# Finding the Path

## UseEffect()

1. **when this `useEffect` called?**
    - This `useEffect` is called after every render of the component.
    - The basic nature or the default behaviour of useEffect is to be called after each render. But if we give it a dependency array, then it is just called once.
        1. but what if we put something inside the dependency array?
        - Then it will only be called when the dependency changes.

## UseEffect() Cases

Syntax : `Without Dependency array`

```
useEffect(()=>{})
```

## Case 1 : *when the dependency array is not included as an argument in the useEffect hook*

In React, when we use the `useEffect` hook without providing a dependency array, the effect will be executed on every render of the component. This means that the code inside the `useEffect` will run both after the initial render and after every subsequent render.

Here's an example of using `useEffect` without a dependency array:

```
import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // This code will run on every render
    console.log('Effect executed');
  });

  return (
    <div>
      {/* Component content */}
    </div>
  );
}
```

In this example, the `useEffect` without a dependency array doesn't specify any dependencies, so it will run after every render of `MyComponent`. This behavior can be useful in some cases, but it's essential to be cautious when using `useEffect` without a dependency array because it can lead to performance issues, especially if the effect contains expensive operations.

When we don't provide a dependency array, the effect is considered to have an empty dependency array, which is equivalent to specifying every value as a dependency. Therefore, it's important to understand the consequences of running the effect on every render and to use this pattern judiciously.

> In many cases, we might want to include a dependency array to control when the effect should run based on changes in specific variables or props. This can help optimize the performance of our component and prevent unnecessary re-renders.

## **Case 2 :** *when the dependency array is empty (i.e., []) in the arguments of the useEffect hook*

Here's the `syntax` for using `useEffect with an empty dependency array`:

```
useEffect(() => {}, []);
```

It's important to consider the use of dependency arrays in `useEffect` to manage the execution of effects efficiently based on the specific requirements of the component.

In React, when the dependency array is empty (`[ ]`) in the arguments of the `useEffect` hook, the callback function will be executed once during the initial render of the component and not on every re-render. The effect will only run after the initial render and will not be triggered on subsequent re-renders unless the component is unmounted and remounted.

Here's the example of using `useEffect` with an empty dependency array:

```
import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // This code will run after the initial render and will not re-run on
subsequent re-renders
    console.log('Effect executed');
  }, []);

  return (
    <div>
      {/* Component content */}
    </div>
  );
}
```

In this example, the callback function in the `useEffect` with an empty dependency array will run once after the initial render and will not run on every subsequent re-render of `MyComponent`. It is suitable for scenarios where you only want the effect to run after the initial render and not on re-renders.

If you intend for the effect to run on every re-render, then you can omit the dependency array altogether, as shown in your first example. However, if you want the effect to run only once after the initial render, you

should specify an empty dependency array (`[]`) as shown in the corrected example above.

## **Case 3 :** *When the dependency array in the arguments of the useEffect hook contains a condition*

(a variable or set of variables), the callback function will be executed once during the initial render of the component and also on re-renders if there is a change in the condition.

Here's an example of using `useEffect with a condition in the dependency array` :

```
import React, { useEffect, useState } from 'react';

function MyComponent() {

  const [count, setCount] = useState(0);

  useEffect(() => {

    // This code will run after the initial render and whenever 'count' changes

    console.log('Effect executed');
  }, [count]);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment C
ount</button>
      <p>Count: {count}</p>
    </div>
  );
}
```

In this case, the useEffect has count as a dependency in the array. This means that the effect will run after the initial render and then again whenever the count variable changes. If we click the Increment Count button, the count state will change, triggering the effect to run again. If the condition specified in the dependency array doesn't change, the effect won't run on re-renders.

This allows us to control when the effect runs based on specific conditions or dependencies. It's a useful way to ensure that the effect only runs when the relevant data or state has changed.

# useState()

**Important Notes :**

- Always call `useState()` inside the functional component, not outside of the component.
- Also, try to call `useState()` hook on the top. So that we don't have a lot of inconsistency in our code.
- Never use or create `useState()` hook inside the `if/else`, `for loop`, `functions` & `conditions`.

2. **What would happen if we do `console.log(useState())`?**

If you use `console.log(useState())` in a React functional component, it will display the result of calling the `useState()` function in our browser's developer console. The `useState()` function is a React Hook that is typically used to declare a state variable in a functional component. When we call `useState()`, it `returns an array with two elements : the current state value and a function to update that state value`.

For example:

```
const [count, setCount] = useState(0);
```

In this example, **count** is the current state value, and **setCount** is the function to update it.

If we do `console.log(useState())`, we will see something like this in the console:

```
[0, Function]
```

The first element of the array is the initial state value (in this case, 0), and the second element is the function to update the state.

> However, using `console.log(useState())` directly in our component without destructuring the array and assigning names to these elements isn't a common or recommended practice.

Normally, we would destructure the array elements when using `useState()` to make our code more readable and maintainable.

So, it's more typical to use useState() like this:

```
const [count, setCount] = useState(0);
console.log(count); // Logs the current state value
console.log(setCount); // Logs the state update function
```

This way, we can access and work with the state and state update function in our component.

## React Router DOM

1. **Install React router DOM**

   ```
   npm i react-router-dom
   ```

2. **Create Routing configuration**

   - For creating a routing configuration, we need to `import` a `createBrowserRouter` & `RouterProvider` from `react-router-dom`.

```
import { createBrowserRouter, RouterProvider } from "react-router-dom";
```

- Created a path for our app such as home, about us, contact us etc using `createBrowserRouter` function. It takes a some configuration & this configuration is a list of objects.
- Each and every object defines a different `paths`.
  - Path is nothing but an object, it takes 2 things as `path` & `element`.

Example :

```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
  },
  {
    path: "/about",
    element: <AboutUs />,
  },
]);
```

- then we need to provide this routing configuration to render it.
- So, We use `RouterProvider`. And this router provider will actually provide this routing configuration to our app.
- `RouterProvider` is a component.

```
const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(<RouterProvider router={appRouter} />);
```

> This is how we create a different routes in React.

3. **Create a Error Component**

```
<!--  This shows that if path has something else which is not defined here
then show error page. -->

const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
    errorElement: <Error />,
  },
  {
    path: "/about",
    element: <AboutUs />,
  },
```

```
  {
    path: "/contact",
    element: <ContactUs />,
  },
]);
```

- There is `useRouteError()` hook provided by `react router dom`, by using this hook we can get more information about the Error.
- It `returns the object` which has several information, we can use that instead of normal content. like "Oops, something went wrong."

4. **Children Routes**

- So, my `<AppLayout/>` will stay as it is. but over here, I will create my children routes.
- Children is a list of paths
- So, this app layout has 3 children, load these 3 children according to the path.

```
const appRouter = createBrowserRouter([
  {
    path: "/",
    element: <AppLayout />,
    children: [
      {
        path: "/",
        element: <Body />,
      },
      {
        path: "/about",
        element: <AboutUs />,
      },
      {
        path: "/contact",
        element: <ContactUs />,
      },
    ],
    errorElement: <Error />,
  },
]);
```

- **Where should I load this?**
  - over here in this `Outlet`

```
return (
    <>
      <div className="container-fluid">
        <Header />
        <Outlet />
      </div>
```

```
    </>
  );
```

- `Outlet` is a Component
- This `Outlet` will be filled with the children, according to the path, on what page we are.
- So, whenever I will be on slash, my body will be filled over here in this `Outlet`.

- **Will I see that `Outlet` in my HTML?**
  - No, Basically, this `Outlet` is replaced by the component. example: `Outlet` is replaced by `ContactUs` component.

5. **Links**

- In react, we should never ever use `anchor` tag for `link` because if we use then whole page got refreshed.
- We can navigate to a new page without reloading the whole page, by using `Link` component from `react-router-dom`.
- This `Link` component works exactly the same as anchor tag.
- The only differences are, We use in anchor tag `href` and Link component `to` for redirecting :

Link component example :

```
<Link className="nav-link" to="/about">
  About Us
</Link>
```

Anchor Tag example :

```
<a className="nav-link" href="/about">
  About Us
</a>
```

6. **Dynamic Routing**

   In the context of React Router DOM, dynamic routing refers to the ability to handle route matching and rendering based on dynamic data or conditions within your React components. This means that the routing logic can be influenced by factors such as user input, API responses, or the current state of the application.

   Here are some common scenarios where dynamic routing in React Router DOM may be used:

   1. **Dynamic Route Parameters**: React Router allows you to define dynamic segments in the route path using parameters. For example, a route path like `/users/:id` can be used to display details of a specific user based on the `id` parameter in the URL.

   Example :

```
// App.js

import React from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';

const UserDetail = ({ match }) => {
  const userId = match.params.id;
  return <div>User ID: {userId}</div>;
};

const App = () => {
  return (
    <Router>
      <Route path="/users/:id" component={UserDetail} />
    </Router>
  );
};

export default App;
```

```
 In this example, the UserDetail component receives the id parameter from the route path /users/:id, allowing it to render dynamic user details based on the id.
```

2. **Conditional Routing**: You can conditionally render components based on certain conditions within your React components. This can be useful for showing different views or redirecting users based on the state of the application.

Example :

```
// App.js

import React from 'react';
import { BrowserRouter as Router, Route, Redirect } from 'react-router-dom';

const App = () => {
  const loggedIn = true; // Example condition for authentication

  return (
    <Router>
      <Route path="/dashboard">
        {loggedIn ? <Dashboard /> : <Redirect to="/login" />}
      </Route>
    </Router>
  );
};

export default App;
```

```
 In this example, the Dashboard component is only rendered if the loggedIn
condition is true. Otherwise, the user is redirected to the /login route.
```

3. **Programmatic Navigation**: React Router DOM provides methods like `history.push()` to programmatically navigate to different routes based on user interactions or other events within your components.

Example :

```
// NavigationButton.js

import React from 'react';
import { useHistory } from 'react-router-dom';

const NavigationButton = ({ path, buttonText }) => {
  const history = useHistory();

  const handleClick = () => {
    history.push(path);
  };

  return <button onClick={handleClick}>{buttonText}</button>;
};

export default NavigationButton;
```

```
 In this example, the NavigationButton component allows for programmatic
navigation to a specified path when the button is clicked using the
history.push() method.
```

4. **Route Configuration based on Data**: You can dynamically generate routes based on data received from an API or other external sources. This can be useful for creating a flexible routing structure that adapts to changing data.

5. **Protected Routes**: Dynamic routing can be used to handle authentication and authorization logic, where certain routes are only accessible to authenticated users. This typically involves conditional rendering based on the authentication state of the user.

Overall, dynamic routing in React Router DOM allows you to create flexible and interactive navigation experiences in your React applications by adapting the routing behavior based on dynamic factors and conditions.

7. **useParams()**

- `useParams()` is a hook provided by React Router that allows you to access the `parameters` of the current `URL`.

  Example : Here's a basic example of how you can use the useParams() hook:

  It is destructuring.

  ```
  const { resId } = useParams();
  ```

- In this example, if the URL is '/restaurants/123', the component will receive '123' as the 'id' parameter.

8. **How do you create Nested Routes react-router-dom configuration ?**

> In React applications using react-router-dom, we can create nested routes by nesting our components inside each other within the route configuration. This allows you to define routes and components hierarchically, making it easier to manage the routing structure of your application.

```
💡Here's a step-by-step guide on how to create nested routes using react-router-dom:
```

1. Install react-router-dom if we haven't already :

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

2. Import the necessary components from react-router-dom in your application file :

```
import { BrowserRouter as Router, Route, Switch } from 'react
-router-dom';
```

- Defining our route hierarchy by nesting components within each other. Typically, this is done within a component that acts as a layout or container for the nested routes. For example, if we have a layout component called Layout:

```
import React from 'react';
import { Route } from 'react-router-dom';
// Import your nested route components
import Home from './Home';
import About from './About';
function Layout() {
  return (
    <div>
```

```
        <h1>My App</h1>
        <Route path="/home" component={Home} />
        <Route path="/about" component={About} />
      </div>
    );
  }
  export default Layout;
```

- In our main application file, wrap our entire application with the Router component, and use the component to render only the first matching route:

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react
-router-dom';
// Import your Layout component that defines nested routes
import Layout from './Layout';
function App() {
  return (
    <Router>
      <Switch>
        <Route path="/" component={Layout} />
      </Switch>
    </Router>
  );
}
export default App;
```

- Now we have a simple example of nested routes. In this case, the Layout component defines the `/home` and `/about` routes, and these nested routes can have their own components and nested routes as well. We can continue to nest routes further by adding more `<Route>` components inside the Home and About components to create a more complex routing structure. Remember that this is just a basic example, and we can customize our routing structure based on the requirements of our application. We can also use the exact prop on routes to ensure that only the exact path is matched if needed.

9. **Read about `createHashRouter`, `createMemoryRouter` from React Router docs.**

    1. `createHashRouter` - createHashRouter is part of the React Router library and provides routing capabilities for single-page applications (SPAs). It's commonly used for building client-side navigation within applications. Unlike traditional server-side routing, it uses the fragment identifier (hash) in the URL to manage and handle routes on the client side. This means that changes in the URL after the

        > symbol do not trigger a full page reload, making it suitable for SPAs.

        To use createHashRouter, we typically import it from the React Router library and define our routes using Route components. Here's a basic example of how you might use it:

```
import { createHashRouter, Route } from 'react-router-dom';
const App = () => (
  <createHashRouter>
    <Route path="/" component={Home} />
    <Route path="/about" component={About} />
    <Route path="/contact" component={Contact} />
  </createHashRouter>
);
```

2. `createMemoryRouter` - createMemoryRouter is another routing component provided by React Router. Unlike createHashRouter or BrowserRouter, createMemoryRouter is not associated with the browser's URL. Instead, it allows you to create an in-memory router for testing or other scenarios where you don't want to interact with the actual browser's URL.

   💡 Here's a simple example of how to use createMemoryRouter:

   ```
   import { createMemoryRouter, Route } from 'react-router-dom';
   const App = () => (
     <createMemoryRouter>
       <Route path="/" component={Home} />
       <Route path="/about" component={About} />
       <Route path="/contact" component={Contact} />
     </createMemoryRouter>
   );
   ```

   - In both cases, we define our application's routes within the router component and specify the components to render for each route. The choice between `createHashRouter` and `createMemoryRouter` depends on our specific use case, such as whether we're building an SPA that interacts with the 's URL or a scenario where we need an in-memory router for testing.

## SPA (Single page application)

1. **What is SPA ?**

   - "SPA stands for Single Page Application, which is a web application or website that interacts with the user by dynamically updating the content on the current page instead of loading complete new pages from the server. In other words, a single HTML page is loaded initially, and then the content is updated dynamically as the user interacts with the application, typically through JavaScript.

`or`

   - It's just one page, the components getting interchanged. that's all is happening via client side routing.

*Key characteristics of SPAs include :*

`Dynamic Updates` : In SPAs, content is loaded and updated without requiring a full page reload. This is achieved using JavaScript and client-side routing.

`Smooth User Experience` : SPAs can provide a smoother and more responsive user experience because they can update parts of the page without the entire page needing to be refreshed.

`Faster Initial Load` : While the initial load of an SPA might take longer as it downloads more JavaScript and assets, subsequent interactions with the application can be faster because only data is exchanged with the server and not entire HTML pages.

`Client-Side Routing` : SPAs often use client-side routing to simulate traditional page navigation while staying on the same HTML page. This is typically achieved using libraries like React Router or Vue Router.

`API-Centric` : SPAs are often designed to be more API-centric, where the client communicates with a backend API to fetch and send data, usually in JSON format. This allows for decoupling the front end and back end.

`State Management` : SPAs often use state management libraries (e.g., Redux for React or Vuex for Vue) to manage the application's state and data flow. Popular JavaScript frameworks and libraries like React, Angular, and Vue are commonly used to build SPAs. They offer tools and patterns to create efficient and maintainable single-page applications.

2. **What is the difference between `Client Side Routing` and `Server Side Routing`?**
    - Client-side routing and server-side routing are two different approaches to handling routing and navigation in web applications. They have distinct characteristics and are often used for different purposes.

Here's an overview of the key differences between them:

**Client-Side Routing :**

`Handling on the Client` : In client-side routing, routing and navigation are managed on the client side, typically within the web browser. JavaScript frameworks and libraries, such as React Router (for React applications) or Vue Router (for Vue.js applications), are commonly used to implement client-side routing.

`Faster Transitions` : Client-side routing allows for faster page transitions since it doesn't require the server to send a new HTML page for each route change. Instead, it updates the DOM and URL dynamically without full page reloads.

`Single-Page Application (SPA)` : Client-side routing is often associated with single-page applications (SPAs), where the initial HTML page is loaded, and subsequent page changes are made by updating the content using JavaScript.

`SEO Challenges` : SPAs can face challenges with search engine optimization (SEO) because search engine crawlers may not fully index the content that relies heavily on client-side rendering. Special techniques like server-side rendering (SSR) or pre-rendering can be used to address this issue.

`Route Management` : Routing configuration is typically defined in code and managed on the client side, allowing for dynamic and flexible route handling.

**Server-Side Routing :**

- Server side routing means you make a network call and the page that `aboutus.html` is coming from server that server side routing.

`Handling on the Server` : Server-side routing manages routing and navigation on the server. When a user requests a different URL, the server generates and sends a new HTML page for that route.

`Slower Transitions` : Server-side routing tends to be slower in terms of page transitions compared to client-side routing, as it involves full page reloads.

`Traditional Websites` : Server-side routing is commonly used for traditional multi-page websites where each page is a separate HTML document generated by the server.

`SEO-Friendly` : Server-side routing is inherently more SEO-friendly, as each page is a separate HTML document that can be easily crawled and indexed by search engines.

`Route Configuration` : Routing configuration in server-side routing is typically managed on the server, and URLs directly correspond to individual HTML files or routes.

In summary, client-side routing is suitable for building SPAs and offers faster, more interactive user experiences but can pose SEO challenges. Server-side routing is more SEO-friendly and is used for traditional websites with separate HTML pages, but it can be slower in terms of page transitions. The choice between these two routing approaches depends on the specific requirements and goals of a web application or website. In some cases, a hybrid approach that combines both client-side and server-side routing techniques may be used to achieve the best of both worlds.

## Various Ways To Add Images

There are several ways to add and display images.

1. Importing images using ES6 Modules
2. Using public folder
3. Loading images from a remote source
4. Using image assets within CSS

- `Importing images using ES6 Modules:` We can import images directly using ES6 modules. This is a common approach for small to medium-sized apps , and it's straightforward. Firstly, We have to place our image in the project directory (e.g., in the src folder or a subfolder).

Example:

```
import React from 'react';
import myImage from './my_image.jpg';
function App() {
    return (
        <div>
            <img src={myImage} alt="My Image" />
```

```
        </div>
    );
}
export default App;
```

- `Using public folder:` If we want to reference images in the public folder, we can do so without importing them explicitly. This method is useful for handling large image assets or for dynamic image URLs. Place your image in the public directory.

```
// public/my_image.jpg
```

Then, reference it in your code:

```
import React from 'react';
function App() {
    return (
        <div>
            <img src={process.env.PUBLIC_URL + '/my_image.jp
g'} alt="My Image" />
        </div>
    );
}
export default App;
```

- `Loading images from a remote source:` We can load images from a remote source, such as an external URL or a backend API, by specifying the image URL directly in our img tag.

Example:

```
import React from 'react';
function App() {
    const imageUrl = 'https://example.com/my_image.jpg';
    return (
        <div>
            <img src={imageUrl} alt="My Image" />
        </div>
    );
}
export default App;
```

- `Using image assets within CSS:` We can also use images as background images or in other CSS styling. In this case, we can reference the image in your CSS file.

Example CSS (styles.css):

```
.image-container {
    background-image: url('/my_image.jpg');
    width: 300px;
    height: 200px;
}
```

Then, apply the CSS class to your JSX:

```
import React from 'react';
import './styles.css';
function App() {
    return (
        <div className="image-container">
            {/* Content goes here */}
        </div>
    );
}
export default App;
```

Choose the method that best fits your project's requirements and organization. Importing images using ES6 modules is the most common and convenient approach for most React applications, especially for small to medium-sized projects. For larger projects with many images, consider the folder structure and organization to keep our code clean and maintainable.