Contact Management App

By: Subhakanta Bhuyan

Table of Contents

- Introduction
- Technologies Used
- Project Structure
- Setup and Installation
- Running the Application
- Application Pages and Components
- Routing and Navigation
- State Management
- Data Fetching
- Styling
- Responsive Design
- Charts and Maps
- Adding New Contacts
- Viewing Contact Details
- Listing and Managing Contacts
- Error Handling
- Deployment
- Best Practices
- Future Improvements
- Conclusion

1. INTRODUCTION

PROJECT OVERVIEW

This project is a comprehensive Contact Management Application developed using modern web technologies. The primary objective of this project is to create a user-friendly interface for managing contacts, supplemented with advanced features like data visualization through charts and geographical data representation using maps. The project leverages the power of React for building the user interface, TypeScript for adding static types, and a host of other libraries to enhance functionality and user experience.

The application allows users to:

- Add new contacts with detailed information.
- View a list of all added contacts.
- View detailed information for each contact.
- Edit and delete contacts as necessary.
- Visualize COVID-19 case data over time using a line chart.
- Explore COVID-19 data across different countries on an interactive map.

By combining these features, the application provides a robust platform for managing contact information and visualizing important data trends.

FEATURES

- **Add Contacts**: Users can fill out a form to add new contacts, including details such as name, email, and phone number.
- **View Contacts**: A comprehensive list of all contacts is displayed, with options to view more detailed information.
- **Edit and Delete Contacts**: Users can edit existing contacts or delete them from the list.
- **Data Visualization:** A line chart shows the fluctuation in COVID-19 cases over time.
- **Interactive Map:** A map displays country-specific COVID-19 data, with markers indicating active cases, recoveries, and deaths.

2. TECHNOLOGIES USED

REACT

React is a popular JavaScript library for building user interfaces, particularly single-page applications where you need a fast, interactive user experience. React allows you to build reusable components, manage state efficiently, and create complex UIs with ease.

TYPESCRIPT

TypeScript is a strongly typed superset of JavaScript that compiles to plain JavaScript. It adds static types to the language, allowing for better code quality, improved debugging, and easier maintenance. TypeScript is especially useful in large projects and collaborative environments.

VITE

Vite is a build tool that provides a fast and lean development experience for modern web projects. It offers instant server start, lightning-fast hot module replacement (HMR), and optimized builds. Vite's architecture leverages native ES modules to improve performance significantly.

TAILWIND CSS

Tailwind CSS is a utility-first CSS framework for rapidly building custom user interfaces. It provides low-level utility classes that let you build completely custom designs without leaving your HTML. Tailwind is highly customizable and promotes a clean, uncluttered codebase.

REACT ROUTER

React Router is a collection of navigational components that compose declaratively with your application. It makes it simple to handle complex routing scenarios and ensures that your application's navigation is intuitive and accessible.

REDUX

Redux is a predictable state container for JavaScript applications. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. Redux centralizes the application's state and logic, making it easier to manage and debug.

REACT QUERY

React Query is a powerful data-fetching library that simplifies the process of fetching, caching, and synchronizing server data in your React applications. It abstracts the complexities of managing asynchronous data, providing a declarative and intuitive API for data fetching.

AXIOS

Axios is a promise-based HTTP client for the browser and Node.js. It simplifies making HTTP requests to fetch or save data and supports a wide range of features such as interceptors, automatic transforms, and request cancellation.

CHART.JS

Chart.js is a simple yet flexible JavaScript charting library that provides a wide range of chart types, including line charts, bar charts, pie charts, and more. It offers extensive customization options and is easy to integrate with React through the react-chartjs-2 library.

LEAFLET

Leaflet is an open-source JavaScript library for mobile-friendly interactive maps. It offers a simple, straightforward API, and is highly performant, making it an excellent choice for integrating maps into your web applications.

3. PROJECT STRUCTURE

FOLDER AND FILE ORGANIZATION

Organizing your project structure effectively is crucial for maintainability and scalability. Here is an overview of the project's folder and file organization:

```
project-root/
│
├── src/
│   ├── components/
│   │   ├── AddContactForm.tsx
│   │   ├── ContactDetails.tsx
│   │   ├── ContactList.tsx
│   │   ├── LineChart.tsx
│   │   ├── MapComponent.tsx
│   │   ├── Sidebar.tsx
│   ├── store/
│   │   ├── contactSlice.ts
│   │   └── index.ts
│   ├── router.tsx
│   ├── App.tsx
│   ├── main.tsx
│   └── index.css
│
├── public/
│   └── index.html
│
├── package.json
├── tsconfig.json
├── tailwind.config.js
└── vite.config.ts
```

EXPLANATION OF FOLDERS AND FILES

- **src/components/:** Contains all the React components used in the application.
- **AddContactForm.tsx:** Component for adding new contacts.
- **ContactDetails.tsx:** Component for displaying contact details.
- **ContactList.tsx:** Component for listing all contacts.
- **LineChart.tsx:** Component for displaying a line chart of COVID-19 cases.
- **MapComponent.tsx:** Component for displaying an interactive map.
- **Sidebar.tsx:** Component for the navigation sidebar.
- src/store/: Contains Redux store setup and contact slice.
- contactSlice.ts: Defines actions and reducers for managing contact state.

- index.ts: Sets up the Redux store.
- src/router.tsx: Configures the application's routes using React Router.
- src/App.tsx: The main application component that includes the layout and routing.
- src/main.tsx: Entry point of the application, rendering the main App component.
- src/index.css: Global CSS styles.
- public/index.html: The HTML template for the application.
- package.json: Lists the project's dependencies and scripts.
- tsconfig.json: TypeScript configuration file.
- tailwind.config.js: Configuration file for Tailwind CSS.
- vite.config.ts: Configuration file for Vite.

4. SETUP AND INSTALLATION

PREREQUISITES

- **Node.js (version 14.x or higher):** Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine.
- **npm (version 6.x or higher) or yarn (version 1.22.x or higher):** Package managers for JavaScript.

INSTALLATION STEPS

1**. Clone the Repository**

Open your terminal and run the following command to clone the repository:

```
git clone https://github.com/your-username/your-repo-name.git
```

This command creates a local copy of the project on your machine.

2. Navigate to the Project Directory

Change your working directory to the project's root directory:

```
cd your-repo-name
```

3. Install Dependencies

Install the project's dependencies using npm or yarn:

```
npm install
# or
yarn install
```

This command installs all the required packages listed in the `package.json` file.

5. RUNNING THE APPLICATION

DEVELOPMENT SERVER

To start the development server and run the application in development mode, use the following command:

```
npm run dev
# or
yarn dev
```

The development server starts and you can open your browser and navigate to `http://localhost:3000` to see the application running. The development server supports hot module replacement (HMR), which means any changes you make to the code will be reflected instantly without needing to refresh the page.

PRODUCTION BUILD

To create an optimized production build of the application, use the following command:

```
npm run build
# or
yarn build
```

This command bundles the application for production, optimizing the build for performance and efficiency. The production build files are output to the `dist` directory.

6. APPLICATION PAGES AND COMPONENTS

AddContactForm

The `AddContactForm` component provides a user interface for adding new contacts. It includes input fields for contact details such as name, email, and phone number. When the form is submitted, the new contact information is dispatched to the Redux store, updating the contact list.

Contactdetails

The `ContactDetails` component displays detailed information about a specific contact. This component is rendered when a user selects a contact from the list, providing a comprehensive view of the contact's information.

ContactList

The `ContactList` component renders a list of all contacts. Each contact entry includes options to view details, edit, or delete the contact. The list is dynamically generated based on the contacts stored in the Redux store.

LineChart

The `LineChart` component visualizes COVID-19 case data over time using Chart.js. It fetches historical data from an API and displays the data in a line chart format, providing a clear visual representation of trends and patterns.

MapComponent

The `MapComponent` integrates Leaflet maps to display COVID-19 data for different countries. It uses markers and popups to indicate active cases, recoveries, and deaths, allowing users to explore the geographical distribution of the pandemic.

Sidebar

The `Sidebar` component serves as the primary navigation for the application. It includes links to different pages such as the contact list, chart view, and map view. The sidebar ensures that navigation is intuitive and accessible from any part of the application.

7. ROUTING AND NAVIGATION

React Router Setup

React Router is used to manage the application's navigation. It allows us to define routes and associate them with specific components, ensuring that users can easily navigate between different views.

Here is an example of the router setup in `router.tsx`:

```
// src/router.tsx
import React from 'react';
import { Routes, Route } from 'react-router-dom';
import AddContactForm from './components/AddContactForm';
import ContactDetails from './components/ContactDetails';
import ContactList from './components/ContactList';
import LineChart from './components/LineChart';
import MapComponent from './components/MapComponent';

const AppRouter: React.FC = () => {
  return (
    <Routes>
      <Route path="/add-contact" element={<AddContactForm />} />
      <Route path="/contacts/:id" element={<ContactDetails />} />
      <Route path="/contact-list" element={<ContactList />} />
      <Route path="/line-chart" element={<LineChart />} />
      <Route path="/map-component" element={<MapComponent />} />
    </Routes>
  );
};

export default AppRouter;
```

SIDEBAR NAVIGATION

The `Sidebar` component provides a consistent navigation experience. Here is an example of the sidebar implementation:

```
import React from 'react';
import { Link } from 'react-router-dom';

const Sidebar: React.FC = () => {
  return (
    <div className="w-64 h-screen bg-gray-800 text-white fixed">
      <div className="p-4 text-lg font-semibold border-b border-gray-700">Navigation</div>
      <ul className="mt-4">
```

```
        <li className="hover:bg-gray-700">
          <Link to="/add-contact" className="block p-4">Add Contact</Link>
        </li>
        <li className="hover:bg-gray-700">
          <Link to="/contact-list" className="block p-4">Contact List</Link>
        </li>
        <li className="hover:bg-gray-700">
          <Link to="/line-chart" className="block p-4">Line Chart</Link>
        </li>
        <li className="hover:bg-gray-700">
          <Link to="/map-component" className="block p-4">Map</Link>
        </li>
      </ul>
    </div>
  );
};

export default Sidebar;
```

The sidebar uses `Link` components from React Router to navigate between different routes without reloading the page.

8. STATE MANAGEMENT

REDUX STORE

Redux is used to manage the application's state, ensuring that state changes are predictable and easy to trace. The Redux store is configured in `src/store/index.ts` and includes a slice for managing contacts.

Here is an example of the store setup:

```
import { configureStore } from '@reduxjs/toolkit';
import contactReducer from './contactSlice';

export const store = configureStore({
  reducer: {
    contacts: contactReducer,
  },
});

export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
```

CONTACT SLICE

The contact slice defines the actions and reducers for managing contact state. It includes actions for adding, editing, and deleting contacts.

Here is an example of the contact slice in `contactSlice.ts`:

```
// src/store/contactSlice.ts
import { createSlice, PayloadAction } from '@reduxjs/toolkit';

interface Contact {
  id: string;
  name: string;
  email: string;
  phone: string;
}

interface ContactState {
  contacts: Contact[];
}

const initialState: ContactState = {
  contacts: [],
};

const contactSlice = createSlice({
  name: 'contacts',
  initialState,
  reducers: {
    addContact: (state, action: PayloadAction<Contact>) => {
      state.contacts.push(action.payload);
    },
    updateContact: (state, action: PayloadAction<Contact>) => {
      const index = state.contacts.findIndex(contact => contact.id === action.payload.id);
      if (index !== -1) {
        state.contacts[index] = action.payload;
      }
    },
    deleteContact: (state, action: PayloadAction<string>) => {
      state.contacts = state.contacts.filter(contact => contact.id !== action.payload);
    },
  },
});

export const { addContact, updateContact, deleteContact } = contactSlice.actions;
export default contactSlice.reducer;
```

9. DATA FETCHING

React Query

React Query is used for fetching and caching data from APIs. It simplifies data fetching logic and provides hooks to manage the state of asynchronous data.

API ENDPOINTS

The application uses the following API endpoints to fetch COVID-19 data:

- Worldwide Data: `https://disease.sh/v3/covid-19/all`
- Country-Specific Data: `https://disease.sh/v3/covid-19/countries`
- Historical Data: `https://disease.sh/v3/covid-19/historical/all?lastdays=all`

Here is an example of using React Query to fetch historical COVID-19 data in the `LineChart` component:

```
// src/components/LineChart.tsx
import React from 'react';
import { Line } from 'react-chartjs-2';
import { useQuery } from '@tanstack/react-query';
import axios from 'axios';
import {
  Chart as ChartJS,
  CategoryScale,
  LinearScale,
  PointElement,
  LineElement,
  Title,
  Tooltip,
  Legend,
} from 'chart.js';

ChartJS.register(
  CategoryScale,
  LinearScale,
  PointElement,
  LineElement,
  Title,
  Tooltip,
  Legend
);

const LineChart: React.FC = () => {
  const { data, error, isLoading } = useQuery({
```

```
    queryKey: ['casesData'], // Change to array
    queryFn: async () => {
      const res = await axios.get('https://disease.sh/v3/covid-
19/historical/all?lastdays=all');
      return res.data;
    }
  });

  if (isLoading) return <p>Loading...</p>;
  if (error) return <p>Error loading data</p>;

  const chartData = {
    labels: Object.keys(data.cases),
    datasets: [
      {
        label: 'Cases',
        data: Object.values(data.cases),
        borderColor: 'rgba(75, 192, 192, 1)',
        borderWidth: 1,
      },
    ],
  };

  return <Line data={chartData} />;
};

export default LineChart;
```

10. STYLING

TAILWIND CSS

Tailwind CSS is used extensively to style the application. It provides a utility-first approach, allowing you to apply styles directly in the HTML or JSX without writing custom CSS classes.

CUSTOM STYLES

In addition to Tailwind CSS, some custom styles are defined in `src/index.css` to handle specific styling needs that Tailwind does not cover.

Here is an example of using Tailwind CSS in the `Sidebar` component:

```
const Sidebar = () => (
  <nav className="w-64 bg-gray-800 h-screen fixed">
    <ul className="mt-8">
```

```
      <li className="text-white text-lg py-4 px-6">
        <Link to="/">Contact List</Link>
      </li>
      <li className="text-white text-lg py-4 px-6">
        <Link to="/add-contact">Add Contact</Link>
      </li>
      <li className="text-white text-lg py-4 px-6">
        <Link to="/chart">COVID-19 Chart</Link>
      </li>
      <li className="text-white text-lg py-4 px-6">
        <Link to="/map">COVID-19 Map</Link>
      </li>
    </ul>
  </nav>
);
```

11. RESPONSIVE DESIGN

Mobile and Desktop Views

The application is designed to be responsive, ensuring it works well on both mobile and desktop devices. Tailwind CSS provides responsive utility classes that make it easy to apply different styles based on the screen size.

Here is an example of responsive design in the `Sidebar` component:

```
const Sidebar = () => (
  <nav className="w-64 bg-gray-800 h-screen fixed md:w-80 lg:w-96">
    <ul className="mt-8">
      <li className="text-white text-lg py-4 px-6">
        <Link to="/">Contact List</Link>
      </li>
      <li className="text-white text-lg py-4 px-6">
        <Link to="/add-contact">Add Contact</Link>
      </li>
      <li className="text-white text-lg py-4 px-6">
        <Link to="/chart">COVID-19 Chart</Link>
      </li>
      <li className="text-white text-lg py-4 px-6">
        <Link to="/map">COVID-19 Map</Link>
      </li>
    </ul>
  </nav>
);
```

The `w-64`, `md:w-80`, and `lg:w-96` classes apply different widths based on the screen size, ensuring the sidebar adjusts appropriately.

12. CHARTS AND MAPS

LINECHART COMPONENT

The `LineChart` component visualizes COVID-19 data over time. It uses the Chart.js library to render a line chart, providing a clear visual representation of case trends.

Here is an example of the `LineChart` component:

```
const LineChart = () => {
  const { data, error, isLoading } = useQuery('historicalData', fetchHistoricalData);

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error fetching data</div>;

  const chartData = {
    labels: Object.keys(data.cases),
    datasets: [
      {
        label: 'Cases',
        data: Object.values(data.cases),
        borderColor: 'rgba(75, 192, 192, 1)',
        fill: false,
      },
    ],
  };

  return <Line data={chartData} />;
};
```

MAP COMPONENT

The `MapComponent` integrates Leaflet maps to display country-specific COVID-19 data. It uses markers to indicate active cases, recoveries, and deaths, providing a geographical representation of the data.

Here is an example of the `MapComponent`:

```
import { MapContainer, TileLayer, Marker, Popup } from 'react-leaflet';
import 'leaflet/dist/leaflet.css';

const MapComponent = () => {
  const { data, error, isLoading } = useQuery('countryData', fetchCountryData);

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error fetching data</div>;

  return (
    <MapContainer center={[51.505, -0.09]} zoom={2} style={{ height: '100vh', width:
'100%' }}>
      <TileLayer url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png" />
      {data.map(country => (
        <Marker key={country.countryInfo.iso2} position={[country.countryInfo.lat,
country.countryInfo.long]}>
          <Popup>
            <strong>{country.country}</strong><br />
            Cases: {country.cases}<br />
            Recovered: {country.recovered}<br />
            Deaths: {country.deaths}
          </Popup>
        </Marker>
      ))}
    </MapContainer>
  );
};

export default MapComponent;
```

13. ADDING NEW CONTACTS

addcontactform implementation

The `AddContactForm` component provides a user interface for adding new contacts. It includes input fields for contact details and handles form submissions by dispatching actions to the Redux store.

Here is an example of the `AddContactForm` component:

```
// src/components/AddContactForm.tsx
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { addContact } from '../store/contactSlice';
import { v4 as uuidv4 } from 'uuid';
```

```
const AddContactForm: React.FC = () => {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [phone, setPhone] = useState('');
  const dispatch = useDispatch();

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    dispatch(addContact({ id: uuidv4(), name, email, phone }));
    setName('');
    setEmail('');
    setPhone('');
  };

  return (
    <form onSubmit={handleSubmit} className="p-4">
      <div>
        <label>Name</label>
        <input type="text" value={name} onChange={(e) => setName(e.target.value)} required
className="border p-2 w-full"/>
      </div>
      <div>
        <label>Email</label>
        <input type="email" value={email} onChange={(e) => setEmail(e.target.value)}
required className="border p-2 w-full"/>
      </div>
      <div>
        <label>Phone</label>
        <input type="tel" value={phone} onChange={(e) => setPhone(e.target.value)}
required className="border p-2 w-full"/>
      </div>
      <button type="submit" className="bg-blue-500 text-white p-2 mt-2">Add
Contact</button>
    </form>
  );
};

export default AddContactForm;
```

HANDLING FORM SUBMISSIONS

The form submission is handled by dispatching the `addContact` action to the Redux store. This action updates the state with the new contact information, ensuring the contact list is kept up-to-date.

14. VIEWING CONTACT DETAILS

ContactDetails Component

The `ContactDetails` component displays detailed information about a specific contact. It is rendered when a user selects a contact from the list, providing a comprehensive view of the contact's information.

Here is an example of the `ContactDetails` component:

```
// src/components/ContactDetails.tsx
import React from 'react';
import { useSelector } from 'react-redux';
import { RootState } from '../store';
import { useParams } from 'react-router-dom';

const ContactDetails: React.FC = () => {
  const { id } = useParams<{ id: string }>();
  const contact = useSelector((state: RootState) => state.contacts.contacts.find(c => c.id
=== id));

  if (!contact) {
    return <p>Contact not found</p>;
  }

  return (
    <div className="p-4">
      <h2>Contact Details</h2>
      <p>Name: {contact.name}</p>
      <p>Email: {contact.email}</p>
      <p>Phone: {contact.phone}</p>
    </div>
  );
};

export default ContactDetails;
```

Dynamic Routing

The `ContactDetails` component uses dynamic routing to render the appropriate contact details based on the URL parameter. This ensures that users can easily view detailed information for any contact by navigating to the corresponding URL.

15. LISTING AND MANAGING CONTACTS

ContactList Component

The `ContactList` component renders a list of all contacts, with options to view details, edit, or delete each contact. The list is dynamically generated based on the contacts stored in the Redux store.

Here is an example of the `ContactList` component:

```
// src/components/ContactList.tsx
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { RootState } from '../store';
import { deleteContact } from '../store/contactSlice';
import { Link } from 'react-router-dom';

const ContactList: React.FC = () => {
  const contacts = useSelector((state: RootState) => state.contacts.contacts);
  const dispatch = useDispatch();

  const handleDelete = (id: string) => {
    dispatch(deleteContact(id));
  };

  return (
    <div className="p-4">
      <h2>Contact List</h2>
      <ul>
        {contacts.map(contact => (
          <li key={contact.id} className="border p-2 mb-2">
            <p>Name: {contact.name}</p>
            <p>Email: {contact.email}</p>
            <p>Phone: {contact.phone}</p>
            <Link to={`/contacts/${contact.id}`} className="text-blue-500">View
Details</Link>
            <button onClick={() => handleDelete(contact.id)} className="text-red-500 ml-
4">Delete</button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default ContactList;
```

EDITING AND DELETING CONTACTS

The `ContactList` component provides buttons to edit or delete each contact. The delete action is handled by dispatching the `deleteContact` action to the Redux store, updating the state to remove the specified contact.

16. ERROR HANDLING

HANDLING API ERRORS

Error handling is an important aspect of any application. In this project, errors during data fetching are handled gracefully, providing user feedback and ensuring the application remains responsive.

Here is an example of error handling in the `LineChart` component:

```
const LineChart = () => {
  const { data, error, isLoading } = useQuery('historicalData', fetchHistoricalData);

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error fetching data</div>;

  const chartData = {
    labels: Object.keys(data.cases),
    datasets: [
      {
        label: 'Cases',
        data: Object.values(data.cases),
        borderColor: 'rgba(75, 192, 192, 1)',
        fill: false,
      },
    ],
  };

  return <Line data={chartData} />;
};
```

FORM VALIDATION

Form validation is implemented in the `AddContactForm` component to ensure that all required fields are filled out correctly before submission.

Here is an example of form validation:

```
const AddContactForm = () => {
  // ... other code ...

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    if (!name || !email || !phone) {
      alert('Please fill out all fields');
      return;
    }
    const newContact = { id: uuidv4(), name, email, phone };
    dispatch(addContact(newContact));
  };

  return (
    <form onSubmit={handleSubmit} className="max-w-md mx-auto p-4">
      {/* form fields */}
      <button type="submit" className="bg-blue-500 hover:bg-blue-700 text-white font-bold
py-2 px-4 rounded">
        Add Contact
      </button>
    </form>
  );
};
```

17. DEPLOYMENT

Netlify Deployment

- **Push to GitHub:** Ensure your code is pushed to a GitHub repository.
- **Import to Netlify:** Go to the Netlify dashboard and import the GitHub repository.
- **Configure Settings:** Set the build command to `npm run build` and the publish directory to `dist`.
- **Deploy:** Click the deploy button to initiate the deployment process.

Netlify provides a URL where your application is accessible once the deployment is complete.

18. CONCLUSION

This documentation provides a comprehensive guide to the React + TypeScript project using VITE. It covers the project structure, setup, key components, state management, data fetching, error handling, and deployment. By following this guide, developers can understand the project's architecture, implement new features, and maintain the codebase effectively.