1. Task: Implementing a Custom Grid System with React and CSS

Objective: Develop a responsive grid system using React components and CSS.

Approach:

- Created a custom grid component in React to handle layout structures.
- Implemented a flexible grid layout with column widths configurable for different screen sizes.
- Utilized React Context to manage grid settings globally within the application.
- Ensured responsiveness by adapting grid layouts to various breakpoints.

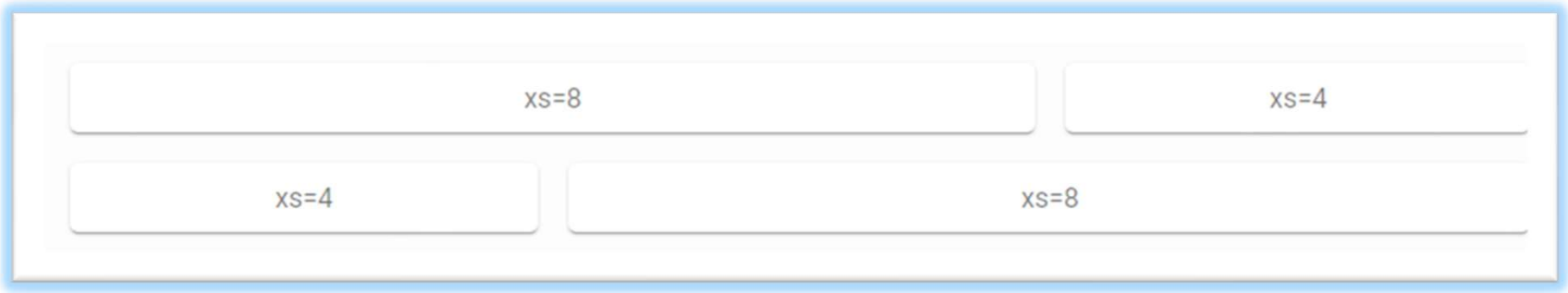


Fig: Implementing a custom grid

Key Achievements:

- **Context Management:** Utilized React Context to manage grid settings globally within the application.
- **Dynamic Column Widths:** Implemented a flexible grid layout where column widths are specified with integer values (1-12) for adaptability across breakpoints.
- **Component Composition:** Created a modular Grid component allowing nested child elements to define their column sizes.
- **CSS Flexbox:** CSS Flexbox for layout flexibility, enabling items to wrap and adjust based on available space.

2. Task: Developing a To-Do List App with RTK Query

Objective: Build a to-do list application integrated with Redux Toolkit Query.

Approach :

- Utilized Redux Toolkit for state management, leveraging RTK Query for data fetching.
- Implemented CRUD (Create, Read, Update, Delete) functionality for tasks.
- Integrated asynchronous actions with Redux Toolkit Query to manage API interactions seamlessly.

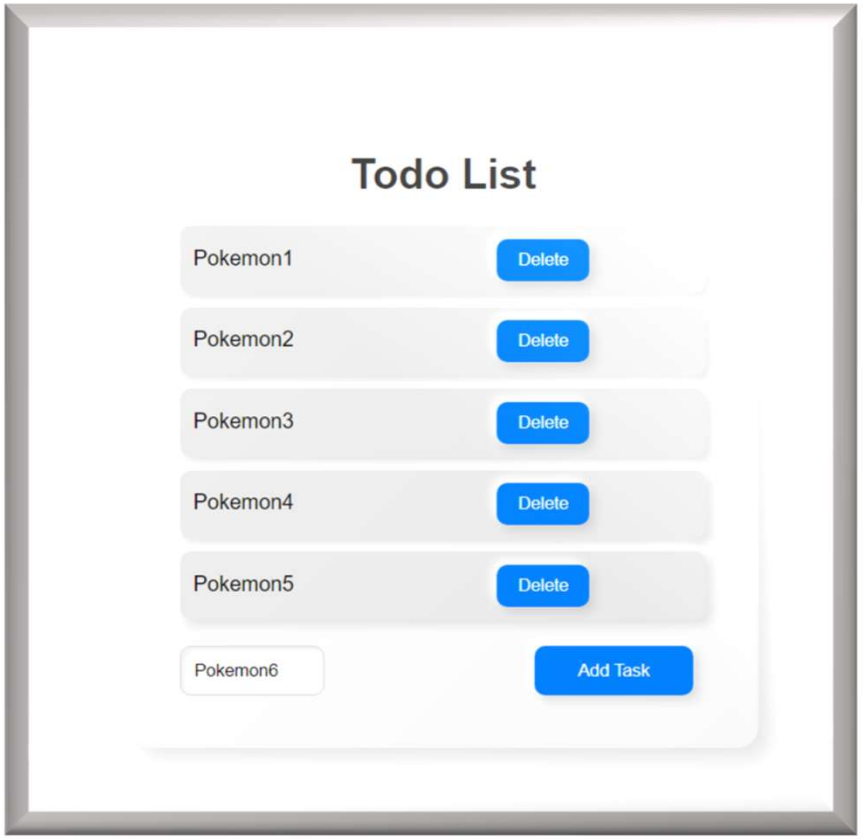


Fig: Implementing a To-Do List

Key Achievements:

1.Redux Toolkit Integration:

- •Integrated Redux Toolkit, leveraging the createSlice and createAsyncThunk utilities for state management and asynchronous data fetching.
- •Created a centralized store with a tasks reducer using configureStore for state initialization.

2.Asynchronous Data Fetching:

- •Implemented asynchronous data fetching using createAsyncThunk to retrieve tasks from a RESTful API endpoint (http://localhost:4000/tasks).
- •Utilized fetchTasks thunk action to populate tasks in the Redux store upon component mount using useEffect.

3.CRUD Operations:

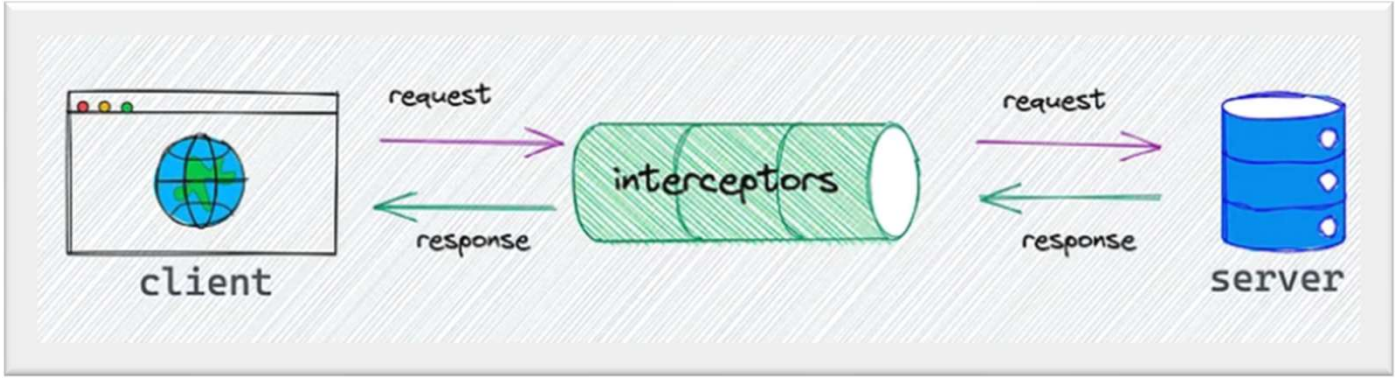
- •Enabled task creation (handleAddTask) by sending POST requests to the API and updating the Redux store with the new task using the addTask reducer.
- •Supported task deletion (handleDeleteTask) by sending DELETE requests to the API and removing the task from the Redux store using the deleteTask reducer.

3. Task: Building a React Vite Application with TypeScript

Objective: Create a modern React application using Vite and TypeScript, integrating Axios with RTK Query.

Approach :

- Set up a project scaffold using Vite for rapid development.
- Implemented TypeScript for type-safe development, enhancing code quality and developer productivity.
- Integrated Axios with Redux Toolkit Query for efficient data fetching and caching.
- Developed an Axios interceptor to handle global request and response logic, adding custom headers.



Key Achievements:

1.Project Setup and Configuration:

- •Utilized Vite for a fast and optimized development environment.
- •Configured Axios as the HTTP client for making API requests with customizable interceptors.

2.Redux Toolkit Query Integration:

- •Implemented Redux Toolkit Query (createApi) to manage API endpoints and data fetching logic.
- •Created a baseQuery function (axiosBaseQuery) to handle HTTP requests using Axios with error handling and response interception.

3.Component Composition:

- •Developed functional components (App and PostList) using React functional components
- •Used hooks (useGetPostsQuery) from Redux Toolkit Query for fetching and managing post data.

4.API Interceptors:

- •Implemented Axios interceptors to modify outgoing requests and incoming responses globally.
- •Added a custom header (Channel: SarveshGupta) to all outgoing requests for identification and logging purposes.

Application Flow:

1.Component Rendering:

- Rendered the App component as the main entry point, which then renders the PostList component for displaying fetched post data.

2.Data Fetching:

- Utilized useGetPostsQuery hook within the PostList component to fetch posts asynchronously from the API endpoint.

3.Error Handling:

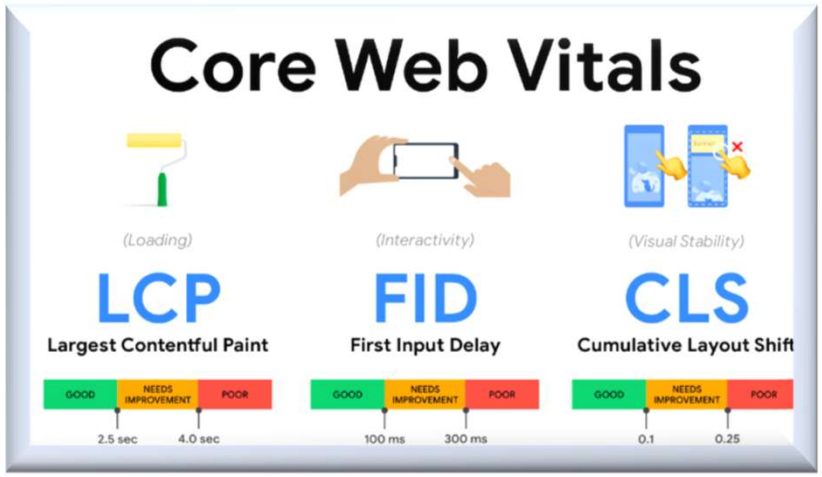
- Managed loading states (isLoading), error handling (error), and data display (data) within the PostList component based on API request status.

4. Task: Web Vitals Presentation

Objective: Prepare an informative presentation on Web Vitals.

Approach :

- Defined Core Web Vitals (CLS, FID, LCP) and their impact on user experience and SEO.
- Discussed strategies to measure and improve Web Vitals scores using tools like Lighthouse.
- Explored optimization techniques such as lazy loading, image optimization, and resource prefetching to enhance performance.



1.Introduction to Web Vitals:

- Explain what Web Vitals are and why they are important for measuring user-centric performance on the web.
- Highlight the three core Web Vitals metrics: Largest Contentful Paint (LCP), First Input Delay (FID), and Cumulative Layout Shift (CLS).

2.Understanding Each Web Vital:

- **Largest Contentful Paint (LCP)**:
  - Define LCP as the metric that measures the render time of the largest content element visible within the viewport.
  - Discuss how LCP impacts perceived loading speed and user experience, emphasizing the importance of optimizing critical rendering paths.
- **First Input Delay (FID)**:
  - Explain FID as the delay between a user's first interaction (e.g., clicking a button) and the browser's response.
  - Explore factors affecting FID, such as JavaScript execution time, and strategies for reducing FID to improve interactivity.
- **Cumulative Layout Shift (CLS)**:
  - Describe CLS as a metric that quantifies the amount of unexpected layout shifts during page load.
  - Highlight how CLS affects visual stability and user frustration, and methods to minimize CLS by ensuring stable element positioning.

3. Optimization Techniques:

- **Performance Budgeting**:
  - Discuss the concept of performance budgeting and setting thresholds for Web Vitals to maintain optimal performance levels.
- **Critical Rendering Path Optimization**:
  - Address techniques like lazy loading, code splitting, and resource prioritization to enhance LCP and reduce time to interactive (TTI).
- **Minimizing JavaScript Execution**:
  - Explore strategies for reducing FID by optimizing JavaScript code, such as deferring non-critical scripts and using efficient event handling.
- **Layout Stability Techniques**:
  - Provide tips for preventing CLS, such as reserving space for dynamic content, specifying image dimensions, and avoiding intrusive elements.

4. Tools for Web Vitals Monitoring:

- Introduce tools like Lighthouse, and Chrome DevTools for assessing and monitoring Web Vitals metrics.
- Demonstrate how to interpret Web Vitals data and use insights to prioritize performance optimizations.

5. Task: Creating a Next.js Project with Specific Features

Objective: Set up a Next.js project with various functionalities.

Approach :

- Implemented user authentication with a login page using Next.js API routes.
- Integrated Open Weather API to display real-time weather information based on user location.
- Utilized props and state management to enable pop-up features for notifications.
- Configured dynamic routing with catch-all routes and route groups for improved navigation and SEO.

Key Features Implemented:

1. Login Page:
- Created a login page using Next.js for user authentication.
- Utilized form components and client-side routing for seamless user interaction.

2. Weather App Integration:
- Integrated OpenWeather API to fetch real-time weather data based on user-provided city names.
- Implemented a search feature (SearchBar) to query weather information and display results dynamically.

3. Pop-Up Feature:
- Implemented a custom Button component with onClick functionality to trigger pop-up alerts or dialogs.
- Utilized useState and onClick event handling to manage dynamic behavior based on user interactions.

4. Catch-All Routes:
- Configured catch-all routes to handle dynamic and nested routing within the Next.js application.
- Implemented routing logic for error handling and redirection based on specific route conditions.

5. Route Groups and Nested Routing:
- Explored nested routing and route groups within Next.js for organizing and managing complex application structures.
- Leveraged route parameters and nested components (CityWeather, HomeDynamic) to display dynamic content based on URL parameters.

Code Highlights:

Login Page (/login):

- Developed a user authentication flow using Next.js routing and form components.

Weather App (/city/cityName):

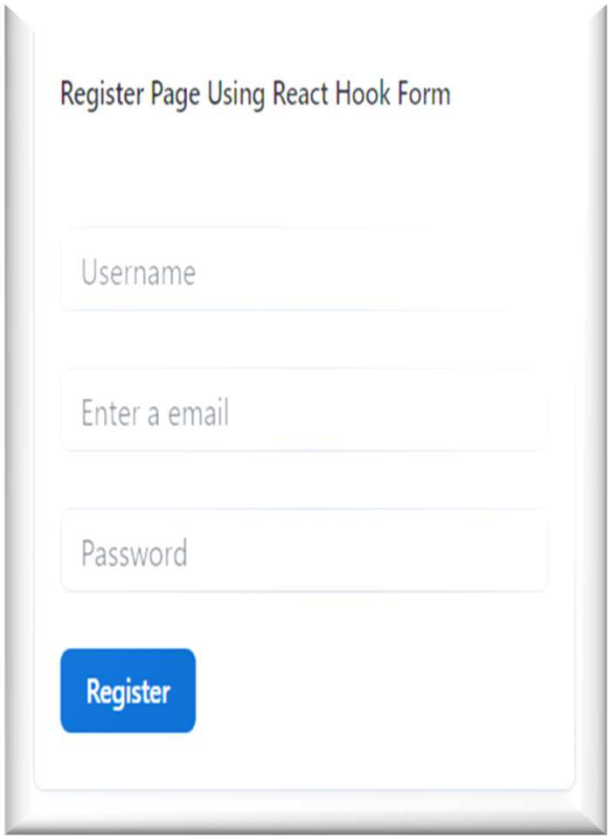
- Implemented dynamic routing to display weather information for specific cities using the OpenWeather API.
- Utilized state management (useState) and API integration (fetchWeather) to fetch and render weather data dynamically.

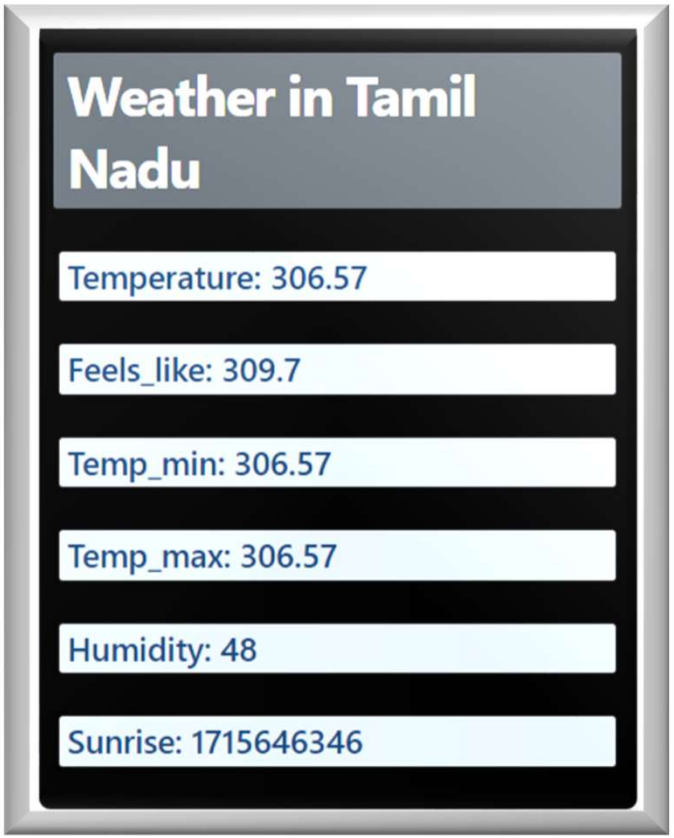
Custom Components:

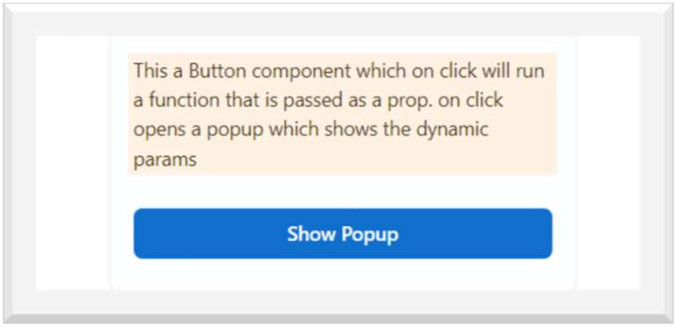
- Created reusable components (Button, Sheet, Typography) using JoyUI (Material-UI) for consistent UI design and interaction.

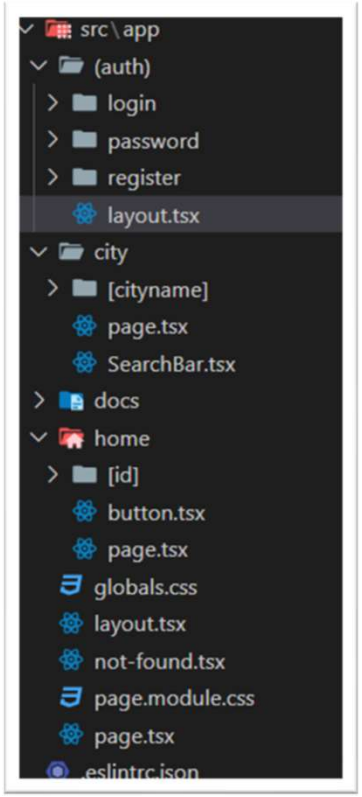
Dynamic Pop-Up Feature:

- Implemented a custom Button component with click event handling to trigger dynamic pop-up alerts or dialogs.


Register Page Using React Hook Form
Username
Enter a email
Password
Register


Weather in Tamil Nadu
Temperature: 306.57
Feels_like: 309.7
Temp_min: 306.57
Temp_max: 306.57
Humidity: 48
Sunrise: 1715646346


This a Button component which on click will run a function that is passed as a prop. on click opens a popup which shows the dynamic params
Show Popup


src\app
(auth)
login
password
register
layout.tsx
city
[cityname]
page.tsx
SearchBar.tsx
docs
home
[id]
button.tsx
page.tsx
globals.css
layout.tsx
not-found.tsx
page.module.css
page.tsx
.eslintrc.json