FSD ASSIGNMENT-1

22AIE457

NAME: AKILESH RAO S

Roll No:CH.EN.U4AIE21004

E-Commerce Product Listing Page

1. **Introduction**

This report outlines the methodology for creating and optimizing a React-based e-commerce product listing page. It covers fetching and displaying product data, implementing search and filter functionalities, using React Router for navigation, styling with Styled-Components, and optimizing image loading.

2. **Methodology**

2.1 ***Fetching and Displaying Products***

Objective: Fetch a list of products from an API and display them on a React component.

Steps:

1. Setup State Management: Use useState to manage the product data.
2. Fetch Data: Utilize useEffect to perform an API call when the component mounts. Libraries such as Axios or the native fetch API can be used.
3. Handle Responses: Process the API response and update the state.
4. Error Handling: Implement error handling to manage any issues during the fetch operation.
5. Conditional Rendering: Display loading indicators, error messages, or the product list based on the current state.

Workflow:

- Initialize state with useState.
- Perform fetch operation within useEffect.
- Update state with fetched data or handle errors.
- Render UI based on state (loading, error, or data).

2.2 ***Implementing Search and Filter Functionalities***

Objective: Allow users to search for products and filter them based on specific criteria.

Steps:

1. Manage Input State: Create state variables for search input and filter selections.
2. Implement Input Handlers: Create functions to handle changes in search and filter inputs.
3. Filter Data: Use the state variables to filter the product list. This can be done by checking if the product title matches the search term and/or if it fits the selected category.
4. Debounce Search: Implement debouncing to optimize search performance and reduce the number of operations.
5. Update Display: Adjust the displayed product list based on the filtered results.

Workflow:

- Set up state for search and filters.
- Implement handlers to update state.
- Filter products based on search term and selected filter.
- Display filtered products.

2.3 ***Using React Router for Navigation***

Objective: Implement navigation between different pages of the application.

Steps:

1. Setup Routing: Wrap the application in BrowserRouter to provide routing capabilities.
2. Define Routes: Use Routes and Route components to define the paths and components for each page.
3. Navigation Links: Use Link or NavLink to create navigational links that enable users to move between pages without reloading the application.
4. Dynamic Routing: Use URL parameters for dynamic routes, allowing navigation to pages with variable content (e.g., product details).

Workflow:

- Define routes in App.js using Routes and Route.
- Use Link components for navigation.
- Access URL parameters in components with useParams for dynamic content.

2.4 ***Styling with Styled-Components***

Objective: Style the product listing page using Styled-Components for a modular and maintainable design.

Steps:

1. Install Styled-Components: Add styled-components to your project.
2. Define Themes: Create theme objects for light and dark modes.
3. Create Styled Components: Define styled components using the styled API to encapsulate styles for various elements (e.g., buttons, cards).
4. Use ThemeProvider: Wrap your application in ThemeProvider to apply the theme throughout the components.
5. Dynamic Styling: Implement dynamic styling based on props to enhance component reusability.

Workflow:

- Define themes in a theme.js file.

- Create styled components for UI elements.
- Apply themes using ThemeProvider.

2.5 ***Optimizing Image Loading***

Objective: Improve the loading time of product images to enhance performance and user experience.

Steps:

1. Implement Lazy Loading: Use libraries like react-lazyload or native browser features to load images only when they come into the viewport.
2. Compress Images: Use image compression tools to reduce file sizes before uploading.
3. Serve Responsive Images: Utilize srcset or responsive image libraries to deliver appropriately sized images based on the user's device.
4. Use a CDN: Serve images via a Content Delivery Network to reduce latency and improve loading times.
5. Enable Caching: Configure HTTP caching headers to store images in the user's browser cache.
6. Placeholder Images: Implement low-quality placeholders or skeleton screens to provide immediate feedback while images load.

Workflow:

- Apply lazy loading to image components.
- Compress images and use modern formats like WebP.
- Configure CDN and caching for image delivery.

3. **Conclusion**

This methodology provides a structured approach to building an e-commerce product listing page using React. By following these steps, you can create a responsive, user-friendly interface with optimized performance and seamless navigation.

1. How would you fetch and display a list of products from an API in a React component?

- **Component Lifecycle**: Use React's lifecycle methods, particularly useEffect, to perform side effects such as data fetching. useEffect allows you to run a piece of code after the component has rendered, which is ideal for making API calls.
- **Data Fetching**: To fetch data, you can use the fetch API or a library like Axios. The API call is typically made inside the useEffect hook. This call will fetch the list of products from the API when the component mounts.
- **State Management**: Use React's useState hook to manage the fetched data. Initially, the state will be set to an empty array or null (to indicate that no data has been loaded yet). Once the API response is received, update the state with the fetched product data.

- **Error Handling**: Always account for potential errors in API calls by wrapping the request in a try/catch block or using promise .catch method. You can update the state to reflect the error, which can then be used to display an appropriate error message to the user.
- **Conditional Rendering**: Use conditional rendering to handle different states (loading, error, or display). Initially, you might display a loading indicator. Once the data is fetched, replace the loading indicator with the product list. If an error occurs, show an error message.
- **Mapping Data to Components**: Once the data is fetched and stored in state, map over the product array to create a list of components (e.g., ProductCard) that display the details of each product, such as the image, title, and price.

2. Describe how you would implement search and filter functionalities in the product listing page.

- **State Management**: Create separate pieces of state to manage the search term and selected filters (e.g., category). These states will store the user's input and selections.
- **Input Handlers**: Implement input handlers for the search bar and filter dropdowns. The handlers update the corresponding state variables whenever the user types in the search bar or selects a filter.
- **Derived State**: Use the state variables for search and filters to derive the list of products to display. This is often done by filtering the main product list based on the current search term and selected filter values.
- **Filtering Logic**: For search functionality, filter the product list based on whether the product titles or descriptions include the search term. For category filtering, only display products that match the selected category.
- **Debouncing**: For search functionality, implement debouncing to prevent excessive API calls or filtering operations. Debouncing ensures that the filtering logic only runs after the user has stopped typing for a certain period.
- **Performance Considerations**: Ensure the filtering logic is optimized for performance, especially if the product list is large. Consider memoization techniques like useMemo to avoid unnecessary re-renders and recalculations.

3. How can you use React Router to navigate between different pages in the application?

- **Router Setup**: Start by wrapping your entire application in the BrowserRouter component (from react-router-dom). This provides the routing context needed for navigation.
- **Define Routes**: Use the Routes and Route components to define the paths in your application and the components they should render. Each Route specifies a path (the URL) and an element (the component to render when the URL matches).
- **Linking Between Pages**: Use the Link component (or NavLink for more styling control) to create navigational links within your app. Link replaces the need for anchor (<a>) tags and provides a more seamless experience by preventing full-page reloads.

- **Dynamic Routing**: For dynamic routes (e.g., a product detail page with a URL like /product/:id), use URL parameters in your route definitions. These parameters can then be accessed in the component via useParams, allowing you to fetch and display data specific to that route.
- **Programmatic Navigation**: Sometimes, you may want to navigate programmatically, such as after a form submission or a button click. For this, use the useNavigate hook to navigate to a different route from within your component logic.
- **Nested Routes**: React Router supports nested routes, where a parent route renders a layout component that includes nested child routes. This is useful for creating complex page layouts where subcomponents need their own routes.

4. Explain how to use Styled-Components to style the product listing page.

- **What are Styled-Components?**: Styled-Components is a library that allows you to write CSS in your JavaScript files, scoped to individual components. It helps keep styles isolated, modular, and easier to manage, especially in large applications.
- **Creating Styled Components**: Styled-Components allows you to define styled elements using template literals. For example, const Button = styled.button``; creates a Button component with its own encapsulated styles. These styles are automatically scoped to the component, preventing conflicts with other styles.
- **Theming**: Styled-Components supports theming, which allows you to define a global theme that can be accessed by any styled component. The theme can contain variables like colors, font sizes, and spacing, making it easy to maintain a consistent design across your application.
- **Styling the Product List**: You can style the product list by creating styled components for the main container, individual product cards, images, titles, and buttons. These styled components can be composed together to create a fully styled product listing page.
- **Dynamic Styles**: Styled-Components supports dynamic styling based on props. For example, you can create a button that changes color based on a primary prop: background: ${props => props.primary ? 'blue' : 'gray'};. This allows for highly customizable and reusable components.
- **Extending Styles**: You can extend the styles of an existing styled component using the .extend method or by creating a new styled component that inherits the styles of another. This promotes reuse and reduces redundancy in your styling code.

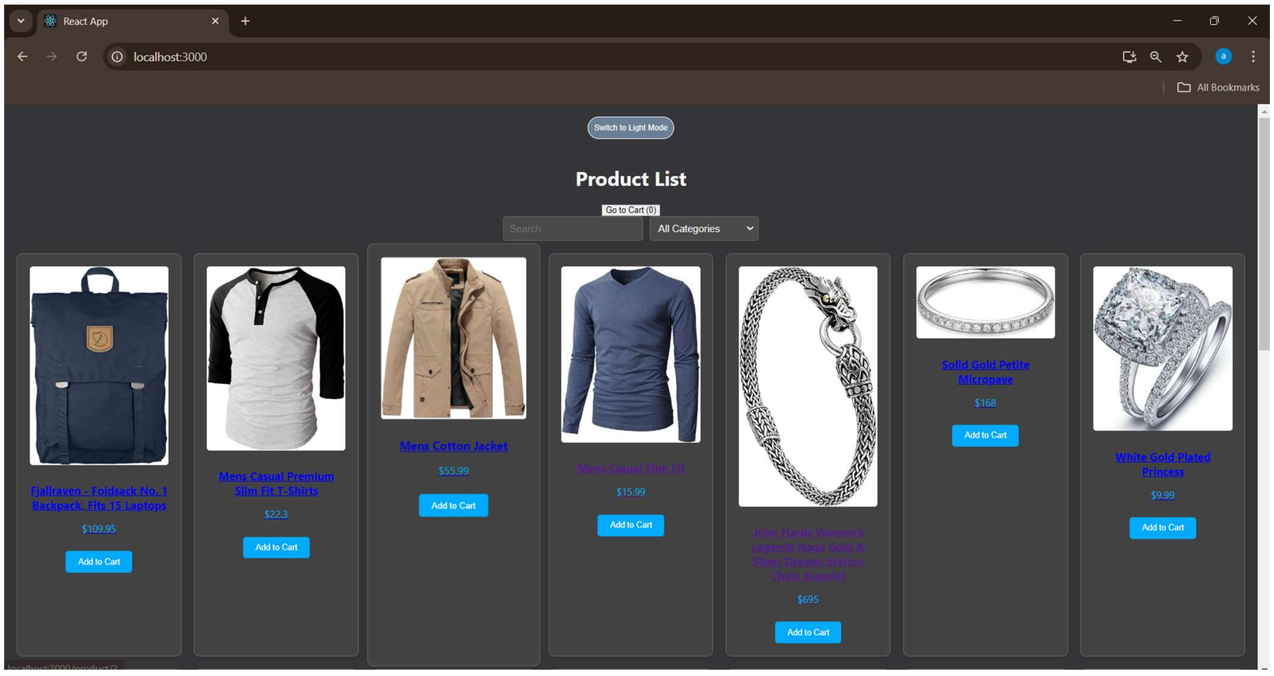
5. What techniques would you use to optimize the loading time of the product images?

- **Lazy Loading**: Implement lazy loading for images so that they are only loaded when they come into the user's viewport. This reduces initial load times, especially on pages with many images, by only loading the images that are visible on the screen.
- **Image Compression**: Use image compression techniques to reduce the file size of images without sacrificing too much quality. Tools like ImageOptim or online services can compress images before they are uploaded. Alternatively, implement automated compression through build tools or server-side processing.

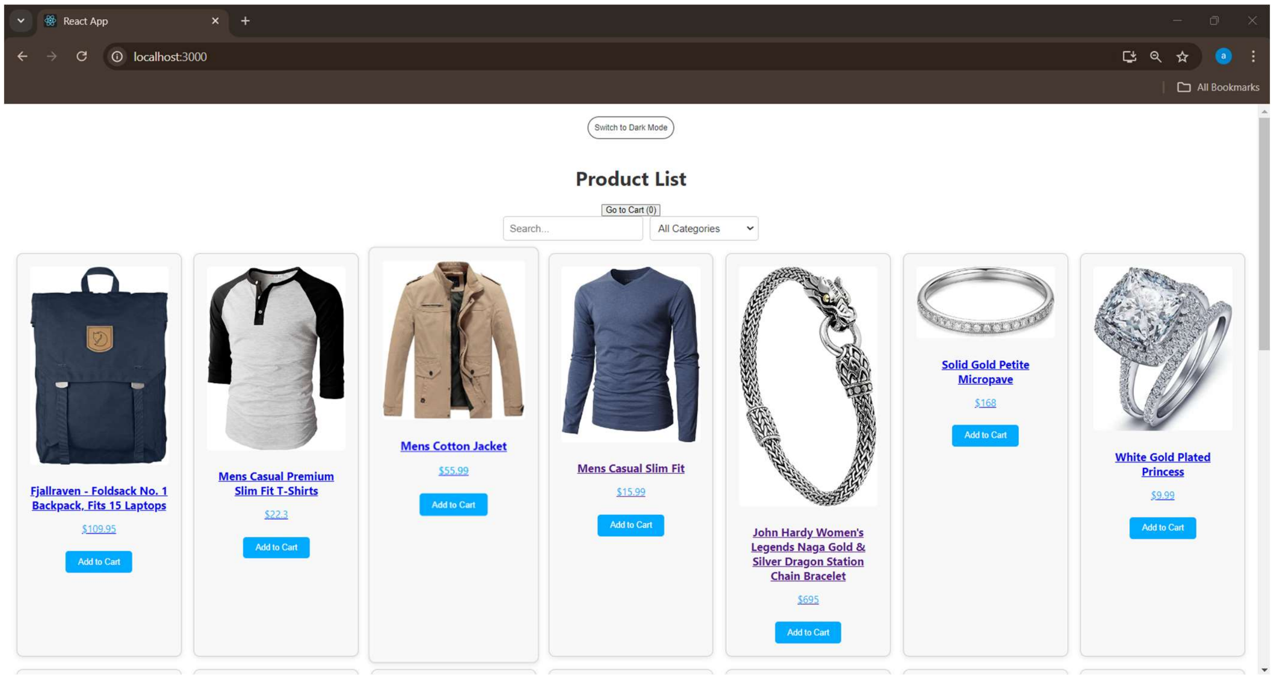
- **Responsive Images**: Serve different image sizes based on the user's device using the srcset attribute or responsive image libraries. This ensures that mobile users don't download unnecessarily large images.
- **CDN Usage**: Serve images through a Content Delivery Network (CDN). CDNs have multiple servers distributed across different locations, allowing users to download images from a server geographically closer to them, thus reducing load times.
- **Caching**: Leverage browser caching by setting appropriate HTTP headers so that images are stored in the user's browser cache. This way, returning users don't need to download the images again, significantly speeding up page load times.
- **Placeholder and Skeleton Screens**: Use low-quality image placeholders (LQIP) or skeleton screens as placeholders while the high-resolution images load. This technique gives the user immediate feedback and reduces the perceived load time.
- **Image Formats**: Use modern image formats like WebP or AVIF that provide better compression and quality ratios compared to traditional formats like JPEG or PNG. These formats are supported by most modern browsers and can significantly reduce image sizes.

Website Screen Shots:

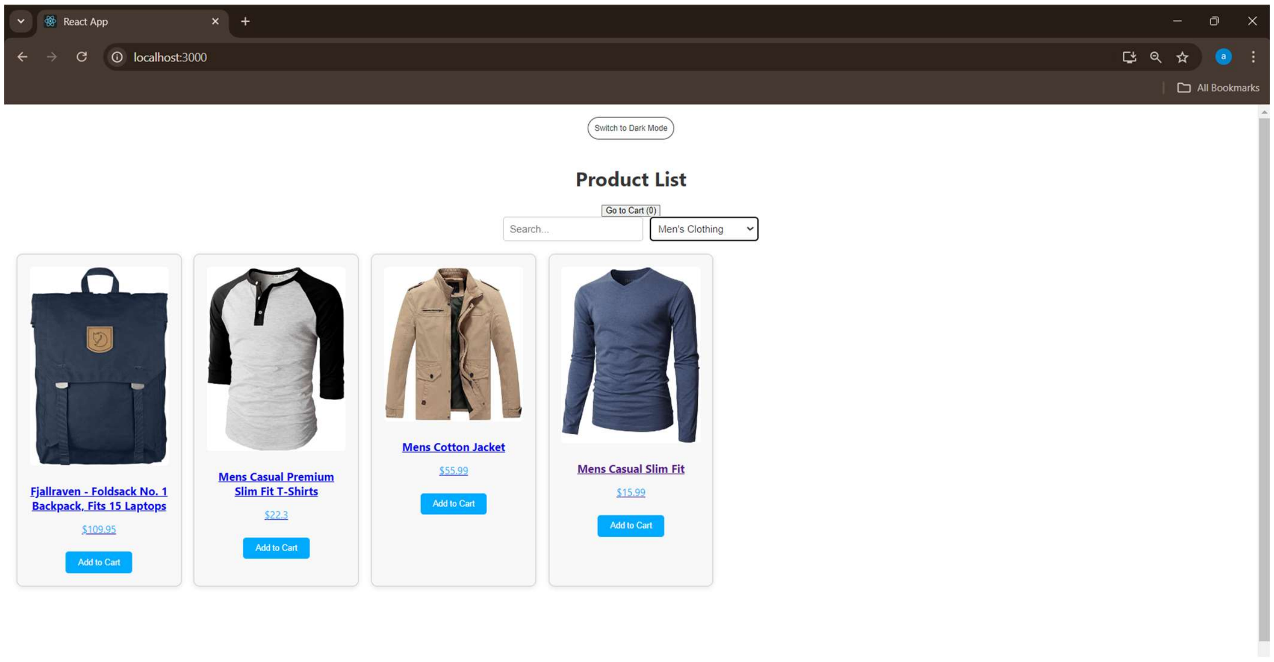
Dark Mode



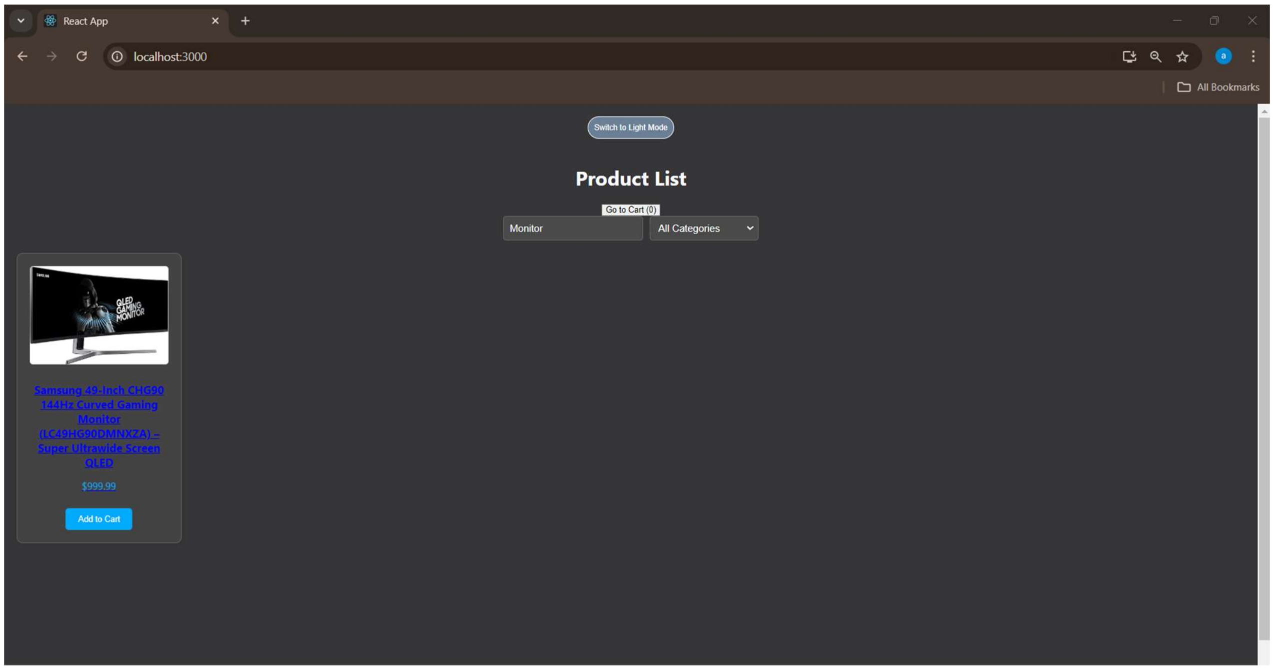
Light Mode



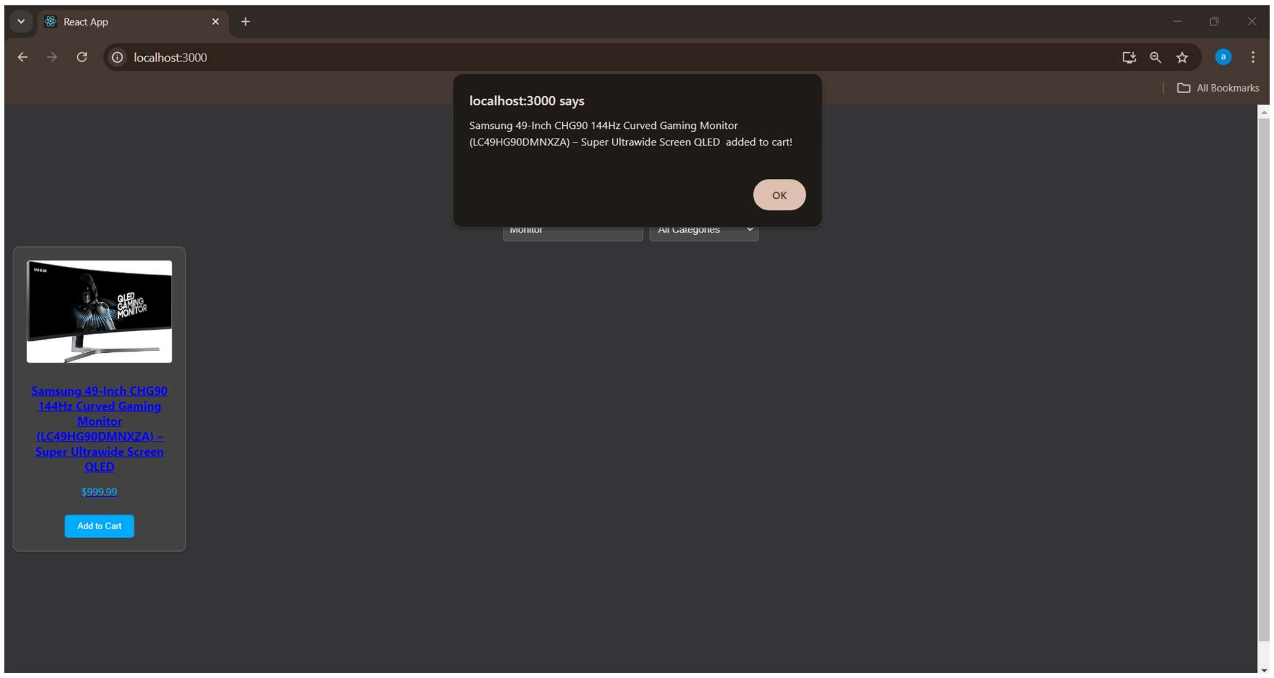
Drop down Menu Filter:



Search:

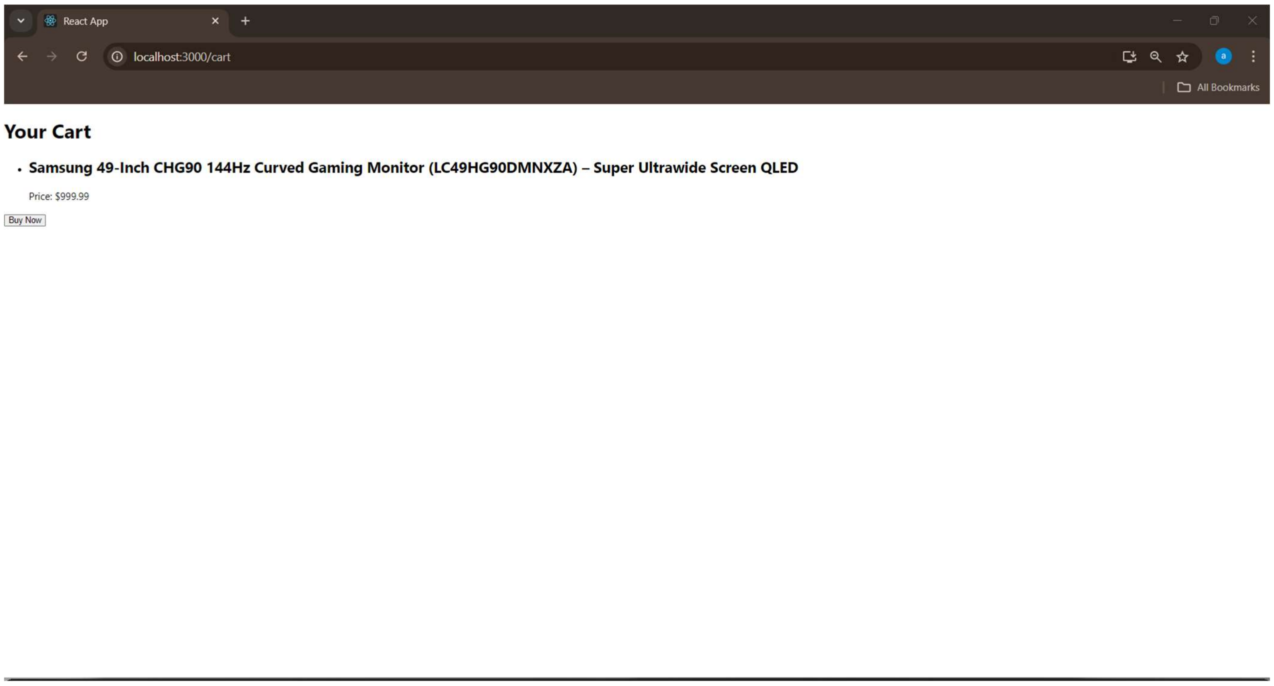


Adding to cart:


React App
localhost:3000
All Bookmarks
Monitor
All Categories
Samsung 49-Inch CHG90 144Hz Curved Gaming Monitor (LC49HG90DMNXZA) – Super Ultrawide Screen QLED
$999.99
Add to Cart
OK

localhost:3000 says

Samsung 49-Inch CHG90 144Hz Curved Gaming Monitor (LC49HG90DMNXZA) – Super Ultrawide Screen QLED added to cart!



Your Cart

- Samsung 49-Inch CHG90 144Hz Curved Gaming Monitor (LC49HG90DMNXZA) – Super Ultrawide Screen QLED

Price: $999.99

Buy Now

Task Management Application with Drag and Drop - Detailed Report

Project Aim and Theory

The primary aim of this project is to develop a Task Management application that allows users to easily organize their tasks using a drag-and-drop interface. This application enables users to categorize their tasks into 'To Do', 'In Progress', and 'Completed' states. Moreover, it supports CRUD operations (Create, Read, Update, Delete) on tasks, ensures data persistence using local storage, and provides a light/dark mode for user preference.

The drag-and-drop functionality enhances user experience by allowing quick and intuitive management of tasks. Through this feature, tasks can be moved between different stages of completion. Additionally, using local storage, tasks are saved so that they persist even after a page reload. The app also offers a modern, responsive user interface with light/dark mode options.

Implementation

1. Drag-and-Drop Functionality

The drag-and-drop feature is implemented using the `react-beautiful-dnd` library. The task list is divided into three droppable areas: 'To Do', 'In Progress', and 'Completed'. Each task is marked as draggable. The following components are used to achieve this:
- `DragDropContext`: Wraps the entire app and listens for the drag end event.
- `Droppable`: Defines the columns (To Do, In Progress, Completed) where tasks can be dropped.
- `Draggable`: Marks each task as draggable, allowing users to move tasks between columns.

2. CRUD Operations

Tasks in the application are managed using state hooks (`useState`) for handling task data, and CRUD operations are implemented as follows:
- ***Create***: Users can input new tasks via a form. These tasks are added to the 'To Do' list.
- ***Read***: The current task list is displayed dynamically under the respective columns based on their status.
- ***Update***: Tasks can be moved between columns using the drag-and-drop functionality, updating their status.
- ***Delete***: Tasks can be removed from the list by clicking the 'Remove' button associated with each task.

3. Local Storage Persistence

To ensure data persistence, the app uses the browser's `localStorage`. When the app loads, the task data is retrieved from localStorage and used to initialize the task state. Every time a task is created, updated, or deleted, the task state is saved back to localStorage. This ensures that users' tasks are retained even after a page reload.

4. Light and Dark Mode

The application provides an option for users to toggle between light and dark modes. This feature is implemented by using a theme state variable that toggles between light and dark themes, adjusting the

CSS variables accordingly. The state of the theme is also stored in `localStorage` to persist the user's preference across sessions.

5. UI Design and Styling

The UI is designed to be modern and responsive, with a clean layout that divides the screen into three columns. Each column represents a different stage in the task management process (To Do, In Progress, Completed). The design also ensures proper alignment, color contrast for both light and dark modes, and responsive design for various screen sizes.

Conclusion

This task management application provides a simple yet powerful interface for managing tasks. With its drag-and-drop functionality, CRUD operations, data persistence through local storage, and modern UI with light/dark modes, the app delivers a smooth and interactive user experience. The use of modern React features ensures scalability and maintainability, making it an efficient tool for task organization.

Questions and Answers

1. How would you implement drag-and-drop functionality in a React component?

The drag-and-drop functionality is implemented using the `react-beautiful-dnd` library. It involves wrapping the app in a `DragDropContext` and defining `Droppable` areas for the task columns. Tasks themselves are defined as `Draggable` components. The `onDragEnd` event is used to update the task's status based on where it was dropped.

2. Describe how to manage the state of tasks and handle CRUD operations.

The state of tasks is managed using `useState` to track the different task lists (`todo`, `inProgress`, `completed`). For CRUD operations, tasks can be created via a form, read by rendering them in their respective columns, updated by dragging them to different columns, and deleted by clicking the remove button.

3. How can you use local storage to persist the state of the tasks across page reloads?

Local storage is utilized to save the task state by storing the task lists in `localStorage` whenever they are updated. Upon loading the app, the task data is fetched from `localStorage` and used to initialize the task state, ensuring that the tasks persist across reloads.

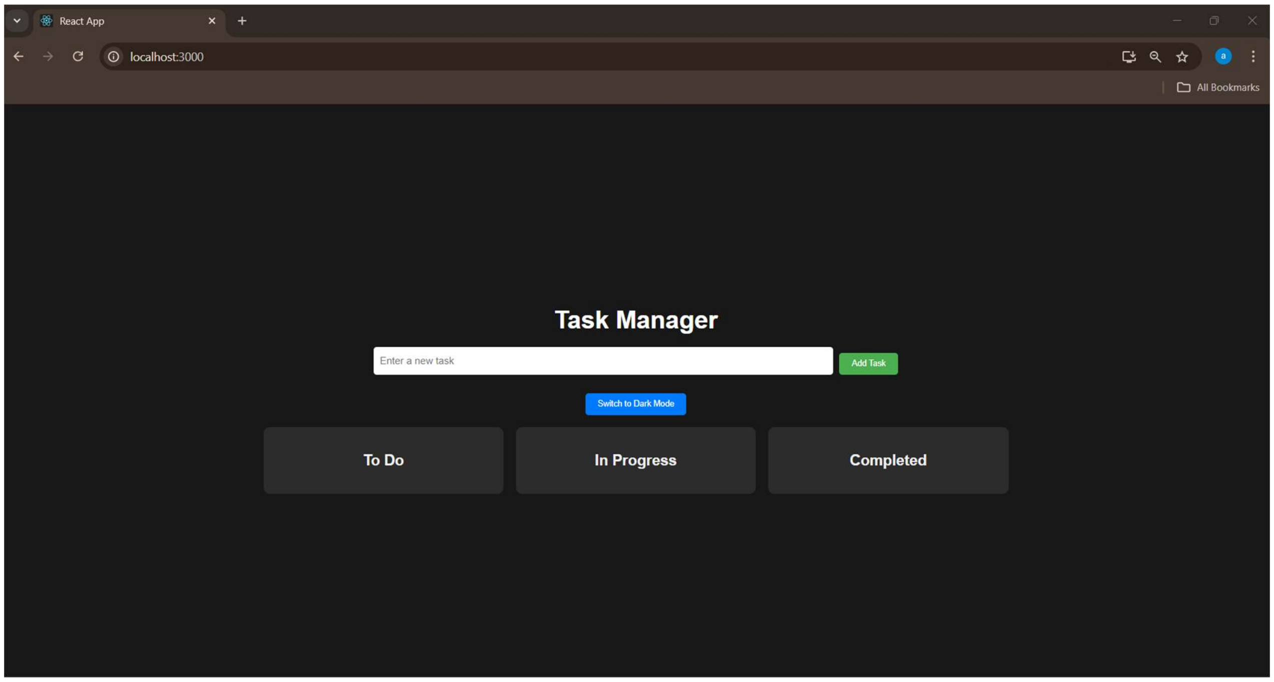
4. Explain the steps to ensure smooth and responsive drag-and-drop interactions.

For smooth drag-and-drop interactions, several optimizations are made:
- Memoize task components using `React.memo` to prevent unnecessary re-renders.
- Use CSS transitions to provide visual feedback and make the dragging process fluid.
- Handle state updates efficiently within the `onDragEnd` event handler.

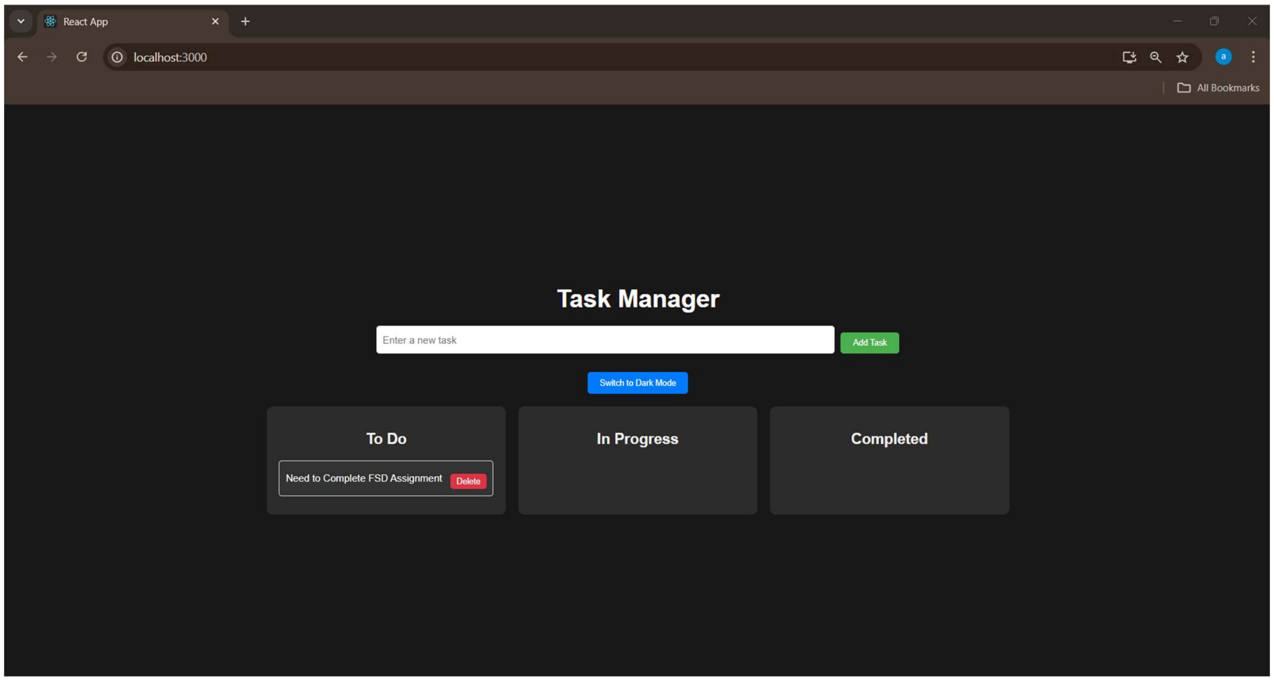
5. What are the best practices for handling large lists of tasks in terms of performance?

Handling large lists of tasks requires performance optimizations such as:
- Using libraries like `react-window` for virtualized lists to only render visible tasks.
- Employing `useMemo` and `useCallback` to avoid unnecessary re-renders.
- Grouping state updates to reduce the number of re-renders and improve performance.

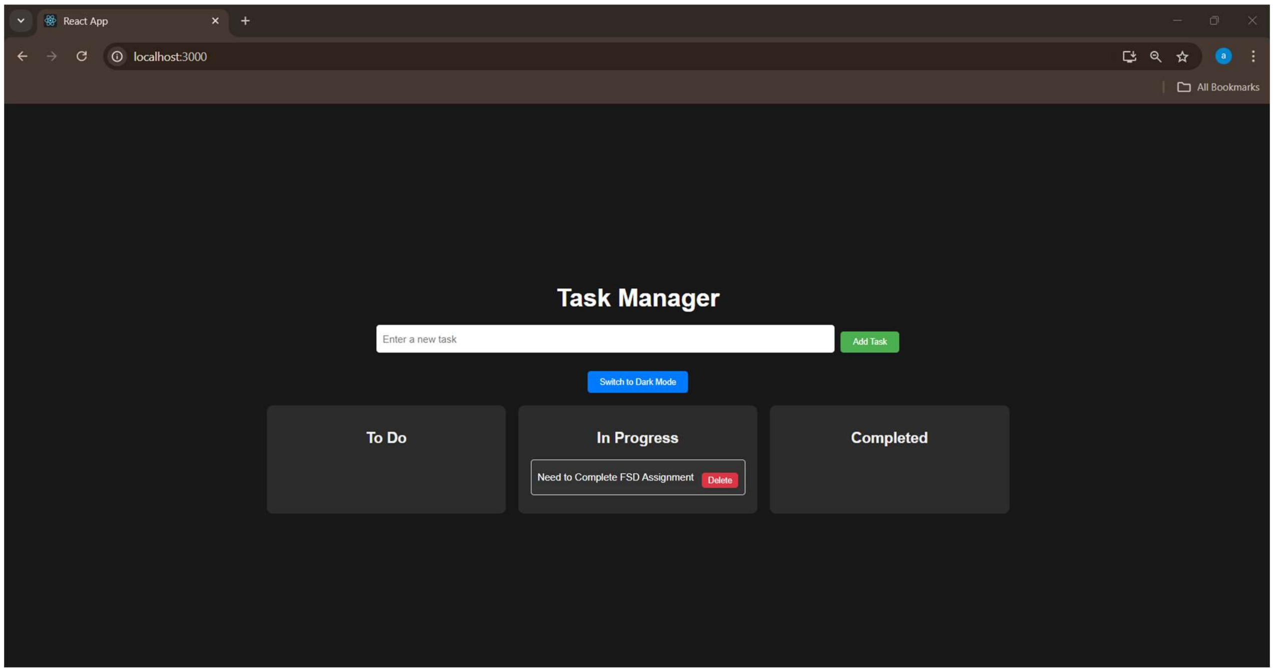
Home Page:



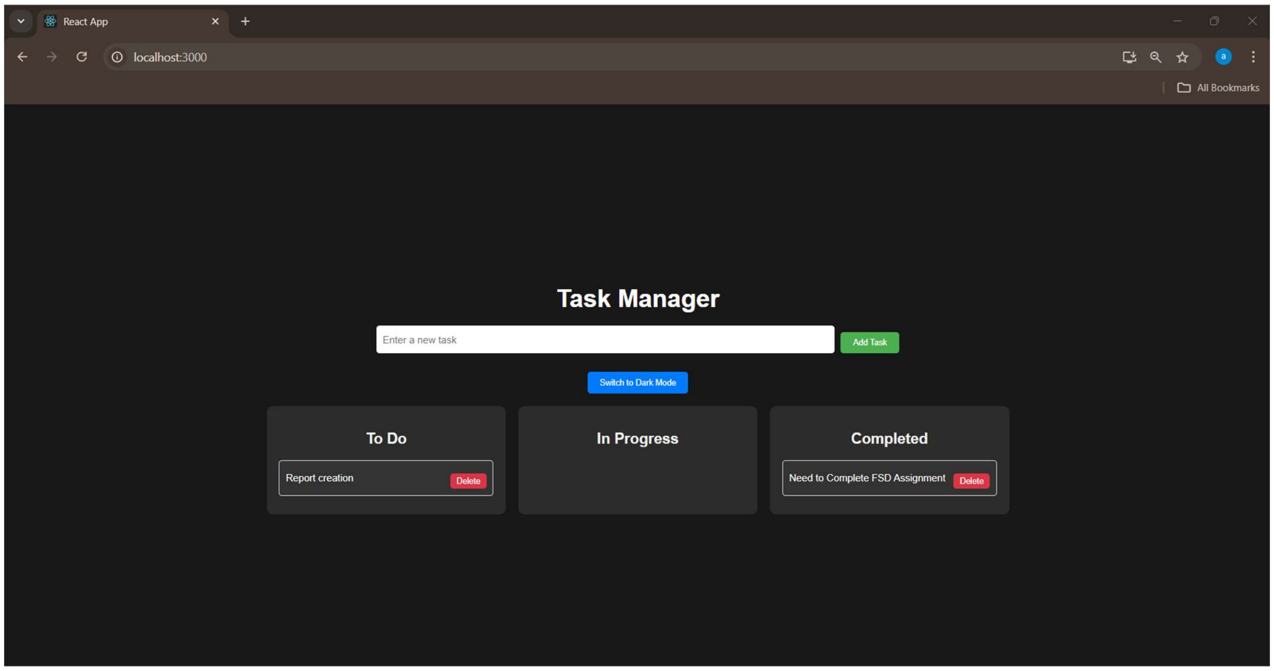
Adding a Task:



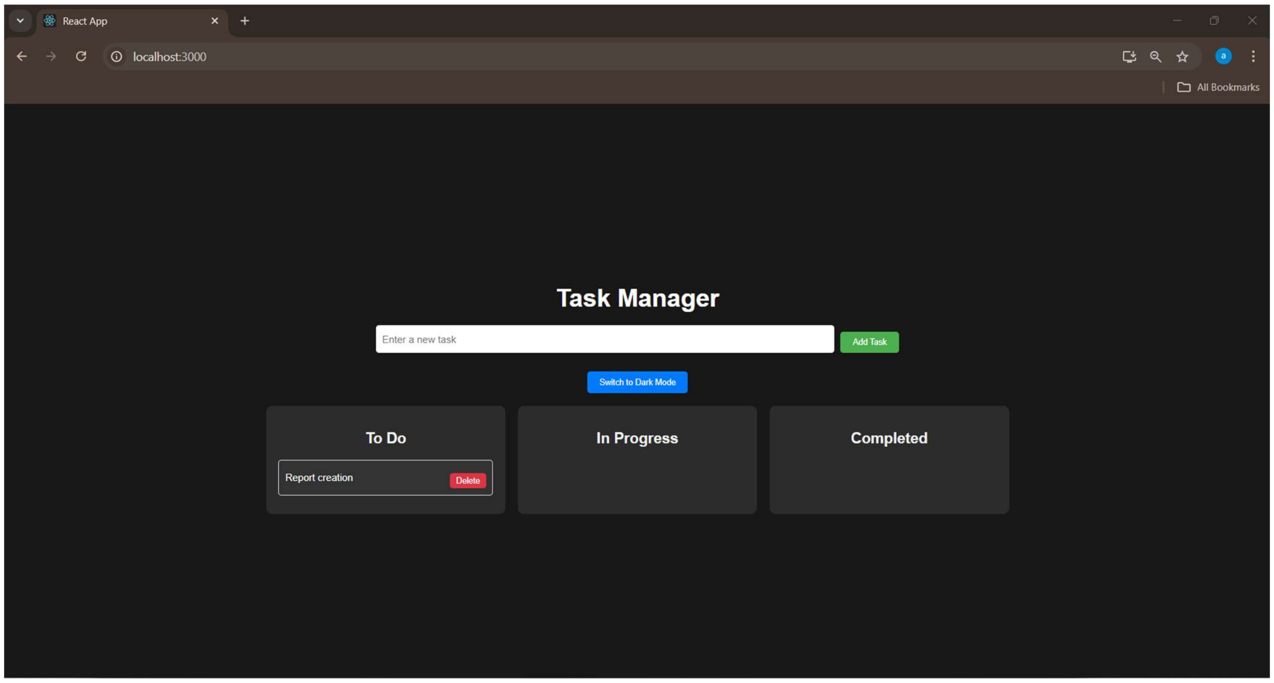
Moving the task to In progress stage by drag and drop:



Drag and drop to completed :



Delete an Task:


React App
localhost:3000
All Bookmarks
Task Manager
Enter a new task
Add Task
Switch to Dark Mode
To Do
Report creation
Delete
In Progress
Completed