LogRocket
Frontend Analytics

useState in React: A complete guide

December 4, 2020 · 12 min read

`useState` is a Hook that allows you to have state variables in functional components. You pass the initial state to this function and it returns a variable with the current state value (not necessarily the initial state) and another function to update this value.

This tutorial serves as a complete guide to the `useState` Hook in React, the equivalent of `this.state` / `this.setSate` for functional components. We'll cover the following in detail:

- Class and functional components in React
- What does the React.useState hook do?
- Declaring state in React
- React Hooks: Update state
- Using an object as a state variable with useState hook
- How to update state in a nested object in React with Hooks
- Multiple state variables or one state object
- Rules for using useState
- The useReducer Hook

If you're just getting started with React Hooks and looking for a visual guide, check out the video tutorial below.

Class and functional components in React

There are two types of components in React: class and functional components.

Class components are ES6 classes that extend from React.Component and can have state and lifecycle methods:

```
class Message extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      message: ‘’
    };
  }

  componentDidMount() {
    /* ... */
  }

  render() {
    return <div>{this.state.message}</div>;
  }
}
```

Functional components are functions that just accept arguments as the properties of the component and return valid JSX:

```
function Message(props) {
  return <div>{props.message}</div>
}
// Or as an arrow function
const Message = (props) =>  <div>{props.message}</div>
```

As you can see, there are no state or lifecycle methods. However, as of React 16.8, we can use Hooks.

React Hooks are functions that add state variables to functional components and instrument the lifecycle methods of classes. They tend to start with `use`.

What does the `React.useState` Hook do?

As stated previously, `useState` enables you to add state to function components. Calling `React.useState` inside a function component generates a single piece of state associated with that component.

Whereas the state in a class is always an object, with Hooks, the state can be any type. Each piece of state holds a single value, which can be an object, an array, a boolean, or any other type you can imagine.

So when should you use the `useState` Hook? It's especially useful for local component state, but larger projects might require additional state management solutions.

Declaring state in React

`useState` is a named export from `react`. To use it, you can write:

```
React.useState
```

Or to import it just write `useState` :

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

But unlike the state object that you can declare in a class, which allows you to declare more than one state variable, like this:

```
import React from 'react';

class Message extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      message: '',
      list: [],
    };
  }
  /* ... */
}
```

The `useState` Hook allows you to declare only one state variable (of any type) at a time, like this:

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

const Message= () => {
   const messageState = useState( '' );
   const listState = useState( [] );
}
```

`useState` takes the initial value of the state variable as an argument.

More great articles from LogRocket:

You can pass it directly, as shown in the previous example, or use a function to lazily initialize the variable (useful when the initial state is the result of an expensive computation):

```
const Message= () => {
   const messageState = useState( () => expensiveComputation() );
   /* ... */
}
```

The initial value will be assigned only on the initial render (if it's a function, it will be executed only on the initial render).

In subsequent renders (due to a change of state in the component or a parent component), the argument of the `useState` Hook will be ignored and the current value will be retrieved.

It is important to keep this in mind because, for example, if you want to update the state based on the new properties the component receives:

```
const Message= (props) => {
   const messageState = useState( props.message );
   /* ... */
}
```

Using `useState` alone won't work because its argument is used the first time only — not every time the property changes (look here for the right way to do this).

But `useState` doesn't return just a variable as the previous examples imply.

It returns an array, where the first element is the state variable and the second element is a function to update the value of the variable:

```
const Message= () => {
   const messageState = useState( '' );
   const message = messageState[0]; // Contains ''
   const setMessage = messageState[1]; // It’s a function
}
```

Usually, you’ll use array destructuring to simplify the code shown above:

```
const Message= () => {
   const [message, setMessage]= useState( '' );
}
```

This way, you can use the state variable in the functional component like any other variable:

```
const Message = () => {
  const [message, setMessage] = useState( '' );

  return (
    <p>
      <strong>{message}</strong>
    </p>
  );
};
```

But why does `useState` return array?

Because compared to an object, an array is more flexible and easy to use.

If the method returned an object with a fixed set of properties, you wouldn’t be able to assign custom names in an easy way.

You’d have to do something like this (assuming the properties of the object are `state` and `setState`):

```
// Without using object destructuring
const messageState = useState( '' );
const message = messageState.state;
const setMessage = messageState

// Using object destructuring
const { state: message, setState: setMessage } = useState( '' );
const { state: list, setState: setList } = useState( [] );
```

React Hooks: Update state

The second element returned by `useState` is a function that takes a new value to update the state variable.

Here’s an example that uses a text box to update the state variable on every change:

```
const Message = () => {
  const [message, setMessage] = useState( '' );

  return (
    <div>
      <input
         type="text"
         value={message}
         placeholder="Enter a message"
         onChange={e => setMessage(e.target.value)}
       />
      <p>
        <strong>{message}</strong>
      </p>
    </div>
  );
};
```

Try it here.

However, this update function doesn’t update the value right away.

Rather, it enqueues the update operation. Then, after re-rendering the component, the argument of `useState` will be ignored and this function will return the most recent value.
If you use the previous value to update state, you must pass a function that receives the previous value and returns the new value:

```
const Message = () => {
  const [message, setMessage] = useState( '' );

  return (
    <div>
      <input
        type="text"
        value={message}
        placeholder="Enter some letters"
        onChange={e => {
          const val = e.target.value;
          setMessage(prev => prev + val)
        } }
      />
      <p>
        <strong>{message}</strong>
      </p>
    </div>
  );
```

Try it here.

Using an object as a state variable with `useState` hook

There are two things you need to keep in mind about updates when using objects:

- The importance of immutability
- And the fact that the setter returned by `useState` doesn't merge objects like `setState()` does in class components

About the first point, if you use the same value as the current state to update the state (React uses Object.is for comparing), React won't trigger a re-render.

When working with objects, it’s easy to make the following mistake:

```
const Message = () => {
  const [messageObj, setMessage] = useState({ message: '' });

  return (
    <div>
      <input
        type="text"
        value={messageObj.message}
        placeholder="Enter a message"
        onChange={e => {
          messageObj.message = e.target.value;
          setMessage(messageObj); // Doesn't work
        }}
      />
      <p>
        <strong>{messageObj.message}</strong>
      </p>
  </div>
  );
```

Try it here.

Instead of creating a new object, the above example mutates the existing state object. To React, that’s the same object.

To make it work, we must create a new object:

```
onChange={e => {
  const newMessageObj = { message: e.target.value };
  setMessage(newMessageObj); // Now it works
}}
```

This leads us to the second important thing you need to remember.

When you update a state variable, unlike `this.setState` in a class component, the function returned by `useState` does not automatically merge update objects, it replaces them.

Following the previous example, if we add another property to the message object (`id`):

```
const Message = () => {
  const [messageObj, setMessage] = useState({ message: '', id: 1 });

  return (
    <div>
      <input
        type="text"
        value={messageObj.message}
        placeholder="Enter a message"
        onChange={e => {
          const newMessageObj = { message: e.target.value };
          setMessage(newMessageObj);
        }}
      />
      <p>
        <strong>{messageObj.id} : {messageObj.message}</strong>
      </p>
  </div>
  );
```

And we only update the `message` property like in the above example, React will replace the original state object:

```
{ message: '', id: 1 }
```

With the object used in the `onChange` event, which only contains the `message` property:

```
{ message: 'message entered' } // id property is lost
```

Try it here, you'll see how the `id` property is lost.

You can replicate the behavior of `setState()` by using the function argument that contains the object to be replaced and the object spread syntax:

```
onChange={e => {
  const val = e.target.value;
  setMessage(prevState => {
    return { ...prevState, message: val }
  });
}}
```

The `...prevState` part will get all of the properties of the object and the `message: val` part will overwrite the `message` property.

This will have the same result as using Object.assign (just remember to create a new object):

```
onChange={e => {
  const val = e.target.value;
  setMessage(prevState => {
    return Object.assign({}, prevState, { message: val });
  });
}}
```

Try it here.

However, the spread syntax simplifies this operation and it also works with arrays.

Basically, when applied to an array, the spread syntax removes the brackets so you can create another one with the values of the original array:

```
[
  ...['a', 'b', 'c'],
  'd'
]
// Is equivalent to
[
  'a', 'b', 'c',
  'd'
]
```

Here's an example that shows how to use `useState` with arrays:

```
const MessageList = () => {
  const [message, setMessage] = useState("");
  const [messageList, setMessageList] = useState([]);

  return (
    <div>
      <input
        type="text"
        value={message}
        placeholder="Enter a message"
        onChange={e => {
          setMessage(e.target.value);
        }}
      />
      <input
        type="button"
        value="Add"
        onClick={e => {
          setMessageList([
```

Try it here.

You just have to be careful when applying the spread syntax to multidimensional arrays because it won't work as you might expect.

This leads us to another thing to consider when working with objects as the state.

How to update state in a nested object in React with Hooks

In JavaScript, multidimensional arrays are arrays within arrays:

```
[
  ['value1','value2'],
  ['value3','value4']
]
```

You could use them to group all your state variables in one place. However, for that purpose, it would be better to use nested objects:

```
{
  'row1' : {
    'key1' : 'value1',
    'key2' : 'value2'
  },
  'row2' : {
    'key3' : 'value3',
    'key4' : 'value4'
  }
}
```

But the problem when working with multidimensional arrays and nested objects is that `Object.assign` and the spread syntax will create a shallow copy instead of a deep copy.

From the spread syntax documentation:

> *Spread syntax effectively goes one level deep while copying an array. Therefore, it may be unsuitable for copying multidimensional arrays, as the following example shows. (The same is true with `Object.assign()` and spread syntax.)*

```
let a = [[1], [2], [3]];
let b = [...a];

b.shift().shift(); //  1
//  Array 'a' is affected as well: [[], [2], [3]]
```

This StackOverflow query offers good explanations for the above example, but the important point is that when using nested objects, we can't just use the spread syntax to update the state object.

For example, consider the following state object:

```
const [messageObj, setMessage] = useState({
  author: '',
  message: {
    id: 1,
    text: ''
  }
});
```

The following code snippets show some incorrect ways to update the `text` field:

```
// Wrong
setMessage(prevState => ({
  ...prevState,
  text: 'My message'
}));

// Wrong
setMessage(prevState => ({
  ...prevState.message,
  text: 'My message'
}));

// Wrong
setMessage(prevState => ({
  ...prevState,
  message: {
    text: 'My message'
  }
}));
```

To properly update the `text` field, we have to copy to a new object the entire set of fields/nested objects of the original object:

```
// Correct
setMessage(prevState => ({
  ...prevState,          // copy all other field/objects
  message: {             // recreate the object that contains the field to
update
    ...prevState.message, // copy all the fields of the object
    text: 'My message'    // overwrite the value of the field to update
  }
}));
```

In the same way, here's how you'd update the `author` field of the state object:

```
// Correct
setMessage(prevState => ({
  author: 'Joe',          // overwrite the value of the field to update
  ...prevState.message    // copy all other field/objects
}));
```

Assuming the `message` object doesn't change. If it does change, you'd have to update the object this way:

```
// Correct
setMessage(prevState => ({
  author: 'Joe',          // update the value of the field
  message: {              // recreate the object that contains the field to
update
    ...prevState.message, // copy all the fields of the object
    text: 'My message'    // overwrite the value of the field to update
  }
}));
```

Multiple state variables or one state object

When working with multiple fields or values as the state of your application, you have the option of organizing the state using multiple state variables:

```
const [id, setId] = useState(-1);
const [message, setMessage] = useState('');
const [author, setAuthor] = useState('');
```

Or an object state variable:

```
const [messageObj, setMessage] = useState({
  id: 1,
  message: '',
  author: ''
});
```

However, you have to be careful when using state objects with a complex structure (nested objects). Consider this example:

```
const [messageObj, setMessage] = useState({
  input: {
    author: {
      id: -1,
      author: {
        fName:'',
        lName: ''
      }
    },
    message: {
      id: -1,
      text: '',
      date: now()
    }
  }
});
```

If you have to update a specific field nested deep in the object, you'll have to copy all the other objects along with the key-value pairs of the object that contains that specific field:

```
setMessage(prevState => ({
  input: {
    ...prevState.input,
    message: {
      ...prevState.input.message,
      text: 'My message'
    }
  }
}));
```

In some cases, cloning deeply nested objects can be expensive because React may re-render parts of your applications that depend on fields that haven't even changed.

For this reason, the first thing you need to consider is trying to flatten your state object(s). In particular, the React documentation recommends splitting the state into multiple state variables based on which values tend to change together.

If this is not possible, the recommendation is to use libraries that help you work with immutable objects, such as immutable.js or immer.

Track state and user interaction with components

It's important to validate that everything works in your production React app as expected. If you're interested in monitoring and tracking issues related to components AND seeing how users interact with specific components, try LogRocket. https://logrocket.com/signup/

LogRocket is like a DVR for web apps, recording literally everything that happens on your site. The LogRocket React plugin allows you to search for user sessions where the user clicks a specific component in your app. With LogRocket you can understand how users interact with components, and surface any errors related to components not rendering.

In addition, LogRocket logs all actions and state from your Redux stores. LogRocket instruments your app to record requests/responses with headers + bodies. It also records the HTML and CSS on the page, recreating pixel-perfect videos of even the most complex single-page apps.

Modernize how you debug your React apps — Start monitoring for free.

Rules for using `useState`

`useState` abides by the same rules that all Hooks follow:

- Only call Hooks at the top level
- Only call Hooks from React functions

The second rule is easy to follow. Don't use `useState` in a class component:

```
class App extends React.Component {
  render() {
    const [message, setMessage] = useState( '' );

    return (
      <p>
        <strong>{message}</strong>
      </p>
    );
  }
}
```

Or regular JavaScript functions (not called inside a functional component):

```
function getState() {
  const messageState = useState( '' );
  return messageState;
}
const [message, setMessage] = getState();
const Message = () => {
 /* ... */
}
```

You'll get an error.

The first rule means that, even inside functional components, you shouldn't call `useState` in loops, conditions, or nested functions because React relies on the order in which `useState` functions are called to get the correct value for a particular state variable.

In that regard, the most common mistake is to wrap `useState` calls in a conditional statement (they won't be executed all the time):

```
if (condition) { // Sometimes it will be executed, making the order of the
useState calls change
  const [message, setMessage] = useState( '' );
  setMessage( aMessage );
}
const [list, setList] = useState( [] );
setList( [1, 2, 3] );
```

A functional component can have many calls to `useState` or other Hooks. Each Hook is stored in a list, and there's a variable that keeps track of the currently executed Hook.

When `useState` is executed, the state of the current Hook is read (or initialized during the first render), and then, the variable is changed to point to the next Hook.

That's why it is important to always maintain the Hook calls in the same order, otherwise, a value belonging to another state variable could be returned.

In general terms, here's an example of how this works step by step:

1. React initializes the list of Hooks and the variable that keeps track of the current Hook
2. React calls your component for the first time
3. React finds a call to `useState` , creates a new Hook object (with the initial state), changes the current Hook variable to point to this object, adds the object to the Hooks list, and return the array with the initial state and the function to update it
4. React finds another call to `useState` and repeats the actions of the previous step, storing a new Hook object and changing the current Hook variable
5. The component state changes
6. React sends the state update operation (performed by the function returned by `useState`) to a queue to be processed
7. React determines it needs to re-render the component
8. React resets the current Hook variable and calls your component
9. React finds a call to `useState` , but this time, since there's already a Hook at the first position of the list of Hooks, it just changes the current Hook variable and returns the array with the current state and the function to update it

10. React finds another call to `useState` and since a Hook exists in the second position, once again, it just changes the current Hook variable and returns the array with the current state and the function to update it

If you like to read code, `ReactFiberHooks` is the class where you can learn how Hooks work under the hood.

The `useReducer` Hook

For advanced use cases, you can use the `useReducer` Hook as an alternative to `useState`. This is especially useful when you have complex state logic that uses multiple sub-values or when a state depends on the previous one.

Here's how to use the `useReducer` Hook:

```
const [state, dispatch] = useReducer(reducer, initialArgument, init)
useReducer returns an array that holds the current state value and a dispatch
method. This should be familiar territory if you have experience using Redux.
```

Whereas, with `useState`, you invoke the state updater function to update state, with `useReducer`, you invoke the `dispatch` function and pass it an action — i.e., an object with at least a `type` property.

```
dispatch({type: 'increase'})
```

Conventionally, an action object may also have a `payload`, e.g., `{action: 'increase', payload: 10}`.

While it's not absolutely necessary to pass an action object that follows this pattern, it's a very common pattern popularized by Redux.

Conclusion

`useState` is a Hook (function) that allows you to have state variables in functional components. You pass the initial state to this function and it returns a variable with the current state value (not necessarily the initial state) and another function to update this value.

Here are some key important points to remember:

- The update function doesn't update the value right away
- If you use the previous value to update state, you must pass a function that receives the previous value and returns an updated value, for example, `setMessage(previousVal => previousVal + currentVal)`
- If you use the same value as the current state (React uses the Object.is for comparing) to update the state, React won't trigger a re-render
- Unlike `this.setState` in class components, `useState` doesn't merge objects when the state is updated. It replaces them
- `useState` follows the same rules that all Hooks do. In particular, pay attention to the order in which these functions are called (there's an ESLint plugin that will help you enforce these rules)

Esteban Herrera Follow

Family man. Java and JavaScript developer. Swift and VR/AR hobbyist. Like books, movies, and still trying many things. Find me at eherrera.net

#react

12 Replies to "useState in React: A complete guide"

Alan Choi Says:
August 13, 2019 at 9:38 pm

Reply

Thanks for a great article. By the way, I don't think this part of the article true.

> If you use the same value as the current state (React uses the Object.is for comparing) to update the state, React won't trigger a re-render

https://codesandbox.io/embed/react-hooks-playground-cfhle
As you can check in this CodeSandBox, even when I call the state updater with the same value as the current state, it still re-renders the component.

Esteban Herrera Says:
August 16, 2019 at 2:31 pm

Reply

Hi Alan, you're right, it's not completely true. From the documentation:
> If you update a State Hook to the same value as the current state, React will bail out without rendering the children or firing effects. (React uses the Object.is comparison algorithm.)
> Note that React may still need to render that specific component again before bailing out. That shouldn't be a concern because React won't unnecessarily go 'deeper' into the tree.

So it doesn't always skip the rendering. I'll update this part of the article. Thank you so much!

SpidaFly Says:
August 30, 2019 at 4:34 am

Reply

This is likely a very elementary, basic JS question, but I'm learning.

Why did you pass the anonymous function that returns the result of expensiveComputation into useState… instead of just passing the result of expensiveComputation itself?

In this line:
const messageState = useState(() => expensiveComputation());

Thank you. Once again I know this isn't a react question but a JS question. Thank you for the clarification. My newbie skills will thank you!

Cefn Hoile Says:
September 2, 2019 at 8:40 am

Reply

I can see why a call to useState() shouldn't be mixed into branching code (ifs, loops etc), because the identity of the state is dictated by the order of useState() calls within the stateless function call.

However, I can't see why having calls to the setMessage function in branching code below would be a problem (as per your example code quoted below).

That's because once the setMessage reference has been created, the association between the function and the specific state hook has been established, and any call to setMessage or setList will manipulate the correct value.

In fact I don't think hooks could work at all if the order of state hook __setting__ calls had to be predictable, so I don't think there's anything wrong with the code example.

By contrast if the useState() call was conditionally called (e.g. based on props), then any state established by a later useState call would have an unpredictable identity on the second render.

const [message, setMessage] = useState(”);
const [list, setList] = useState([]);
if (condition) {
setMessage(aMessage); // Sometimes it will be executed, making the order change
}
setList([1, 2, 3]);

Esteban Herrera Says:
September 23, 2019 at 12:33 pm

Reply

Hi Cefn. Sorry for the late response, you are right, that was a bad example, I have updated the sample code to place the call to useState inside the if block. Thank you so much!

Esteban Herrera Says:
September 23, 2019 at 12:40 pm

Reply

Hi SpidaFly, sorry for the late response, and no problem at all answering your question. If we have an expression like this:
let result = expensiveComputation();

It will be evaluated immediately, blocking the execution of the code until the method returns.

On the other hand, something like this:
let result = () => expensiveComputation();

Returns a function that is not executed when the line is evaluated, so the execution of the code is not blocked at that particular time. The function is executed only when you can it (result()), if ever. That’s the benefit of laziness (https://en.wikipedia.org/wiki/Lazy_evaluation).

Hope this answers your question!

Konstantin Meiklyar Says:
October 22, 2019 at 7:06 am

Reply

Hi
There is a typo in the name of useState function in the following line:

“This article is a guide to the useSate (state) hook, the equivalent of this.state/this.setSate for functional components.|

Disillusia Says:
November 3, 2019 at 8:31 am

Reply

Hey Alan,

It seems that the initial statement made in the article is true in v16.8.0 and above. I changed the react version in your CodeSandBox and true enough, React doesn't seem to trigger a re-render if the value is the same as the current state.

Trevor Holmes Says:
April 18, 2020 at 12:18 pm

Reply

This article should be the offical documentation of the useState hook. Well done.

Diwakar Pant Says:
April 28, 2020 at 1:02 pm

Reply

{
const val = e.target.value;
setMessage(prev => prev + val)
} }
/>

In this, I do not seem to understand how "prev" refers to the ACTUAL PREVIOUS value of input ??

Gabor Says:
February 8, 2021 at 2:13 pm

Reply

Thanks for the great article!
I have a question about the 'here's how you'd update the author field of the state object' section.
Should this not be
setMessageObj((prevState) => ({
...prevState, // copy all other field/objects
author: "Joe" // overwrite the value of the field to update
}))

As far as I understant, if the message child object has not changed, we can use the spread operator to copy all values as-is, then we just overwrite the author field on the top level of the state object.

AGuy Says:
September 8, 2021 at 10:39 am

Reply

Very helpful article. thank you very much👍🏽

Leave a Reply

Enter your comment here...