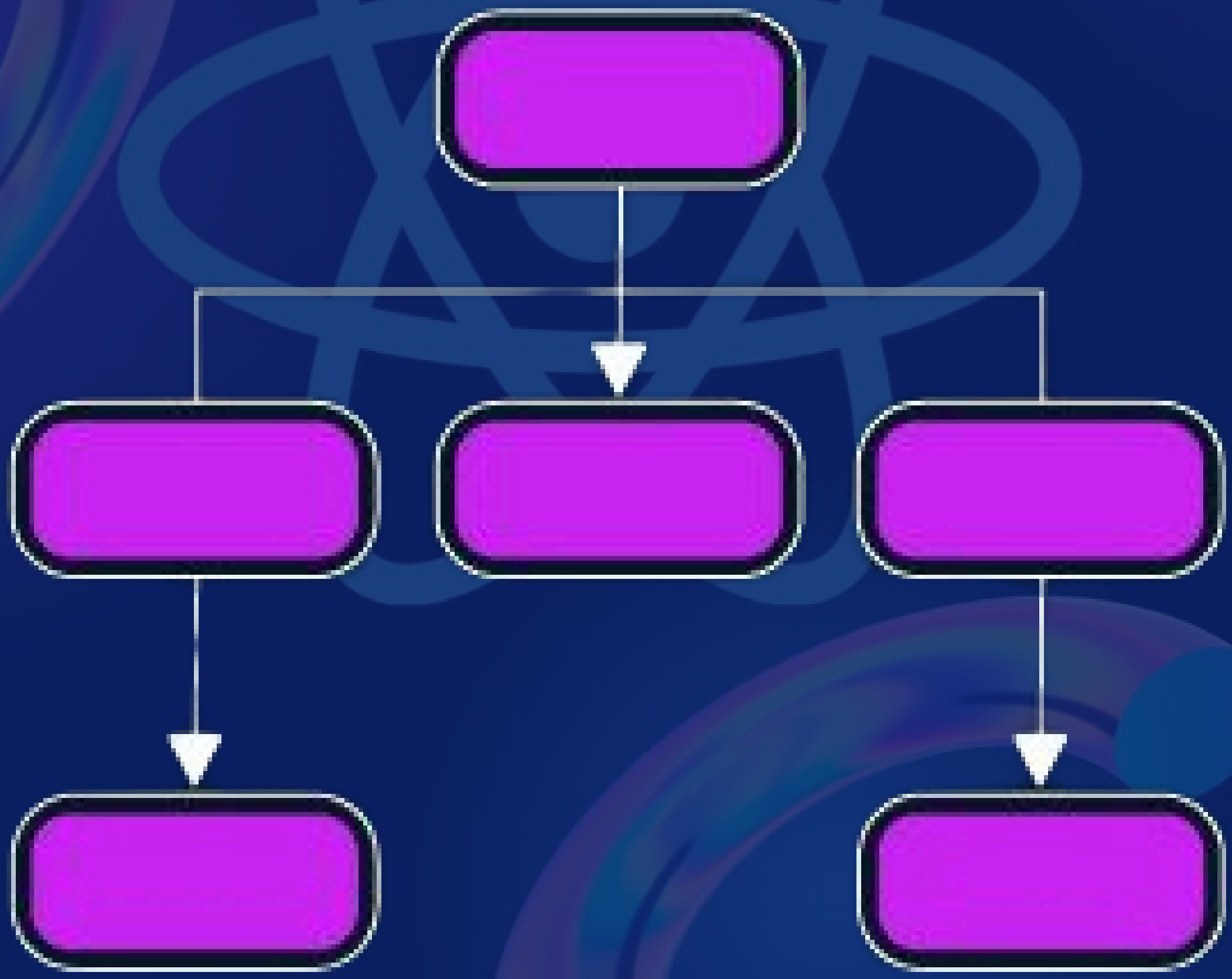
Day 21

Global State Management In React (Part 3)



@oluwakemi Oluwadahunsi

The “All-mighty” Redux

Redux is a **predictable state container** for JavaScript apps. In React applications, it is commonly used to manage global state, which means **keeping state outside** of individual components and sharing it across the app.

While using **Context API + Reducers** for global state management, it is relatively basic and works best for **smaller applications** where global state management is not too complex, **Redux** is more **powerful** and **structured**, designed to handle **complex applications** with large state needs.

It is built around the concept of a **single source of truth** (a centralized store) and enforces a strict pattern of how state changes (through actions and reducers), which makes the data flow more predictable and maintainable in larger apps.

@oluwakemi Oluwadahunsi

Redux enables centralized state management, where all state is managed in one place (the store), and changes to state follow a unidirectional data flow pattern, making your application more predictable and easier to debug.

Key Concepts of Redux

- **Store:** The central place where the state of your entire application lives.
- **Actions:** Descriptions of what should be done (e.g., fetching data, adding an item). Actions are plain JavaScript objects with a type field.
- **Reducers:** Pure functions that take the current state and an action as arguments and return a new state.
- **Dispatch:** A function used to send actions to the Redux store.
- **Selectors:** Functions used to extract data from the Redux store.

@oluwakemi Oluwadahunsi

How Redux Works:

Redux operates based on a few principles:

- Single source of truth: The state of your whole application is stored in an object tree within a single store.

- State is read-only: The only way to change the state is to dispatch an action.

- Changes are made with pure functions: To specify how the state tree is transformed by actions, you write pure reducers.

@oluwakemi Oluwadahunsi

Setting Up Redux In React

To show the steps involved in managing global state with Redux in a React project, let's walk through setting up Redux and using it to fetch data from an API.

There are 2 patterns of working with Redux in a React project, either by the **basic method** (separating the actions, states, reducers, and thunk in different files), or by directly installing the Redux full kit known as the Redux Toolkit, which has everything needed for your setup and you can declare all states, reducers, etc. un a single file.

Although, the first method is deprecated but still works, this is for the purpose of explanation only, **it is recommended that you use the Redux Toolkit method.**

@oluwakemi Oluwadahunsi

The Basic Method

Step 1: Setting Up Redux with React

To begin, let's install the required packages for Redux and React integration.

In your terminal, navigate to your project’s root folder (You must have created a React project already), and type in:
“**npm install redux react-redux**” like this:

```
kemil@DESKTOP-QKVDT62 MINGW64 /c/Desktop/My-Projects/my-vite-app
$ npm install redux react-redux
```

- **redux:** The core Redux library.
- **react-redux:** Bindings that allow React components to interact with Redux.

Step 2: Create a Redux Store

In Redux, the **store** holds the global state. We'll create a store using the **createStore** method from Redux.

For more organization, create a new folder in your **root folder**, named “**utils**”, and create a file in the utils folder named “**store.js**”

```
// /utils/store.js
import { createStore } from 'redux';
import rootReducer from './reducers';

const store = createStore(rootReducer);

export default store;
```

Here, we use **rootReducer**, which combines all the reducers in the application (more on reducers in a moment).

You will get a linting warning that the **createStore** method is deprecated, but don’t worry about it, it will still work.

@oluwakemi Oluwadahunsi

Step 3: Defining Actions

Actions in Redux are plain JavaScript objects that describe an event. You can think of them as “messages” that tell Redux what you want to do.

In your “**utils**” folder, create another folder named “**actions**”, in this folder, create a file named “**dataActions.js**”

```
// /utils/actions/dataActions.js
export const fetchDataRequest = () => ({
  type: 'FETCH_DATA_REQUEST',
});

export const fetchDataSuccess = (data) => ({
  type: 'FETCH_DATA_SUCCESS',
  payload: data,
});

export const fetchDataFailure = (error) => ({
  type: 'FETCH_DATA_FAILURE',
  payload: error,
});
```

@oluwakemi Oluwadahunsi

- **fetchDataRequest:** Indicates the start of the data fetching process.
- **fetchDataSuccess:** Fired when the data is successfully fetched.
- **fetchDataFailure:** Fired if there's an error fetching data.

Step 4: Creating a Reducer

Reducers specify **how the application's state changes in response to actions**. It takes the **current state** and **an action** and returns a **new state,** just like we learned yesterday.

In your “utils” folder, create another folder named “reducers”, in this folder, create a file named “dataReducers.js”

In your dataReducers.js file:

```
// /utils/reducers/dataReducer.js
const initialState = {
  loading: false,
  data: [],
  error: '',
};

const dataReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'FETCH_DATA_REQUEST':
      return {
        ...state,
        loading: true,
      };
    case 'FETCH_DATA_SUCCESS':
      return {
        ...state,
        loading: false,
        data: action.payload,
        error: '',
      };
    case 'FETCH_DATA_FAILURE':
      return {
        ...state,
        loading: false,
        data: [],
        error: action.payload,
      };
    default:
      return state;
  }
};

export default dataReducer;
```

initial state of the reducers

each state is updated

@oluwakemi Oluwadahunsi

In the file above:

- **Initial State:** loading, data, and error represent the states during the API call.
- **FETCH_DATA_REQUEST:** Sets loading to true while data is being fetched.
- **FETCH_DATA_SUCCESS:** Sets data when the API call is successful.
- **FETCH_DATA_FAILURE:** Updates the error if the API call fails.

Note that all “**case**” values are written in UPPER CASE letters

Step 5: Combine Reducers (if multiple reducers)

We often have **multiple reducers** in a large app. You can combine them using Redux's **combineReducers** method.

```
// /utils/reducers/index.js
import { combineReducers } from 'redux';
import dataReducer from './dataReducer';

const rootReducer = combineReducers({
  data: dataReducer,
});

export default rootReducer;

```

You can import as many reducers as you have in your project, but for this project we only have dataReducers.

@oluwakemi Oluwadahunsi

Step 6: Connecting Redux to React

To make the **Redux store** accessible to React components, we use the **Provider** component from react-redux.

You will have to wrap your entire application with the Provider component either in your “**index.js**” file (if using CRA method) or “**main.jsx**” file (if using Vite method) like this:

```
// src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import App from './App';
import store from './store';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Provider wrapping around the entire app

Provider: This makes the Redux store available to all components in your app.

This is a CRA method, so it’s in index.js

Step 7: Fetching Data from API using Redux and React

For this example, we will fetch a list of users from the JSONPlaceholder API.

Add this in your dataActions file:

```
// utils/actions/dataActions.js
export const fetchData = () => {
  return async (dispatch) => {
    dispatch(fetchDataRequest());
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/users');
      const data = await response.json();
      dispatch(fetchDataSuccess(data));
    } catch (error) {
      dispatch(fetchDataFailure(error.message));
    }
  };
};
```

Remember the actions functions we declared previously, the response from the endpoint, if successfull is dispatched using the **fetchDataSuccess** functions and if it fails, the error is dispatched using the **fetchDataFailure** function.

@oluwakemi Oluwadahunsi

This **fetchData** function uses **thunk middleware** to handle asynchronous logic. To use this, install redux-thunk

In your terminal, type: "**npm install redux-thunk**"

Thunk in Redux is a **middleware** that allows you to write action creators that return a function instead of an action object. This function can perform asynchronous operations (like API calls) and dispatch actions based on the result, enabling **side effects in Redux**.

Now, update the **store** to use **thunk**:

```
// /utils/store.js
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));

export default store;
```

Step 8: Create React Component to Display Fetched Data

Create a component named “UserList.jsx/jsx” in your “components” folder”

```jsx

import { useEffect } from "react";
import { useDispatch, useSelector } from "react-redux";
import { fetchData } from "../actions/dataActions";

const UserList = () => {
  const dispatch = useDispatch();
  const { loading, data, error } = useSelector((state) => state.data);

  useEffect(() => {
    dispatch(fetchData());
  }, [dispatch]);

  return (
    <div>
      {loading && <p>Loading ...</p>}
      {error && <p>Error: {error}</p>}
      <div className="container">
        {data.map((user) => (
          <div key={user.id} className="wrapper">
            <img
              src={user.img}
              alt="user-image"
              title="user"
              className="avatar"
            />
            <h2>{user.name}</h2>
            <p>{user.role}</p>
            <p>Phone: {user.phone}</p>
            <address>
              Address: {user.address.suite}, {user.address.street}, {user.address.city}.
            </address>
            <a href={`mailto:${user.email}`}>{user.email} </a>
            <p>Website: {user.website}</p>
          </div>
        ))}
      </div>
    </div>
  );
};

export default UserList;
```

- **useDispatch:** Dispatches the fetchData action when the component mounts.
- **useSelector:** Selects the loading, data, and error state from the Redux store.
- The **useEffect hook** is used to trigger the API call when the component renders.

Step 9: Putting it All Together

In your “App.js/jsx” file, import the UserList component:

```
// src/App.js
import React from 'react';
import UserList from './components/UserList';

function App() {
  return (
    <div>
      <h1>Redux API Fetch Example</h1>
      <UserList />
    </div>
  );
}

export default App;
```

Setup Using Redux Toolkit

Redux Toolkit (RTK) is the official, recommended way to write Redux logic. One of its key features is **slices**, which help to streamline the process of writing reducers, actions, and handling side effects.

In the example we used for the basic Redux to fetch data from an API, **slices** can simplify and organize our code by bundling action creators and reducers together.

In Redux Toolkit, a **slice** represents a "**slice of state**" and includes all the logic related to that state (reducers, actions, and initial state) in a **single file**.

Let's refactor our previous API fetching example using Redux Toolkit.

@oluwakemi Oluwadahunsi

Step 1: Setting Up Redux Toolkit

First, install Redux Toolkit by typing this in your terminal (create a fresh project please):
“**npm install @reduxjs/toolkit**”

We no longer need to install **redux-thunk** separately, as **Redux Toolkit** includes **thunk** by default.

Step 2: Creating a Slice

We will create a slice to manage the global state for fetching users. This slice will automatically generate **action creators** and a **reducer**.

Create a **utils** folder as we did in the first example, and create another folder in it named “**slices**”. In the “slices” folder, create a file named “**userSlice.js**” like this:

```
// utils/slices/userSlice.js
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

// Initial state for the slice
const initialState = {
  loading: false,
  data: [],
  error: '',
};

// Async thunk to fetch users
export const fetchUsers = createAsyncThunk('users/fetchUsers', async () => {
  const response = await fetch('https://jsonplaceholder.typicode.com/users');
  const data = await response.json();
  return data;
});

// Create the slice
const userSlice = createSlice({
  name: 'users',
  initialState,
  reducers: {}, // You can add more reducers here if needed
  extraReducers: (builder) => {
    builder
      .addCase(fetchUsers.pending, (state) => {
        state.loading = true;
      })
      .addCase(fetchUsers.fulfilled, (state, action) => {
        state.loading = false;
        state.data = action.payload;
        state.error = '';
      })
      .addCase(fetchUsers.rejected, (state, action) => {
        state.loading = false;
        state.data = [];
        state.error = action.error.message;
      });
  },
});

export default userSlice.reducer;
```

initial state defined

async function to fetch users

reducers used for updating the states

In the userSlice.js file:

- **createSlice:** This function helps you write Redux logic by automatically generating **action creators** and a **reducer**.

- **createAsyncThunk:** Handles asynchronous actions (like API calls). It automatically dispatches the **pending**, **fulfilled**, and **rejected** actions based on the promise lifecycle.

- **extraReducers:** Handles actions generated by **createAsyncThunk**. When the API call is pending, it sets loading to true. If the call is successful, it sets the data with the fetched results. If it fails, it sets an error message.

Step 3: Setting Up the Redux Store

Next, let's configure the Redux store with the userSlice reducer.

Create a “store” file in your utils folder:

```js
// src/store.js
import { configureStore } from '@reduxjs/toolkit';
import userReducer from './features/userSlice';

const store = configureStore({
  reducer: {
    users: userReducer, // You can add other reducers here if needed
  },
});

export default store;
```

Instead of using the deprecated **createStore** method in the old setup, we use Redux Toolkit's **configureStore**, and you don’t need to worry about adding middleware like **thunk**, it’s included by default.

Step 4: Connecting Redux Store to React

As before, we need to wrap the app with the **Provider** to give the Redux store access to the components (refer to step 6 in page 13) .

Step 5: Fetching Data and Using the Slice in a React Component

Now, let's update the React component (UserList.js/jsx) to fetch users using the **fetchUsers thunk** and display the data.

```jsx

import { useEffect } from "react";
import { useDispatch, useSelector } from "react-redux";
import { fetchUsers } from "../features/userSlice";

const UserList = () => {
  const dispatch = useDispatch();
  const { loading, data, error } = useSelector((state) => state.users);

  useEffect(() => {
    dispatch(fetchUsers());
  }, [dispatch]);

  return (
    <div>
      <h1>Users List</h1>
      {loading && <p>Loading ...</p>}
      {error && <p>Error: {error}</p>}
      <div className="container">
        {data.map((user) => (
          <div key={user.id} className="wrapper">
            <img
              src={user.img}
              alt="user-image"
              title="user"
              className="avatar"
            />
            <h2>{user.name}</h2>
            <p>{user.role}</p>
            <p>Phone: {user.phone}</p>
            <address>
              Address: {user.address.suite}, {user.address.street},{" "}
              {user.address.city}.{" "}
            </address>
            <a href={`mailto:${user.email}`}>{user.email}</a>
            <p>Website: {user.website}</p>
          </div>
        ))}
      </div>
    </div>
  );
};

export default UserList;
```

In this userList.js file:

- **useDispatch():** Used to dispatch the **fetchUsers** action when the component mounts.
- **useSelector():** Selects the **loading**, **data**, and **error** from the Redux store's state managed by **userSlice**.
- **useEffect():** Calls the **fetchUsers** action as soon as the component is rendered.

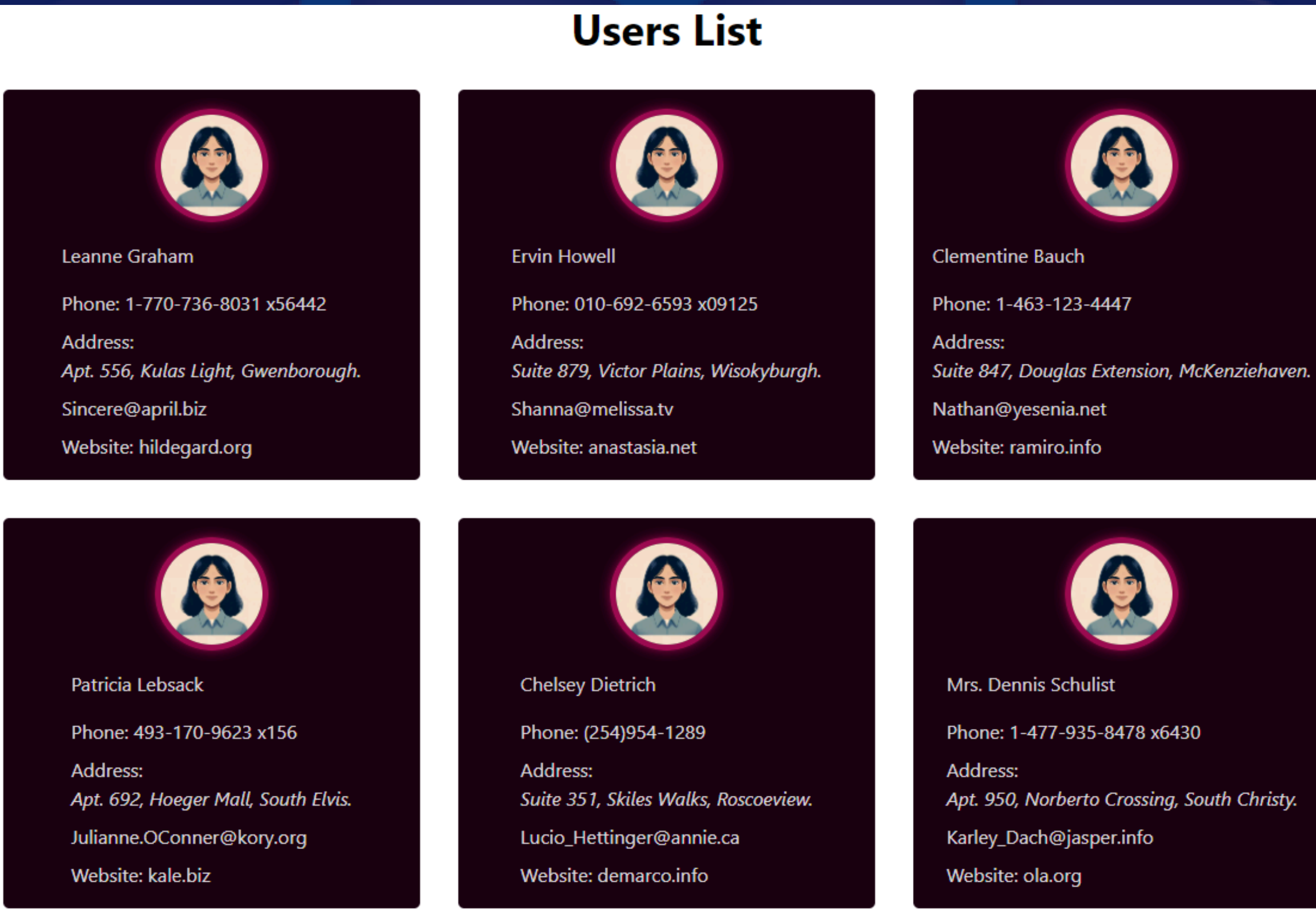
Step 6: Putting Everything Together

Now, import the UserList component in your "App.js/jsx" file for output (refer to step 9 in page 17 above).

Do note that the component extension is .js for CRA method and .jsx for Vite method.

Now run your application to view the ftеched data in your browser. You should see a list of user's data displayed.

I added some stylings to mine using Tailwind CSS. You can add yours too. Here is mine:



Remember to add an image in your folder as there are no images from the endpoint.

@oluwakemi Oluwadahunsi

Best Practices for Using Redux

1. Use Redux only for Global State:

If a piece of state is only used by one component or its immediate children, local state (using useState) is better.

2. Keep Reducers Pure:

Reducers should always be pure functions, returning new state without side effects.

3. Use Redux Toolkit:

It simplifies the syntax of Redux and reduces boilerplate.

4. Combine Thunks or Sagas:

For complex asynchronous logic, you can combine Redux with **redux-thunk** (for simple async logic) or **redux-saga** (for complex workflows).

@oluwakemi Oluwadahunsi

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app