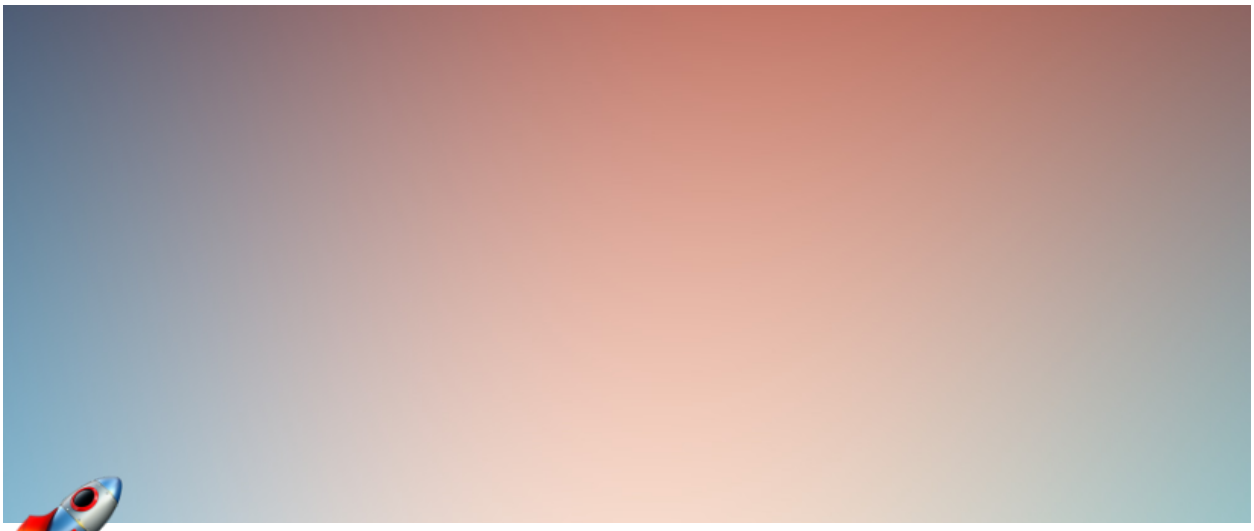

Episode-07 | Finding The Path

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-07** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) What are various ways to add images to our App? Explain with code examples

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
```

1 `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;
```

2 `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;
```

3 `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;
```

4 `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):

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

Then, apply the CSS class to your JSX:

```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.

Q) 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.

Q) How will useEffect behave if we don't add a dependency array?

> 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.

💡 **Syntax**

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

`Case 1, when the dependency array is not included as an argument in the useEffect hook`, the callback function inside useEffect will be executed every time the component is initially rendered and subsequently re-rendered. This means that the effect runs on every render cycle, and there are no dependencies that control when it should or should not execute.

Here's the relevant code again for reference:

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

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

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

The callback function in the useEffect will log `Effect executed` to the console every time MyComponent is rendered or re-rendered. This behavior can be useful in some cases but should be used carefully to avoid excessive or unnecessary

executions of the effect. If we want more control over when the effect should run, we can include a dependency array to specify the dependencies that trigger the effect when they change.

`In Case 2, when the dependency array is empty (i.e., []) in the arguments of the useEffect hook`, the callback function will indeed be executed once during the initial render of the component. However, it won't be limited to the initial render only. It will run after the initial render and then on every re-render of the component.

Here's an 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 on eve
ry re-render
    console.log('Effect executed');
  }, []);

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

That's not accurate. In Case 2, when the dependency array is empty (i.e., []) in the arguments of the useEffect hook, the callback function will indeed be executed once during the initial render of the component. However, it won't be limited to the initial render only. It will run after the initial render and then on every re-render of the component.

Here's an 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 on eve
ry re-render
    console.log('Effect executed');
  }, []);

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

In this case, the callback function in the useEffect with an empty dependency array will run once after the initial render and then on every subsequent re-render of MyComponent. It won't run if the component is unmounted and then re-mounted, but it will run whenever the component is re-rendered, even if there are no dependencies to watch for changes.

If you want the effect to run only once, and not re-run on re-renders, you can specify an empty dependency array like this:

```
useEffect(() => {
  // This code will run only once, after the initial render
  console.log('Effect executed');
}, []);
```

In this case, the effect will run only after the initial render, and it won't run again on subsequent re-renders.

`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`:

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

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

  useEffect(() => {
    // This code will run after the initial render and whenev
er '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.

Q) What is `SPA` ?

> SPA stands for `Single Page Application`. It's a type of web application or website that interacts with the user by

> dynamically rewriting the current web page rather than loading entire 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.

`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.

Q) What is the difference between `Client Side Routing` and `Server Side Routing`?

A: 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`:

`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.