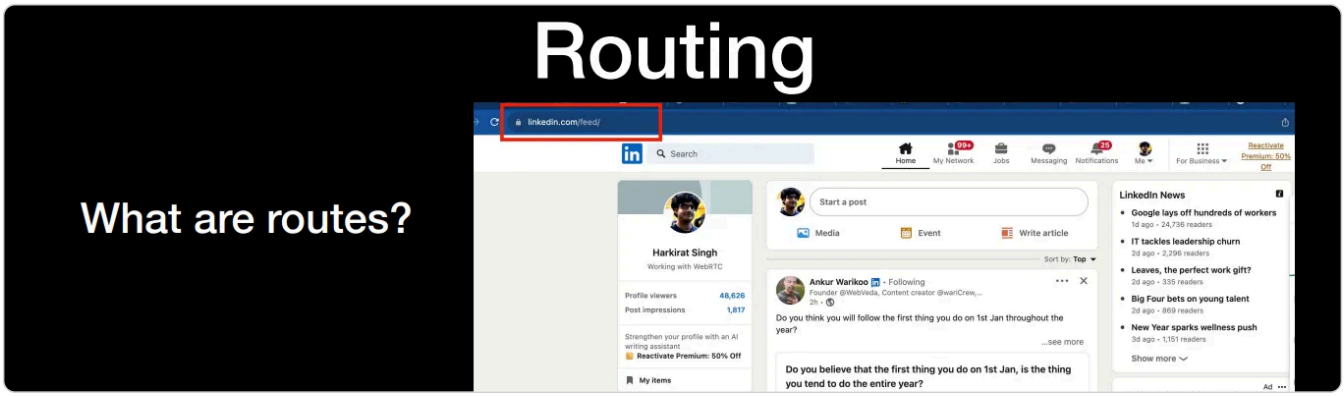
Show Contents

Filter

Week 7.1

Context API & Prop Drilling

In this lecture, Harkirat covers key concepts in React development, specifically focusing on routing, prop drilling, and the Context API. `Routing` is vital for managing navigation in React applications, while `prop drilling` and the `Context API` address challenges related to passing data between components. These insights provide essential knowledge for building well-organized and effective React projects.



React Routing

Routing in React is a mechanism that allows you to manage navigation and control the content displayed in your application based on the URL. It's essential for several reasons:

1. **`Multi-Page Applications (MPAs)`** : In traditional web development, navigating between pages required a full page reload. React, being a Single Page Application (SPA) library, loads a single HTML page and dynamically updates the content as users navigate. Routing enables SPAs to mimic the behavior of traditional MPAs by updating the view based on the URL.
2. **`User Experience`** : Routing enhances the overall user experience by providing a seamless and dynamic interface. Users can navigate between different views or sections of your application without experiencing the delays associated with full-page reloads.
3. **`Bookmarking and Sharing`** : With routing, each view in your React application can have a unique URL. This allows users

5. `State Preservation` : When users navigate between different views, routing helps preserve the state of the application. React Router, a popular routing library for React, allows you to pass parameters and state between different components based on the route.
6. `Conditional Rendering` : Routing enables conditional rendering of components based on the current URL. Different components or views can be displayed depending on the route, allowing you to create dynamic and context-aware user interfaces.

> To implement routing in a React application, developers often use libraries like React Router. React Router provides a set of components and functions to define routes, handle navigation, and manage the application's history, making it an essential tool for building robust and navigable React applications.

Some Jargons

1. SPA (Single Page Application):

A Single Page Application (SPA) is a type of web application or website that interacts with the user by dynamically rewriting the current page, rather than loading entire new pages from the server.

- Key Characteristics:
  - Loads a single HTML page initially.
  - Subsequent interactions and navigation are handled by dynamically updating the content on the page through JavaScript.
  - Utilizes AJAX or Fetch API to communicate with the server and fetch data without reloading the entire page.
  - Provides a more fluid and seamless user experience by avoiding full-page reloads.

2. Client-side Bundle:

In the context of web development, a client-side bundle refers to a collection of JavaScript files and other assets bundled together to be delivered to the client's web browser.

- Key Components:
  - **JavaScript Files:** The application's logic and functionality are written in JavaScript files. Bundling involves combining these files into a single or multiple bundles.
  - **Stylesheets, Images, and Other Assets:** Along with JavaScript, other assets like stylesheets, images, and fonts may be included in the bundle for efficient delivery to the client.
- Advantages:

3. Client-side Routing:

Client-side routing refers to the process of managing navigation within a Single Page Application (SPA) entirely on the client side, without making additional requests to the server for each new view.

- Key Features:
  - Utilizes the browser's History API to manipulate the URL without triggering full page reloads.
  - Enables dynamic content updates based on the route, improving user experience.
  - Typically implemented using libraries like React Router for React applications or Vue Router for Vue.js applications.
- Advantages:
  - Enhances the performance of SPAs by avoiding the need for server round-trips during navigation.
  - Allows for a smoother and more responsive user interface as content is updated dynamically.
  - Enables bookmarking, sharing, and direct linking to specific views within the SPA.

React Router DOM

In React, routing is commonly achieved using the React Router DOM library, which provides a set of components for handling navigation within a React application. The main components involved in React Router DOM are `BrowserRouter`, `Routes`, and `Route`. Here's an overview of how routing is typically implemented using these components:

1. BrowserRouter:

- The `BrowserRouter` component is a top-level component that should be used to wrap your entire application. It enables the use of routing features throughout your React application.
- It utilizes the HTML5 History API to manipulate the URL without triggering full page reloads.

```
import { BrowserRouter } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      {/* Other components and routing components go here */}
    </BrowserRouter>
  );
}
```

2. Routes:

```
import { Routes } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        {/* Define Route components here */}
      </Routes>
    </BrowserRouter>
  );
}
```

3. Route:

- The `Route` component is responsible for rendering specific components based on the current URL path. It takes two main props: `path` and `element`.
- The `path` prop defines the URL path that should match for the route to be rendered, and the `element` prop specifies the component to render when the path matches.

```
import { Route } from 'react-router-dom';
import Home from './components/Home';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        {/* Add more Route components for other views */}
      </Routes>
    </BrowserRouter>
  );
}
```

In the above example, when the URL path is "/", the `Home` component will be rendered.

> This is a basic setup for using React Router DOM. You can extend this by adding nested routes, handling dynamic route parameters, and incorporating additional features provided by React Router DOM for more advanced routing scenarios.

Using React Router DOM

If you haven't installed React Router DOM, you can do so by running the following command:

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

1. Setting up Routes:
   Create two components for the landing page and the dashboard.

```
// Landing.jsx
import React from 'react';

const Landing = () => {
  return (
    <div>
      <h1>Welcome to the Landing Page</h1>
    </div>
  );
};

export default Landing;
```

```
// Dashboard.jsx
import React from 'react';

const Dashboard = () => {
  return (
    <div>
      <h1>Dashboard</h1>
    </div>
  );
};

export default Dashboard;
```

1. Create the Main App Component:
   Set up your main App component with React Router to handle routing.

```
// App.jsx
import React from 'react';
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
import Landing from './Landing';
import Dashboard from './Dashboard';

const App = () => {
  return (
```

```
            <li>
              <Link to="/dashboard">Dashboard</Link>
            </li>
          </ul>
        </nav>

        <Routes>
          <Route path="/" element={<Landing />} />
          <Route path="/dashboard" element={<Dashboard />} />
        </Routes>
      </BrowserRouter>
    );
  };

  export default App;
```

In this example, we use the `Link` component from React Router to create navigation links. The `Routes` component contains individual `Route` components for each page.

1. Navigate Programmatically:
   If you want to navigate programmatically, you can use `window.location.href` . For example, in a function:

```
const navigateToDashboard = () => {
  window.location.href = '/dashboard';
};
```

However, using `Link` from React Router is a preferred way for declarative navigation within a React app.

1. Shared UI:
   If you want to share UI components between the landing page and the dashboard, you can create a common component and use it in both `Landing` and `Dashboard` components.

```
// SharedComponent.jsx
import React from 'react';

const SharedComponent = () => {
  return (
    <div>
      <p>This component is shared between Landing and Dashboard.</p>
    </div>
  );
};

export default SharedComponent;
```

between different pages.

Issue with `window.location.href`

When using `window.location.href` for navigation in a React application, it triggers a full page reload, which is not desirable in client-side routing. A full page reload involves fetching the HTML, CSS, and other assets again, leading to a slower and less efficient user experience.

To address this issue, React Router DOM provides a solution in the form of the `useNavigate` hook. This hook is designed for programmatic navigation within a React component without triggering a full page reload. By using `useNavigate`, you can ensure smoother transitions between different views in a single-page application (SPA) without unnecessary overhead.

Here's an example of how to use `useNavigate`:

```
// App.jsx
import React from 'react';
import { BrowserRouter as Router, Routes, Route, Link, useNavigate } from 'react-route
import Landing from './Landing';
import Dashboard from './Dashboard';

const App = () => {
  const navigate = useNavigate();

  const navigateToDashboard = () => {
    // Use navigate function instead of window.location.href
    navigate('/dashboard');
  };

  return (
    <Router>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            {/* Use the navigateToDashboard function for navigation */}
            <button onClick={navigateToDashboard}>Go to Dashboard</button>
          </li>
        </ul>
      </nav>

      <Routes>
        <Route path="/" element={<Landing />} />
        <Route path="/dashboard" element={<Dashboard />} />
```

```
export default App;
```

> In this example, the `useNavigate` hook is used to get the `navigate` function, which can be called to navigate to different routes without causing a full page reload. By using this approach, you maintain the benefits of client-side routing in React, ensuring a faster and more seamless user experience.

Note

The `useNavigate` hook in React Router DOM is designed to work within the context of a `BrowserRouter` . It should be used inside a component that is a descendant of `BrowserRouter` to ensure access to the correct router context. This limitation is intentional, as `useNavigate` relies on the router context for scoped navigation, enabling seamless client-side routing without triggering a full page reload. Placing the hook within the correct context ensures its proper functionality for dynamic view and URL updates.

Lazy Loading

Lazy loading in React is a technique used to optimize the performance of a web application by deferring the loading of certain components until they are actually needed. This can significantly reduce the initial bundle size and improve the overall loading time of the application.

In React, lazy loading is typically achieved using the `React.lazy` function along with the `Suspense` component. The `React.lazy` function allows you to load a component lazily, meaning it is only fetched when the component is actually rendered. Here's a simple example:

React Suspense

In React, `Suspense` is a component that enables a better experience for handling asynchronous operations such as code-splitting and lazy loading. It's used in conjunction with `React.lazy` for lazy loading components or with data fetching functions.

When you're using `React.lazy` to load a component lazily, you wrap it with `Suspense` to specify a fallback UI that will be rendered while the component is being loaded. The `fallback` prop of `Suspense` defines what to display during the loading period.

Code Implementation

```
const MyLazyComponent = React.lazy(() => import('./MyComponent'));

function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <MyLazyComponent />
      </Suspense>
    </div>
  );
}
```

In this example, if `MyLazyComponent` is not yet loaded, the `Suspense` component will render the "Loading..." message as the fallback until the component is fully loaded and ready to be displayed.

> This mechanism is particularly useful for improving the user experience when dealing with dynamic loading of components or fetching data asynchronously. The `fallback` UI gives users feedback that something is happening in the background, making the application feel more responsive.

Prop Drilling in React

Prop drilling refers to the process of passing data from a top-level component down to deeper levels through intermediate components. It happens when a piece of state needs to be accessible by a component deep in the component tree, and it gets passed down as a prop through all the intermediate components.

Why Prop Drilling?

1. State Management:
   Prop drilling is often used to manage state in a React application. By passing state down through the component tree, you can share data between components without resorting to more advanced state management solutions like context or state management libraries.
2. Simplicity:
   Prop drilling keeps the application structure simple and makes it easier to understand the flow of data. It's a straightforward way of handling data without introducing more complex tools.

Code Implementation

```
// Top-level component
function App() {
```

```
function ChildComponent({ data }) {
  return <GrandchildComponent data={data} />;
}

// Deepest component
function GrandchildComponent({ data }) {
  return <p>{data}</p>;
}
```

In this example, `App` has a piece of data that needs to be accessed by `GrandchildComponent`. Instead of using more advanced state management tools, we pass the `data` as a prop through `ChildComponent`. This is prop drilling in action.

Drawbacks

1. **Readability:**
   Prop drilling can make the code less readable, especially when you have many levels of components. It might be hard to trace where a particular prop is coming from.
2. **Maintenance:**
   If the structure of the component tree changes, and the prop needs to be passed through additional components, it requires modifications in multiple places.

> While prop drilling is a simple and effective way to manage state in some cases, for larger applications or more complex state management, consider using tools like React Context or state management libraries. These can help avoid the drawbacks of prop drilling while providing a cleaner solution for state sharing.

Context API in React

Context API is a feature in React that provides a way to share values like props between components without explicitly passing them through each level of the component tree. It helps solve the prop drilling problem by allowing data to be accessed by components at any level without the need to pass it through intermediate components.

Key Components of Context API:

1. **`createContext`:**
   The `createContext` function is used to create a context. It returns an object with two components - `Provider` and `Consumer`.

```
const MyContext = React.createContext();
```

component tree.

```
<MyContext.Provider value={/* some value */}>
  {/* Components that can access the context value */}
</MyContext.Provider>
```

1. **`Consumer`** (or **`useContext`** hook):
   The `Consumer` component allows components to consume the context value. Alternatively, the `useContext` hook can be used for a more concise syntax.

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

or

```
const value = useContext(MyContext);
```

Code Implementation

```
// Create a context
const UserContext = React.createContext();

// Top-level component with a Provider
function App() {
  const user = { username: "john_doe", role: "user" };

  return (
    <UserContext.Provider value={user}>
      <Profile />
    </UserContext.Provider>
  );
}

// Intermediate component
function Profile() {
  return <Navbar />;
}

// Deepest component consuming the context value
function Navbar() {
  const user = useContext(UserContext);
```

In this example, the `UserContext.Provider` in the `App` component provides the `user` object to all its descendants. The `Navbar` component, which is deeply nested, consumes the `user` context value without the need for prop drilling.

Advantages of Context API:

1. **Avoids Prop Drilling:**
   Context API eliminates the need for passing props through intermediate components, making the code cleaner and more maintainable.
2. **Global State:**
   It allows you to manage global state that can be accessed by components across the application.

> While Context API is a powerful tool, it's essential to use it judiciously and consider factors like the size and complexity of the application. For complex state management needs, additional tools like Redux might be more suitable.

Other Solutions

Recoil, Redux, and Context API are all solutions for managing state in React applications, each offering different features and trade-offs.

1. Context API

- **Role:** Context API is a feature provided by React that allows components to share state without prop drilling. It creates a context and a provider to wrap components that need access to that context.
- **Usage:**

```
// Context creation
import { createContext, useContext } from 'react';

const UserContext = createContext();

// Context provider
function UserProvider({ children }) {
  const user = { name: 'John' };

  return <UserContext.Provider value={user}>{children}</UserContext.Provider>;
}

// Accessing context in a component
```

- **Advantages:** Simplicity, built-in React feature.

2. Recoil

- **Role:** Recoil is a state management library developed by Facebook for React applications. It introduces the concept of atoms and selectors to manage state globally. It can be considered a more advanced and feature-rich alternative to Context API.
- **Usage:**

```
// Atom creation
import { atom, useRecoilState } from 'recoil';

export const userState = atom({
  key: 'userState',
  default: { name: 'John' },
});

// Accessing Recoil state in a component
function Profile() {
  const [user, setUser] = useRecoilState(userState);

  return (
    <div>
      <p>Welcome, {user.name}</p>
      <button onClick={() => setUser({ name: 'Jane' })}>Change Name</button>
    </div>
  );
}
```

- **Advantages:** Advanced features like selectors, better performance optimizations.

3. Redux:

- **Role:** Redux is a powerful state management library often used with React. It introduces a global store and follows a unidirectional data flow. While Redux provides more features than Context API, it comes with additional concepts and boilerplate.
- **Usage:**

```
// Store creation
import { createStore } from 'redux';

const initialState = { user: { name: 'John' } };
```

```
  }
};

const store = createStore(rootReducer);

// Accessing Redux state in a component
function Profile() {
  const user = useSelector((state) => state.user);
  const dispatch = useDispatch();

  return (
    <div>
      <p>Welcome, {user.name}</p>
      <button onClick={() => dispatch({ type: 'CHANGE_NAME' })}>Change Name</button>
    </div>
  );
}
```

- **Advantages:** Middleware support, time-travel debugging, broader ecosystem.

Considerations:

- **Complexity:** Context API is simple and built into React, making it a good choice for simpler state management. Recoil provides more features and optimizations, while Redux is powerful but comes with additional complexity.
- **Scalability:** Recoil and Redux are often preferred for larger applications due to their ability to manage complex state logic.
- **Community Support:** Redux has a large and established community with a wide range of middleware and tools. Recoil is newer but gaining popularity, while Context API is part of the React core.

Choosing Between Them:

- **Use Context API for Simplicity:** For simpler state management needs, especially in smaller applications or when simplicity is a priority.
- **Consider Recoil for Advanced Features:** When advanced state management features, like selectors and performance optimizations, are needed.
- **Opt for Redux for Scalability:** In larger applications where scalability, middleware, and a broader ecosystem are important factors.

6 Comments

Most upvotes All comments

Add a comment...

Comment

A **Ankur Masih** 7 months ago

is there a way to download the notes?

6 0 1 Reply

A **ADNAN KHAN** a year ago

well written 👍

2 0 0 Replies

S **Syed Ahmedullah Jaser** 9 months ago

Very insightful notes!

0 0 0 Replies

S **Shreyansh Dubey** 7 months ago

How can i download these notes ?

0 0 1 Reply

L **Lokesh Negi** 9 months ago

so you mean we can't use recoil for scalable projects

0 0 0 Replies