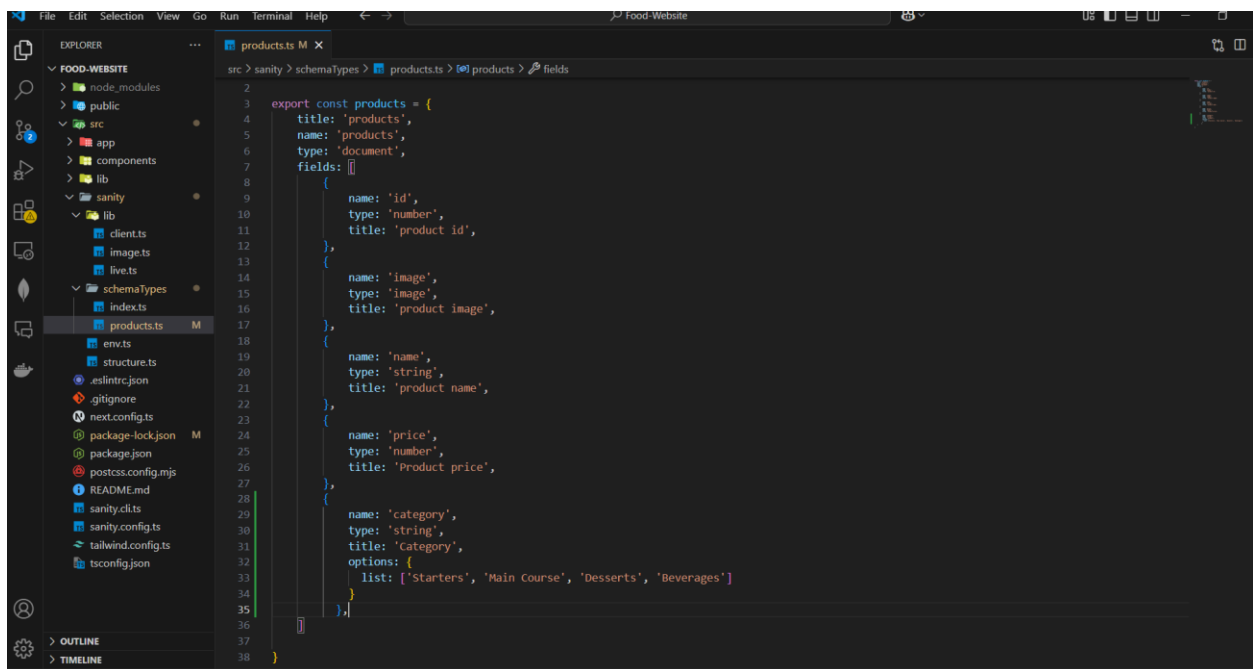


Day 3: API Integration and Data Setup for Food Tunk Restaurant Website

This documentation outlines the work completed on Day 3 of the Food Tunk website development. It covers API integration with Sanity, creating a custom schema for restaurant data, and using GROQ queries to display the content in a Next.js application. Since data is manually set in the Sanity dashboard, this guide focuses on setting up the schema and querying data rather than data migration.

Custom Schema Setup in Sanity CMS

The custom schema defines how restaurant data, such as menu items, specials, and location details, are structured in the Sanity CMS. Here's an example of how to set up a schema for the "Product Item" data:

A screenshot of a code editor (VS Code) showing the Sanity schema setup for 'Product Item' data. The file explorer on the left shows the project structure, with 'products.ts' selected under 'schemaTypes'. The main editor displays the following TypeScript code:

```
2 export const products = {
3   title: 'products',
4   name: 'products',
5   type: 'document',
6   fields: [
7     {
8       name: 'id',
9       type: 'number',
10      title: 'product id',
11    },
12    {
13      name: 'image',
14      type: 'image',
15      title: 'product image',
16    },
17    {
18      name: 'name',
19      type: 'string',
20      title: 'product name',
21    },
22    {
23      name: 'price',
24      type: 'number',
25      title: 'Product price',
26    },
27    {
28      name: 'category',
29      type: 'string',
30      title: 'Category',
31      options: {
32        list: ['Starters', 'Main Course', 'Desserts', 'Beverages']
33      },
34    },
35  ],
36 },
```

Custom Validation:

The schema includes validation rules to ensure data integrity. For example: Price must be a positive number. Title and description cannot be empty.

Sanity API Setup:

The code connects to Sanity's API using a configured dataset and project ID. It authenticates using an API token. Environment variables (e.g., project ID, dataset) are used for security

Fetching Data from Sanity:

GROQ queries are used to fetch structured content from the Sanity CMS. These queries retrieve restaurant-related data such as menu items, categories (e.g., appetizers, main courses, desserts, beverages), prices, descriptions, and other relevant details.

Example Query:

A typical query fetches data for restaurant menu items, including dish names, descriptions, prices, images, and categories.

Mapping and Formatting:

Once the data is fetched, it is mapped to align with the website's schema. Each record, such as a menu item or restaurant information, is restructured to match the front-end display requirements of the "Food Tunk" application. For example, each menu item's title, description, price, and image will be organized in a way that can be easily rendered by the website.

Displaying Data:

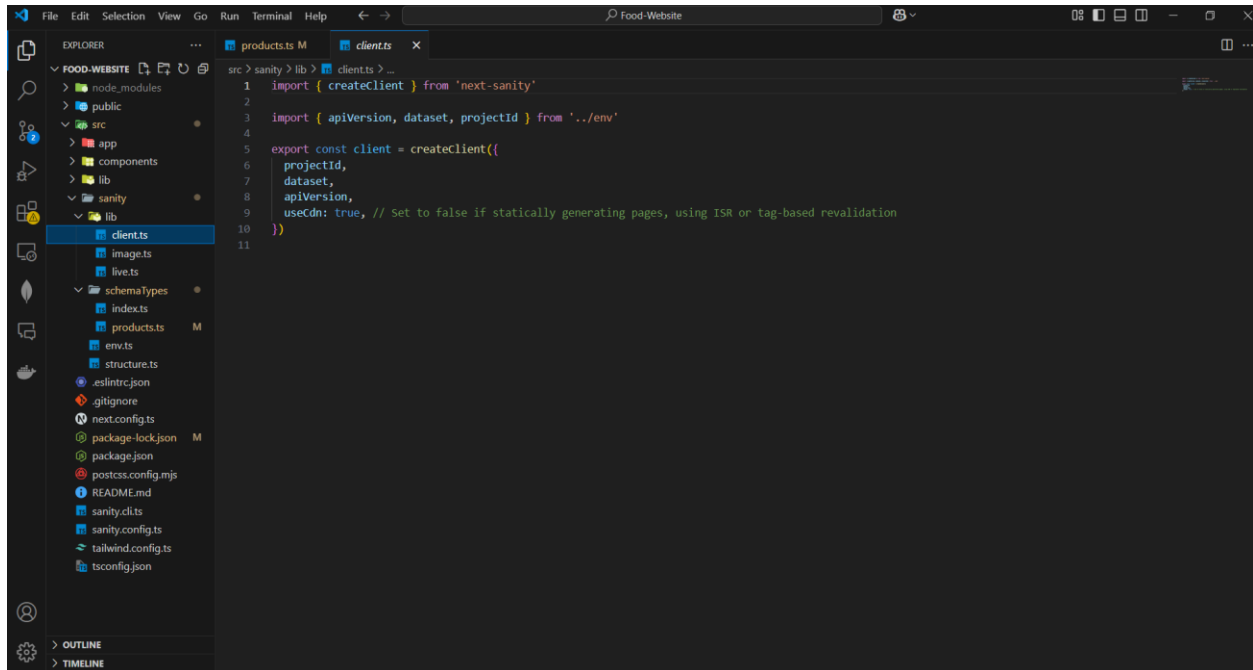
The structured data fetched from Sanity is displayed on the restaurant website dynamically. The data includes:

- Menu items displayed in categories (e.g., starters, mains, desserts).

No need for database saving or direct API insertion since the data is pulled live from Sanity through GROQ queries for real-time updates.

Client Page Code:

This is the client-side code for rendering the Food Tunk data in a Next.js page. Here's how the code works:



GROQ Query to Fetch Data:

The `getServerSideProps` function uses a GROQ query to fetch data from Sanity during server-side rendering (SSR). This ensures that data is pre-fetched and injected into the page before the user sees it.

Rendering Items:

The `ClientPage` component renders the list of items passed from props. React's `.map()` method iterates over the fetched data, creating a list of Product cards.

Dynamic Routing:

The code includes dynamic routing links for individual Product items. Clicking an item navigates the user to a detailed page (e.g., `/product/[id]`).

Code Highlights:

SSR Optimization: Server-side rendering ensures faster loading times and better SEO.

Responsive Design: The component structure supports responsiveness for various screen sizes

GROQ Query ability:

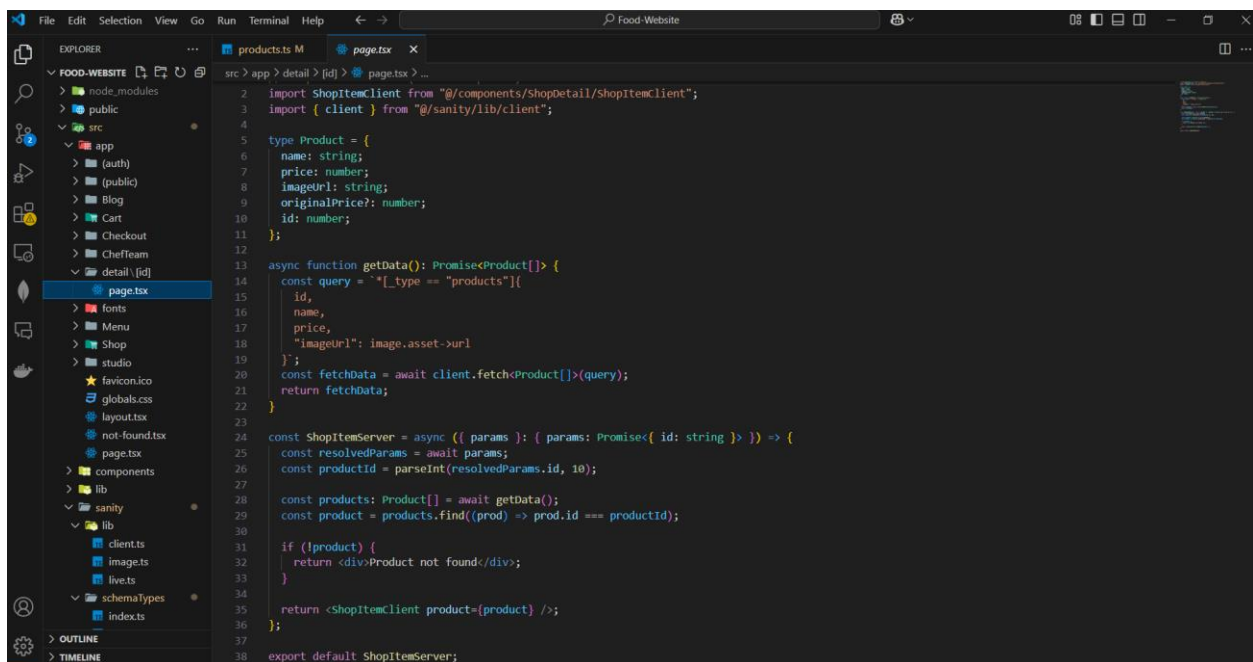
The schema design ensures that all fields are easily accessible and queryable using GROQ in Next.js.

Scalability:

Additional fields like tags or categories can be added as needed to accommodate new features.

Product Card Code:

This code represents the design and functionality of a single product card. It's used within the client page to display individual products.

A screenshot of a code editor window titled 'Food-Website'. The left sidebar shows a file explorer with a project structure including folders like 'node_modules', 'public', 'src', 'components', 'lib', 'sanity', and 'schemaTypes'. The main editor area displays the code for 'page.tsx'. The code defines a 'Product' type with fields like 'name', 'price', 'imageUrl', 'originalPrice', and 'id'. It includes an async function 'getData()' that fetches product data from a Sanity API using GROQ. The 'ShopItemServer' component is defined as an async function that takes 'params' and returns a 'ShopItemClient' component. The 'ShopItemClient' component is a React component that displays product details dynamically based on the props passed to it.

```
1 import ShopItemClient from "@components/ShopDetail/ShopItemClient";
2 import { client } from "@sanity/lib/client";
3
4
5 type Product = {
6   name: string;
7   price: number;
8   imageUrl: string;
9   originalPrice?: number;
10  id: number;
11 };
12
13 async function getData(): Promise<Product[]> {
14   const query = "[*_type == 'products']{
15     id,
16     name,
17     price,
18     'imageUrl': image.asset->url
19   }";
20   const fetchData = await client.fetch<Product[]>(query);
21   return fetchData;
22 }
23
24 const ShopItemServer = async ({ params }: { params: Promise<{ id: string }> }) => {
25   const resolvedParams = await params;
26   const productId = parseInt(resolvedParams.id, 10);
27
28   const products: Product[] = await getData();
29   const product = products.find((prod) => prod.id === productId);
30
31   if (!product) {
32     return <div>Product not found</div>;
33   }
34   return <ShopItemClient product={product} />;
35 };
36
37
38 export default ShopItemServer;
```

Props Destructuring:

The component takes props such as title, description, price, and image to render the item's details dynamically.

Design and Styling:

Styled using CSS classes or Tailwind (depending on the implementation). Ensures responsiveness and accessibility.

Dynamic Features:

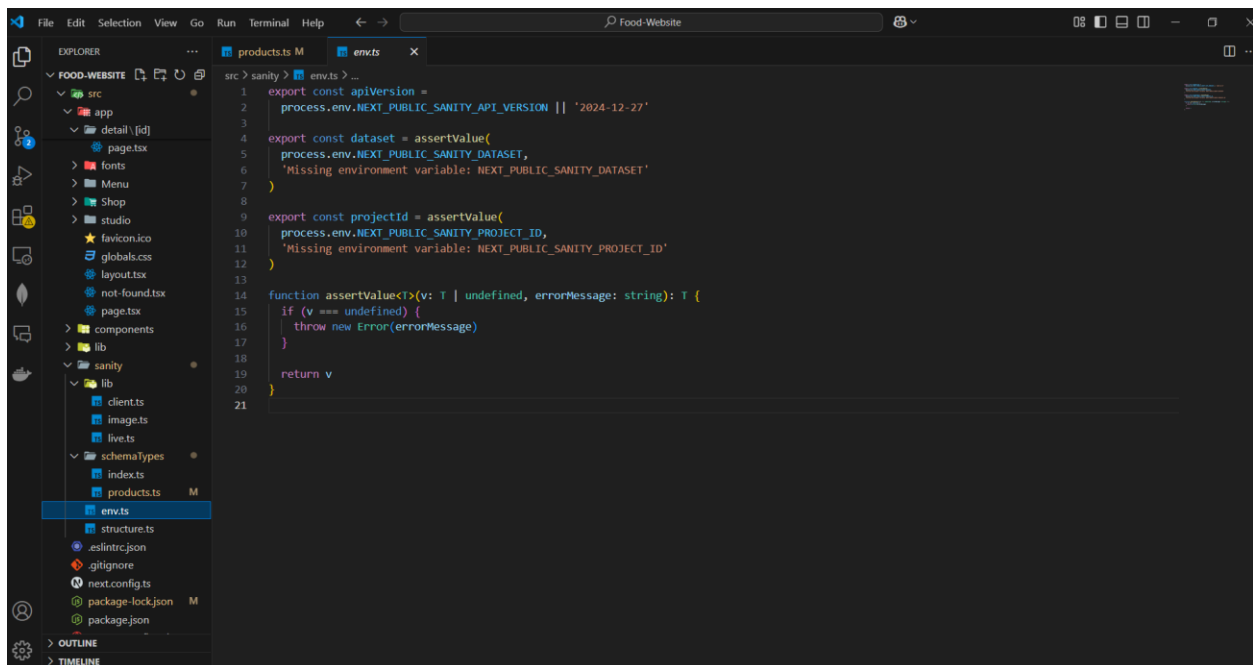
Includes a button for adding the item to the cart or viewing more details. Optimized image rendering using libraries like next/image.

Reusability:

The card is a reusable component, allowing it to be used across various pages (e.g., homepage, category pages).

Environment Variables:

The .env file includes sensitive configurations for the Food Tunk application. Key entries:

A screenshot of a code editor interface. The Explorer panel on the left shows a file tree for 'FOOD-WEBSITE' with folders like 'src', 'components', 'lib', and 'sanity'. The 'env.ts' file is selected. The main editor shows the content of 'env.ts', which includes configuration for Sanity, such as 'apiVersion', 'dataset', and 'projectId', along with an 'assertValue' utility function. The code is as follows:

```
1 export const apiVersion =  
2   process.env.NEXT_PUBLIC_SANITY_API_VERSION || '2024-12-27'  
3  
4 export const dataset = assertValue(  
5   process.env.NEXT_PUBLIC_SANITY_DATASET,  
6   'Missing environment variable: NEXT_PUBLIC_SANITY_DATASET'  
7 )  
8  
9 export const projectId = assertValue(  
10  process.env.NEXT_PUBLIC_SANITY_PROJECT_ID,  
11  'Missing environment variable: NEXT_PUBLIC_SANITY_PROJECT_ID'  
12 )  
13  
14 function assertValue<T>(v: T | undefined, errorMessage: string): T {  
15   if (v === undefined) {  
16     throw new Error(errorMessage)  
17   }  
18  
19   return v  
20 }  
21
```

Sanity Configuration:

SANITY_PROJECT_ID: The unique identifier for the Sanity project.

SANITY_DATASET: Specifies the dataset (e.g., production or development).

API Security:

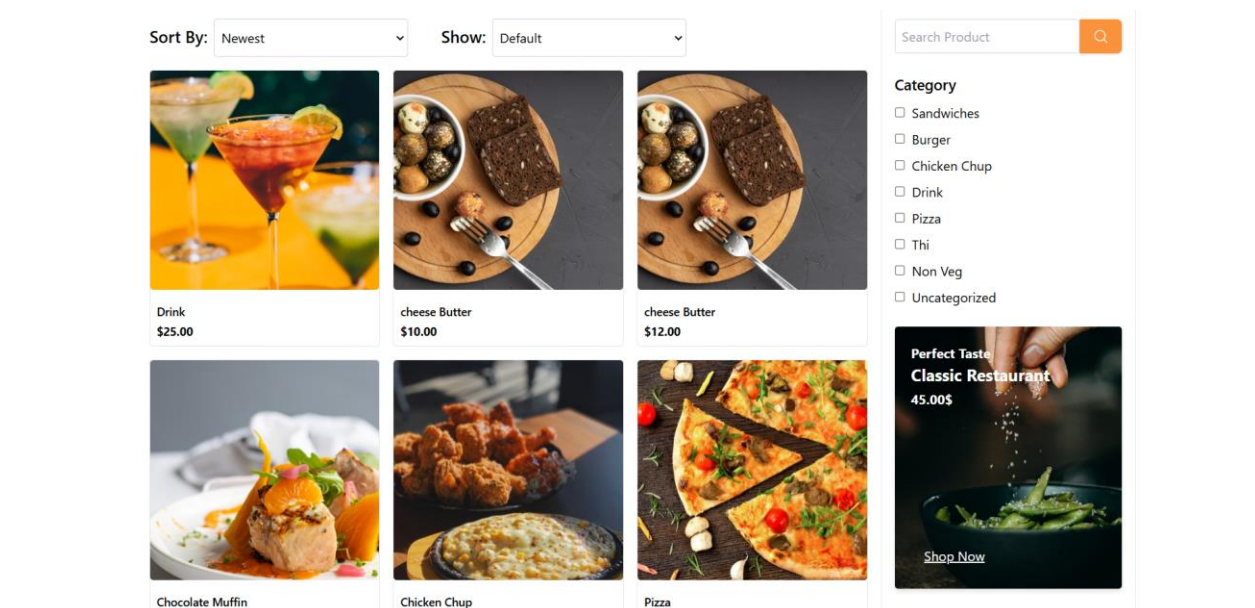
SANITY_API_TOKEN: A secure token used to authenticate API calls to Sanity. It should never be exposed to the client.

Database and API: Other environment variables for database connections and backend endpoints might also be included

Security Notes:

Environment variables are stored securely and accessed using process.env. They are not exposed in the frontend.

Frontend Outcome Overview:



Conclusion:

On Day 3, the focus was on setting up the **Food Tunk** website's backend and integrating dynamic data from Sanity. The key achievements were:

- **Schema Setup:** Created a structured schema in Sanity for Product items .
- **Data Entry:** Manually added data through the Sanity dashboard.

- **Data Fetching:** Used GROQ queries to fetch and display data dynamically on the frontend.
- **Responsive Display:** Designed a mobile-friendly, attractive menu and restaurant info layout.
- **Secure Configuration:** Managed API keys and sensitive data securely with environment variables.

This work laid the foundation for a dynamic, easy-to-update restaurant website that can be easily extended in the future.