4. Event Handling in React

What

In React, **event handling** allows components to respond to user interactions, such as clicks, mouse movements, or form submissions. Just like in regular JavaScript, React provides a way to handle these events, but with some React-specific patterns.

Why

Handling events is essential for building interactive applications. With event handling, you can create buttons that perform actions, inputs that respond to typing, and components that update dynamically based on user interactions.

How

We'll set up a simple button with an **onClick** event and then add more event types (like mouse events and form handling) to demonstrate different ways to handle user interactions in React.

Step 1: Creating a Button with an onClick Event

1. **Create a New Component**: In your src folder, create a file called EventDemo.jsx.
2. **Define the Component and Add a Button**: Write the following code to create a button that logs a message to the console when clicked.

```
import React from 'react';
function EventDemo() {
  const handleClick = () => {
    console.log('Button was clicked!');
  };
  return (
    <div>
      <button onClick={handleClick}>Click Me</button>
    </div>
  );
}
export default EventDemo;
```

-handleClick is a **handler function** that runs when the button is clicked.

-We use the onClick attribute on the <button> element and set it to handleClick to define what should happen when the button is clicked.

3. **Use the Component**: In App.jsx, import and add EventDemo to test the button.

```
import React from 'react';
import EventDemo from './EventDemo';
function App() {
return (
<div>
<h1>Event Handling in React</h1>
<EventDemo />
</div>
);
}
export default App;
```

-When you click the button, "Button was clicked!" should appear in the console.

Step 2: Adding an onMouseEnter and onMouseLeave Event

Let's add two new event handlers for when the mouse enters and leaves an element.

1. **Modify EventDemo.jsx** to include these events:

```
import React from 'react';
function EventDemo() {
  const handleClick = () => {
    console.log('Button was clicked!');
  };

  const handleMouseEnter = () => {
    console.log('Mouse entered!');
  };

  const handleMouseLeave = () => {
    console.log('Mouse left!'); };;
 return (
    <div>
      <button
        onClick={handleClick}
        onMouseEnter={handleMouseEnter}
        onMouseLeave={handleMouseLeave}
      >
        Hover or Click Me
      </button>
    </div>
  );
}
export default EventDemo;
```

- onMouseEnter runs when the mouse enters the button area, and onMouseLeave runs when the mouse leaves the area.

Step 3: Handling Form Events

Forms are a key part of most web applications. Let's add an input field with an onChange event to track what the user types.

1. **Update EventDemo.jsx** to add an input field:

```
import React, { useState } from 'react';
function EventDemo() {
  const [inputValue, setInputValue] = useState('');

  const handleClick = () => {
    console.log('Button was clicked!');
  };

  const handleMouseEnter = () => {
    console.log('Mouse entered!');
  };

  const handleMouseLeave = () => {
    console.log('Mouse left!');
  };

  const handleInputChange = (event) => {
    setInputValue(event.target.value);
    console.log('Input value:', event.target.value);
  };
  return (
    <div>
      <button
        onClick={handleClick}
        onMouseEnter={handleMouseEnter}
        onMouseLeave={handleMouseLeave}
      >
```

```
        Hover or Click Me
      </button>

      <div style={{ marginTop: '20px' }}>
        <input
          type="text"
          value={inputValue}
          onChange={handleInputChange}
          placeholder="Type something..."
        />
        <p>Current input: {inputValue}</p>
      </div>
    </div>
  );
}
export default EventDemo;
```

- handleInputChange is triggered whenever the user types in the input field. The event.target.value contains the current input value.

-We use **state** (inputValue) to store and display the input's current value dynamically.

Summary

In this example, we've covered:

- **onClick**: Handles button clicks.
- **onMouseEnter and onMouseLeave**: Track when the mouse enters or leaves an element.
- **onChange**: Captures and stores user input in form elements.

Event handling allows us to make our app interactive and respond to user actions, which is a core part of creating dynamic UIs in React. Next, we'll explore how to display lists dynamically using props and state.

5. Rendering Lists and Using Hooks

What

In React, **rendering lists** allows us to display multiple items dynamically by looping through data, such as an array. **Hooks** like `useState` help manage and update data in components, making them interactive and responsive to user actions.

Why

Dynamic lists are a fundamental part of building real applications. For example, a shopping app may need to display a list of products, and a to-do app would display tasks. Using `useState`, we can add interactivity, like adding, removing, or updating items in the list.

How

We'll create a simple to-do list application where users can add and delete tasks. This will introduce the concept of rendering lists and show how `useState` allows us to manage the list dynamically.

Step 1: Set Up the To-Do List Component

1. Create a New Component File: Inside the `src` folder, create a file called `TodoList.jsx`.

2. Define the Component and State: In `TodoList.jsx`, define the component and set up `useState` to manage the list of tasks.

```
import React, { useState } from 'react';
function TodoList() {
  const [tasks, setTasks] = useState(['Learn React', 'Build a project']);
  const [newTask, setNewTask] = useState('');
  return (
    <div>
      <h2>To-Do List</h2>
      <ul>
        {tasks.map((task, index) => (
          <li key={index}>{task}</li>
        ))}
      </ul>
    </div>
  );
}
export default TodoList;
```

- Here, `tasks` is an array of initial to-do items, managed by `useState`.

- We use the `.map()` method to loop over the `tasks` array and render each item inside a `<li>` element.

- The `key` prop, here set to `index`, uniquely identifies each list item to help React track changes to the list efficiently.

Step 2: Add New Tasks to the List

Next, we'll add a form with an input field and a button to allow users to add tasks.

1. Update `TodoList.jsx` to add an input and a button to add tasks:

```
import React, { useState } from 'react';
function TodoList() {
  const [tasks, setTasks] = useState(['Learn React', 'Build a project']);
  const [newTask, setNewTask] = useState('');
  const handleAddTask = () => {
    if (newTask.trim() !== '') {
      setTasks([...tasks, newTask]);
      setNewTask(''); // Clear the input field after adding
    }
  };
  return (
    <div>
      <h2>To-Do List</h2>
      <ul>
        {tasks.map((task, index) => (
          <li key={index}>{task}</li>
        ))}
      </ul>

      <input
        type="text"
        value={newTask}
        onChange={(e) => setNewTask(e.target.value)}
        placeholder="Enter a new task"/>
    <button onClick={handleAddTask}>Add Task</button>
```

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

- `newTask` stores the value of the input field, while `handleAddTask` updates the `tasks` array with the new task.

- We use `[...tasks, newTask]` to create a new array with the existing tasks and the new task, then set it as the updated list.

Step 3: Remove Tasks from the List

To make the list more interactive, let's add a delete button for each task.

1. Modify `TodoList.jsx` to add a delete button that removes tasks:

```
import React, { useState } from 'react';
function TodoList() {
  const [tasks, setTasks] = useState(['Learn React', 'Build a project']);
  const [newTask, setNewTask] = useState('');

  const handleAddTask = () => {
    if (newTask.trim() !== '') {
      setTasks([...tasks, newTask]);
      setNewTask('');
    }
  };

  const handleDeleteTask = (index) => {
    const updatedTasks = tasks.filter((_, i) => i !== index);
    setTasks(updatedTasks);
  };

  return (
    <div>
      <h2>To-Do List</h2>
      <ul>
        {tasks.map((task, index) => (
          <li key={index}>
            {task}
            <button onClick={() => handleDeleteTask(index)}>Delete</button>
          </li>
```

```
        ))}
      </ul>

      <input
        type="text"
        value={newTask}
        onChange={(e) => setNewTask(e.target.value)}
        placeholder="Enter a new task"
      />
      <button onClick={handleAddTask}>Add Task</button>
    </div>
  );
}

export default TodoList;
```

- handleDeleteTask` takes the index of the task and removes it from the array by creating a new array without that item.

- We use the `.filter()` method to create a new array that excludes the task with the matching index, then set this updated array as the new list.

Summary

In this example, we've covered:

- useState: Manages the list of tasks and the input field's value.

- map(): Loops through the `tasks` array to render each task dynamically.

- Event Handling: We add new tasks and delete existing ones based on user interactions.

Rendering lists and managing state are essential skills in React, and this example demonstrates how to make a dynamic, interactive list. Next, we'll move on to working with more advanced state management techniques.

6. Using Objects and Arrays in `useState`

What

In React, `useState` isn't limited to simple values like strings or numbers. We can also use it to manage complex data structures like objects and arrays. This is helpful when you need to manage a list of items or store multiple related properties together in a single component.

Why

Handling complex state with `useState` allows you to build more sophisticated components. For instance, if you're building a shopping cart, you might use an array to store all cart items or an object to track a product's details (like name, price, and quantity). Knowing how to work with these structures helps you manage and update data effectively in your React components.

How

We'll build a small profile card component that allows us to manage an object's properties, and a list of hobbies using an array. We'll then use `useState` to update both types of state.

Step 1: Using Objects in `useState`

Let’s start by creating a user profile component with an object that stores user details.

1. Create a New Component File: In the `src` folder, create a file called `UserProfile.jsx`.

2. Define the Component and Set Up State: In `UserProfile.jsx`, initialize `useState` with an object to hold user details.

```
import React, { useState } from 'react';
function UserProfile() {
  const [user, setUser] = useState({
    name: 'John Doe',
    age: 25,
    location: 'New York',
  });
const handleUpdateLocation = () => {
    setUser((prevUser) => ({
      ...prevUser,
      location: 'Los Angeles',
    })) };

  return (
    <div>
      <h2>User Profile</h2>
      <p>Name: {user.name}</p>
      <p>Age: {user.age}</p>
      <p>Location: {user.location}</p>
      <button onClick={handleUpdateLocation}>Move to Los Angeles</button>
    </div>
  );
}
export default UserProfile;
```

- We define `user` as an object containing properties: `name`, `age`, and `location`.

- The `handleUpdateLocation` function uses the updater function in `setUser` to change the `location` property while preserving the other properties.

- We use the spread operator (`...prevUser`) to keep the existing properties intact when updating only the `location`.

Step 2: Using Arrays in `useState`

Next, let's add an array to manage a list of hobbies, which we'll allow the user to add to dynamically.

1. Extend `UserProfile.jsx` by adding an array of hobbies and allowing the user to add a new hobby.

```
import React, { useState } from 'react';
function UserProfile() {
  const [user, setUser] = useState({
    name: 'John Doe',
    age: 25,
    location: 'New York',
  });

  const [hobbies, setHobbies] = useState(['Reading', 'Traveling']);

  const handleUpdateLocation = () => {
    setUser((prevUser) => ({
      ...prevUser,
      location: 'Los Angeles',
    }));
  };

  const addHobby = () => {
    setHobbies((prevHobbies) => [...prevHobbies, 'New Hobby']);
  };

  return (
    <div>
      <h2>User Profile</h2>
      <p>Name: {user.name}</p>
      <p>Age: {user.age}</p>
```

```
      <p>Location: {user.location}</p>
      <button onClick={handleUpdateLocation}>Move to Los Angeles</button>

      <h3>Hobbies</h3>
      <ul>
        {hobbies.map((hobby, index) => (
          <li key={index}>{hobby}</li>
        ))}
      </ul>
      <button onClick={addHobby}>Add Hobby</button>
    </div>
  );
}
export default UserProfile;
```

- `hobbies` is an array of strings representing different hobbies.

- `addHobby` uses the updater function with `setHobbies` to add a new hobby. The spread operator (`...prevHobbies`) is used to keep all existing hobbies and add a new one at the end.

Step 3: Updating Array and Object State Together

Now, let's add an input to allow users to add custom hobbies, demonstrating how to update both `user` (an object) and `hobbies` (an array) interactively.

1. Modify `UserProfile.jsx` to include an input field for adding a custom hobby.

```
import React, { useState } from 'react';
function UserProfile() {
  const [user, setUser] = useState({
    name: 'John Doe',
    age: 25,
    location: 'New York',
  });

  const [hobbies, setHobbies] = useState(['Reading', 'Traveling']);
  const [newHobby, setNewHobby] = useState('');

  const handleUpdateLocation = () => {
    setUser((prevUser) => ({
      ...prevUser,
      location: 'Los Angeles',
    }));
  };

  const addHobby = () => {
    if (newHobby.trim()) {
      setHobbies((prevHobbies) => [...prevHobbies, newHobby]);
      setNewHobby('');
    }
  };
```

```
  return (
    <div>
      <h2>User Profile</h2>
      <p>Name: {user.name}</p>
      <p>Age: {user.age}</p>
      <p>Location: {user.location}</p>
      <button onClick={handleUpdateLocation}>Move to Los Angeles</button>

      <h3>Hobbies</h3>
      <ul>
        {hobbies.map((hobby, index) => (
          <li key={index}>{hobby}</li>
        ))}
      </ul>

      <input
        type="text"
        value={newHobby}
        onChange={(e) => setNewHobby(e.target.value)}
        placeholder="Enter a new hobby"
      />
      <button onClick={addHobby}>Add Hobby</button>
    </div>
  );
}
export default UserProfile;
```

-`newHobby` is a separate piece of state to track the input field's value.

- `addHobby` checks if `newHobby` is non-empty, adds it to the `hobbies` array, and clears the input field afterward.

Summary

In this example, we've learned how to:

- Use objects in `useState` to store multiple related properties.

- Use arrays in `useState` to manage lists of items.

- Use the spread operator to update only parts of an object or array while keeping the rest of the data intact.

Understanding how to manage objects and arrays in `useState` is key to handling complex data in React apps. Up next, we'll explore the `useEffect` hook, which helps with side effects like data fetching.

7. `useEffect` Hook

What

The `useEffect` hook allows you to run side effects in your components. Side effects are actions that occur outside the React rendering process, such as data fetching, updating the DOM, or setting up timers. `useEffect` is React's way of performing tasks that don't directly affect rendering but need to happen after a component is rendered.

Why

In React, we want to keep rendering and side effects separate. `useEffect` allows us to handle these effects at specific times in a component's lifecycle, such as when it mounts (appears on the screen), updates (changes), or unmounts (is removed from the screen). This makes it possible to handle complex behavior, like fetching data from an API, updating document titles, or cleaning up resources when a component is no longer needed.

How

We'll use `useEffect` to create a simple component that fetches and displays data from an API when it mounts. We'll also add an effect that updates the page title based on the data.

Step 1: Basic Use of `useEffect` for Logging:

Let's start with a basic example to see how `useEffect` works. We'll create a simple component that logs a message to the console every time it renders.

1. Create a New Component File: Inside the `src` folder, create a file called `EffectDemo.jsx`.

2. Set Up the Component and Basic `useEffect` Hook: In `EffectDemo.jsx`, add a `useEffect` hook to log a message whenever the component mounts or updates.

```
import React, { useEffect } from 'react';
function EffectDemo() {
  useEffect(() => {
    console.log('Component has been rendered or updated!');
  });
  return (
    <div>
      <h2>Effect Demo</h2>
      <p>Check your console to see the effect in action.</p>
    </div>
  )}
}
export default EffectDemo;
```

- `useEffect` takes a function as its first argument. This function contains the code that should run as a side effect (in this case, logging a message).

- Without any dependencies, `useEffect` runs every time the component renders or re-renders.

Step 2: Running `useEffect` Only Once on Component Mount

To make `useEffect` run only once when the component mounts, we can provide an empty dependency array. This setup is common for actions that should only happen once, like fetching data.

1. Modify `EffectDemo.jsx` to only log when the component mounts:

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

function EffectDemo() {
  useEffect(() => {
    console.log('Component mounted!');
  }, [] );
  return (
    <div>
      <h2>Effect Demo</h2>
      <p>Check your console for a one-time log on component mount.</p>
    </div>
  );
}
export default EffectDemo;
```

- By adding `[]` as the second argument to `useEffect`, we tell React that this effect has no dependencies and should run only once after the initial render (when the component mounts).

- This approach is helpful for actions like fetching data when a component loads.

Step 3: Fetching Data with `useEffect`

Now let's create a component that fetches data from an API using `useEffect`. We'll use the JSONPlaceholder API to fetch a list of users.

1. Update `EffectDemo.jsx`** to fetch data on mount:

```
import React, { useEffect, useState } from 'react';
  function EffectDemo() {
    const [users, setUsers] = useState([]);


    useEffect(() => {
      fetch('https://jsonplaceholder.typicode.com/users')
        .then((response) => response.json())
        .then((data) => setUsers(data))
        .catch((error) => console.error('Error fetching data:', error));
    }, [] );
    return (
      <div>
        <h2>User List</h2>
        <ul>
          {users.map((user) => (
            <li key={user.id}>{user.name}</li>
          ))}
        </ul>
      </div>
    );
  }
  export default EffectDemo;
```

- We initialize `users` as an empty array and update it with the fetched data.

- `useEffect` fetches data from the API only once when the component mounts (thanks to the empty dependency array `[]`) and use `.then()` to handle the response and update `users` with the fetched data, which is then rendered in a list.

Step 4: Cleanup with `useEffect`

Sometimes, we need to clean up effects to prevent memory leaks. For example, if we set up a timer or subscription, we should remove it when the component unmounts. `useEffect` allows us to return a cleanup function that runs when the component unmounts.

1. Modify `EffectDemo.jsx` to add a timer and clear it on unmount:

```
import React, { useEffect, useState } from 'react';
function EffectDemo() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    const timer = setInterval(() => {
      setCount((prevCount) => prevCount + 1);
    }, 1000);
    // Cleanup function to clear the timer
    return () => {
      clearInterval(timer);
      console.log('Component unmounted, timer cleared');
    };
  }, [] );
  return (
    <div>
      <h2>Timer</h2>
      <p>Count: {count}</p>
    </div>
  )}
export default EffectDemo;
```

- Here, we use `setInterval` to increment `count` every second.

- The cleanup function (inside `return`) clears the timer when the component unmounts, preventing memory leaks.

Summary

In this example, we've learned how to:

- Use `useEffect` for side effects that run on every render, only once, or when specific dependencies change.

- Fetch data with `useEffect` and store it in state.

- Clean up effects by returning a cleanup function to prevent memory leaks.

The `useEffect` hook is crucial for handling side effects in React and is widely used for tasks like data fetching, setting up subscriptions, and managing timers. This wraps up our exploration of `useEffect`. Next, we'll dive into state management using `useContext`!