Working with complex data in useState

In this reading, you will learn how to use objects as state variables when using `useState`. You will also discover the proper way to only update specific properties, such as state objects and why this is done. This will be demonstrated by exploring what happens when changing the string data type to an object.

An example of holding state in an object and updating it based on user-generated events

When you need to hold state in an object and update it, initially, you might try something like this:

```
import { useState } from "react";

export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });
  console.log(greeting, setGreeting);

  function updateGreeting() {
    setGreeting({ greet: "Hello, World-Wide Web" });
  }

  return (
    <div>
      <h1>{greeting.greet}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

While this works, it's not the recommended way of working with state objects in React, this is because the state object usually has more than a single property, and it is costly to update the entire object just for the sake of updating only a small part of it.

The correct way to update the state object in React when using useState

The suggested approach for updating the state object in React when using `useState` is to copy the state object and then update the copy.

This usually involves using the spread operator (`...`).

Keeping this in mind, here's the updated code:

```
import { useState } from "react";

export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });
  console.log(greeting, setGreeting);

  function updateGreeting() {
    const newGreeting = {...greeting};
    newGreeting.greet = "Hello, World-Wide Web";
    setGreeting(newGreeting);
  }

  return (
    <div>
      <h1>{greeting.greet}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

Incorrect ways of trying to update the state object

To prove that a copy of the old state object is needed to update state, let's explore what happens when you try to update the old state object directly:

```
import { useState } from "react";

export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });
  console.log(greeting, setGreeting);

  function updateGreeting() {
    greeting = {greet: "Hello, World-Wide Web};
    setGreeting(greeting);
  }

  return (
    <div>
      <h1>{greeting.greet}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

The above code does not work because it has a `TypeError` hiding inside of it.

Specifically, the `TypeError` is: "Assignment to constant variable".

In other words, you cannot reassign a variable declared using const, such as in the case of the `useState` hook's array destructuring:

```
const [greeting, setGreeting] = useState({ greet: "Hello, World" });
```

Another approach you might attempt to use to work around the suggested way of updating state when working with a state object might be the following:

```
import { useState } from "react";

export default function App() {
  const [greeting, setGreeting] = useState({ greet: "Hello, World" });
  console.log(greeting, setGreeting);

  function updateGreeting() {
```

```
    greeting.greet = "Hello, World-Wide Web";
    setGreeting(greeting);
  }

  return (
    <div>
      <h1>{greeting.greet}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

The above code is problematic because it doesn't throw any errors; however, it also doesn't update the heading, so it is not working correctly. This means that, regardless of how many times you click the "Update greeting" button, it will still be "Hello, World".

To reiterate, the proper way of working with state when it's saved as an object is to:

1. Copy the old state object using the spread (`...`) operator and save it into a new variable and
2. Pass the new variable to the state-updating function

Updating the state object using arrow functions

Now, let's use a more complex object to update state.

The state object now has two properties: greet and location.

The intention of this update is to demonstrate what to do when only a specific property of the state object is changing, while keeping the remaining properties unchanged:

```
import { useState } from "react";

export default function App() {
  const [greeting, setGreeting] = useState(
    {
      greet: "Hello",
      place: "World"
    }
  );
  console.log(greeting, setGreeting);

  function updateGreeting() {
    setGreeting(prevState => {
      return {...prevState, place: "World-Wide Web"}
    });
  }

  return (
    <div>
      <h1>{greeting.greet}, {greeting.place}</h1>
      <button onClick={updateGreeting}>Update greeting</button>
    </div>
  );
}
```

The reason this works is because it uses the previous state, which is named `prevState`, and this is the previous value of the greeting variable. In other words, it makes a copy of the `prevState` object, and updates only the place property on the copied object. It then returns a brand-new object:

```
return {...prevState, place: "World-Wide Web"}
```

Everything is wrapped in curly braces so that this new object is built correctly, and it is returned from the call to `setGreeting`.

Conclusion

You have learned what happens when changing the string data type to an object, with examples of holding state in an object and updating it based on user-generated events. You also learned about correct and incorrect ways to update the state object in React when using `useState`, and about updating the state object using arrow functions.

Mark as completed

Like Dislike Report an issue

coach