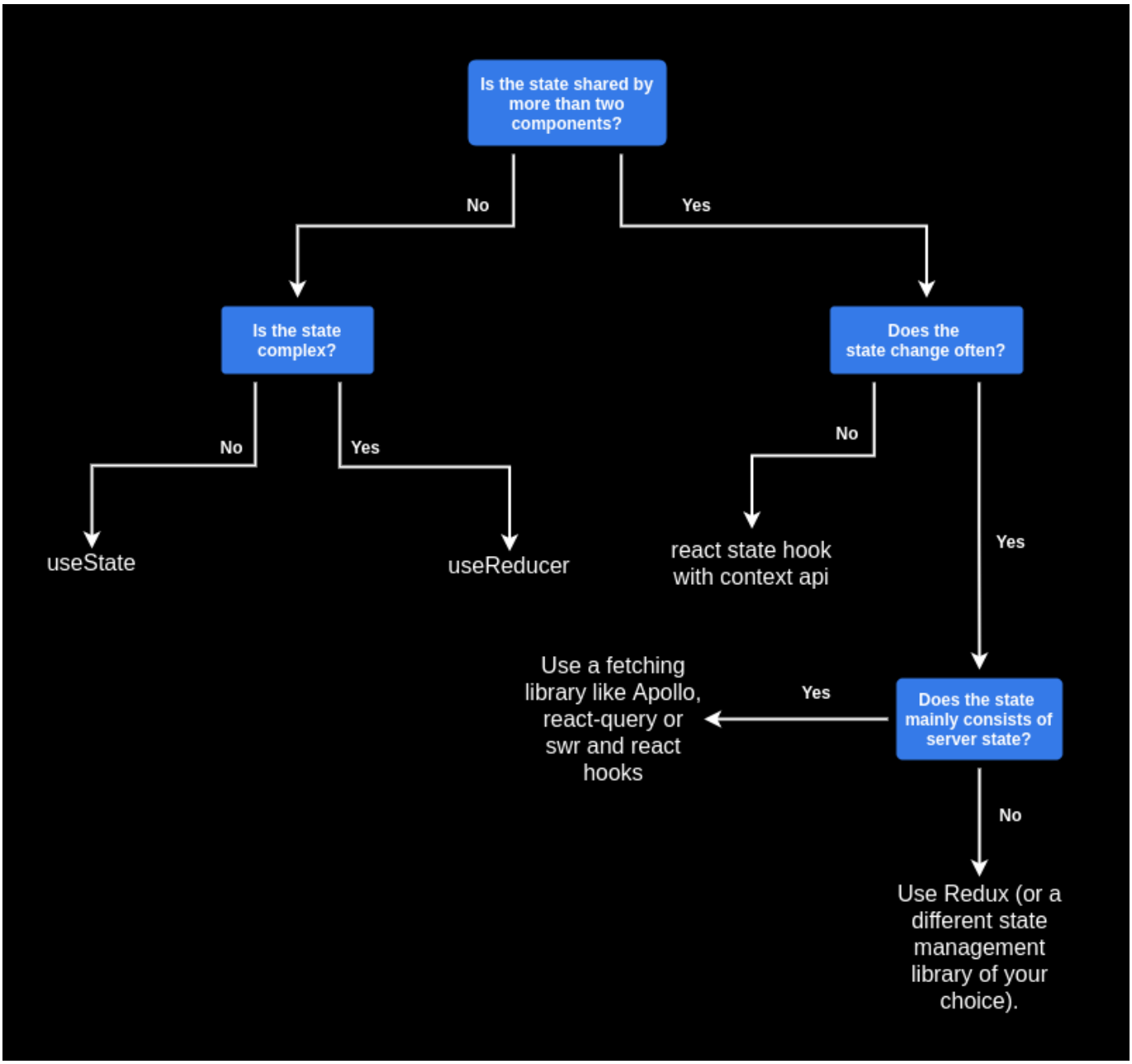
Khóa học Front-end ATT Lab - ReactJS

1. Tìm hiểu về global state



1.1. Redux và Redux Toolkit: Từ Cơ Bản Đến Thực Hành

1. Giới thiệu về Redux và các khái niệm cần nắm

Redux là một thư viện quản lý trạng thái (state management) phổ biến cho các ứng dụng JavaScript, đặc biệt là trong React. Nó giúp bạn quản lý trạng thái của ứng dụng một cách hiệu quả và dễ dàng hơn. Các khái niệm chính trong Redux:

Store: Nơi lưu trữ toàn bộ trạng thái của ứng dụng. **Action:** Một object mô tả sự kiện xảy ra trong ứng dụng. **Reducer:** Hàm xử lý các action và cập nhật state. **Dispatch:** Phương thức để gửi action đến store.

Luồng dữ liệu trong Redux:

- User tương tác với UI
- Tương tác kích hoạt một action
- Action được dispatch đến store
- Reducer xử lý action và cập nhật state
- UI cập nhật dựa trên state mới

2. Ưu điểm của Redux Toolkit so với Redux thuần

Redux Toolkit là một bộ công cụ chính thức để phát triển ứng dụng Redux hiệu quả. Nó cung cấp các tiện ích giúp đơn giản hóa quy trình phát triển Redux.

Ưu điểm chính:

- Cấu hình đơn giản: Redux Toolkit cung cấp hàm `configureStore()` giúp cấu hình store dễ dàng hơn.
- Giảm boilerplate code: Với `createSlice()`, bạn có thể định nghĩa reducers và actions trong một nơi.
- Immutability mặc định: Sử dụng `Immer` để cho phép viết "mutating" logic trong reducers.
- Tích hợp DevTools: Redux DevTools được cấu hình sẵn.
- Thunk middleware: Được tích hợp sẵn để xử lý async logic.

3. Thực hành tạo ứng dụng đơn giản với Redux Toolkit

Chúng ta sẽ tạo một ứng dụng Todo List đơn giản sử dụng React và Redux Toolkit.

Bước 1: Cài đặt các thư viện cần thiết

```
npx create-react-app redux-toolkit-todo
```

```
cd redux-toolkit-todo
```

```
npm install @reduxjs/toolkit react-redux
```

Bước 2: Tạo Redux Store

- Tạo file src/app/store.js:

```
import { configureStore } from "@reduxjs/toolkit";
import todoReducer from "../features/todo/todoSlice";

export const store = configureStore({
  reducer: {
    todos: todoReducer,
```

```
  },
});
```

Bước 3: Tạo Todo Slice

- Tạo file src/features/todo/todoSlice.js:

```
import { createSlice } from "@reduxjs/toolkit";

const initialState = {
  todos: [],
};

export const todoSlice = createSlice({
  name: "todo",
  initialState,
  reducers: {
    addTodo: (state, action) => {
      state.todos.push({
        id: Date.now(),
        text: action.payload,
        completed: false,
      });
    },
    toggleTodo: (state, action) => {
      const todo = state.todos.find((todo) => todo.id === action.payload);
      if (todo) {
        todo.completed = !todo.completed;
      }
    },
    removeTodo: (state, action) => {
      state.todos = state.todos.filter((todo) => todo.id !==
action.payload);
    },
  },
});

export const { addTodo, toggleTodo, removeTodo } = todoSlice.actions;

export default todoSlice.reducer;
```

Bước 4: Cung cấp Store cho ứng dụng React

- Sửa file src/index.js:

```
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import { store } from "./app/store";
import App from "./App";
```

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

Bước 5: Tạo các components

Tạo file src/components/AddTodo.js:

```
import React, { useState } from "react";
import { useDispatch } from "react-redux";
import { addTodo } from "../features/todo/todoSlice";

function AddTodo() {
  const [text, setText] = useState("");
  const dispatch = useDispatch();

  const handleSubmit = (e) => {
    e.preventDefault();
    if (text.trim()) {
      dispatch(addTodo(text));
      setText("");
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Add a new todo"
      />
      <button type="submit">Add</button>
    </form>
  );
}

export default AddTodo;
```

- Tạo file src/components/TodoList.js:

```
import React from "react";
import { useSelector, useDispatch } from "react-redux";
```

```
import { toggleTodo, removeTodo } from "../features/todo/todoSlice";

function TodoList() {
  const todos = useSelector((state) => state.todos.todos);
  const dispatch = useDispatch();

  return (
    <ul>
      {todos.map((todo) => (
        <li key={todo.id}>
          <span
            style={{ textDecoration: todo.completed ? "line-through" :
"none" }}
            onClick={() => dispatch(toggleTodo(todo.id))}
          >
            {todo.text}
          </span>
          <button onClick={() =>
dispatch(removeTodo(todo.id))}>Delete</button>
        </li>
      ))}
    </ul>
  );
}

export default TodoList;
```

Bước 6: Cập nhật App component

- Sửa file src/App.js:

```
import React from "react";
import AddTodo from "./components/AddTodo";
import TodoList from "./components/TodoList";

function App() {
  return (
    <div>
      <h1>Todo List</h1>
      <AddTodo />
      <TodoList />
    </div>
  );
}

export default App;
```

1.2. Giới thiệu về Zustand và các concept cần nắm

> Zustand là một thư viện quản lý state nhỏ gọn, nhanh chóng và dễ sử dụng cho React. Nó cung cấp một cách tiếp cận đơn giản để quản lý state toàn cục mà không cần nhiều boilerplate code như Redux.

Các khái niệm chính trong Zustand:

1. **Store**: Nơi lưu trữ state và các phương thức để thay đổi state.
2. **Actions**: Các hàm được định nghĩa trong store để thay đổi state.
3. **Selectors**: Các hàm để truy xuất state từ store.

Ưu điểm của Zustand:

1. **Đơn giản**: Ít boilerplate code, dễ học và sử dụng.
2. **Linh hoạt**: Có thể sử dụng bên trong và bên ngoài components React.
3. **Hiệu suất cao**: Chỉ re-render khi cần thiết.
4. **TypeScript support**: Hỗ trợ tốt cho TypeScript.

Hướng dẫn thực hành: Tạo ứng dụng Todo List với Zustand

Bước 1: Cài đặt project và thư viện

```
npx create-react-app zustand-todo
```

```
cd zustand-todo
```

```
npm install zustand
```

Bước 2: Tạo Zustand Store

- Tạo file src/store.js:

```
import create from "zustand";

const useStore = create((set) => ({
  todos: [],
  addTodo: (text) =>
    set((state) => ({
      todos: [...state.todos, { id: Date.now(), text, completed: false }],
    })),
  toggleTodo: (id) =>
    set((state) => ({
      todos: state.todos.map((todo) =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      ),
```

```
    })),
  removeTodo: (id) =>
    set((state) => ({
      todos: state.todos.filter((todo) => todo.id !== id),
    })),
}));

export default useStore;
```

Bước 3: Tạo component AddTodo

- Tạo file src/components/AddTodo.js:

```
import React, { useState } from "react";
import useStore from "../store";

function AddTodo() {
  const [text, setText] = useState("");
  const addTodo = useStore((state) => state.addTodo);

  const handleSubmit = (e) => {
    e.preventDefault();
    if (text.trim()) {
      addTodo(text);
      setText("");
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Add a new todo"
      />
      <button type="submit">Add</button>
    </form>
  );
}

export default AddTodo;
```

Bước 4: Tạo component TodoList

- Tạo file src/components/TodoList.js:

```
import React from "react";
import useStore from "../store";
```

```
function TodoList() {
  const todos = useStore((state) => state.todos);
  const toggleTodo = useStore((state) => state.toggleTodo);
  const removeTodo = useStore((state) => state.removeTodo);

  return (
    <ul>
      {todos.map((todo) => (
        <li key={todo.id}>
          <span
            style={{ textDecoration: todo.completed ? "line-through" :
"none" }}
            onClick={() => toggleTodo(todo.id)}
          >
            {todo.text}
          </span>
          <button onClick={() => removeTodo(todo.id)}>Delete</button>
        </li>
      ))}
    </ul>
  );
}

export default TodoList;
```

Bước 5: Cập nhật App component

- Sửa file src/App.js:

```
import React from "react";
import AddTodo from "./components/AddTodo";
import TodoList from "./components/TodoList";

function App() {
  return (
    <div>
      <h1>Todo List with Zustand</h1>
      <AddTodo />
      <TodoList />
    </div>
  );
}

export default App;
```

Bước 6: Chạy ứng dụng

```
npm start
```

Giải thích chi tiết

```
Tạo Store:

Sử dụng create từ Zustand để tạo store.

Định nghĩa state ban đầu (todos) và các actions (addTodo, toggleTodo,
removeTodo).
Mỗi action sử dụng set để cập nhật state.


Sử dụng Store trong Components:

Import useStore và sử dụng nó để truy cập state và actions.
Ví dụ: const addTodo = useStore(state => state.addTodo);
```

Zustand cập nhật UI ntn:

Zustand tự động cập nhật UI khi state thay đổi. Không cần wrap ứng dụng trong Provider như Redux.

1.3 So sánh useContext với Redux và Các Global State Khác

1. Giới thiệu

Trong phát triển ứng dụng React, việc quản lý state là một trong những thách thức lớn. Hai phương pháp phổ biến là sử dụng `useContext` hook của React và thư viện quản lý state như Redux. Trong bài viết này, chúng ta sẽ tìm hiểu về nhược điểm của `useContext` so với Redux và các giải pháp global state khác.

2. Nhược điểm của useContext

2.1. Hiệu suất

`useContext` có thể gây ra re-render không cần thiết trong các component con khi state thay đổi, ngay cả khi component đó không sử dụng phần state đã thay đổi.

2.2. Khó khăn trong việc tổ chức code

Với các ứng dụng lớn, việc sử dụng nhiều context có thể dẫn đến "provider hell", làm cho code trở nên khó đọc và bảo trì.

2.3. Không có công cụ debug tích hợp

Không như Redux DevTools, `useContext` không có công cụ debug mạnh mẽ được tích hợp sẵn.

2.4. Khó khăn trong việc xử lý logic phức tạp

`useContext` không cung cấp cơ chế built-in để xử lý các side effects hoặc logic phức tạp như Redux middleware.

2. Giới thiệu axios để call API dễ dàng hơn

1. Giới thiệu

> Trong phát triển ứng dụng web hiện đại, việc gọi API và xử lý authentication là một phần quan trọng. Bài viết này sẽ hướng dẫn bạn cách sử dụng Axios interceptors để tự động thêm token vào mỗi request và xử lý refresh token khi cần thiết.

2. Cài đặt

Đầu tiên, hãy cài đặt Axios trong project của bạn:

```
npm install axios
```

3. Thiết lập Axios Instance

- Tạo một file mới, ví dụ api.js, để cấu hình Axios:

```
import axios from "axios";

const api = axios.create({
  baseURL: "https://api.example.com", // URL của API của bạn
  timeout: 5000, // Thời gian timeout cho mỗi request
  headers: {
    "Content-Type": "application/json",
  },
});

export default api;
```

4. Tạo Interceptor để Thêm Token

- Thêm đoạn code sau vào file api.js:

```
// Hàm để lấy access token từ local storage
const getAccessToken = () => {
  return localStorage.getItem("access_token");
};

// Request interceptor để thêm token vào header
api.interceptors.request.use(
  (config) => {
    const token = getAccessToken();
    if (token) {
      config.headers["Authorization"] = `Bearer ${token}`;
    }
    return config;
```

```
  },
  (error) => {
    return Promise.reject(error);
  }
);
```

6. Thêm Response Interceptor để trích xuất data

- Thêm đoạn code sau vào file `api.js`:

```
// Response interceptor để tự động trích xuất data
api.interceptors.response.use(
  (response) => {
    // Trả về trực tiếp data từ response
    return response.data;
  },
  (error) => {
    // Nếu response lỗi có chứa data, trả về data đó
    if (error.response && error.response.data) {
      return Promise.reject(error.response.data);
    }
    return Promise.reject(error);
  }
);
```

5. Kết luận

> Sử dụng Axios interceptors là một cách hiệu quả để quản lý authentication trong ứng dụng của bạn. Nó giúp tự động hóa quá trình thêm token vào requests và xử lý refresh token, giúp code của bạn sạch sẽ và dễ bảo trì hơn. Hãy nhớ rằng, bảo mật là rất quan trọng khi làm việc với tokens. Luôn đảm bảo bạn đang sử dụng các phương pháp bảo mật tốt nhất cho ứng dụng của mình.

3. Giới thiệu react-hook-form

1. Giới thiệu

> React Hook Form là một thư viện mạnh mẽ để quản lý form trong React. Nó cung cấp một cách tiếp cận dựa trên hook, giúp tạo và quản lý form một cách hiệu quả với ít code hơn và hiệu suất tốt hơn.

Ưu điểm chính:

1. Ít re-render: Sử dụng uncontrolled components.
2. Hiệu suất cao: Tối ưu hóa việc render và validation.
3. Dễ sử dụng: API đơn giản và trực quan.
4. Tích hợp tốt: Hoạt động tốt với các thư viện UI khác.

2. Cài đặt

- Bắt đầu bằng cách cài đặt React Hook Form trong project của bạn:

```
npm install react-hook-form
```

3. Ví dụ Cơ bản

- Hãy bắt đầu với một form đăng ký đơn giản.

Bước 1: Tạo component form

- Tạo một file mới, ví dụ RegistrationForm.js:

```
import React from "react";
import { useForm } from "react-hook-form";

function RegistrationForm() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm();

  const onSubmit = (data) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <div>
        <label htmlFor="username">Username</label>
        <input
          id="username"
          {...register("username", { required: "Username is required" })}
        />
        {errors.username && <p>{errors.username.message}</p>}
      </div>

      <div>
        <label htmlFor="email">Email</label>
        <input
          id="email"
          {...register("email", {
            required: "Email is required",
            pattern: {
              value: /^\S+@\S+$/i,
              message: "Invalid email format",
            },
          })}
        />
        {errors.email && <p>{errors.email.message}</p>}
      </div>
```

```
        <div>
          <label htmlFor="password">Password</label>
          <input
            id="password"
            type="password"
            {...register("password", {
              required: "Password is required",
              minLength: {
                value: 6,
                message: "Password must be at least 6 characters",
              },
            })}
          />
          {errors.password && <p>{errors.password.message}</p>}
        </div>

        <button type="submit">Register</button>
      </form>
    );
}

export default RegistrationForm;
```

Bước 2: Sử dụng component trong ứng dụng

- Trong file App.js hoặc component chính của bạn:

```
import React from "react";
import RegistrationForm from "./RegistrationForm";

function App() {
  return (
    <div className="App">
      <h1>Registration Form</h1>
      <RegistrationForm />
    </div>
  );
}

export default App;
```

4. Giải thích Chi tiết

4.1 Hook useForm

```
const {
  register,
  handleSubmit,
```

```
  formState: { errors },
} = useForm();
```

- register: Dùng để đăng ký các trường input với form.
- handleSubmit: Xử lý việc submit form.
- errors: Chứa các lỗi validation.

4.2 Đăng ký Input

```
<input {...register("username", { required: "Username is required" })} />
```

- register được sử dụng để đăng ký input với form.
- Các rules validation được định nghĩa trong object thứ hai.

4.3 Xử lý Submit

```
const onSubmit = data => console.log(data);

// Trong JSX
<form onSubmit={handleSubmit(onSubmit)}>
```

- handleSubmit bọc hàm onSubmit, đảm bảo rằng form chỉ được submit khi tất cả validation pass.

4.4 Hiển thị Lỗi

```
{
  errors.username && <p>{errors.username.message}</p>;
}
```

- Hiển thị thông báo lỗi nếu trường username có lỗi.

5. Ví dụ Nâng cao

5.1 Validation Tùy chỉnh

```
const {
  register,
  handleSubmit,
  formState: { errors },
  watch,
} = useForm();
```

```
// Trong JSX
<input
  {...register("confirmPassword", {
    validate: (value) =>
      value === watch("password") || "Passwords do not match",
  })}
/>;
```

5.2 Sử dụng Yup cho Schema Validation

- Đầu tiên, cài đặt Yup và hook form resolver:

```
npm install @hookform/resolvers yup
```

- Sau đó, sử dụng Yup trong form:

```
import { useForm } from "react-hook-form";
import { yupResolver } from "@hookform/resolvers/yup";
import * as yup from "yup";

const schema = yup
  .object({
    username: yup.string().required("Username is required"),
    email: yup.string().email("Invalid email").required("Email is
required"),
    password: yup
      .string()
      .min(6, "Password must be at least 6 characters")
      .required("Password is required"),
  })
  .required();

function RegistrationForm() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm({
    resolver: yupResolver(schema),
  });

  // ... rest of the component
}
```

5.3 Điều khiển Form Động

```
import React from "react";
import { useForm, useFieldArray } from "react-hook-form";

function DynamicForm() {
  const { register, control, handleSubmit } = useForm();
  const { fields, append, remove } = useFieldArray({
    control,
    name: "items",
  });

  const onSubmit = (data) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      {fields.map((field, index) => (
        <div key={field.id}>
          <input {...register(`items.${index}.name`)} />
          <button type="button" onClick={() => remove(index)}>
            Delete
          </button>
        </div>
      ))}
      <button type="button" onClick={() => append({ name: "" })}>
        Add Item
      </button>
      <button type="submit">Submit</button>
    </form>
  );
}
```

6. Tích hợp với UI Libraries

- React Hook Form dễ dàng tích hợp với các thư viện UI phổ biến như Material-UI hoặc Ant Design. Ví dụ với Material-UI:

```
import React from "react";
import { useForm, Controller } from "react-hook-form";
import { TextField, Button } from "@material-ui/core";

function MaterialUIForm() {
  const { control, handleSubmit } = useForm();

  const onSubmit = (data) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <Controller
        name="firstName"
        control={control}
        defaultValue=""
```

```
          rules={{ required: "First name is required" }}
          render={({ field, fieldState: { error } }) => (
            <TextField
              {...field}
              label="First Name"
              error={!!error}
              helperText={error?.message}
            />
          )}
        />
        <Button type="submit" variant="contained" color="primary">
          Submit
        </Button>
      </form>
    );
  }
```

7. Kết luận

> React Hook Form cung cấp một cách tiếp cận mạnh mẽ và linh hoạt để xử lý form trong React. Với API đơn giản và hiệu suất tốt, nó là một lựa chọn tuyệt vời cho cả ứng dụng nhỏ và lớn. Bằng cách sử dụng React Hook Form, bạn có thể:

- Giảm số lượng code cần viết
- Tăng hiệu suất của ứng dụng
- Dễ dàng xử lý các trường hợp validation phức tạp
- Tích hợp tốt với các thư viện UI khác

4. Giới thiệu React Router Dom v6

1. Giới thiệu

> React Router Dom là thư viện routing phổ biến nhất cho React applications. Phiên bản 6 mang đến nhiều cải tiến và đơn giản hóa API, giúp việc xây dựng ứng dụng single-page (SPA) trở nên dễ dàng hơn.

2. Cài đặt

- Bắt đầu bằng việc cài đặt React Router Dom v6 trong project của bạn:

```
npm install react-router-dom
```

3. Cấu Hình Cơ Bản

Bước 1: Tạo các components

- Tạo một số components cơ bản để sử dụng trong ví dụ:

```
// src/components/Home.js
function Home() {
  return <h1>Trang Chủ</h1>;
}

// src/components/About.js
function About() {
  return <h1>Về Chúng Tôi</h1>;
}

// src/components/Contact.js
function Contact() {
  return <h1>Liên Hệ</h1>;
}
```

Bước 2: Cấu hình Router

- Cập nhật file src/App.js:

```
import React from "react";
import { BrowserRouter as Router, Routes, Route, Link } from "react-
router-dom";
import Home from "./components/Home";
import About from "./components/About";
import Contact from "./components/Contact";

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Trang Chủ</Link>
            </li>
            <li>
              <Link to="/about">Về Chúng Tôi</Link>
            </li>
            <li>
              <Link to="/contact">Liên Hệ</Link>
            </li>
          </ul>
        </nav>

        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/contact" element={<Contact />} />
        </Routes>
      </div>
    </Router>
```

```
  );
}

export default App;
```

4. Các Tính Năng Chính của React Router Dom v6

4.1 Nested Routes

- React Router v6 hỗ trợ nested routes một cách tự nhiên. Ví dụ:

```
function Dashboard() {
  return (
    <div>
      <h1>Dashboard</h1>
      <nav>
        <Link to="main">Main Dashboard</Link>
        <Link to="settings">Settings</Link>
      </nav>
      <Outlet />
    </div>
  );
}

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="dashboard" element={<Dashboard />}>
          <Route path="main" element={<MainDashboard />} />
          <Route path="settings" element={<Settings />} />
        </Route>
      </Routes>
    </Router>
  );
}
```

4.2 useNavigate Hook

- useNavigate thay thế cho useHistory trong các phiên bản trước:

```
import { useNavigate } from "react-router-dom";

function LoginButton() {
  let navigate = useNavigate();

  function handleClick() {
    navigate("/dashboard");
```

```
  }

  return <button onClick={handleClick}>Login</button>;
}
```

4.3 No More Switch, Welcome Routes

- Routes thay thế cho Switch trong v6:

```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="about" element={<About />} />
  <Route path="*" element={<NotFound />} />
</Routes>
```

4.4 useParams Hook

- useParams vẫn được giữ lại từ phiên bản trước, nhưng cách sử dụng đã được đơn giản hóa:

```
import { useParams } from "react-router-dom";

function User() {
  let { id } = useParams();
  return <h1>User ID: {id}</h1>;
}

// Trong component cha
<Route path="user/:id" element={<User />} />;
```

4.5 Link Component

- Link component vẫn được sử dụng để tạo liên kết:

```
import { Link } from "react-router-dom";

function Nav() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>
  );
}
```

5. Ví Dụ Nâng Cao: Protected Routes

- Tạo một Protected Route để kiểm soát truy cập:

```
import { Navigate, Outlet } from "react-router-dom";

const ProtectedRoute = ({ isAllowed, redirectPath = "/login", children })
=> {
  if (!isAllowed) {
    return <Navigate to={redirectPath} replace />;
  }

  return children ? children : <Outlet />;
};

function App() {
  const isLoggedIn = checkIfUserIsLoggedIn(); // Implement this function

  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/login" element={<Login />} />
        <Route element={<ProtectedRoute isAllowed={isLoggedIn} />}>
          <Route path="/dashboard" element={<Dashboard />} />
          <Route path="/profile" element={<Profile />} />
        </Route>
      </Routes>
    </Router>
  );
}
```

6. Lazy Loading với React Router

- Sử dụng React.lazy và Suspense để lazy load các components:

```
import React, { Suspense } from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";

const Home = React.lazy(() => import("./components/Home"));
const About = React.lazy(() => import("./components/About"));

function App() {
  return (
    <Router>
      <Suspense fallback={<div>Loading...</div>}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
        </Routes>
      </Suspense>
    </Router>
```

```
    );
}
```

7. Kết Luận

> React Router Dom v6 mang đến nhiều cải tiến đáng kể, giúp việc xây dựng và quản lý routing trong ứng dụng React trở nên đơn giản và hiệu quả hơn. Với API được đơn giản hóa và các tính năng mới, nó là một công cụ mạnh mẽ cho việc phát triển single-page applications. Các điểm chính cần nhớ:

- Sử dụng `Routes` thay cho `Switch`.
- `Nested routes` được hỗ trợ tự nhiên.
- `useNavigate` thay thế cho `useHistory.`
- `Protected routes` và `lazy loading` dễ dàng triển khai.

5. Giới thiệu react-query

1. Giới Thiệu

> React Query là một thư viện mạnh mẽ để quản lý, lưu trữ và đồng bộ hóa dữ liệu server trong ứng dụng React. Nó giúp đơn giản hóa quá trình fetching, caching, và cập nhật dữ liệu từ server mà không cần quản lý state phức tạp.

Mục Đích Sử Dụng:

1. **Fetching Data**: Dễ dàng lấy dữ liệu từ API.
2. **Caching**: Tự động cache dữ liệu và quản lý việc làm mới cache.
3. **Synchronization**: Đồng bộ hóa dữ liệu giữa client và server.
4. **Updating Server State**: Quản lý mutations (cập nhật, thêm, xóa dữ liệu).
5. **Performance Optimization**: Tối ưu hiệu suất bằng cách giảm số lượng requests không cần thiết.

2. Cài Đặt

Bắt đầu bằng việc cài đặt React Query trong project của bạn:

```
npm install react-query
```

3. Cấu Hình Cơ Bản

- Bước 1: Cấu hình QueryClient

Trong file src/index.js hoặc src/App.js:

```
import React from "react";
import ReactDOM from "react-dom";
import { QueryClient, QueryClientProvider } from "react-query";
import App from "./App";
```

```
const queryClient = new QueryClient();

ReactDOM.render(
  <React.StrictMode>
    <QueryClientProvider client={queryClient}>
      <App />
    </QueryClientProvider>
  </React.StrictMode>,
  document.getElementById("root")
);
```

4. Sử Dụng React Query

4.1 Fetching Data với useQuery

- Tạo một component để fetch và hiển thị danh sách bài viết:

```
import React from "react";
import { useQuery } from "react-query";
import axios from "axios";

const fetchPosts = async () => {
  const { data } = await axios.get(
    "https://jsonplaceholder.typicode.com/posts"
  );
  return data;
};

function Posts() {
  const { data, isLoading, error } = useQuery("posts", fetchPosts);

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>An error occurred: {error.message}</div>;

  return (
    <ul>
      {data.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default Posts;
```

4.2 Mutations với useMutation

- Tạo một component để thêm bài viết mới:

```
import React from "react";
import { useMutation, useQueryClient } from "react-query";
import axios from "axios";

const addPost = async (newPost) => {
  const { data } = await axios.post(
    "https://jsonplaceholder.typicode.com/posts",
    newPost
  );
  return data;
};

function AddPost() {
  const queryClient = useQueryClient();
  const mutation = useMutation(addPost, {
    onSuccess: () => {
      queryClient.invalidateQueries("posts");
    },
  });

  const handleSubmit = (event) => {
    event.preventDefault();
    const title = event.target.title.value;
    const body = event.target.body.value;
    mutation.mutate({ title, body });
  };

  return (
    <form onSubmit={handleSubmit}>
      <input name="title" placeholder="Title" />
      <textarea name="body" placeholder="Body" />
      <button type="submit">Add Post</button>
    </form>
  );
}

export default AddPost;
```

5. Tính Năng Nâng Cao

5.1 Parallel Queries

- Fetch nhiều resource cùng một lúc:

```
import { useQueries } from "react-query";

function ParallelQueries() {
  const results = useQueries([
    { queryKey: ["users"], queryFn: fetchUsers },
    { queryKey: ["posts"], queryFn: fetchPosts },
```

```
  ]);

  const isLoading = results.some((result) => result.isLoading);
  const isError = results.some((result) => result.isError);

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>An error occurred</div>;

  const [usersData, postsData] = results.map((result) => result.data);

  return (
    <div>
      <h2>Users</h2>
      <ul>
        {usersData.map((user) => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
      <h2>Posts</h2>
      <ul>
        {postsData.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

5.2 Dependent Queries

- Thực hiện query phụ thuộc vào kết quả của query khác:

```
function UserPosts({ userId }) {
  const { data: user } = useQuery(["user", userId], () =>
fetchUser(userId));
  const { data: posts } = useQuery(
    ["posts", userId],
    () => fetchUserPosts(userId),
    { enabled: !!user }
  );

  // Render user and their posts
}
```

5.3 Pagination

- Xử lý phân trang:

```
function PaginatedPosts() {
  const [page, setPage] = React.useState(1);
  const { data, isLoading } = useQuery(
    ["posts", page],
    () => fetchPosts(page),
    { keepPreviousData: true }
  );

  if (isLoading) return <div>Loading...</div>;

  return (
    <div>
      {data.map((post) => (
        <p key={post.id}>{post.title}</p>
      ))}
      <button onClick={() => setPage((old) => Math.max(old - 1, 1))}>
        Previous Page
      </button>
      <button onClick={() => setPage((old) => old + 1)}>Next Page</button>
    </div>
  );
}
```

6. Tối Ưu Hóa Hiệu Suất

6.1 Caching và Stale Time

- Cấu hình thời gian cache:

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 5 * 60 * 1000, // 5 minutes
      cacheTime: 10 * 60 * 1000, // 10 minutes
    },
  },
});
```

6.2 Prefetching

- Prefetch dữ liệu trước khi cần:

```
function App() {
  const queryClient = useQueryClient();

  React.useEffect(() => {
    queryClient.prefetchQuery("posts", fetchPosts);
  }, []);
```

```
  // ...
}
```

7. Xử Lý Lỗi

- Xử lý lỗi một cách tinh tế:

```
function Posts() {
  const { data, isLoading, error, refetch } = useQuery("posts",
fetchPosts, {
    retry: 3,
    onError: (error) => {
      console.log("An error occurred:", error.message);
    },
  });

  if (isLoading) return <div>Loading...</div>;
  if (error)
    return (
      <div>
        An error occurred: {error.message}
        <button onClick={() => refetch()}>Try Again</button>
      </div>
    );

  // Render posts
}
```

8. Kết Luận

> React Query cung cấp một giải pháp mạnh mẽ và linh hoạt để quản lý state server trong ứng dụng React. Với các tính năng như caching thông minh, quản lý mutations, và tối ưu hóa hiệu suất, React Query giúp đơn giản hóa quá trình xây dựng ứng dụng với trải nghiệm người dùng mượt mà. Các lợi ích chính:

- Giảm boilerplate code
- Tự động quản lý cache
- Tối ưu hóa hiệu suất
- Dễ dàng xử lý các tình huống phức tạp như pagination, dependent queries