React Hooks

useState

The fundamental of storing information. `useState` hook allow you to create local state returning you both the value that you can access as well as a setter to modify the value. The modification of the state will trigger a re-rendering of the component.

A simple example of using state would be just a counter with a button that you can click an increment on.

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

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

useEffect

`useEffect` replaces the old lifecycle method in react like `componentDidMount, componentDidUpdate, componentWillunmount`. You can provide a callback method to run whenever the condition triggers. The condition is specify through the second parameter which is called the dependency array.

- `[]` -> Whenever the component is mounted (The component has been loaded into the DOM)
- `no array` -> Run after every single render
- `[state]` -> Monitor a state so that whenever the state changes the method will be invoked

This is just a more extensive example of on how to use `useEffect` to trigger the update to `displayString` to append an extra pipe character whenever you increment by pressing the button. As you can see the button is not directly affecting the `displayString` but rather through the side

effects of the hook.

```
function Counter() {

    let [counter, setCounter] = useState(0);
    let [displayString, setDisplayString] = useState("|")

    function increment() {
        setCounter(counter + 1);
    }

    useEffect(() => {
        setDisplayString(displayString + "|")
    }, [counter])

    return (
        <>
            <p>{displayString}</p>
            <button onClick={increment}>Increment</button>
        </>
    )
}

export default function SidePage() {
    return (
        <div>
            <Counter />
        </div>
    )
}
```

One subtle thing if you have clean up functions is that:

```
  useEffect( () => {
      console.log('Effect is applied')
      //some logic
      return () => {
          console.log('cleaning up')
          //cleanup logic
      }
```

```
  })
  return (<>
      {console.log('rendering...')}
   </>)
```

Say you have a component that does the following, if the passed prop is changed then the component will be re-rendered. You might think the order goes like this:

1. 'cleaning up'
2. New prop is passed
3. 'rendering'
4. 'Effect is applied'

In reality that is not the case, in reality the ordering goes like this:

1. New prop is passed
2. 'rendering'
3. 'cleaning up'
4. 'Effect is applied'

> That is clean up function is ran **AFTER** the new rendering of the updated props but **BEFORE** the new effect is applied.
>
> https://stackoverflow.com/a/58099371 done for performance reason.

useContext

`useContext` solve one of the common problem that you might encounter when using React which is passing states around from the parent to deeply nested children.

Say the state is held by the highest parent and you would need to pass the state to one of the deeply nested children in order for it to display the value properly.

> In React the manner of passing props deeply down to one of the child component is called props drilling.

You can see from this example that the Parent had to pass the value as prop through `ChildA` and then from `ChildA` to `ChildB` in order to display the value. Wouldn't it be easier if `ChildB` was able to just access the `value` directly from the `Parent`? With `useContext` hook you can!

```
function Parent() {
    let [value, setValue] = useState("Display from me nested Child B");
```

```
    return (
        <ChildA display={value} />
    )
}

function ChildA({display}) {
    return (
        <>
            <p>Child A</p>
            <ChildB display={display} />
        </>
    )
}

function ChildB({display}) {
    return (
        <p>{display}</p>
    )
}

export default function SidePage() {
    return (
        <div>
            <Parent />
        </div>
    )
}
```

In order to leverage `useContext` you must first create the context through the `createContext` and then wrap your child component that would like to have access to those states through the context provider.

```
const ParentContext = createContext();

function Parent() {
    let [display, setDisplay] = useState("Display from me nested Child B");

    return (
        <ParentContext value={display}>
```

```
            <ChildA />
        </ParentContext>
    )
}

function ChildA() {
    return (
        <>
            <p>Child A</p>
            <ChildB />
        </>
    )
}

function ChildB() {
    let display = useContext(ParentContext);

    return (
        <p>{display}</p>
    )
}

export default function SidePage() {
    return (
        <div>
            <ReferenceTest />
        </div>
    )
}
```

With the `ParentContext.Provider` you set the value that you would like to set within context through the `value` attribute. In this case we are sharing the `display` state. Now once you wrap the component that you would like to have access to state, all of the children would be able to have access to the context by calling `useContext` hook.

As you can see within `ChildB` component we don't need the prop anymore, we can just access it through the context block.

> Context can also be nested, if a component is nested in nested context then it will be using the closest context provider for resolving the value.

useRef

`useRef` is the last of the four most common hook that you will be encountering in React. It is very similar to `useState` in that they both create a mutable variable that you can use within your component. When you create a reference you can access the value through the `current` value.

The difference between a state and reference is that, everytime a state is updated through the setter method it will trigger a re-rendering of the component, **however, with a reference when you make modification to the `current` value, it will not trigger a re-render of the component.**

An example of creating a reference would be:

```
function ReferenceTest() {
    let counter = useRef(0);

    function handleIncrement() {
        counter.current++;
        console.log(counter.current);
    }

    return (
        <div>
            <p>{counter.current}</p>
            <button onClick={handleIncrement}>Click me</button>
        </div>
    )
}

export default function SidePage() {
    return (
        <div>
            <ReferenceTest />
        </div>
    )
}
```

> You can see in this example, if you were to click on the button, the `counter.current` value displayed in the HTML page will not be incremented, but if you observe in the console output you will see that the value is indeed being incremented. This is because changing a

> reference will not trigger a re-rendering of the component.

Practical Usage

One of the common usage of `useRef` is to point to the real DOM element that's within the HTML and using that as reference you can call DOM methods on it such as focus or select, to focus user's input field to those elements.

```
function ReferenceTest() {
    let nodeReference = useRef(null);

    useEffect(() => {
        nodeReference.current.focus()
    }, [])

    return (
        <div>
            <input ref={nodeReference} defaultValue="Focus on me" />
        </div>
    )
}
```

This example component when render will focus the user's texting cursor to the input field when it is loaded. `nodeReference.current` in this case is pointing to the actual DOM `<input>` element that's within HTML and therefore that's why you're able to use `.focus` on it.

> The usage of reference comes around when you want to have a mutable variable to store something but you don't want the DOM to react to it. Like an ID or something. Reference also survives re-render as oppose to variables declare with let which is reset to whatever the default value is after re-render.

useMachine / createMachine

> Note this hook comes from the xstate library and is not part of the standard React library. If you want to use machine you would need to pull this library in first.

In a standard React app you would use states within your application, but as your application get complex, you end up with bunch of state `isLoading, isError, isSuccess, isWorking`. They could all be say true by accident because of poorly managed states. The XState library's State Machine aims to help avoid those issues.

Quick recap, XState's definition of machine (or state machine) is a model that basically can only be in one status (mode) at any given time. It can be transitioned to other mode by moving the state machine to another state.

In order to leverage XState's State Machine you would have to first define the specs of the machine which can be done like below:

```
import { createMachine, assign } from 'xstate';

export const toggleMachine = createMachine({
  id: 'toggle',
  initial: 'inactive',
  context: { count: 0 },
  states: {
    inactive: {
      on: { TOGGLE: { target: 'active', actions: assign({ count: ({ context }) =>
context.count + 1 }) } }
    },
    active: {
      on: { TOGGLE: { target: 'inactive' } }
    }
  }
});
```

There are couple interesting things.

1. `id`: This is just a unique string to identify the machine. This will become helpful when you have multiple machines running
2. `context`: This is where you will be storing any relevant information that you would want along with the different states. You don't necessary need to use context, it is just any information that you would to attach to the machine as it transition to different states. Method of access will be discussed later. **Data inside here are reactive meaning React will re-render the component used any of the value and the value changes**
3. `initial`: The initial starting point of the machine
4. `states`: This is heart of the object. It has information about the states that this machine can be under, as well as what actions to take as it assumes a specific state. For example, in the above code snippet. There are only two states available to the machine `inactive / active`. The `on` property describes the action that the machine will take as it becomes that particular state. When the state is inactive and it receives an event called `TOGGLE` then it will transition itself to `active` while incrementing the `count` state inside the

context object.

Using the Machine

Now that you have created the machine specs and you would like to use it below is an example of how:

```
import { useMachine } from '@xstate/react';
import { toggleMachine } from './toggleMachine';

function ToggleButton() {
  // useMachine creates + starts an actor for you automatically
  const [state, send, actor] = useMachine(toggleMachine);

  return (
    <button onClick={() => send({ type: 'TOGGLE' })}>
      {state.value === 'inactive' ? 'Turn on' : 'Turn off'}
    </button>
  );
}
```

Taking the same machine defined above, to leverage the machine you would need to invoke the `useMachine` hook and provide the machine. The hook returns `[state, send, actor]`.

The first argument is a snapshot of the machine to give you rich information about the current state of the machine. For example:

1. You can access the current state of the machine through `state.value`
2. `state.matches('x')` to check the state as it returns a boolean
3. `state.context.<>` to check the context data that's stored within the machine

The second argument is the function that you call in order to transition the state machine into different states.

```
send({ type: 'TOGGLE' }); // If you only want to send the event without any additional
information
send({ type: 'SUBMIT', values: formData }); // Sending along with the a payload which the
state machine can access
send('RESET'); // Shorthand if there are no payload needed
```

The state machine will check if within the current state, does it react to the given event, if it does then it will transition to it along with any other additional actions, if it cannot then it will not transition to the new state.

The third argument isn't often used and is called actor. It represents the running instance of the state machine. You would typically need to interact with this if you want to interact with the lower level APIs.

Overriding Initial Data

If you want to override the defaults of the defined machine specs you can also do so when you're creating the machine:

```
const [state, send] = useMachine(toggleMachine, {
  // These override machine defaults
  context: { count: 5 },
  actions,
  guards,
  services
});
```

You may notice that there are `actions, guards, services` being defined. These are important behavior that you can attach to a state machine to decide when something happens, what happens, and how long complex asynchronous things take respectively.

When you define it through the override object, they will be available to use as a string name when you define the states property.

Guards

This is the if-statement checks that decides whether or not the transition to the state is permitted or not.

Writing guards requires a pure function that returns true or false, without any side effects. If it returns false, then the state transition is skipped.

The method signature for each of the guard is below:

```
({ context, event }) => boolean
```

So for example:

```
createMachine(machine, {
  guards: {
    isAdult: ({context}) => context.age >= 18
  }
});
```

To use it within the states step you would just refer to it with the guard property:

```
{
  id: 'checkout',
  initial: 'cart',
  states: {
    cart: {
      on: {
        CHECKOUT: [
          { target: 'payment', guard: 'isAdult' }
        ]
      }
    }
  }
}
```

Actions

This runs a specific code when a transition happens. You're able to mutate the state within this handler and has the method signature:

```
({context, event, self }) => void
```

You can also just use the `assign` for context updates. Actions are ran **after** the transition has happened. You're also able to run an array of actions.

Services

Services handle asynchronous work, meaning you're able to use the `async / await` keyword, dealing with promises, callbacks will be done with services.

The machine will wait for the asynchronous work to complete and receives completion events.

Revision #6
Created 2025-11-22 03:28:00 UTC by Tamarine
Updated 2026-01-30 17:25:25 UTC by Tamarine