Rules of Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

Hooks are JavaScript functions, but you need to follow two rules when using them. We provide a linter plugin to enforce these rules automatically:

Only Call Hooks at the Top Level

Don't call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function. By following this rule, you ensure that Hooks are called in the same order each time a component renders. That's what allows React to correctly preserve the state of Hooks between multiple `useState` and `useEffect` calls. (If you're curious, we'll explain this in depth below.)

Only Call Hooks from React Functions

Don't call Hooks from regular JavaScript functions. Instead, you can:

- ✅ Call Hooks from React function components.
- ✅ Call Hooks from custom Hooks (we'll learn about them on the next page).

By following this rule, you ensure that all stateful logic in a component is clearly visible from its source code.

ESLint Plugin

We released an ESLint plugin called `eslint-plugin-react-hooks` that enforces these two rules. You can add this plugin to your project if you'd like to try it:

This plugin is included by default in Create React App.

```
npm install eslint-plugin-react-hooks --save-dev
```

```
// Your ESLint configuration
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Checks rules of Hooks
    "react-hooks/exhaustive-deps": "warn" // Checks effect dependencies
  }
}
```

You can skip to the next page explaining how to write your own Hooks now. On this page, we'll continue by explaining the reasoning behind these rules.

Explanation

As we learned earlier, we can use multiple State or Effect Hooks in a single component:

```
function Form() {
  // 1. Use the name state variable
  const [name, setName] = useState('Mary');

  // 2. Use an effect for persisting the form
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. Use the surname state variable
  const [surname, setSurname] = useState('Poppins');

  // 4. Use an effect for updating the title
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}
```

So how does React know which state corresponds to which `useState` call? The answer is that **React relies on the order in which Hooks are called**. Our example works because the order of the Hook calls is the same on every render:

```
// ------------
// First render
// ------------
useState('Mary')           // 1. Initialize the name state variable with 'Mary'
useEffect(persistForm)     // 2. Add an effect for persisting the form
useState('Poppins')        // 3. Initialize the surname state variable with 'Poppins'
useEffect(updateTitle)     // 4. Add an effect for updating the title

// -------------
// Second render
// -------------
useState('Mary')           // 1. Read the name state variable (argument is ignored)
useEffect(persistForm)     // 2. Replace the effect for persisting the form
useState('Poppins')        // 3. Read the surname state variable (argument is ignored)
useEffect(updateTitle)     // 4. Replace the effect for updating the title

// ...
```

As long as the order of the Hook calls is the same between renders, React can associate some local state with each of them. But what happens if we put a Hook call (for example, the `persistForm` effect) inside a condition?

```
  // 🔴 We're breaking the first rule by using a Hook in a condition
  if (name !== '') {
    useEffect(function persistForm() {
      localStorage.setItem('formData', name);
    });
  }
```

The `name !== ''` condition is `true` on the first render, so we run this Hook. However, on the next render the user might clear the form, making the condition `false`. Now that we skip this Hook during rendering, the order of the Hook calls becomes different:

```
useState('Mary')           // 1. Read the name state variable (argument is ignored)
// useEffect(persistForm)  // 🔴 This Hook was skipped!
useState('Poppins')        // 🔴 2 (but was 3). Fail to read the surname state variable
useEffect(updateTitle)     // 🔴 3 (but was 4). Fail to replace the effect
```

React wouldn't know what to return for the second `useState` Hook call. React expected that the second Hook call in this component corresponds to the `persistForm` effect, just like during the previous render, but it doesn't anymore. From that point, every next Hook call after the one we skipped would also shift by one, leading to bugs.

This is why Hooks must be called on the top level of our components. If we want to run an effect conditionally, we can put that condition *inside* our Hook:

```
useEffect(function persistForm() {
  // 👍 We're not breaking the first rule anymore
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

Note that you don't need to worry about this problem if you use the provided lint rule. But now you also know *why* Hooks work this way, and which issues the rule is preventing.

Next Steps

Finally, we're ready to learn about writing your own Hooks! Custom Hooks let you combine Hooks provided by React into your own abstractions, and reuse common stateful logic between different components.

Edit this page

Previous article
Using the Effect Hook

Next article
Building Your Own Hooks