Day 3 - API Integration Report – Foodtuck.com

Table of Contents:

1. API Integration Process

Overview

- **Objective:** Integrate the provided API into the **Next.js** frontend to dynamically fetch and display product data

.

- **API Used:** custom API.
- **Tools Used:**
- **Postman** for testing API endpoints.
- **Next.js** for frontend development.
- **Sanity CMS** for managing product data.

Steps Taken

1. API Understanding:

- Reviewed the API documentation to identify key endpoints (e.g., `/products`, `/categories`).
- Tested API endpoints using **Postman** to understand the response structure.

2. Frontend Integration:

- Created utility functions in **Next.js** to fetch data from the API.
- Integrated the API into the front end to display product listings, categories, and prices.
- Implemented error handling to manage API failures (e.g., network errors, invalid data).

3. Testing:

- Tested the API integration using **Postman** and browser developer tools.
- Simulated error scenarios (e.g., empty responses, slow network) to ensure robust error handling.

Challenges Faced

- **Schema Mismatch:** Initially, the API fields did not match the **Sanity CMS** schema. Adjusted the schema to align with the API data structure.
- **Error Handling:** Implemented fallback data and user-friendly error messages to improve the user experience.

2. Schema Adjustments

Original Schema

- The provided schema was basic, with fields like `name`, `price`, and `category`.

Improved Schema

- Added additional fields for better representation of product data:
- Applied validation rules:

- `description:` Detailed product description.
- `images:` Array of product images (limit of 5 images).
- `tags:` Array of tags for better searchability.
- `reviews:` Array of references to customer reviews.
- `name:` and `price` are required fields. - `price:` must be a positive number.
- `images:` are limited to 5 per product.

Schema Code

```
export default {
    name: 'food',
    type: 'document',
    title: 'Food',
    fields: [
      {
        name: 'name',
        type: 'string',
        title: 'Food Name',
      },
      {
        name: 'slug',
        type: 'slug',
        title: 'Slug',
        options: {
          source: 'name',
          maxLength: 96,
      },
      },
      {
        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',
    },
  ],
};
```

3. Data Migration

Migration Method

- Used **migration scripts** to fetch data from the custom API and populate **Sanity CMS** - Additional data was manually imported using **Sanity's** built-in import tools.

Steps Taken

1. Fetched data from the API using a script.
2. Transformed the data to match the **Sanity CMS** schema.
3. Imported the data into **Sanity CMS**
4. Verified that all fields were correctly populated.

Challenges Faced

- **Data Transformation:** Some API fields required transformation to match the schema (e.g., converting `product_title` to `name`).
- **Validation Errors:** Fixed validation errors during data import (e.g., missing required fields).

4. Error Handling

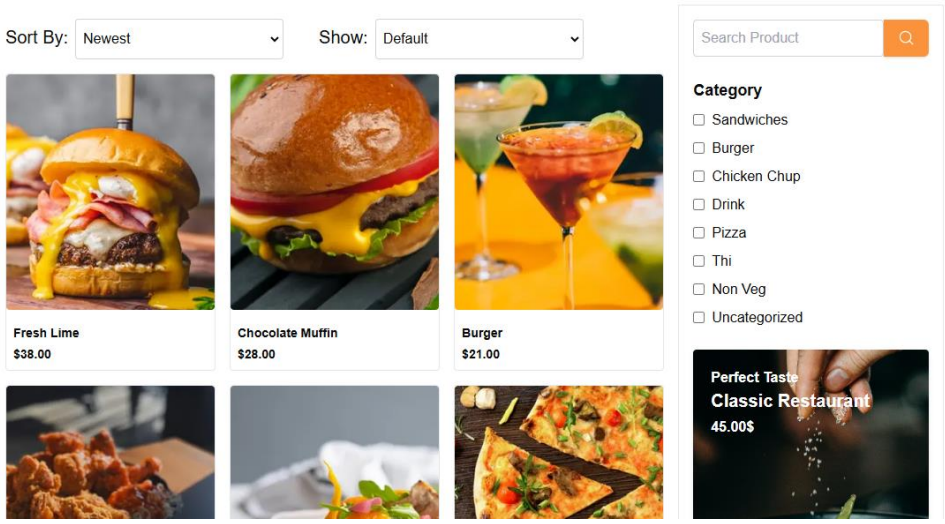
Implementation

- Added error handling for API calls:
- Displayed user-friendly error messages (e.g., "Failed to load products. Please try again later.").
- Used fallback data or skeleton loaders to improve the user experience during loading or errors.

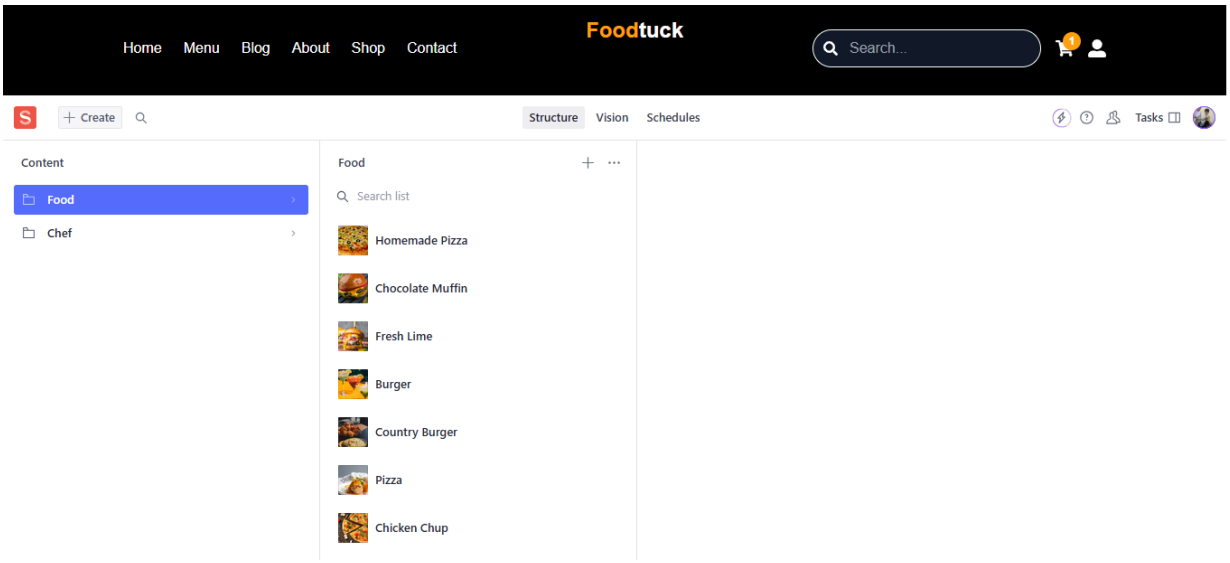
Error Scenarios Tested

- **API Downtime**: Displayed an error message when the API was unavailable.
- **Invalid Data**: Handled cases where the API returned incomplete or invalid data.

Frontend Data Display



Populated Sanity CMS Fields



5. Code Snippets

API Integration

```
// Utility function to fetch products export async function fetchProducts() {   try {     const response = await fetch('/api/products');
```

```
    if (!response.ok) throw new Error('Failed to fetch products');
return await response.json();
  } catch (error) {
console.error(error);
return [];
  }
}
```

Error Handling

```
// Display error message in UI function
ProductList() {   const [products, setProducts]
= useState([]);   const [error, setError] =
useState('');

  useEffect(() => {
fetchProducts()
      .then(setProducts)
      .catch(() => setError('Failed to load products. Please try again
later.'));
  }, []);

  if (error) return <div className="error">{error}</div>;   return
<div>{products.map((product) => <ProductCard key={product.id}
{...product} />)}</div>;
}
```

6. Best Practices Followed

- Used `.env` files to store sensitive data like API keys.
- Followed clean coding practices (e.g., modular functions, descriptive variable names).
- Documented every step of the process for future reference.
- Used version control (Git) to track changes and tag milestones.

7. Conclusion

- Successfully integrated the API into the **Next.js** frontend.
- Migrated data into **Sanity CMS** and adjusted the schema for better compatibility.
- Implemented robust error handling to ensure a smooth user experience.
- Prepared for submission with detailed documentation, screenshots, and code snippets.