In React, the useCallback hook is used to memoize a callback function so that it's not recreated on every render, improving performance by preventing unnecessary re-renders of child components. Here's an example of how to use useCallback in a React application:

Suppose you have a parent component and a child component, and you want to pass a callback function from the parent to the child, memoizing it with useCallback to avoid unnecessary re-renders.

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

// Child component
const ChildComponent = ({ callback }) => {
  console.log('ChildComponent rendering');
  return (
    <div>
      <button onClick={callback}>Click me</button>
    </div>
  );
};

// Parent component
const ParentComponent = () => {
  const [count, setCount] = useState(0);

  // Define a callback function using useCallback
  const handleClick = useCallback(() => {
    console.log('Button clicked');
    setCount(count + 1);
  }, [count]);

  console.log('ParentComponent rendering');

  return (
    <div>
      <h1>Parent Component</h1>
      <p>Count: {count}</p>
      {/* Pass the memoized callback to the child component */}
      <ChildComponent callback={handleClick} />
    </div>
  );
};

export default ParentComponent;
```

In this example:

We have a ParentComponent that maintains a count state variable and defines a callback function handleClick. We use useCallback to memoize this callback, and we specify [count] as its dependency array. This means that the callback will only be recreated if the count value changes.

The ChildComponent receives the callback prop and renders a button that triggers the callback when clicked.

When you run this code, you'll notice that the ChildComponent does not re-render unnecessarily when the parent component re-renders. The handleClick function is only recreated if the count state changes, thanks to useCallback. This optimization can be particularly useful when you pass callbacks down to child components that rely on props or state from the parent.