React Forms, Data Fetching and Routing Review

Working with Forms in React

- **Controlled Inputs**: This is when you store the input field value in state and update it through `onChange` events. This gives you complete control over the form data and allows instant validation and conditional rendering.

```
import { useState } from "react";

function App() {
  const [name, setName] = useState("");

  const handleChange = (e) => {
    setName(e.target.value);
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(name);
  };

  return (
    <>
      <form onSubmit={handleSubmit}>
        <label htmlFor="name">Your name</label> <br />
        <input value={name} id="name" onChange={handleChange} type="text" />
        <button type="submit">Submit</button>
      </form>
    </>
  );
}

export default App;
```

- **Uncontrolled Inputs**: Instead of handling the inputs through the `useState` hook, uncontrolled inputs in HTML maintain their own internal state with the help of the DOM. Since the DOM controls the input values, you need to pull in the values of the input fields with a `ref`.

```
import { useRef } from "react";

function App() {
```

```
  const nameRef = useRef();

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(nameRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="name">Your</label>{" "}
      <input type="text" ref={nameRef} id="name" />
      <button type="submit">Submit</button>
    </form>
  );
}

export default App;
```

Working with the `useActionState` Hook

- **Server Actions**: These are functions that run on the server to allow form handling right on the server without the need for API endpoints. Here is an example from a Next.js application:

```
"use server";

async function submitForm(formData) {
  const name = formData.get("name");
  return { message: `Hello, ${name}!` };
}
```

The `"user server"` directive marks the function as a server action.

- **`useActionState` Hook**: This hook updates state based on the outcome of a form submission. Here's the basic syntax of the `useActionState` hook:

```
const [state, action, isPending] = useActionState(actionFunction, initialState, permalink);
```

- `state` is the current state the action returns.
- `action` is the function that triggers the server action.
- `isPending` is a boolean that indicates whether the action is currently running or not.
- `actionFunction` parameter is the server action itself.
- `initialState` is the parameter that represents the starting point for the state before the action runs.
- `permalink` is an optional string that contains the unique page URL the form modifies.

Data Fetching in React

- **Options For Fetching Data**: There are many different ways to fetch data in React. You can use the native Fetch API, or third party tools like Axios or SWR.
- **Commonly Used State Variables When Fetching Data**: Regardless of which way you choose to fetch your data in React, there are some pieces of state you will need to keep track of. The first is the data itself. The second will track whether the data is still being fetched. The third is a state variable that will capture any errors that might occur during the data fetching process.

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);
```

Since data fetching is a side effect, it's best to use the `Fetch API` inside of a `useEffect` hook.

```
useEffect(() => {
  const fetchData = async () => {
    try {
      const res = await fetch("https://jsonplaceholder.typicode.com/posts");

      if (!res.ok) {
        throw new Error("Network response was not ok");
      }

      const data = await res.json();
      setData(data);
    } catch (err) {
      setError(err);
    } finally {
      setLoading(false);
    }
  };

  fetchData();
}, []);
```

Then you can render a loading message if the data fetching is not complete, an error message if there was an error fetching the data, or the results.

```
if (loading) {
  return <p>Loading...</p>;
}

if (error) {
```

```
  return <p>{error.message}</p>;
}

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

If you want to use Axios, you need to install and import it:

```
npm i axios
```

```
import axios from "axios";
```

Then you can fetch the data using `axios.get`:

```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);

useEffect(() => {
  const fetchData = async () => {
    try {
      const res = await axios.get(
        "https://jsonplaceholder.typicode.com/users"
      );
      setData(res.data);
    } catch (err) {
      setError(err);
    } finally {
      setLoading(false);
    }
  };

  fetchData();
}, []);
```

To fetch data using the `useSWR` hook, you need to first install and import it.

```
npm i swr
```

```
import useSWR from "swr";
```

Here is how you can use the hook to fetch data:

```
import useSWR from "swr";

const fetcher = (url) => fetch(url).then((res) => res.json());

const FetchTodos = () => {
 const { data, error } = useSWR(
   "https://jsonplaceholder.typicode.com/todos",
   fetcher
 );

 if (!data) {
   return <h2>Loading...</h2>;
 }
 if (error) {
   return <h2>Error: {error.message}</h2>;
 }

 return (
   <>
     <h2>Todos</h2>
     <div>
       {data.map((todo) => (
         <h3 key={todo.id}>{todo.title}</h3>
       ))}
     </div>
   </>
 );
};

export default FetchTodos;
```

Working with the `useOptimistic` Hook

- **`useOptimistic` Hook**: This hook is used to keep UIs responsive while waiting for an async action to complete in the background. It helps manage "optimistic updates" in the UI, a strategy in which you provide immediate updates to the UI based on the expected outcome of an action, like waiting for a server response.

Here is the basic syntax:

```
const [optimisticState, addOptimistic] = useOptimistic(actualState, updateFunction);
```

- `optimisticState` is the temporary state that updates right away for a better user experience.
- `addOptimistic` is the function that applies the optimistic update before the actual state changes.
- `actualState` is the real state value that comes from the result of an action, like fetching data from a server.
- `updateFunction` is the function that determines how the optimistic state should update when called.

Here is an example of using the `useOptimistic` hook in a `TaskList` component:

```
"use client";

import { useOptimistic } from "react";

export default function TaskList({ tasks, addTask }) {
  const [optimisticTasks, addOptimisticTask] = useOptimistic(
    tasks,
    (state, newTask) => [...state, { text: newTask, pending: true }]
  );

  async function handleSubmit(e) {
    e.preventDefault();
    const formData = new FormData(e.target);

    addOptimisticTask(formData.get("task"));

    addTask(formData);
    e.target.reset();
  }

  return <>{/* UI */}</>;
}
```

- **`startTransition`**: This is used to render part of the UI and mark a state update as a non-urgent transition. This allows the UI to be responsive during expensive updates. Here is the basic syntax:

```
startTransition(action);
```

The `action` performs a state update or triggers some transition-related logic. This ensures that urgent UI updates (like typing or clicking) are not blocked.

Working with the `useMemo` Hook

- **Memoization**: This is an optimization technique in which the result of expensive function calls are cached (remembered) based on specific arguments. When the same arguments are provided again, the cached result is returned instead of re-computing the function.
- **`useMemo` Hook**: This hook is used to memoize computed values. Here is an example of memoizing the result of sorting a large array. The `expensiveSortFunction` will only run when `largeArray` changes:

```
const memoizedSortedArray = useMemo(
  () => expensiveSortFunction(largeArray),
  [largeArray]
);
```

Working with the `useCallback` Hook

- **`useCallback` Hook**: This is used to memoize function references.

```
const handleClick = useCallback(() => {
  // code goes here
}, [dependency]);
```

- **`React.memo`**: This is used to memoize a component to prevent it from unnecessary re-renders when its prop has not changed.

```
const MemoizedComponent = React.memo(({ prop }) => {
 return (
   <>
     {/* Presentation */}
   </>
 )
});
```

Dependency Management Tools

- **Dependency Definition**: In software, a dependency is where one component or module in an application relies on another to function properly. Dependencies are common in software applications because they allow developers to use pre-built functions or tools created by others. The two core dependencies needed for a React project will be the `react` and `react-dom` packages:

```
"dependencies": {
  "react": "^18.3.1",
  "react-dom": "^18.3.1"
}
```

- **Package Manager Definition**: To manage software dependencies in a project, you will need to use a package manager. A package manager is a tool used for installation, updates and removal of dependencies. Many popular programming languages like JavaScript, Python, Ruby and Java, all use package managers. Popular package managers for JavaScript include npm, Yarn and pnpm.
- `package.json` **File**: This is a key configuration file in projects that contains metadata about your project, including its name, version, and dependencies. It also defines scripts, licensing information, and other settings that help manage the project and its dependencies.
- `package-lock.json` **File**: This file will lock down the exact versions of all packages that your project is using. When you update a package, then the new versions will be updated in the lock file as well.
- `node_modules` **Folder**: This folder contains the actual code for the dependencies listed in your `package.json` file, including both your project's direct dependencies and any dependencies of those dependencies.
- **Dev Dependencies**: These are packages that are only used for development and not in production. An example of this would be a testing library like Jest. You would install Jest as a dev dependency because it is needed for testing your application locally but not needed to have the application run in production.

```
"devDependencies": {
  "@eslint/js": "^9.17.0",
  "@types/react": "^18.3.18",
  "@types/react-dom": "^18.3.5",
  "@vitejs/plugin-react": "^4.3.4",
  "eslint": "^9.17.0",
  "eslint-plugin-react": "^7.37.2",
  "eslint-plugin-react-hooks": "^5.0.0",
  "eslint-plugin-react-refresh": "^0.4.16",
  "globals": "^15.14.0",
  "vite": "^6.0.5"
}
```

React Router

- **Introduction**: React Router is a third party library that allows you to add routing to your React applications. To begin, you will need to install React Router in an existing React project like this:

```
npm i react-router
```

Then inside of the `main.jsx` or `index.jsx` file, you will need to setup the route structure like this:

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router";
import App from "./App.jsx";

import "./index.css";

createRoot(document.getElementById("root")).render(
```

```
  <StrictMode>
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<App />} />
      </Routes>
    </BrowserRouter>
  </StrictMode>
);
```

The `path` and `element` are used to couple the URL and UI components together. In this case, we are setting up a route for the homepage that points to the `App` component.

- **Multiple Views and Route Setup**: It is common in larger applications to have multiple views and routes setup like this:

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

  <Route path="products">
    <Route index element={<ProductsHome />} />
    <Route path=":category" element={<Category />} />
    <Route path=":category/:productId" element={<ProductDetail />} />
    <Route path="trending" element={<Trending />} />
  </Route>
</Routes>
```

The `index` prop in these examples is meant to represent the default route for a given path segment. So the `Home` component will be shown at the root `/` path while the `ProductsHome` component will be shown at the `/products` path.

- **Nesting Routes**: You can nest routes inside other routes which results in the path of the child route being appended to the parent route's path.

```
<Route path="products">
  <Route path="trending" element={<Trending />} />
</Route>
```

In the example above, the path for the trending products will be products/trending.

- **Dynamic Segments**: A dynamic segment is where any part of the URL path is dynamic.

```
<Route path=":category" element={<Category />} />
```

In this example we have a dynamic segment called `category`. When a user navigates to a URL like products/brass-instruments, then the view will change to the `Category` component and you can dynamically fetch the appropriate data based on the segment.

- `useParams` **Hook**: This hook is used to access the dynamic parameters from a URL path.

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

export default function Category() {
  let params = useParams();
  {/* Accessing the category param: params.category */}
  {/* rest of code goes here */}
}
```

React Frameworks

- **Introduction**: React frameworks provide features like routing, image optimizations, data fetching, authentication and more. This means that you might not need to set up separate frontend and backend applications for certain use cases. Examples of React Frameworks include Next.js and Remix.
- **Next.js Routing**: This routing system includes support for dynamic routes, parallel routes, route handlers, redirects, internalization and more.

Here is an example of creating a custom request handler:

```
export async function GET() {
  const res = await fetch("https://example-api.com");
  const data = await res.json();

  return Response.json({ data });
}
```

- **Next.js Image Optimization**: The `Image` component extends the native HTML `img` element and allows for faster page loads and size optimizations. This means that images will only load when they enter the viewport and the `Image` component will automatically serve correctly sized images for each device.

```
import Image from "next/image";

export default function Page() {
  return (
    <Image src="link-to-image-goes-here" alt="descriptive-title-goes-here" />
  );
}
```

Prop Drilling

- **Definition**: Prop drilling is the process of passing props from a parent component to deeply nested child components, even when some of the child components don't need the props.

State Management

- **Context API**: Context refers to when a parent component makes information available to child components without needing to pass it explicitly through props. `createContext` is used to create a context object which represent the context that other components will read. The `Provider` is used to supply context values to the child components.

```
import { useState, createContext } from "react";

const CounterContext = createContext();

const CounterProvider = ({ children }) => {
  const [count, setCount] = useState(0);

  return (
    <CounterContext.Provider value={{ count, setCount }}>
      {children}
    </CounterContext.Provider>
  );
};

export { CounterContext, CounterProvider };
```

- **Redux**: Redux handles state management by providing a central store and strict control over state updates. It uses a predictable pattern with actions, reducers, and middleware. Actions are payloads of information that send data from your application to the Redux store, often triggered by user interactions. Reducers are functions that specify how the state should change in response to those actions, ensuring the state is updated in an immutable way. Middleware, on the other hand, acts as a bridge between the action dispatching and the reducer, allowing you to extend Redux's functionality (e.g., logging, handling async operations) without modifying the core flow.
- **Zustand**: This state management solution is ideal for small to medium-scale applications. It works by using a `useStore` hook to access access state directly in components and pages. This lets you modify and access data without needing actions, reducers, or a provider.

Debugging React Components Using the React DevTools

- **React Developer Tools**: This is a browser extension you can use in Chrome, Firefox and Edge to inspect React components and identify performance issues. For Safari, you will need to install the `react-devtools` npm package. After installing React DevTools and opening a React app in the browser, open the browser developer tools to access the two extra tabs provided for debugging React – Components and Profiler.
- **Components Tab**: This tab displays each component for you in a tree view format. Here are some things you can do in this tab:
  - view the app's component hierarchy
  - check and modify props, states, and context values in real time
  - check the source code for each selected component
  - log the component data to the console
  - inspect the DOM elements for the component
- **Profiler Tab**: This tab helps you analyze component performance. You can record component performance so you can identify unnecessary re-renders, view commit durations, and subsequently optimize slow components.

React Server Components

- **Definition**: React Server Components are React components that render exclusively on the server, sending only the final HTML to the client. This means those components can directly access server-side resources and dramatically reduce the amount of JavaScript sent to the browser.