Day 5-Testing and Backend Refinement- Quick Commerce Coffee Shop

Project Overview

The **Quick Commerce Coffee Shop** is a coffee delivery service built during a week long hackathon. This app enables customers to order coffee and related products with an optimized, user-friendly experience. On **Day 5**, we focused on refining the backend integration, implementing robust error handling, and optimizing assets for improved performance. This ensures a smoother user journey, quicker load times, and better reliability, both in terms of frontend and backend operations.

Main Objectives

1. Error Handling

Error handling is crucial for any application to ensure that users are not left confused or frustrated during failures. During this stage of the project, we integrated error handling mechanisms in both the frontend and backend.

Frontend Error Handling

We incorporated error handling in the FeaturedProducts component to ensure that if data fetching fails, users receive meaningful feedback instead of a broken or empty page.

Code Example:

```
const FeaturedProducts: React.FC = () => {
  const [products, setProducts] = useState<Product[]>([]);
  const [loading, setLoading] = useState<boolean>(true);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        setLoading(true);
        setError(null);
        const query = `*[_type == "product"][0...8] {
          _id,
          title,
          price,
          "productImage": productImage.asset->url,
          description,
          tags,
          dicountPercentage,
          isNew
        }`;
        const data: Product[] = await client.fetch(query);
        setProducts(data);
      } catch (err) {
        console.error("Error fetching products:", err);
        setError("Failed to load products. Please try again later.");
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, []);

  if (loading) {
    return <div className='text-black'>Loading featured
products...</div>;
  }

  if (error) {
    return <div className="text-red-500">{error}</div>;
  }
```

- Error State: If the `fetch` request fails (due to network issues or a server error), the app shows a helpful error message.
- Loading State: A loading spinner or text is shown while data is being fetched.

2. Backend Integration

The backend of this project integrates with Sanity CMS for fetching dynamic product data. We've made refinements to ensure smooth communication between the frontend and the backend, including proper error handling during data retrieval.

Product Fetching and Backend Error Handling

Backend error handling ensures that the app behaves predictably even if there is a failure in the API requests. For instance, if the product data fails to load, it triggers an error and notifies the user.

Code Example:

```
 const fetchProduct = async () => {
      try {
        setLoading(true);
        setError(null);
        const query = `*[_type == "product" && _id == $id][0] {
          _id,
          title,
          price,
          "productImage": productImage.asset->url,
          description,
          tags,
          dicountPercentage,
          isNew
        }`;
        const data: Product | null = await client.fetch(query, { id:
params.id });
        if (!data) {
          throw new Error("Product not found");
        }
        setProduct(data);
      } catch (err) {
        console.error("Error fetching product details:", err);
        setError("Failed to load product details. Please try again
later.");
      } finally {
        setLoading(false);
      }
    };
```

- Backend Failures: If there's an issue with Sanity's API (e.g., downtime or incorrect data), an error is logged and the user is shown a message informing them of the failure.

- Optimized Queries: We ensure the queries are optimized, and the system gracefully handles the absence of data or other anomalies.

3. Optimization of Assets

To improve performance, assets like images, CSS, and scripts are optimized to reduce loading times and enhance the overall user experience. Some key optimizations include:

Lazy Loading Images

Images are lazily loaded to ensure that only the necessary images are loaded initially, which improves the page load time. We use Next.js's `next/image` component for efficient image handling, including automatic lazy loading.

Code Example:

```
<div className="search-result-image">
                <Image
                  src={product.productImage.asset.url}
                  alt={product.title}
                  width={50}
                  height={50}
                  className="search-result-product-image"
                  priority // Optimize loading
                />
              </div>
```

- Lazy Loading: Images are loaded only when they come into the viewport, reducing the initial load time and conserving bandwidth.
- Priority Images: For critical images (such as featured products), the `priority` attribute is used to load them immediately.

Debounced Search Input

The search functionality uses debouncing to delay the API call until the user stops typing, which reduces unnecessary calls to the backend.

Code Example:

```
useEffect(() => {
    const debounceTimeout = setTimeout(() => searchProducts(), 300);
    return () => clearTimeout(debounceTimeout);
  }, [searchQuery, searchProducts]);
```

- Debounced API Requests: By waiting for the user to stop typing, we prevent multiple rapid requests to the backend, optimizing server load and reducing the potential for unnecessary data fetching.

Server-Side Rendering (SSR)

Data for products is fetched on the server side to ensure that the page is populated with content before it is delivered to the user, improving performance and SEO.

Example:

```
export async function getServerSideProps() {

  const products = await client.fetch(`*[_type == "product"]`);

  return {

    props: {

      products,

    },

  };

}
```

- SEO Benefits: SSR ensures that search engines can index content properly, boosting visibility.
- Faster Initial Load: SSR helps with faster initial page loads by delivering pre-rendered content.

Code Splitting

We implemented code splitting to reduce the initial JavaScript bundle size. Non-essential components are loaded asynchronously only when needed.

```
const Cart = dynamic(() => import('./Cart'), { ssr: false });
```

- Lazy Load Non-Critical Components: Cart components and other parts of the application are only loaded when necessary, improving the first-page load performance.

4. Cross-Browser/Device Compatibility Verification

We ensured that the application is fully responsive and works seamlessly across all major browsers and devices.

Responsive Design

Using Tailwind CSS, we made sure the layout adapts fluidly to different screen sizes, ensuring a mobile-friendly experience.

```
@media (max-width: 768px) {
  .grid {
    grid-template-columns: 1fr;
  }
}
```

- Mobile Optimization: The grid and layout automatically adjust to fit smaller screens, providing a better experience on mobile devices.
- Browser Compatibility: The application has been tested across multiple browsers like Chrome, Firefox, and Safari, ensuring consistent behavior.

Conclusion

The Quick Commerce Coffee Shop project now includes robust error handling, optimized backend integration, and asset optimization to enhance performance. These refinements help ensure that the application remains responsive, efficient, and user-friendly under various conditions. By addressing backend challenges, optimizing content delivery, and maintaining compatibility, we have created a reliable and high-performance platform for quick commerce.