DAY05 &06 MIGRATION API SANITY:

Error Handling in Q-commerce Marketplace Project

Introduction

In this document, we outline common errors that might arise in the development of the Food Restaurant Marketplace project and strategies to handle them effectively.

---

1. Client-Side Errors

1.1. Missing or Incorrect State Updates

Error: State variables like cart, wishlist, or userInfo not updating as expected.

Cause:

- Incorrect useState or useEffect logic.
- Local storage data being fetched asynchronously but the state trying to render synchronously.

Solution:

- Ensure state updates are done properly using useEffect for side effects.
- Always check if the data is available before rendering the component.

```
useEffect(() => {
  const savedCart = JSON.parse(localStorage.getItem("cart") || "[]");
  setCart(savedCart);
}, []);
```

1.2. Missing Icons or Broken Components

Error: Icons not appearing on the page (e.g., Cart, Wishlist, User icons).

Cause:

- Incorrect import or missing dependencies for icon libraries like lucide-react.
- File path errors for assets.

Solution:

- Double-check imports and ensure proper installation of required libraries.

npm install lucide-react

- Make sure to use the correct path for components or assets.
- Test with simple static components to confirm the issue.

---

2. Navigation and Routing Errors

2.1. Incorrect Routing for Pages

Error: Clicking on links (e.g., Cart, Wishlist) leads to 404 or broken routes.

Cause:

- Misconfiguration of routes in Next.js.
- Incorrect pathing in Link components.

Solution:

- Ensure routes are correctly set up in the pages folder.
- Double-check the href attributes in the Link components.

<Link href="/cart">Cart</Link>

2.2. Incorrect Page Display or Content Not Loading

Error: Content or pages not loading, or showing incorrect data.

Cause:

- Missing or incorrect data fetching logic.
- Misconfigured page routing (_app.tsx).

Solution:

- Add error boundaries in your components to catch rendering errors.

```
import React, { ErrorInfo } from "react";

class ErrorBoundary extends React.Component {
  state = { hasError: false };

  static getDerivedStateFromError(error: Error) {
    return { hasError: true };
  }

  componentDidCatch(error: Error, errorInfo: ErrorInfo) {
    console.log(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

export default ErrorBoundary;

---

3. Authentication and Authorization Errors

3.1. User Login/Signup Failure

Error: Login or signup not working as expected (e.g., invalid credentials or no data persistence).

Cause:

- Missing API calls or incorrect response handling for login/signup.
- Incorrect usage of local storage or session management.

Solution:

- Verify that API calls are correctly made and handled with proper status codes.
- Use localStorage or cookies to store session data and validate user sessions.

```
const handleLogin = async () => {
  const response = await fetch("/api/login", { method: "POST", body: JSON.stringify(credentials) });
  const data = await response.json();
  if (data.success) {
    localStorage.setItem("user", JSON.stringify(data.user));
    // redirect to dashboard or home page
  } else {
    alert("Login failed");
  }
};
```

3.2. User Logout Doesn't Clear Session

Error: User remains logged in even after hitting the logout button.

Cause:

- Session or local storage data not cleared on logout.

Solution:

- Ensure localStorage.removeItem or cookie clearing happens during the logout process.

```
const handleLogout = () => {
  localStorage.removeItem("user");
  window.location.href = "/login";
};
```

---

4. API Integration Errors

4.1. API Data Fetch Failures

Error: Data fetching errors (e.g., fetching cart, wishlist, or product details) due to incorrect API calls.

Cause:

- Incorrect API endpoint or network issues.
- Improper response handling or missing error handling for failed API requests.

Solution:

- Always check the API response status code and handle errors gracefully.
- Implement a fallback UI for errors (e.g., loading spinners or error messages).

```
const fetchData = async () => {
  try {
    const response = await fetch("/api/cart");
    if (!response.ok) {
      throw new Error("Failed to fetch data");
    }
    const data = await response.json();
    setCart(data);
  } catch (error) {
    console.error(error);
    setError(true);
  }
};
```

4.2. Missing or Incorrect API Response Data

Error: Data is missing or incorrectly displayed in the UI due to issues with API response data format.

Cause:

- API response format changed or incorrect data mapping in the frontend.

Solution:

- Always log API responses to verify the data structure.
- Use TypeScript interfaces or PropTypes to ensure correct data types are used.

```
interface CartItem {
  id: string;
  name: string;
```

```
  quantity: number;
}

const [cart, setCart] = useState<CartItem[]>([]);
```

---

5. User Interface and Styling Errors

5.1. Misaligned UI Elements or Missing Styles

Error: UI components are misaligned, or some styles aren't being applied.

Cause:

- Missing CSS classes or incorrect Tailwind CSS usage.
- Global styles not properly loaded.

Solution:

- Inspect the page using browser developer tools to identify missing classes or styles.
- Make sure you have configured Tailwind correctly, especially if you are using custom configurations.

npm install tailwindcss

- Make sure your custom styles are scoped properly, and if using CSS modules, check for className conflicts.

---

6. Performance Issues

6.1. Slow Page Load or Poor Performance

Error: Page load times are slow, affecting user experience.

Cause:

- Heavy JavaScript or large images being loaded.

- Unoptimized data fetching or unnecessary re-renders.

Solution:

- Lazy load images and components wherever possible.
- Use React.memo to optimize re-renders of components.
- Optimize API calls to avoid unnecessary requests.
