demonstrates how API calls are made to fetch shop data:

```

20 | import { fetchShopData } from "@lib/utils";
21 |
22 | interface MenuType {
23 |   id: string;
24 |   name: string;
25 |   price: number;
26 |   originalPrice?: number;
27 |   imageUrl: string;
28 | }
29 |
30 | export default function ShopPage() {
31 |   // const products = [ ...
32 |   // ]
33 |   const [menu, setMenu] = useState<MenuType[] | null>(null);
34 |   const [error, setError] = useState<string>('');
35 |   const [loading, setLoading] = useState<boolean>(true);
36 |
37 |   useEffect(() => {
38 |     const loadData = async () => {
39 |       try {
40 |         const data = await fetchShopData();
41 |         setMenu(data);
42 |       } catch (err) {
43 |         setError("Failed to load data");
44 |       } finally {
45 |         setLoading(false);
46 |       }
47 |     };
48 |     loadData();
49 |   }, []);
50 |
51 |   if (loading) return <div>Loading...</div>;
52 |   if (error) return <div>{error}</div>;
53 |

```

Error Handling: The `try-catch` block ensures that any errors encountered during the data fetching process are handled gracefully. The `setError()` function updates the state to display an error message on the frontend if the API call fails.

Displaying Data in Frontend: Once the data is fetched successfully, it is stored in the state (`menu`) and rendered in the UI. Each item is displayed within a `ShopCard` component, showcasing the product name, price, and image.

```

168      {/* Food Cards */}
169      <div className="col-span-9 row-span-7 flex flex-wrap md:justify-normal justify-center mt-4 md:mt-0 gap-4">
170        {menu ? (
171          menu.map((food, idx) => (
172            <Link key={idx} href={` /shop/${food.name}`} >
173              <ShopCard
174                imagePath={food.imageUrl}
175                altText={food.name}
176                imageHeight={100}
177                imageWidth={244}
178                dishName={food.name}
179                currentPrice={food.price}
180                oldPrice={food.originalPrice}
181              />
182            </Link>
183          )) : (
184            <p>Loading...</p>
185          )}

```

This code snippet checks if menu has data, then maps over each food item and renders it in the form of a **ShopCard** component.

Modularization and Scalability

The API integration is designed to be modular, with `fetchShopData()` and `fetchChefData()` utility functions handling data fetching independently. This modular approach ensures that the platform can easily be scaled by adding more APIs for additional functionalities, such as user authentication or order management, without impacting the existing codebase.

Schema Adjustments

In the process of adapting the food and chef data models to the given API requirements, the schema for the food item has been modified, and a new schema for chefs was added. The following outlines the changes made:

1. Original Schema for Food:

Before the adjustments, the schema for the food item was as follows:

```

export default {
  name: "product",
  title: "Product",
  type: "document",
  fields: [

```

```
{
  name: "name",
  title: "Food Name",
  type: "string",
  validation: (Rule) => Rule.required(),
},
// {
//   name: "slug",
//   title: "Slug",
//   type: "slug",
//   options: { source: "name", maxLength: 96 },
//   validation: (Rule) => Rule.required(),
// },
{
  name: "description",
  title: "Description",
  type: "text",
  description: 'Short description of the food item',
},
{
  name: "price",
  title: "Current Price",
  type: "number",
  // validation: (Rule) => Rule.required().min(0),
},
{
  name: 'originalPrice',
  type: 'number',
  title: 'Original Price',
  description: 'Price before discount (if any)',
},
{
  name: 'tags',
  type: 'array',
  title: 'Tags',
  of: [{ type: 'string' }],
  options: {
    layout: 'tags',
  },
},
```

```

        description: 'Tags for categorization (e.g., Best Seller, Popular,
New)',
    },
    {
        name: 'image',
        type: 'image',
        title: 'Food Image',
        options: {
            hotspot: true,
        },
    },
    {
        name: "category",
        title: "Category",
        // type: "reference",
        type: "string",
        description:
            'Category of the food item (e.g., Burger, Sandwich, Drink, etc.)',
        // to: [{ type: "category" }],
    },
    {
        name: "available",
        title: "Available",
        type: "boolean",
        description: 'Availability status of the food item',
    },
    {
        name: "createdAt",
        title: "Created At",
        type: "datetime",
        initialValue: new Date().toISOString(),
    },
],
};

```

This schema featured fields such as name, slug, description, price, images, category, and availability status (inStock).

2. Modified Schema for Food:

The schema was adjusted to fit the given API specifications. The revised schema is now as follows:

```
export default {
  name: 'food',
  type: 'document',
  title: 'Food',
  fields: [
    {
      name: 'name',
      type: 'string',
      title: 'Food Name',
    },
    {
      name: 'category',
      type: 'string',
      title: 'Category',
      description:
        'Category of the food item (e.g., Burger, Sandwich, Drink, etc.)',
    },
    {
      name: 'price',
      type: 'number',
      title: 'Current Price',
    },
    {
      name: 'originalPrice',
      type: 'number',
      title: 'Original Price',
      description: 'Price before discount (if any)',
    },
    {
      name: 'tags',
      type: 'array',
      title: 'Tags',
      of: [{ type: 'string' }],
      options: {
```

```

        layout: 'tags',
      },
      description: 'Tags for categorization (e.g., Best Seller, Popular,
New)',
    },
    {
      name: 'image',
      type: 'image',
      title: 'Food Image',
      options: {
        hotspot: true,
      },
    },
    {
      name: 'description',
      type: 'text',
      title: 'Description',
      description: 'Short description of the food item',
    },
    {
      name: 'available',
      type: 'boolean',
      title: 'Available',
      description: 'Availability status of the food item',
    },
  ],
};

```

Key changes to the food schema include:

- **Category** field now accepts a string value instead of a reference type.
- Added the **originalPrice** field to capture the price before discounts.
- **Tags** field was introduced to add a categorization for products (e.g., Best Seller, Popular).
- **Food Image** field was made clearer by including an **image** type with hotspot options.
- **Available** field for availability status was added.

3. Chefs Schema (New Addition):

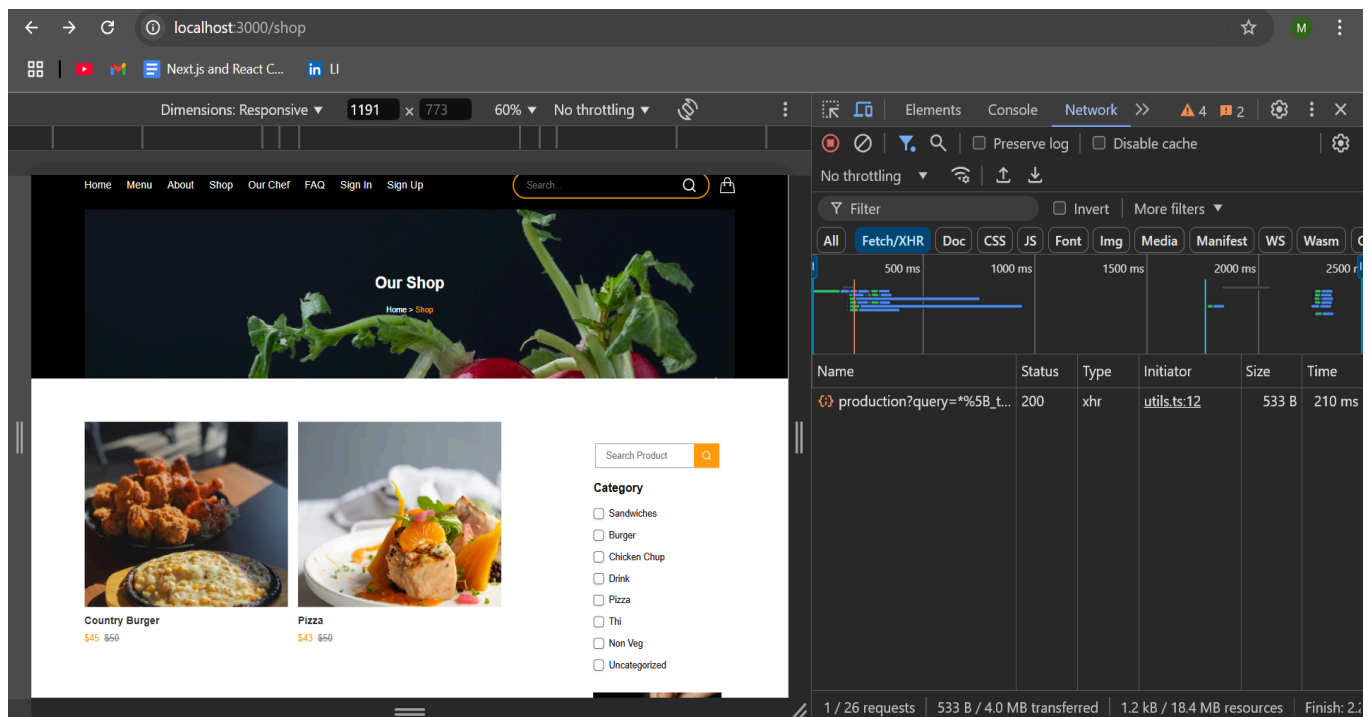
Since there was no schema for chefs before, a new `chefs.ts` schema has been created to align with the requirements.

Migration Steps

Steps:

- 1- Cloned the repository provided in the Day3 Document.
- 2- Installed dependencies in the repo directory using `npm install`.
- 3- Configured environment variables from my Sanity project (token with developer mode to read and write).
- 4- Run the following command to import sample data for Food and Chef models:
`npm run import-data`
- 5- Verified the data in Sanity project.

Screenshots



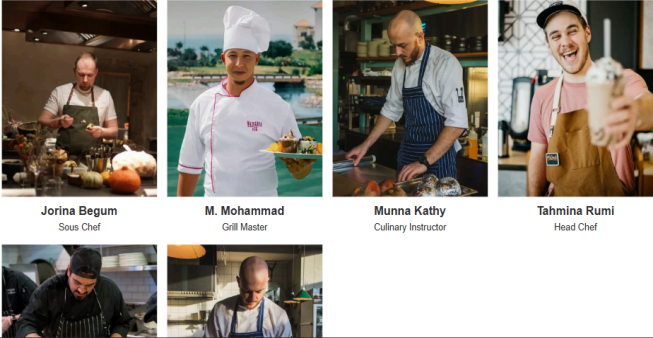
localhost:3000/ourchef


Next.js and React C...[in](#)

Dimensions: Responsive 1191 x 772 60% No throttling


Sign In Page

[Home](#) [Sign In](#)







Jorina Begum
Sous Chef



M. Mohammad
Grill Master



Munna Kathy
Culinary Instructor



Tahmina Rumi
Head Chef

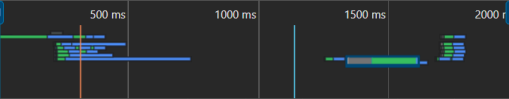
ElementsNetwork

Preserve logDisable cache

No throttling

Filter

AllFetch/XHRDocCSSJSFontImgMediaManifestWSWasm



Name	Status	Type	Initiator	Size	Time
production?query=%5...	200	xhr	utils.ts:29	1.2 kB	262 ms

1 / 24 requests | 1.2 kB / 2.2 MB transferred | 1.1 kB / 10.6 MB resources | Fir