React hooks: not magic, just arrays

Untangling the rules around the proposal using diagrams

Rudi Yardley Follow
Oct 31, 2018 · 5 min read

I am a huge fan of the new hooks API. However, it has some odd constraints about how you need to use it. Here I present a model for how to think about using the new API for those that are struggling to understand the reasons for those rules.

. . .

Photo by rawpixel.com from Pexels

Unpacking how hooks work

I have heard some people struggling with the 'magic' around the new hooks API draft proposal so I thought I would attempt to unpack how the syntax proposal works at least

The rules of hooks

There are two main usage rules the React core team stipulates you need to follow to use hooks which they outline in the hooks proposal documentation.

- **Don't call Hooks inside loops, conditions, or nested functions**
- **Only Call Hooks from React Functions**

The latter I think is self evident. To attach behaviour to a functional component you need to be able to associate that behaviour with the component somehow.

The former however I think can be confusing as it might seem unnatural to program using an API like this and that is what I want to explore today.

State management in hooks is all about arrays

To get a clearer mental model, let's have a look at what a simple implementation of the hooks API might look like.

Please note this is speculation and only one possible way of implementing the API to show how you might want to think about it. This is not necessarily how the API works internally.

How could we implement `useState()` ?

Let's unpack an example here to demonstrate how an implementation of the state hook might work.

First let's start with a component:

```
function RenderFunctionComponent() {
  const [firstName, setFirstName] = useState("Rudi");
  const [lastName, setLastName] = useState("Yardley");

  return (
    <Button onClick={() => setFirstName("Fred")}>Fred</Button>
  );
}
```

hooks-state-example.js hosted with ❤ by GitHub view raw

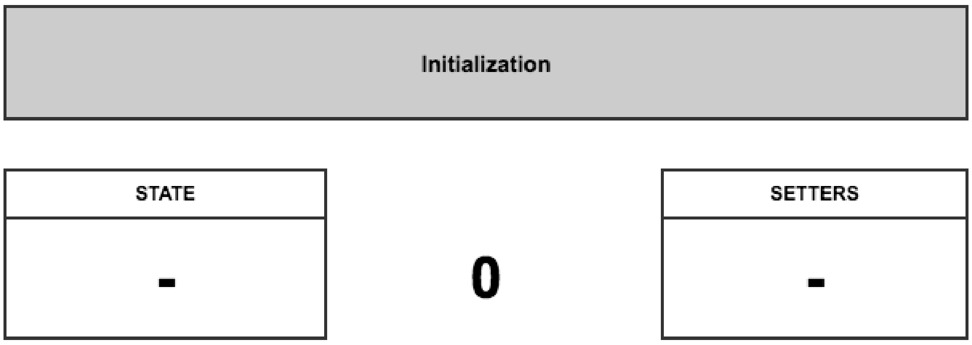
The idea behind the hooks API is that you can use a setter function returned as the second array item from the hook function and that setter will control state which is managed by the hook.

So what's React going to do with this?

Let's annotate how this might work internally within React. The following would work within the execution context for rendering a particular component. That means that the data stored here lives one level outside of the component being rendered. This state is not shared with other components but it is maintained in a scope that is accessible to subsequent rendering of the specific component.

Create two empty arrays: `setters` and `state`

Set a cursor to 0



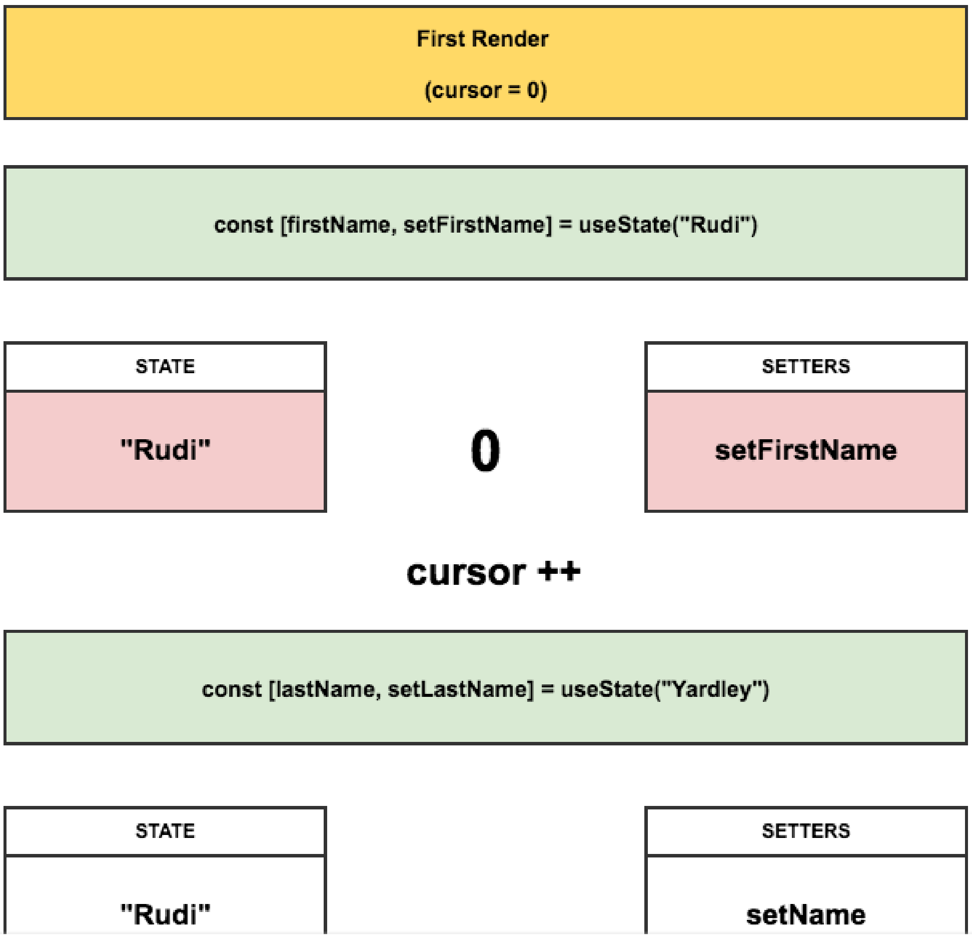
Initialisation: Two empty arrays, Cursor is 0

. . .

2) First render

Run the component function for the first time.

Each `useState()` call, when first run, pushes a setter function (bound to a cursor position) onto the `setters` array and then pushes some state on to the `state` array.

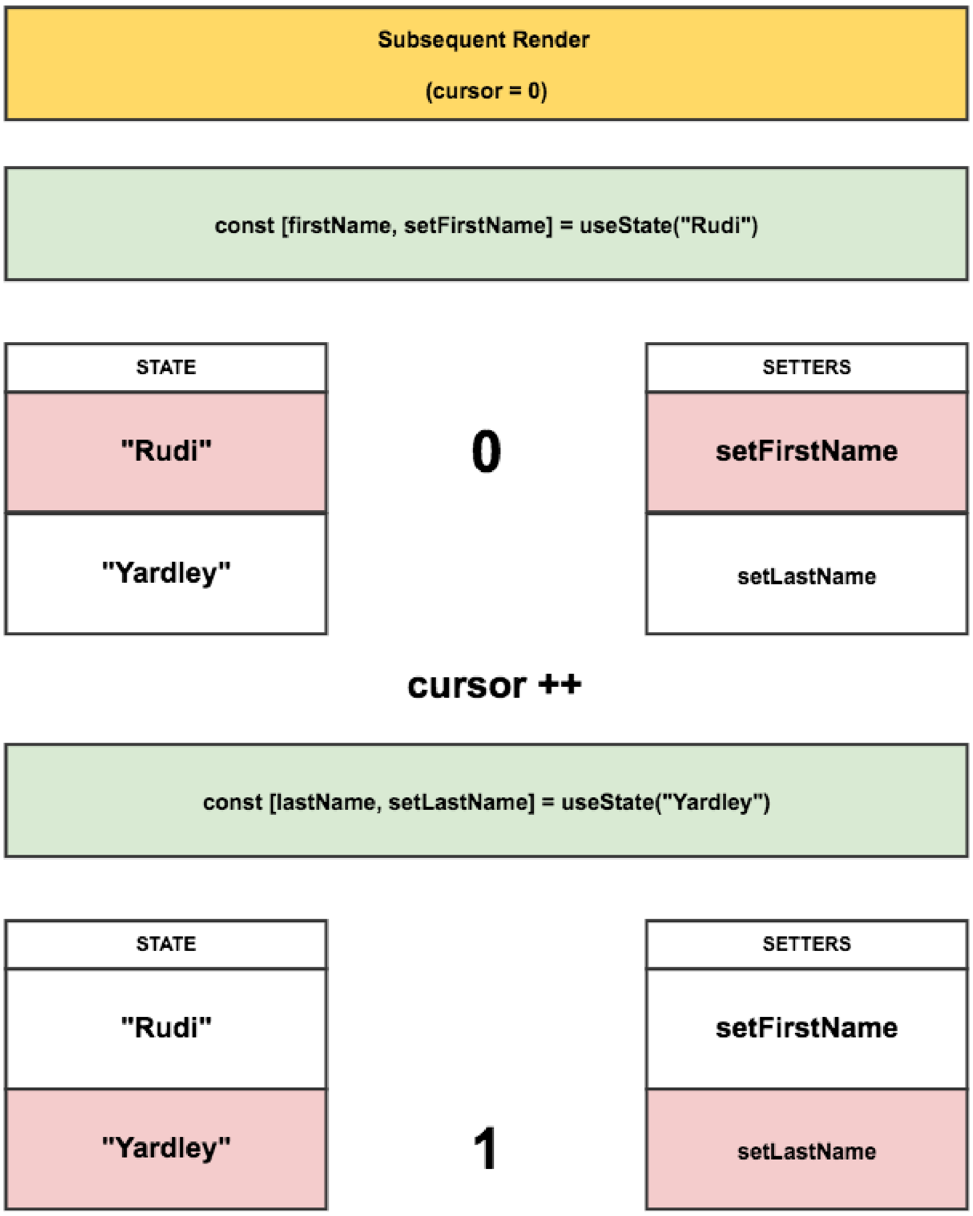

First Render
(cursor = 0)
const [firstName, setFirstName] = useState("Rudi")
STATE
"Rudi"
0
SETTERS
setFirstName
cursor ++
const [lastName, setLastName] = useState("Yardley")
STATE
"Rudi"
SETTERS
setName

First render: Items written to the arrays as cursor increments.

. . .

3) Subsequent render

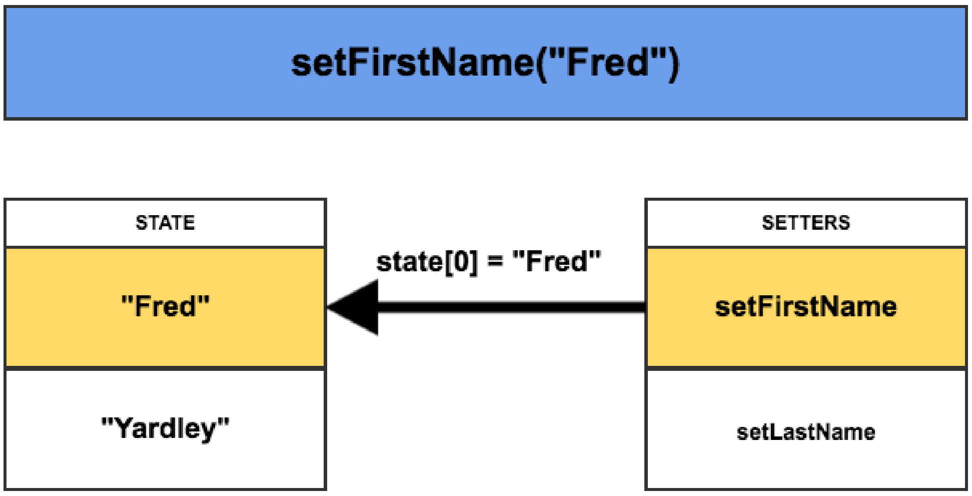
Each subsequent render the cursor is reset and those values are just read from each array.



Subsequent render: Items read from the arrays as cursor increments

. . .

Each setter has a reference to its cursor position so by triggering the call to any `setter` it will change the state value at that position in the state array.



Setters "remember" their index and set memory according to it.

. . .

And the naive implementation

Following here is a naive code example to demonstrate that implementation.

Note: This is not representative of how hooks work at all under the hood but it should give you an idea of a good way to think about how hooks work for a single component. That is why we are using module level vars etc.

```
let state = [];
let setters = [];
let firstRun = true;
let cursor = 0;

function createSetter(cursor) {
  return function setterWithCursor(newVal) {
    state[cursor] = newVal;
  };
}

// This is the pseudocode for the useState helper
export function useState(initVal) {
  if (firstRun) {
    state.push(initVal);
    setters.push(createSetter(cursor));
    firstRun = false;
  }

  const setter = setters[cursor];
  const value = state[cursor];

  cursor++;
```

```

// Our component code that uses hooks
function RenderFunctionComponent() {
  const [firstName, setFirstName] = useState("Rudi"); // cursor: 0
  const [lastName, setLastName] = useState("Yardley"); // cursor: 1

  return (
    <div>
      <Button onClick={() => setFirstName("Richard")}>Richard</Button>
      <Button onClick={() => setFirstName("Fred")}>Fred</Button>
    </div>
  );
}

// This is sort of simulating Reacts rendering cycle
function MyComponent() {
  cursor = 0; // resetting the cursor
  return <RenderFunctionComponent />; // render
}

console.log(state); // Pre-render: []
MyComponent();
console.log(state); // First-render: ['Rudi', 'Yardley']
MyComponent();
console.log(state); // Subsequent-render: ['Rudi', 'Yardley']

// click the 'Fred' button

console.log(state); // After-click: ['Fred', 'Yardley']
```

Why order is important

Now what happens if we change the order of the hooks for a render cycle based on some external factor or even component state?

Let’s do the thing the React team say you should not do:

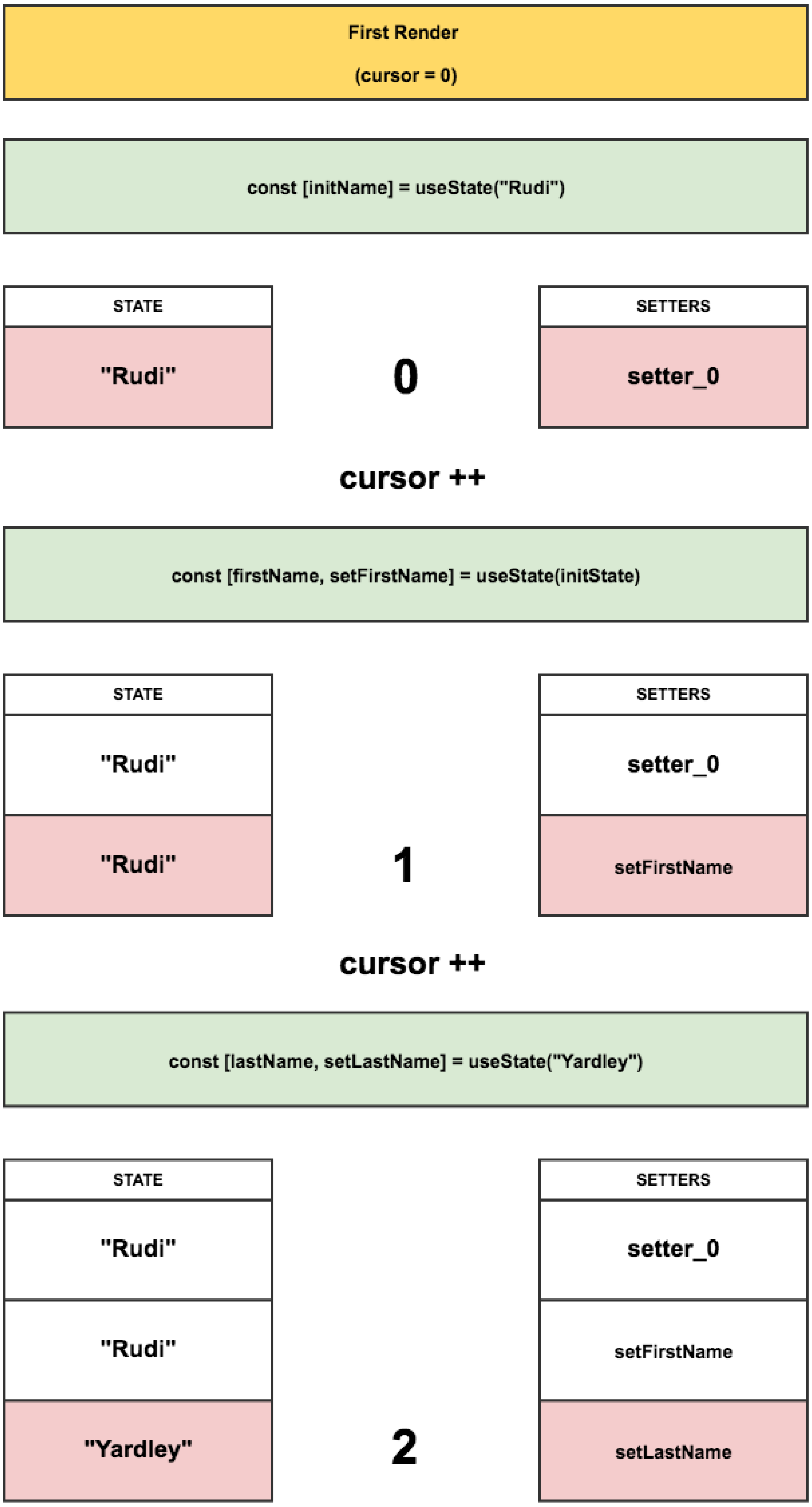
```
let firstRender = true;

function RenderFunctionComponent() {
  let initName;

  if(firstRender){
    [initName] = useState("Rudi");
    firstRender = false;
  }
  const [firstName, setFirstName] = useState(initName);
  const [lastName, setLastName] = useState("Yardley");

  return (
    <Button onClick={() => setFirstName("Fred")}>Fred</Button>
  );
}
```

hooks-order-dont-do-this-example.jsx hosted with ❤ by GitHub view raw

This breaks the rules!

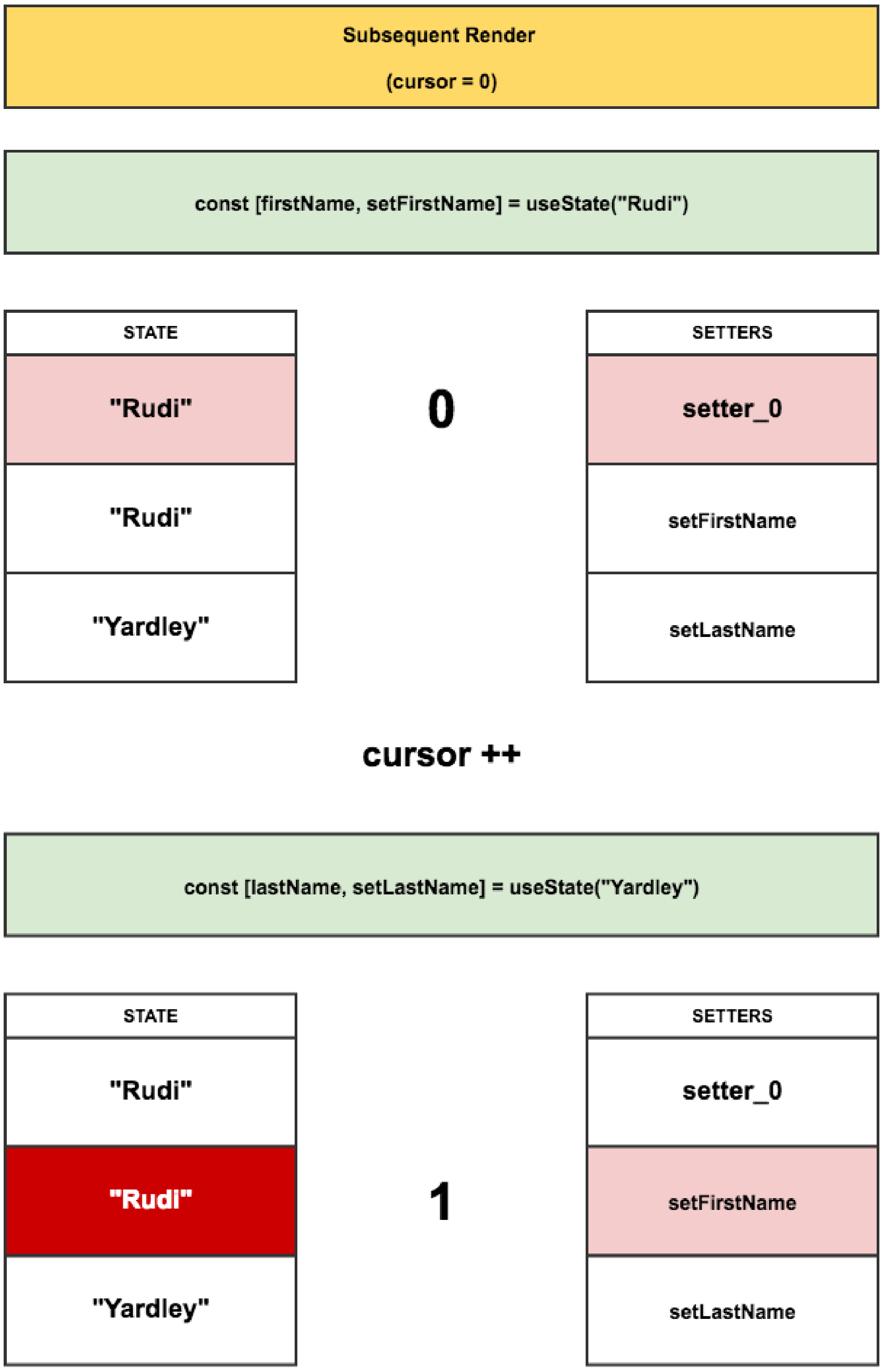
system.

Bad Component First Render



At this point our instance vars `firstName` and `lastName` contain the correct data but let's have a look what happens on the second render:

Bad Component Second Render



By removing the hook between renders we get an error.

Now both `firstName` and `lastName` are set to `"Rudi"` as our state storage becomes inconsistent. This is clearly erroneous and doesn't work but it gives us an idea of why the

> The React team are stipulating the usage rules because not following them will lead to inconsistent data

Think about hooks manipulating a set of arrays and you wont break the rules

So now it should be clear as to why you cannot call `use` hooks within conditionals or loops. Because we are dealing with a cursor pointing to a set of arrays, if you change the order of the calls within render, the cursor will not match up to the data and your use calls will not point to the correct data or handlers.

So the trick is to think about hooks managing its business as a set of arrays that need a consistent cursor. If you do this everything should work.

Conclusion

Hopefully I have laid out a clearer mental model for how to think about what is going on under the hood with the new hooks API. Remember the true value here is being able to group concerns together so being careful about order and using the hooks API will have a high payoff.

Hooks is an effective plugin API for React Components. There is a reason why people are excited about this and if you think about this kind of model where state exists as a set of arrays then you should not find yourselves breaking the rules around their usage.

. . .

This article is a living document please reach out to me if you want to contribute or see anything inaccurate here.

You can follow Rudi Yardley on Twitter as @rudiyardley or on Github as @ryardley

React JavaScript Reactjs