Amrita School of Computing-Department of Computer Science and Engineering - AIE

Amrita Vishwa Vidyapeetham Chennai Campus.

ASSIGNMENT – 1

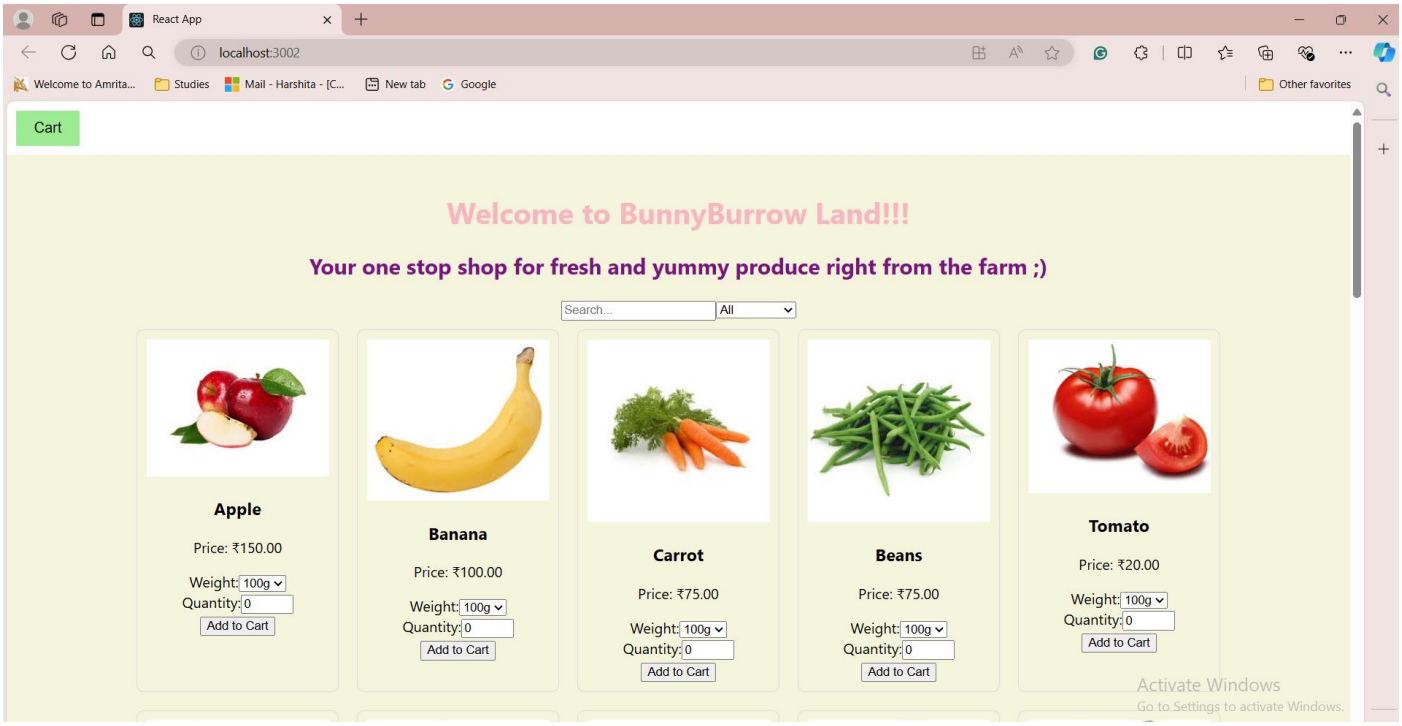
22AIE457 – Full Stack Development

Name: S.Harshita

Roll No. : CH.EN.U4AIE21015

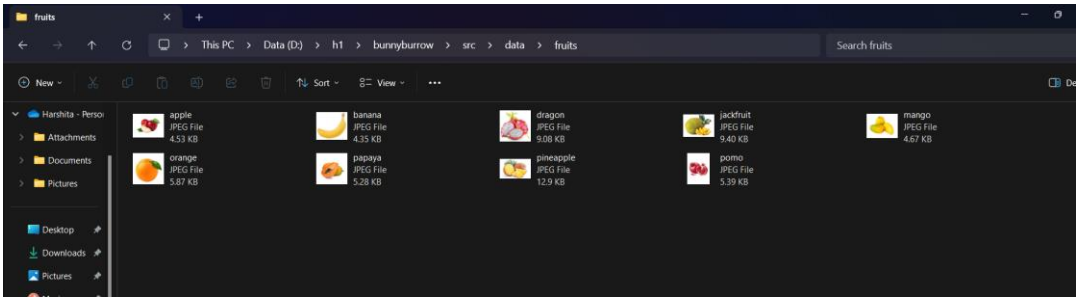
Case Study 2: E-Commerce Product Listing Page

For this I have created my own website BunnyBurrow which is a React-based e-commerce platform featuring a product listing page where users can search, filter, and add fresh produce (fruits and vegetables) to their cart. The application utilizes React Router for navigation, Styled-Components for styling, and optimized image loading techniques for enhanced performance.



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

Ans: To fetch and display products, we utilized the fetch API in a React component to get the product data which is a self-created dataset, and then rendered it using state management. We used useEffect to simulate fetching product data from an API and set it in the state. This data is then displayed in the ProductList component. The image given below is an example for just fruits dataset, similar data was created for vegetables.



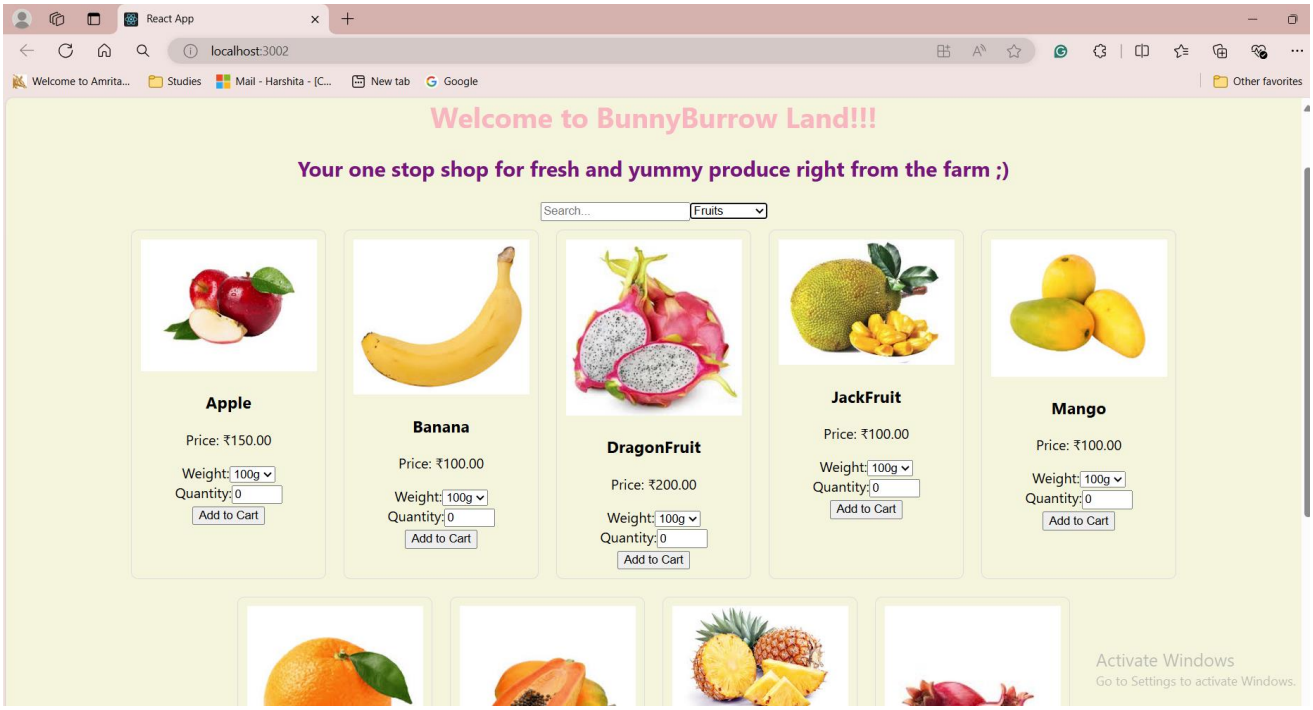
The code used:

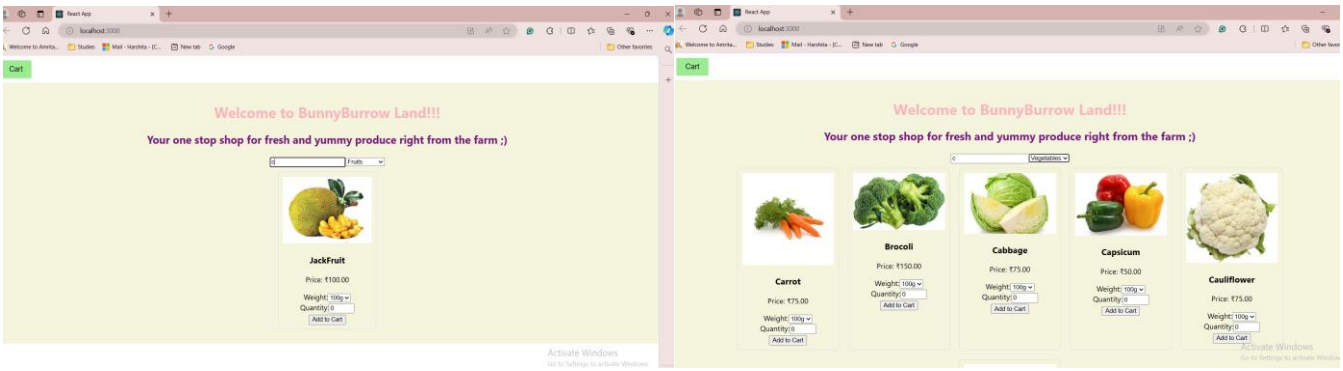
```
import React, { useEffect, useState } from 'react';

import { products as localProducts } from '../data/products';

import { Wrapper, Title, Subtitle, ProductContainer, ProductCard, ProductImage } from '../styles';


const ProductList = ({ addToCart }) => {

  const [products, setProducts] = useState([]);
```

```
  useEffect(() => {
    // Simulate fetching data from an API
    // Replace with fetch('api/products') for real API
    setProducts(localProducts);
  }, []);

  return (
    <Wrapper>
      <Title>Welcome to BunnyBurrow Land!!!</Title>
      <Subtitle>Your one stop shop for fresh and yummy produce right from the farm ;)</Subtitle>
      <ProductContainer>
        {products.map((product) => (
          <ProductCard key={product.id}>
            <ProductImage src={product.image} alt={product.title} />
            <h3>{product.title}</h3>
            <p>Price: ₹{product.price}</p>
            <button onClick={() => addToCart(product)}>Add to Cart</button>
          </ProductCard>
        ))}
      </ProductContainer>
    </Wrapper>
  );
};
export default ProductList;
```

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

Ans: To implement search and filter functionalities, we used state to manage search input and filter selection. We then filtered the list of products based on these criteria. We manage search and filter states, then filter the products based on these states. The filtered products are then rendered dynamically.





The above images show you how I can filter the fruits alone out of the product list, the second set of images shows how when I search for Letter C in Fruit Filter vs Vegetable Filter. The code I have used for implementation of this is as follows:

```
import React, { useState } from 'react';

import { products as localProducts } from '../data/products';


const ProductList = ({ addToCart }) => {
  const [searchTerm, setSearchTerm] = useState('');
  const [filter, setFilter] = useState('all');


  const filteredProducts = localProducts.filter((product) => {
    const productName = product.title.toLowerCase();
    return (
```

```
    (filter === 'all' || product.category === filter) &&
    productName.includes(searchTerm.toLowerCase())
  );
});

return (
  <div>
    {/* Search Input */}
    <input
      type="text"
      placeholder="Search..."
      value={searchTerm}
      onChange={(e) => setSearchTerm(e.target.value)}
    />

    {/* Filter Dropdown */}
    <select onChange={(e) => setFilter(e.target.value)} value={filter}>
      <option value="all">All</option>
      <option value="fruits">Fruits</option>
      <option value="vegetables">Vegetables</option>
    </select>

    {/* Display Filtered Products */}
    <div>
      {filteredProducts.map((product) => (
        <div key={product.id}>
          <img src={product.image} alt={product.title} />
          <h3>{product.title}</h3>
          <p>Price: ₹{product.price}</p>
          <button onClick={() => addToCart(product)}>Add to Cart</button>
        </div>
      ))}
    </div>
  </div>
```

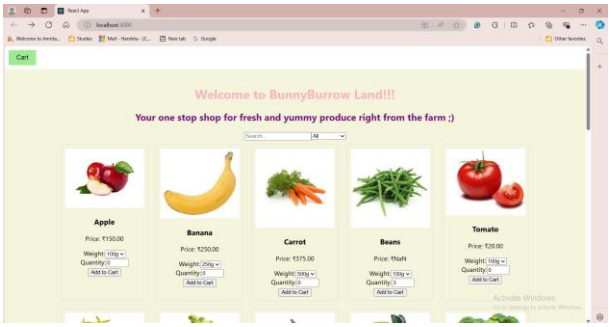
```
  );
};
export default ProductList;
```

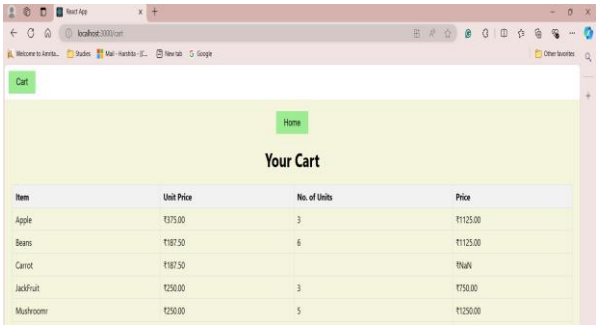
Key Code Explanation:

- Search Input: The searchTerm state is used to capture the user's search query. It is updated on every change to the input field.
- Products are filtered using productName.includes(searchTerm.toLowerCase()) to check if the search term matches the product title.
- Filter Dropdown: The filter state determines whether to show "All," "Fruits," or "Vegetables."
- Products are filtered by their category using (filter === 'all' || product.category === filter).

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

In BunnyBurrow, React Router is used to navigate between different pages, such as the Product Listing page and the Cart page. Here's how I have implemented React Router for navigation:





So, after adding my products I can easily go to my Cart by clicking the cart button which easily navigates me to the cart page once my shopping is over, I can again go back to my home page by clicking the home button in the Cart page. The code snippets that I used for this implementation is:

- Define Routes in App.js: Use the Routes and Route components from react-router-dom to define your application's routes.

```
import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import ProductList from './components/ProductList';
import CartPage from './components/CartPage';
import Navbar from './components/Navbar';
const App = () => {
  return (
    <Router>
      <Navbar />
      <Routes>
        <Route path="/" element={<ProductList />} />
        <Route path="/cart" element={<CartPage />} />
```

```
    </Routes>
  </Router>
 );
};
export default App;
```

- Add Links for Navigation: In the Navbar.js, use the Link component to allow navigation between pages.

```
import React from 'react';
import { Link } from 'react-router-dom';
const Navbar = () => {
 return (
  <nav>
   <Link to="/">Home</Link>
   <Link to="/cart">Cart</Link>
  </nav>
 );
};
export default Navbar;
```

- Cart Button in Product List: Add a "Go to Cart" button that links to the Cart page.

```
import { Link } from 'react-router-dom';
const ProductList = () => {
 return (
  <div>
   {/* Products Listing */}
   <Link to="/cart">
    <button>Go to Cart</button>
   </Link>
  </div>
 ); };
```

Key Code Explanation:

- Routes define different pages (e.g., / for the product listing and /cart for the cart).
- Link provides clickable navigation between these pages without a full page reload.
- The Router component wraps your app to provide routing functionality.
- This setup enables smooth navigation between the product listing and the cart page in BunnyBurrow.

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

For this first I had to install npm install styled-components package and used it my js files. The code snippet on how it was executed in bunnyburrow is as follows:

```
import React, { useState } from 'react';

import styled from 'styled-components';

import { products as localProducts } from '../data/products';

// Styled Components

const ProductListContainer = styled.div`

  background-color: beige;

  padding: 20px;

  text-align: center;

`;

const Heading = styled.h1`

  color: lightpink;

  font-weight: bold;

`;

const SubHeading = styled.h2`

  color: purple;

`;

const SearchBar = styled.input`

  padding: 10px;

  margin-bottom: 20px;

  width: 60%;

  font-size: 16px;

`;

const FilterDropdown = styled.select`

  margin-left: 20px;

  padding: 10px;

  font-size: 16px;

`;

const ProductCard = styled.div`

  display: inline-block;

  margin: 20px;

  padding: 20px;
```

```
  border: 1px solid #ddd;
  border-radius: 8px;
  width: 200px;
  text-align: center;
  background-color: #fff;
`;
const ProductImage = styled.img`
  max-width: 100px;
  max-height: 100px;
  object-fit: cover;
`;
const AddToCartButton = styled.button`
  margin-top: 10px;
  padding: 10px;
  background-color: lightgreen;
  border: none;
  cursor: pointer;
  &:hover {
    background-color: green;
    color: white;
  }
`;
const ProductList = ({ addToCart }) => {
  const [searchTerm, setSearchTerm] = useState('');
  const [filter, setFilter] = useState('all');
  const filteredProducts = localProducts.filter((product) => {
    const productName = product.title.toLowerCase();
    return (
      (filter === 'all' || product.category === filter) &&
      productName.includes(searchTerm.toLowerCase())
    );
  });

  return (
```

```
   <ProductListContainer>
    {/* Heading and Subheading */}
    <Heading>Welcome to BunnyBurrow Land!!!</Heading>
    <SubHeading>Your one-stop shop for fresh and yummy produce right from the farm ;)</SubHeading>

    {/* Search and Filter */}
    <SearchBar
     type="text"
     placeholder="Search for products..."
     value={searchTerm}
     onChange={(e) => setSearchTerm(e.target.value)}
    />
    <FilterDropdown onChange={(e) => setFilter(e.target.value)} value={filter}>
     <option value="all">All</option>
     <option value="fruits">Fruits</option>
     <option value="vegetables">Vegetables</option>
    </FilterDropdown>

    {/* Product Cards */}
    {filteredProducts.map((product) => (
     <ProductCard key={product.id}>
      <ProductImage src={product.image} alt={product.title} />
      <h3>{product.title}</h3>
      <p>Price: ₹{product.price}</p>
      <AddToCartButton onClick={() => addToCart(product)}>Add to Cart</AddToCartButton>
     </ProductCard>
    ))}
   </ProductListContainer>
  );
};
Export default ProductList;
```

Key Code Explanation:

- Styled Components Creation: ProductListContainer, Heading, SubHeading, SearchBar, FilterDropdown, ProductCard, ProductImage, and AddToCartButton are examples of styled components that encapsulate the CSS directly in the JavaScript file.
- The styles are scoped to the specific component, ensuring no conflict with other styles.
- How Styled-Components Are Used: Instead of using traditional className for styling, each component is styled using styled.[element].
- Styled-components are used as regular React components, but they contain CSS rules to style them.

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

To optimize the loading time of product images in BunnyBurrow, I have implemented the following techniques to enhance performance and reduce page load times. Here are a few approaches with code snippets:

1. Lazy Loading Images:

Lazy loading ensures that images are only loaded when they enter the viewport, reducing the initial load time.

```
const ProductImage = styled.img`
  max-width: 100px;
  max-height: 100px;
  object-fit: cover;
  loading: lazy; // Add lazy loading attribute
`;
```

By adding `loading="lazy"` to the `<img>` tag, browsers will defer loading the images until they're about to appear in the viewport.

2. Using Optimized Image Formats:

Use modern image formats like WebP, which offer better compression and faster loading times compared to traditional formats like JPEG or PNG.

```
<ProductImage src={product.imageWebP ? product.imageWebP : product.image} alt={product.title} />
```

3. Responsive Images (srcset):

Providing multiple versions of images at different resolutions, and let the browser choose the most appropriate one based on the user's device or screen size.

```
const ProductImage = styled.img`
  max-width: 100px;
  max-height: 100px;
  object-fit: cover;
`;
<ProductImage
  srcSet={`${product.imageSmall} 200w, ${product.imageMedium} 400w, ${product.imageLarge} 800w`}
```

```
    sizes="(max-width: 600px) 200px, (max-width: 1200px) 400px, 800px"
    src={product.imageMedium}
    alt={product.title}
  />
```

4. Image Compression: Compress the images beforehand using tools like [TinyPNG](https://tinypng.com/) or [ImageOptim](https://imageoptim.com/) to reduce file sizes without noticeable loss of quality.

5. Content Delivery Network (CDN): Serve images from a CDN to ensure that they are loaded from servers closest to the user's location, reducing latency and speeding up image loading times.

6. Placeholder or Skeleton Images: While the actual images are loading, you can display a lightweight placeholder or skeleton to improve user experience.

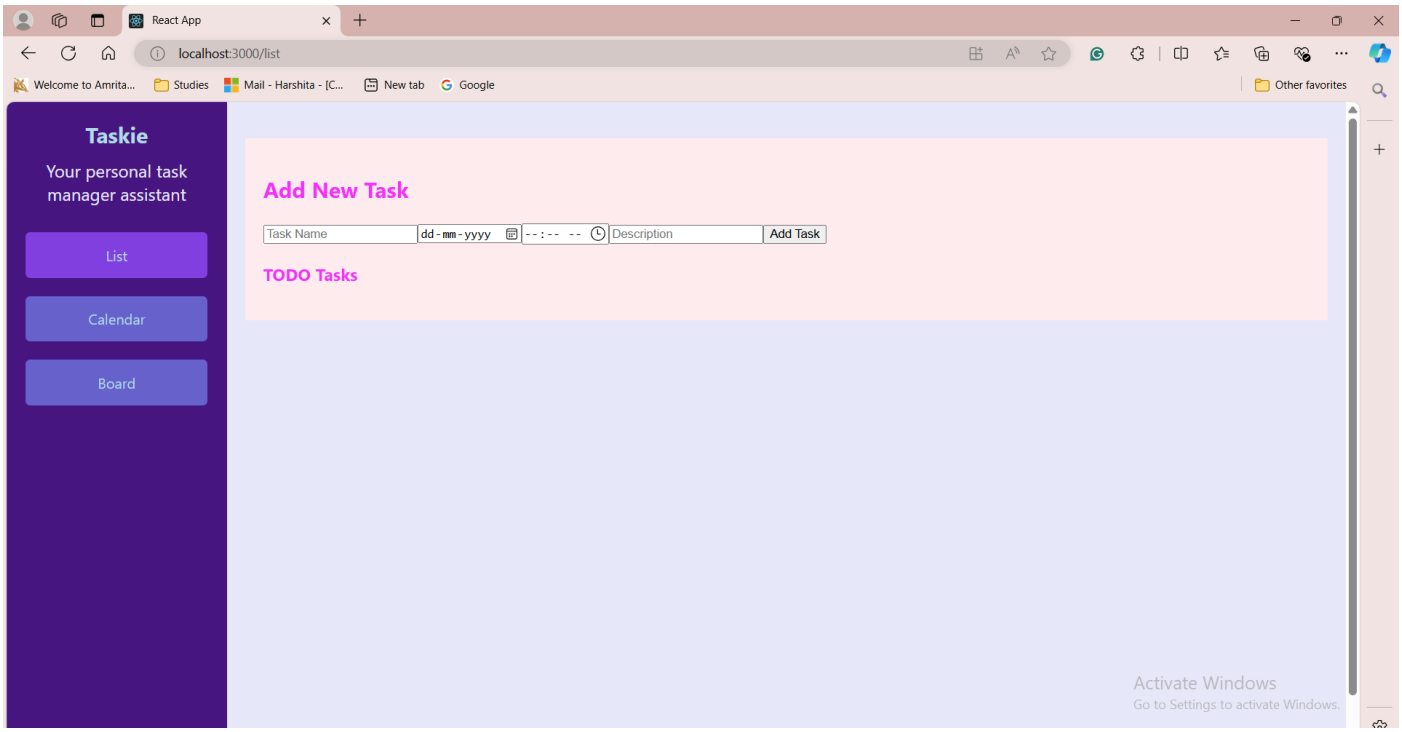
```
  const ProductImagePlaceholder = styled.div`
    width: 100px;
    height: 100px;
    background-color: #f0f0f0;
  `;
  const ProductImage = ({ src, alt }) => {
    const [loaded, setLoaded] = useState(false);
    return (
      {!loaded && <ProductImagePlaceholder />}
      <img
        src={src}
        alt={alt}
        style={{ display: loaded ? 'block' : 'none' }}
        onLoad={() => setLoaded(true)}
    ); };
```

7. Cache and Image Expiration Headers: Setting proper cache headers for images, allowing browsers to cache them and avoid reloading the same image repeatedly. Combining these techniques ensured that BunnyBurrow's product images load quickly and efficiently, improving both performance and user experience.

Conclusion:

Building the BunnyBurrow my own React.js application has taught me essential skills in web development, including API integration for fetching product data, state management for features like search, filtering, and cart functionality, and navigation using React Router. I also learned to style components efficiently with Styled-Components and implemented optimization techniques such as lazy loading and image compression to enhance performance. This project helped me deepen my understanding of creating scalable, user-friendly e-commerce applications while maintaining smooth performance and clean, modular code.

Case Study 3: Task Management Application with Drag and Drop

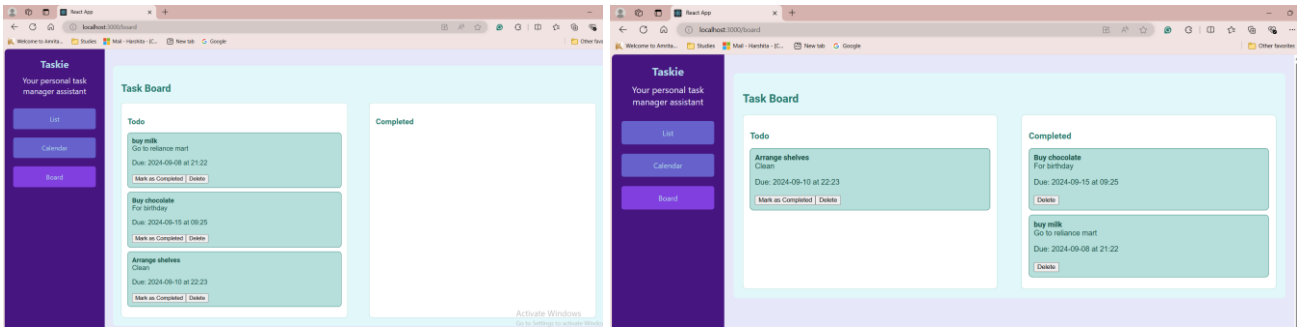


Taskie is a my React.js-based task management application designed to help users efficiently manage their tasks. It features a task list component that allows users to add new tasks with details such as due dates and descriptions. The calendar component helps users visualize their tasks by displaying tasks planned for specific dates, making it easy to track deadlines. Additionally, Taskie includes a task board, where users can organize their tasks between "To Do" and "Completed" columns using a drag-and-drop interface. This combination of features ensures streamlined task management and better productivity.

Questions:

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

In Taskie, drag-and-drop functionality is implemented on the Board component, which allows users to move tasks between the "To Do" and "Completed" lists. The react-beautiful-dnd library is used for handling this feature. When a user drags a task card and drops it in a new position or list, the app updates the state to reflect the changes. The following is a small code snippet:




```
import React from 'react';
import { DragDropContext, Droppable, Draggable } from 'react-beautiful-dnd';
import { useTasksBoard } from '../contexts/TaskContextBoard';
const Board = () => {
const { tasks, moveTask } = useTasksBoard();
  const onDragEnd = (result) => {
```

```
  const { source, destination } = result;
  if (!destination) return;
  moveTask(source.droppableId, destination.droppableId, source.index, destination.index);
};

return (
  <DragDropContext onDragEnd={onDragEnd}>
    <Droppable droppableId="todo">
      {(provided) => (
        <div ref={provided.innerRef} {...provided.droppableProps}>
          <h3>TODO</h3>
          {tasks.todo.map((task, index) => (
            <Draggable key={task.id} draggableId={task.id} index={index}>
              {(provided) => (
                <div ref={provided.innerRef} {...provided.draggableProps} {...provided.dragHandleProps}>
                  <h4>{task.name}</h4>
                </div>
              )}
            </Draggable>
          ))}
          {provided.placeholder}
        </div>
      )}
    </Droppable>

    <Droppable droppableId="completed">
      {(provided) => (
        <div ref={provided.innerRef} {...provided.droppableProps}>
          <h3>Completed</h3>
          {tasks.completed.map((task, index) => (
            <Draggable key={task.id} draggableId={task.id} index={index}>
              {(provided) => (
                <div ref={provided.innerRef} {...provided.draggableProps} {...provided.dragHandleProps}>
                  <h4>{task.name}</h4>
                </div>
```

```
            )}
          </Draggable>
        ))}
        {provided.placeholder}
      </div>
    )}
  </Droppable>
</DragDropContext>
);
};
```

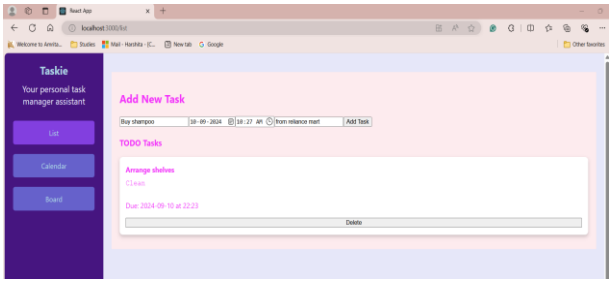
Key Code Explanation:

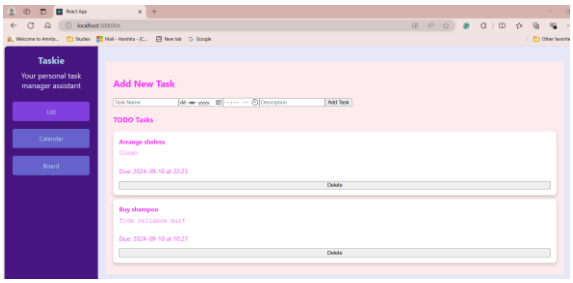
- DragDropContext: Wraps the entire board and handles the drag-and-drop behavior.
- Droppable: Defines the areas (lists) where tasks can be dropped.
- Draggable: Each task card is draggable, allowing users to move it between lists.
- onDragEnd: Updates task positioning after the user finishes dragging.

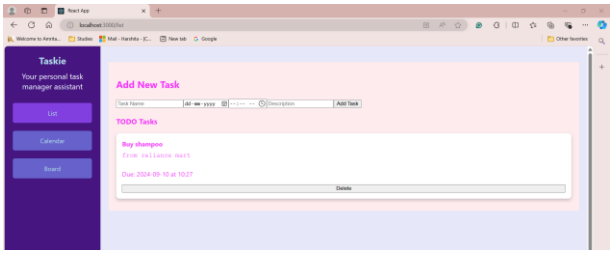
This makes Taskie's task board flexible and easy to interact with, providing users with smooth drag-and-drop task management.

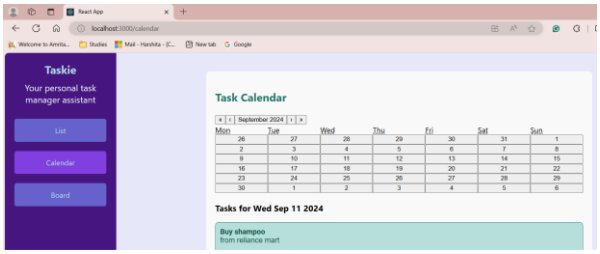
2. Describe how to manage the state of tasks and handle CRUD operations.

In Taskie, the state of tasks and the CRUD (Create, Read, Update, Delete) operations are managed using React's useState hook and context (TaskContext). The state of tasks is stored globally so that it can be accessed and modified by different components like the Task List, Calendar, and Task Board. Here's a small code snippet:







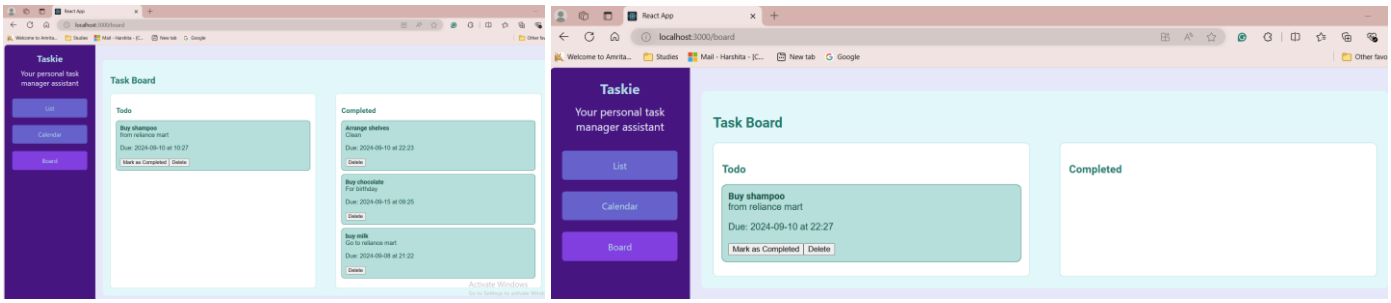



```
import { createContext, useState, useContext } from 'react';

const TaskContext = createContext();

export const TaskProvider = ({ children }) => {
  const [tasks, setTasks] = useState([]);
  const addTask = (newTask) => setTasks([...tasks, newTask]);
  const updateTask = (updatedTask) => setTasks(tasks.map(task => task.id === updatedTask.id ? updatedTask : task));
  const deleteTask = (taskId) => setTasks(tasks.filter(task => task.id !== taskId));
```

```
  return (
    <TaskContext.Provider value={{ tasks, addTask, updateTask, deleteTask }}>
      {children}
    </TaskContext.Provider>
  );
};
export const useTasks = () => useContext(TaskContext);
```

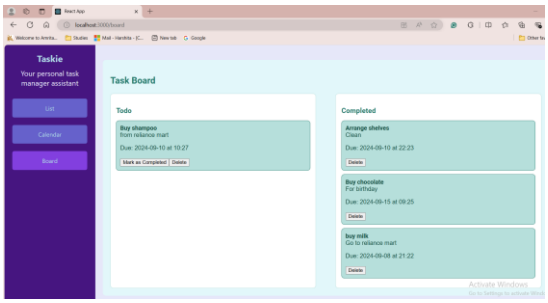
3. How can you use local storage to persist the state of the tasks across page reloads?

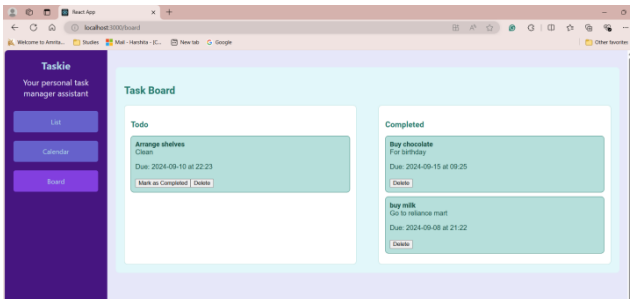
In Taskie, when the app is opened, tasks are retrieved from local storage and displayed. When a user adds or deletes tasks, the changes are automatically saved to local storage. If the user refreshes or closes the page, the tasks persist, and they are loaded back from local storage when the app is reopened. By integrating local storage with the React state through useEffect, Taskie ensures that tasks are saved across page reloads.




```
import React, { useState, useEffect } from 'react';
const TaskContext = React.createContext();
export const TaskProvider = ({ children }) => {
  const [tasks, setTasks] = useState([]);
  // Load tasks from local storage when the component mounts
  useEffect(() => {
    const storedTasks = localStorage.getItem('tasks');
    if (storedTasks) {
      setTasks(JSON.parse(storedTasks));  // Parse and set the tasks if they exist in local storage
    }
  }, []);
  return (
    <TaskContext.Provider value={{ tasks, setTasks }}>
      {children}
    </TaskContext.Provider>
  ); };
```

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






```
import React from 'react';
import { DragDropContext, Droppable, Draggable } from 'react-beautiful-dnd';
import { useTasks } from '../contexts/TaskContext';
import './Board.css';
const Board = () => {
const { tasks, moveTask, deleteTask } = useTasks();
const onDragEnd = (result) => {
const { source, destination } = result;

  // If there's no destination, do nothing
  if (!destination) return;

  const sourceList = source.droppableId;
  const destinationList = destination.droppableId;
  moveTask(sourceList, destinationList, source.index, destination.index);
};
const markAsCompleted = (taskId) => {
moveTask('todo', 'completed', tasks['todo'].findIndex(task => task.id === taskId), 0);
};
return (
  <DragDropContext onDragEnd={onDragEnd}>
    <div className="board-container">
      <h2 className="board-heading">Task Board</h2>
      <div className="columns">
        {['todo', 'completed'].map((listName) => (
          <Droppable key={listName} droppableId={listName}>
            {(provided) => (
              <div
                className="column"
```

```
      ref={provided.innerRef}
      {...provided.droppableProps}
    >
      <h3>{listName.charAt(0).toUpperCase() + listName.slice(1)}</h3>
      {tasks[listName].map((task, index) => (
        <Draggable key={task.id} draggableId={task.id} index={index}>
          {(provided) => (
            <div
              className="task-card"
              ref={provided.innerRef}
              {...provided.draggableProps}
              {...provided.dragHandleProps}
            >
              <h4>{task.name}</h4>
              <p className="task-description">{task.description}</p>
              <p>Due: {task.endDate} at {task.time}</p>
              {listName === 'todo' && (
                <button onClick={() => markAsCompleted(task.id)}>
                  Mark as Completed
                </button>
              )}
              <button onClick={() => {
                console.log('Deleting task from:', listName, 'Task ID:', task.id);
                deleteTask(listName, task.id);
              }}>
                Delete
              </button>
            </div>
          )}
        </Draggable>
      ))}
      {provided.placeholder}
    </div>
  )}
</Droppable>
```

```
        ))}
      </div>
    </div>
  </DragDropContext>
);};
export default Board;
```

In Taskie, smooth and responsive drag-and-drop interactions are achieved by:

- Optimizing component rendering using React.memo.
- Batching state updates to prevent multiple re-renders.
- Leveraging requestAnimationFrame for smoother animations.
- Adding CSS transitions for visual feedback.
- Passing minimal data during drag events to reduce the payload.
- Debouncing expensive operations like drag-end callbacks.
- Improving accessibility for keyboard users.

These practices ensure that drag-and-drop in Taskie is fast, responsive, and user-friendly.

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

When handling large lists of tasks in terms of performance, here are some best practices:

- Virtualization: Using libraries like react-window or react-virtualized to render only the visible portion of the task list, which reduces the number of DOM elements created and improves rendering performance.
- Pagination or Infinite Scrolling: Instead of loading all tasks at once, load tasks in chunks using pagination or infinite scrolling to reduce the initial page load time.
- Memoization: Use React.memo, useMemo, or useCallback to prevent unnecessary re-renders of list items and avoid recomputation for unchanged tasks.
- Lazy Loading: Load task-related data or components only when they are needed. For example, defer loading of completed tasks until a user scrolls down or clicks on the completed section.
- Batch State Updates: Group multiple state changes into a single update to prevent excessive re-renders. React automatically batches updates within event handlers.
- Efficient Data Structures: Using appropriate data structures like Maps or Sets for faster lookups, and avoid modifying the original task array directly to keep updates efficient.
- Optimized Drag-and-Drop: For drag-and-drop functionality, batch the movement of tasks and use lightweight data structures to handle task reordering. Avoid complex computations during drag operations.
- Debouncing or Throttling: When performing operations like filtering or searching within large lists, debounce or throttle these operations to reduce the number of updates triggered during user input.
- CSS Transitions and Animations: Avoid heavy animations or transitions for each task item when dealing with large lists, or use GPU-accelerated CSS properties like transform and opacity.
- Use Web Workers: Offload heavy computations (like filtering or sorting large datasets) to a Web Worker to prevent blocking the main thread and improve UI responsiveness.

These best practices can help ensure smooth performance even when dealing with large numbers of tasks.

Conclusion:

By creating Taskie, I gained a deep understanding of building a comprehensive task management system in React. I learned how to effectively manage state across different components using React Context and how to handle CRUD operations for task management. Implementing drag-and-drop functionality with `react-beautiful-dnd` enhanced my understanding of creating intuitive and interactive UI experiences. Additionally, I explored techniques for optimizing performance when dealing with large data sets, such as virtualization and memoization. The project also helped me reinforce my knowledge of persisting data using local storage and ensuring smooth, responsive user interactions.