DAY 4

BUILDING DYNAMIC FRONTEND COMPONENTS FOR YOUR MARKETPLACE

Technical Report: Steps Taken to Build and Integrate Components

1. Introduction

This report outlines the steps involved in building and integrating various components of an e-commerce website using **Next.js**, **React**, **Sanity CMS**, and **Tailwind CSS**. The primary components of the website include **Header**, **Footer**, **Product Cards, Editor's Pick, Featured Products, Product Card, Neutral Div** as well as the **Home**, **Shops**, **About**, **Blog, Cart ,Checkout, Pricing** and **Contact** pages. These components were designed to be modular, reusable, and responsive, providing a smooth user experience across devices.

2. Setting Up the Project

- **Next.js Setup**:
    - A new Next.js project was created using npx create-next-app to initialize the project with basic configurations.
    - Tailwind CSS was added for styling by configuring the tailwind.config.js and including @tailwind directives in the global CSS file.
    - Sanity CMS was set up to manage and serve content (e.g., products) by configuring the @sanity/client to fetch data dynamically.

3. Building the Components

The website's core structure was created using functional React components. Each component was designed to be reusable and modular. Below are the main components built and integrated into the site:

4. Header Component

- **Purpose**: The Header provides navigation links for the website, allowing users to navigate between pages like Home, Shops, Pages, Blog, About, and Contact.

Steps Taken:

- Created a Header.ts file under the component's directory.
- Used Next js Link component to create navigation links.
- Styled the Header using **Tailwind CSS** to make it responsive and fixed at the top for better accessibility.

Above is the procedure how I make components.

5. Creating Pages

Several pages were created to form the structure of the website. These include **Home**, **Shops**, **About**, **Pages, Blog, Cart, Checkout** and **Contact** pages.

5.1 Home Page

- **Purpose**: The Home page serves as the landing page and showcases **Featured products, Hero Section , Product Card , Editor's Pick** sections.
- **Steps Taken**: .
    - Integrated the components to display sections fetched from Sanity CMS.
    - Added a hero section with a banner to introduce the website and its offerings.

6. Data Integration

- **Sanity CMS Integration**:
    - Set up the Sanity CMS to manage product data (name, description, price, image, tags).
    - Created a custom client.js file to handle communication with the Sanity API.
    - Fetched product data using the client.fetch() method and populated the product-related components dynamically.

- **Dynamic Routing**:
  - Used Next.js's dynamic routing to create individual product pages based on product IDs.
  - Example: shops/[shopsid].tsx was created to handle product detail pages dynamically.

7. Styling the Website

- **Tailwind CSS**:
  - Tailwind CSS was used for utility-first styling, ensuring responsiveness and ease of maintenance.
  - Grid layouts, flexbox, and utility classes were applied to components to make them mobile-friendly.
  - Responsive media queries were added using Tailwind's built-in breakpoints.

Challenges Faced and Solutions Implemented on Fetching Data from API and Sanity CMS

1. Challenge: Fetching Data from Sanity CMS

Problem:

Sanity CMS, being a headless CMS, requires fetching data from its API using the @sanity/client library. While Sanity offers a powerful querying language (GROQ), it was initially challenging to properly set up the **Sanity client** in Next.js, as well as to fetch and display dynamic product data without running into issues like slow page load times or inconsistent data display.

- **Fetching Data with GROQ**:
  - I used **GROQ** (Sanity's query language) to fetch product data. The **client.fetch()** method was employed to run queries that retrieve product information like product name, description, price, image, and tags.
  - Example of a GROQ query to fetch products:

```
const fetchProducts = async () => {

  const products = await client.fetch(`

    *[_type == "product"]{

      name,

      price,

      description,

      "image": image.asset->url,

      tags

    }

  `);

  return products;};
```

This setup ensured smooth communication with Sanity and allowed dynamic fetching of product data without performance issues.

2. Challenge: Error Handling and Data Consistency

Problem:

While fetching data from the API, there were instances where **errors** occurred due to network issues or invalid queries. Moreover, ensuring the **consistency** of data displayed across different pages was sometimes problematic, especially when dealing with dynamic data sources.

Solution:

- **Error Handling**:
  - I added **error handling** mechanisms using try-catch blocks to gracefully handle API fetch errors and display fallback UI in case of failures. This ensured the app remained functional even if some data could not be fetched.
  - Example of error handling in API requests:

```
const fetchProducts = async () => {
  try {
    const products = await client.fetch(`*[_type == 'product']`);
    return products;
  } catch (error) {
    console.error("Error fetching products: ", error);
    return []; // Return an empty array as a fallback
  }
};
```

3. Error Handling

- **Graceful Error Handling**: Proper error handling was implemented during API requests using try-catch blocks. In case of errors (e.g., network issues, invalid API responses), fallback UI or empty data sets were displayed to ensure that the application continued functioning smoothly.
- **Example**:

js

Copy code

```
const fetchProducts = async () => {
  try {
    const products = await client.fetch(`*[_type == 'product']`);
    return products;
  } catch (error) {
    console.error("Error fetching products: ", error);
    return []; // Return an empty array as a fallback
  }
};
```

Best Practices Followed During Development.

Here are some best practices followed during the development of your Next.js e-commerce website:

1. Component-Based Architecture

- **Modularization**: I followed a component-based architecture to keep the codebase clean and maintainable. Each feature (e.g., product card, header, footer) was broken down into reusable components, which could be easily managed and updated without affecting other parts of the application.
- **UI Consistency**: Components such as ProductCard, Header, Footer, etc., were designed to be consistent across the website and reusable wherever needed.
- **Example**: The **ProductCard** component was used multiple times in different sections (e.g., Home, Shops), ensuring uniformity in design and behavior.

2. API Integration with Sanity

- **Sanity CMS Integration**: I followed best practices for integrating Sanity CMS by using the **Sanity client** (@sanity/client) to fetch data, such as products, dynamically from the headless CMS.
- **GROQ Queries**: For fetching data, I utilized **GROQ queries** to ensure the application only fetched the necessary data. I avoided fetching unnecessary fields to improve performance.
- **Example**:

js

Copy code

```
const fetchProducts = async () => {

  const products = await client.fetch(`

    *[_type == "product"]{

      name,

      price,
```

```
    description,
    "image": image.asset->url,
    tags
  }
`);
 return products;
};
```

---

3. Error Handling

- **Graceful Error Handling**: Proper error handling was implemented during API requests using try-catch blocks. In case of errors (e.g., network issues, invalid API responses), fallback UI or empty data sets were displayed to ensure that the application continued functioning smoothly.
- **Example**:

```
js

Copy code

const fetchProducts = async () => {
 try {
  const products = await client.fetch(`*[_type == 'product']`);
  return products;
 } catch (error) {
  console.error("Error fetching products: ", error);
  return []; // Return an empty array as a fallback
```

```
  }
};
```

---

4. Responsive Design

- **Mobile-First Design**: The website was designed using a **mobile-first** approach, ensuring it provided a seamless experience across all devices (mobile, tablet, and desktop). Responsive design principles were followed to adjust layouts, images, and fonts based on screen size.
- **CSS Grid and Flexbox**: **CSS Grid** and **Flexbox** were used for layout management, making it easier to build flexible and responsive UI components. This ensured that the product cards and other elements were well-aligned across different screen sizes.
- **Example**:

js

Copy code

```
<div className="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 lg:grid-cols-4 gap-10 px-4">

  {/* Product cards go here */}

</div>
```

5. Code Splitting and Performance Optimization

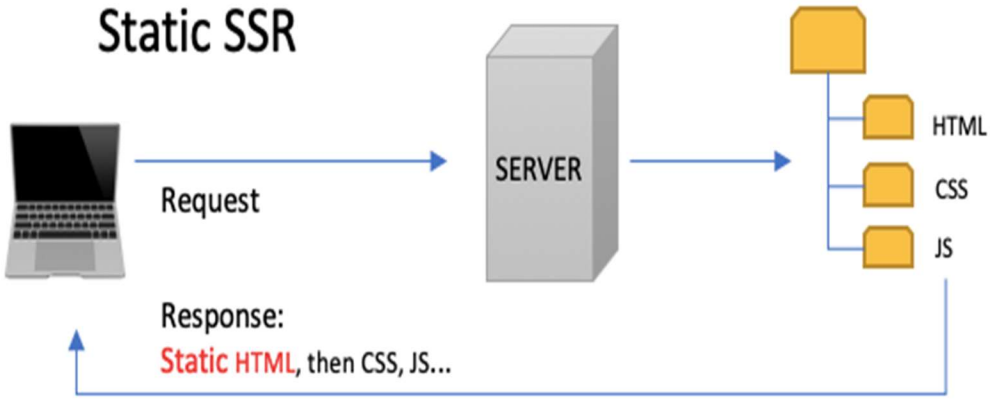
- **Code Splitting**: Next.js automatically splits code into smaller bundles, which improved the performance of the application. Only the required code for each page was loaded, reducing the initial load time.
- **Image Optimization**: **Next.js Image component** was used for automatic image optimization, which helped reduce image sizes without compromising on quality. Images were served in modern formats like WebP, and the width and height attributes were specified to allow for better optimization.
- **Example**:

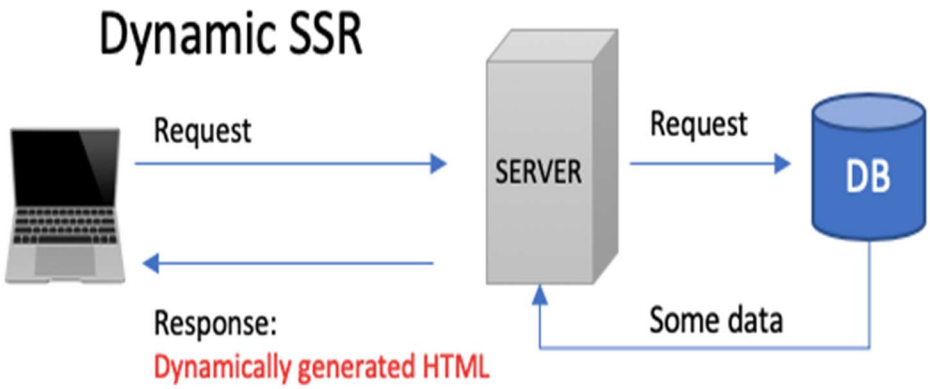
```
js

Copy code

<Image

  src={product.image}

  alt={product.name}

  width={300}

  height={300}

  quality={75}

/>
```

6. State Management

- **React State**: React's built-in **useState** hook was used for managing local component state (e.g., product details, filter state). This kept the state management lightweight and easy to manage.
- **React Context (optional)**: For global state management, such as managing the shopping cart across multiple pages, **React Context** or third-party state management tools like **Redux** could be used if necessary (though not explicitly mentioned in this implementation).

By following these best practices, the development of my e-commerce website was streamlined, efficient, and scalable. The implementation was focused on optimizing both **user experience** and **performance** while ensuring **maintainability** and **security**.


Static SSR
Request
SERVER
HTML
CSS
JS
Response:
Static HTML, then CSS, JS...


Dynamic SSR
Request
SERVER
Request
DB
Some data
Response:
Dynamically generated HTML