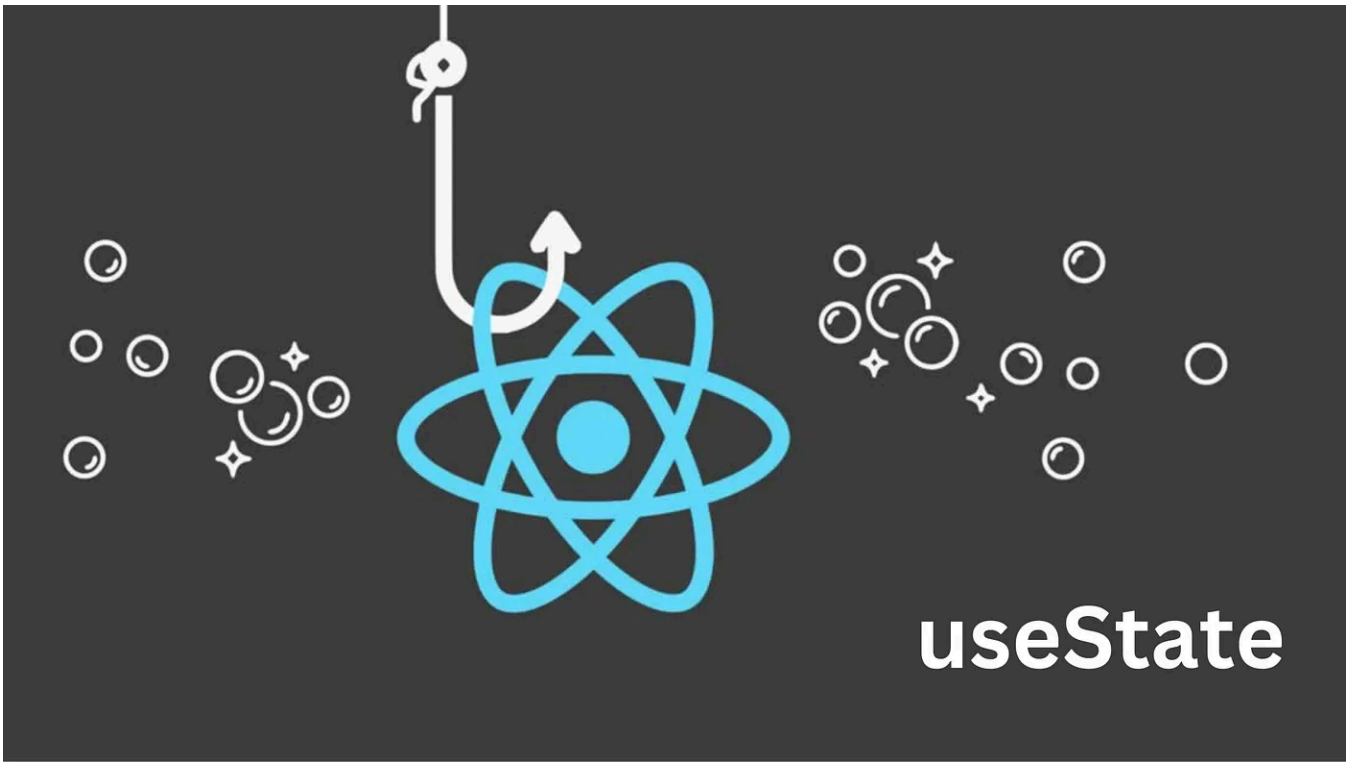
React Hooks: useState (With Practical Examples)

Tito Adeoye · Follow

7 min read · Feb 5, 2024



Hello there👋

This is the first article of the React Hooks Unboxed Series. In this article, we would be discussing the useState hook with practical examples.

An Introduction

The useState hook is used to manage state in React. State is simply data that can change over time. The useState hook lets us create a state variable, initialize it with data and also gives us access to a setter function that lets us update this state.

This is especially important in React which, just like the name suggests, is a library that is designed to efficiently update and **react** to changes in the user interface. The library efficiently updates the DOM when the application **state** changes, providing a reactive programming model.

The ability to update state is essential in the creation of modern, responsive web applications, and can be useful to execute the following operations:

1. **Dynamic UI Updates:** State allows React to monitor and re-render components and update the UI based on changes in data or user interactions.
2. **User Input Handling:** State is often used to manage user input, such as form data. When users interact with an application, the state can capture and reflect the current input values.
3. **Component Communication:** Parent components can pass down state as props to child components, enabling data flow and communication between components and synchronization.
4. **Asynchronous Operations:** State is essential for handling asynchronous operations, such as fetching data from an API. It helps to manage loading states, error states, and the display of fetched data.
5. **Conditional Rendering:** Components can use state to conditionally render different parts of the UI based on certain conditions. This way we can show or hide elements dynamically.

Create A State Variable

Here's how we can create a state variable using the useState hook.

```
const [state, setState] = useState(initialValue)
```

`state` is the state variable.

`setState` is the *setter function* that lets us update the state variable, triggering React to re-render the component.

`initialValue` is the value we use to initialize the state variable.

A simple example is if we want to monitor and display the name provided by a user.

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

export default function App() {
  const [name, setName] = useState("");

  return (
    <div className="App">
      <input
        type="text"
        placeholder="Enter name"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <p>Hello {name}</p>
    </div>
  );
}
```

In this example, we initialize the `name` state variable with an empty string, `""` , and monitor input changes using the event handler, `onChange` . Once a user updates the text in the input element, the *setter function*, `setName` , is run and the argument passed into it, i.e. the current value in the input element, `e.target.value` , is stored in the `name` state. Once the state has been updated, React re-renders the component and updates the UI, letting us display the updated name.

You can also pass a function instead of a variable as the initial value. This function is called the *initializer function* and it runs only once, at the first render of that component. It should be pure and and should return a value, which is used to initialize the variable.

You should pass the function and not call the function when initializing with the useState hook. Passing a function ensures it is only called on the first render of that component and consequently ignored.

```
function getName () {
    return 'Paul'
}

// wrong👎
const [name, setName] = useState(getName());

// right 👍
const [name, setName] = useState(getName);
```

Another way of updating state using the *setter function* is to pass a pure function into it instead of a value. This pure function has access to the current state which you can use as a parameter in your function to calculate the next state. This pure function that updates state is called an *updater function*.

```
const [name, setName] = useState('TITO');

setName(currentName => currentName.toLowerCase());
// console.log(name) => 'tito'
```

Here, our *setter function* accesses the current `name` , *'TITO'*, and updates it to lowercase, *'tito'*.

Do not try to set state in this function. The *updater function* must be pure and should take only the current state as an argument.

An Example

Now, we will create a login form to authenticate the users of our app. What do we need this login form to do:

- Collect email and password user input. We can collect user input using the form element.
- Monitor and store these values in state as they change e.g. when a user is entering input or changing it
- Submit this form using user provided data
- Display feedback after authentication

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

export default function App() {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [success, setSuccess] = useState(false);
  const [error, setError] = useState(false);

  function handleSubmit(e) {
    e.preventDefault();
```

```
      // submit email and password via post request for server authentication
      if (email === "dev@gmail.com" && password === "password") {
        // setTimeout is used to simulate server delay of 2s
        setTimeout(() => {
          // simulate message pop up
          setSuccess(true);
          setTimeout(() => setSuccess(false), 2000);
        }, 2000);
      } else {
        setTimeout(() => {
          // simulate message pop up
          setError(true);
          setTimeout(() => setError(false), 2000);
        }, 2000);
      }
    }

    return (
      <div className="App">
        <form onSubmit={handleSubmit}>
          <input
            type="email"
            placeholder="Enter email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
            required
          />
          <br />
          <input
            type="password"
            placeholder="Enter password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
            required
          />
          <br />
          <button type="submit">Log in</button>
        </form>
        <!-- display notification messages -->
        {success && <p>Welcome back!</p>}
        {error && <p>Are you a hacker or just forgetful? Try again😄 </p>}
      </div>
    );
  }
```

In this example, we have been able to show the power of useState as stated at the beginning of the article. We handled user input by monitoring changes and storing them in the email and password states, conditionally rendered UI (notification

messages) and dynamically updated the UI with a message based on the `success` and `error` states, and handled updated states asynchronously.

Typically, the `handleSubmit()` function makes an asynchronous POST API request to a server which is either successful or not depending on if the user submitted the right data. We can update the success and error states based on the server response.

```
// ...
try {
  // make POST API request
  // await response
  // if status 200, setSuccess(true)
  // if status 400, throw Error
} catch(err) {
  // catch error, setError(true)
}
```

What if we want to add interactivity for better UX? Typically, response after an API request may be delayed due to varying factors. So, we can display a loading message while our user waits. Our form can be in two different states in terms of submission, loading and not-loading. We can use the useState hook to manage this state!

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

export default function App() {
  // ...
  const [loading, setLoading] = useState(false);

  function handleSubmit(e) {
    e.preventDefault();

    // simulate loading
    setLoading(true);
    // submit email and password via post request for server authentication
    if (email === "dev@gmail.com" && password === "password") {
      setTimeout(() => {
        setSuccess(true);
        // stop loading after success
        setLoading(false);
        // simulate message pop up
        setTimeout(() => setSuccess(false), 2000);
      }, 2000);
    } else {
```

```
        setTimeout(() => {
          setError(true);
          // stop loading after error
          setLoading(false);
          // simulate message pop up
          setTimeout(() => setError(false), 2000);
        }, 2000);
      }
    }

    return (
      <div className="App">
        <form onSubmit={handleSubmit}>
          <!-- ... -->
          <button type="submit">{loading ? "Loading..." : "Log in"}</button>
        </form>
        <!-- ... -->
      </div>
    );
  }
```

Libraries like react-query help provide abstraction, managing and giving us access to our loading, error, success and response(data) states, so our code is not so verbose.

Updating arrays and objects

One thing that stumps beginners is updating array and object states. When managing array and object state, one should aim to **replace** with a new value instead of mutate the existing one. For instance, if we have a user state variable that stores the following properties; name, age and sex, we can update the `name` property by using the spread syntax(. . .) to copy existing data and then override the `name` property with the new value.

```
const [user, setUser] = useState({
    name: 'tito',
    age: 22,
    sex: female
});

// wrong👎
user.name = 'esther';

// right👍
setUser({...user, name: 'esther'})
```

One should also aim to treat state arrays as immutable and read-only, utilizing methods that return a copy of the array instead of mutating the existing one.

Examples of such non-mutating methods include filter, concat, slice and map.

```
const [names, setNames] = useState([]);

// wrong👎
names.push('john');

// right👍
setNames([...names, 'john'])  // adds 'john' to end of list
setNames(['john', ...names])  // adds 'john' to beginning of list
setNames(names.filter(name => name.startsWith('e')) ); // deletes 'esther'
```

The last line creates a new array of names that contain elements that **do not** start with an *'e'*, effectively deleting *'esther'* from the array.

Component Communication

State can be passed as props from a parent to a child component.

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

export default function App() {
  const [email, setEmail] = useState("");

  return (
    <div className="App">
      <button
        onClick={() => setEmail("dev@gmail.com") }
      >
        Show
      </button>
      <ChildComponent email={email} />
    </div>
  );
}

const ChildComponent = ({ email }) => {
  return <>{email && <p>This {email} was passed from my dad!</p>}</>;
};
```

The `email` state variable is passed from the parent component, `App.jsx`, to `ChildComponent.jsx` which destructures the props object to access `email`. When a user clicks the show button, you get:

React State Variables vs Regular Variables

You may wonder why we don't just use regular variables. We can, after all, declare and initialize a variable and also update/replace the value stored in it by reassignment.

```
// declare and initialize variable
var name = 'tito';
const [name, setName] = useState('tito');

// update name value
```

```
name = 'esther';
setName('esther')
```

These are primary reasons why this won't work:

- Changes in regular variables do not trigger React to initiate a render.
- Regular variables do not persist between renders. Therefore, if that component re-renders, 'esther' goes back to being 'tito'. React simply won't consider any changes to regular variables.

This is the problem the useState hook solves. It lets us persist data between renders and also initiate re-renders with the newly updated state data.

Recap

In this article we have discussed what the useState hook is, its use cases and why it is important and preferable to regular variables. We've also covered examples to properly illustrate how to use it.

Please leave a like if this was helpful. Comment also if you have questions.

Happy Coding🙌

JavaScript React Usestate React Hook

Follow

Written by Tito Adeoye

2 Followers

Hello World! I am Tito, a full-stack software engineer, obsessed with building great products. Currently, I am specializing in building products for the web.

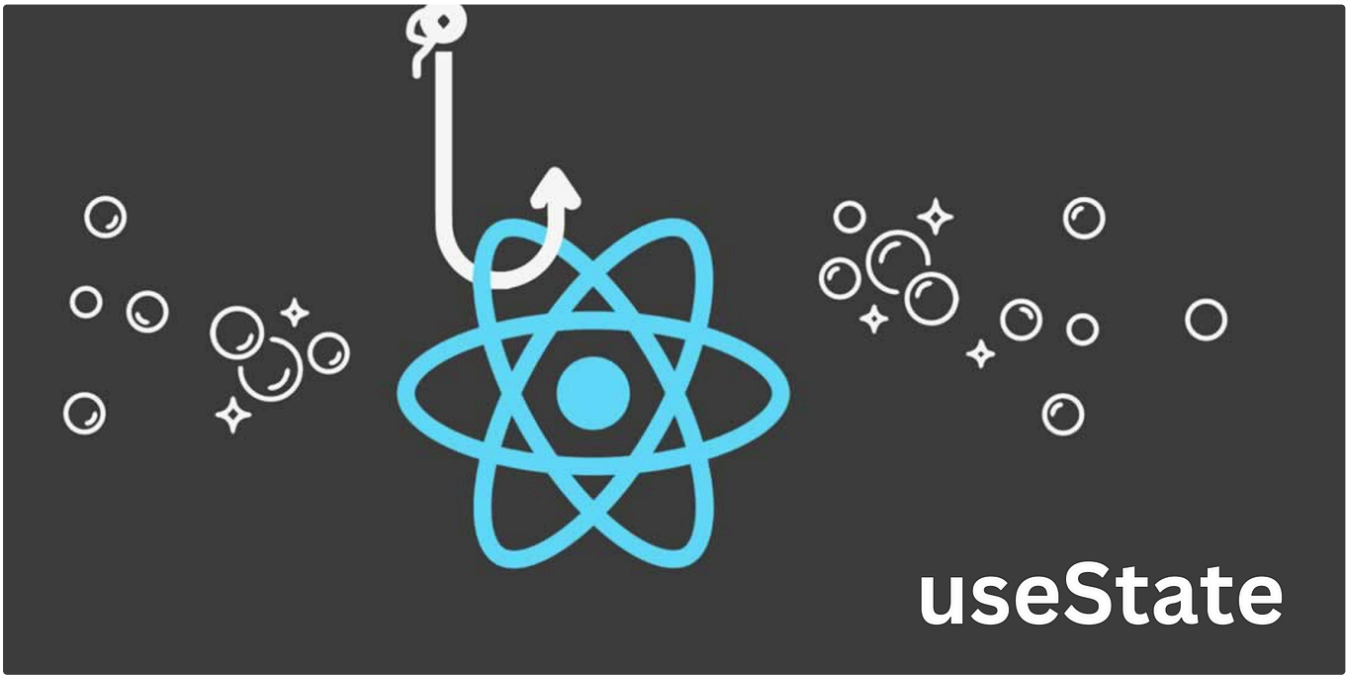
More from Tito Adeoye

Tito Adeoye

The DOM: Light & Shadow

Hello there👋

Feb 8 36

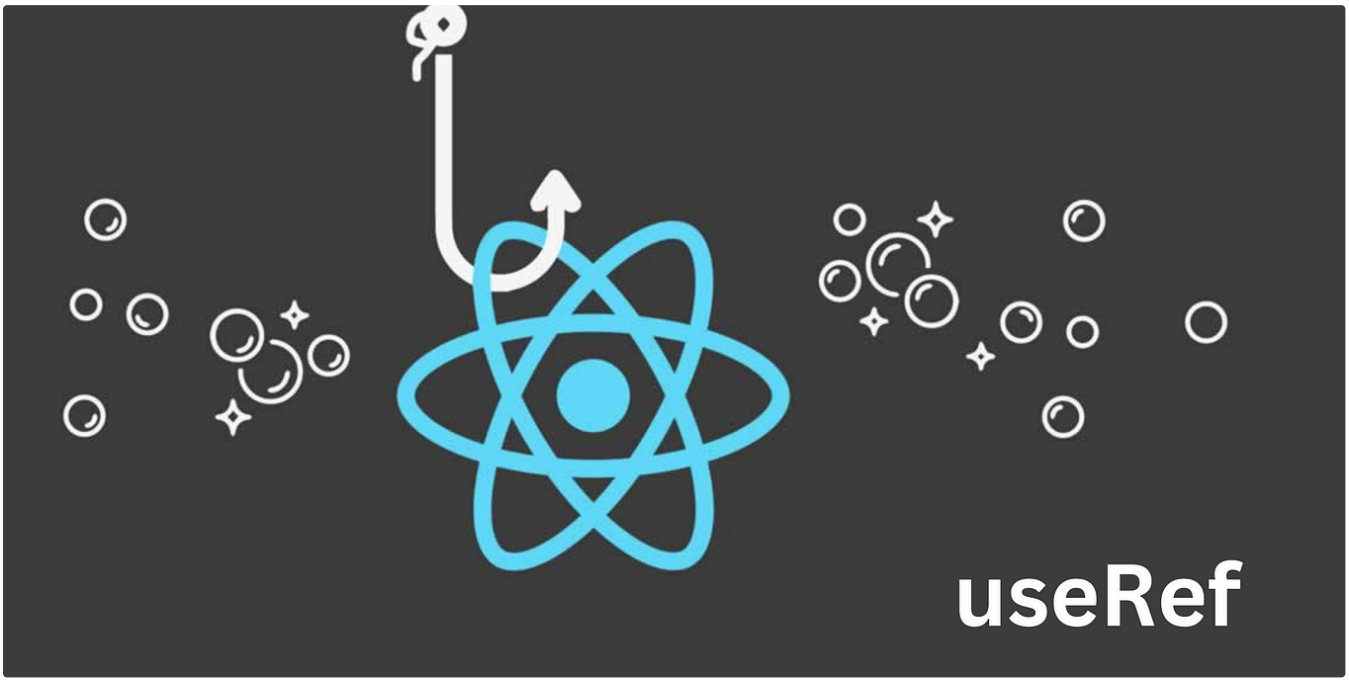


Tito Adeoye

State Behaves Like a Snapshot 📸

Ever tried to access a state variable in React just after updating it and wondered why you still get the previous value? Yes? Well, by the...

Feb 18 👏 1

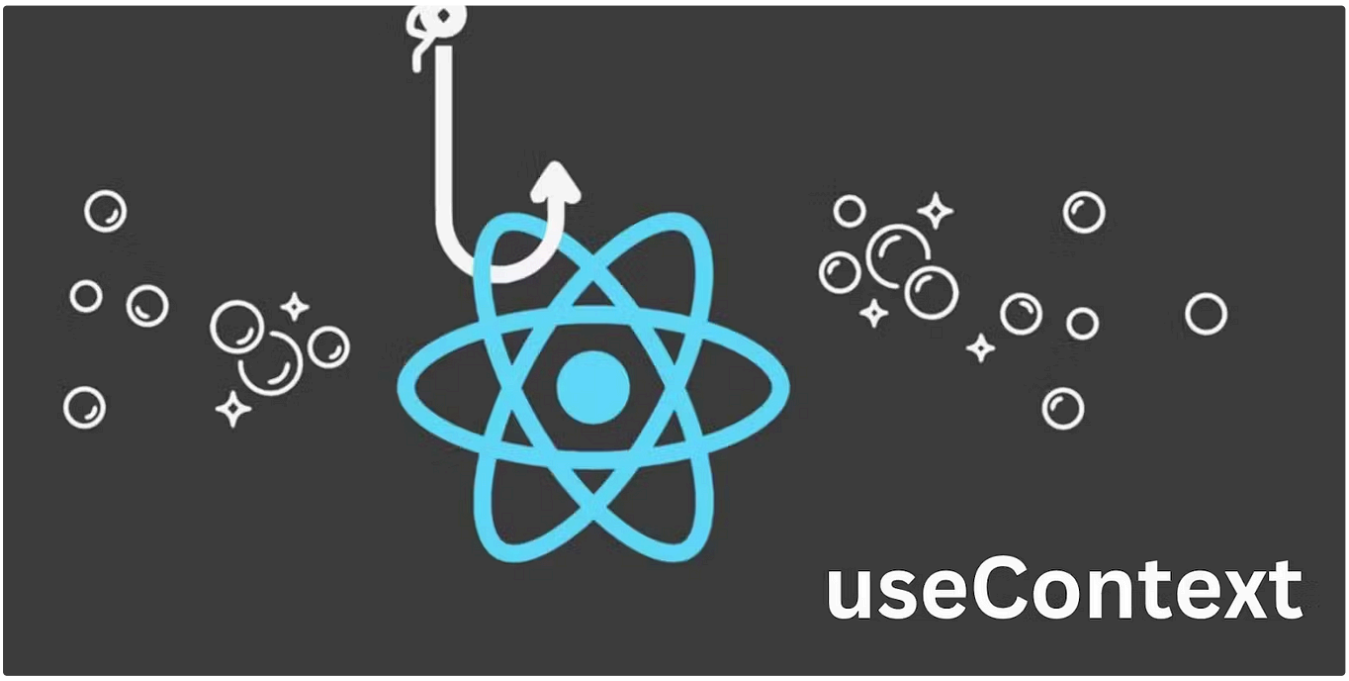


Tito Adeoye

React Hooks: useRef (With Practical Examples!)

Hello there👋

Feb 5 👏 5



Tito Adeoye

React Hooks: useContext

Hello again👋

Jun 17 1

See all from Tito Adeoye

Recommended from Medium

amandeep kumar

useCallback hook in react

What is useCallback?

Feb 17 17 4

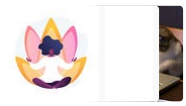
Sviat Kuzhelev in Level Up Coding

Why Using React Context for State Management Could Be a Big Mistake | 5 Core Examples

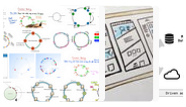
When diving into state management in React, React Context often appears as a straightforward and attractive option.

3d ago 1K 2

Lists



Stories to Help You Grow as a Software Developer
19 stories · 1232 saves



General Coding Knowledge
20 stories · 1430 saves

Medium's Huge List of Publications Accepting Submissions
334 stories · 3171 saves

Generative AI Recommended Reading
52 stories · 1238 saves

Afan Khan in JavaScript in Plain English

Microsoft is ditching React

Here's why Microsoft considers React a mistake for Edge.

Jun 6 2.9K 66

asierr.dev

How to Write Cleaner JavaScript Code: Tips, Tricks, and Best Practices

We all know that clean code is the foundation of maintainable and scalable applications. Writing clean, readable, and maintainable...

3d ago 56

Sam Ho

Zustand: How is it better than React Context (with an example)

As a web engineer, you may be in a situation where you need to manage the state of your application. When comes to state management, there...

Apr 13 61 1

Louis Trinh

React useDebounce with typescript

The useDebounce hook is a custom React hook that helps to prevent excessive re-renders or function calls by introducing a delay between...

Feb 1 7