React Hooks cheat sheet: Best practices with examples

January 15, 2021 · 14 min read

***Editor's note**: This React Hooks tutorial was last updated in January 2021 to include more React Hooks best practices and examples.*

React Hooks have a very simple API, but given its massive community and variety of use cases, questions are bound to arise around React Hooks best practices and how to solve common problems.

In this tutorial, we'll outline some React Hooks best practices and highlight some use cases with examples, from simple to advanced scenarios. To help demonstrate how to solve common React Hooks questions, I built an accompanying web app for live interaction with the examples herein.

React Hooks cheat sheet: Best practices and examples

This React Hooks cheat sheet includes a lot of code snippets and assumes some Hooks fluency. If you're completely new to Hooks, you may want to start with our React Hooks API reference guide.

Included in this React Hooks cheat sheet are best practices related to the following Hooks:

- `useState`
- `useEffect`
- `useContext`

- `useLayoutEffect`
- `useReducer`
- `useCallback`
- `useMemo`
- `useRef`

`useState`

`useState` lets you use local state within a function component. 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.

Check out this React `useState` video tutorial:

Declare state variable

Declaring a state variable is as simple as calling `useState` with some initial state value, like so: `useState(initialStateValue)`.

```
const DeclareStateVar = () => {
  const [count] = useState(100)
  return <div> State variable is {count}</div>
}
```

Update state variable

Updating a state variable is as simple as invoking the updater function returned by the `useState` invocation: `const [stateValue, updaterFn] = useState(initialStateValue);`.

> ***Note how the age state variable is being updated.***

Here's the code responsible for the screencast above:

```
const UpdateStateVar = () => {
  const [age, setAge] = useState(19)
  const handleClick = () => setAge(age + 1)

  return (
    <div>
      Today I am {age} Years of Age
      <div>
        <button onClick={handleClick}>Get older! </button>
      </div>
    </div>
  )
}
```

Why does the React `useState` Hook not update immediately?

If you find that `useState/setState` are not updating immediately, the answer is simple: they're just queues.

React `useState` and `setState` don't make changes directly to the state object; they create queues to optimize performance, which is why the changes don't update immediately.

React Hooks and multiple state variables

Multiple state variables may be used and updated from within a functional component, as shown below:

Here's the code responsible for the screencast above:

```
  const handleSiblingsNum = () =>
      setSiblingsNum(siblingsNum + 1)



  return (
    <div>
      <p>Today I am {age} Years of Age</p>
      <p>I have {siblingsNum} siblings</p>

      <div>
        <button onClick={handleAge}>
          Get older!
        </button>
        <button onClick={handleSiblingsNum}>
            More siblings!

        </button>
      </div>
    </div>
  )
}
```

Use object state variable

As opposed to strings and numbers, you could also use an object as the initial value passed to `useState`.

Note that you have to pass the entire object to the `useState` updater function because the object is replaced, not merged.

```
// 🐢 setState (object merge) vs useState (object replace)
// assume initial state is {name: "Ohans"}

setState({ age: 'unknown' })
// new state object will be
// {name: "Ohans", age: "unknown"}

useStateUpdater({ age: 'unknown' })
// new state object will be
// {age: "unknown"} - initial object is replaced
```

Multiple state objects updated via a state object variable.

Here's the code for the screencast above:

```
    setState({
      ...state,
      [val]: state[val] + 1
    })
  const { age, siblingsNum } = state

  return (
    <div>
      <p>Today I am {age} Years of Age</p>
      <p>I have {siblingsNum} siblings</p>

      <div>
        <button onClick={handleClick.bind(null, 'age')}>Get older!</button>
        <button onClick={handleClick.bind(null, 'siblingsNum')}>
          More siblings!
        </button>
      </div>
    </div>
  )
}
```

Initialize state from function

As opposed to just passing an initial state value, state could also be initialized from a function, as shown below:

```
const StateFromFn = () => {
  const [token] = useState(() => {
    let token = window.localStorage.getItem("my-token");
    return token || "default#-token#"
  })

  return <div>Token is {token}</div>
}
```

Functional `setState`

The updater function returned from invoking `useState` can also take a function similar to the good ol' `setState`:

```
const [value, updateValue] = useState(0)
// both forms of invoking "updateValue" below are valid 👇
updateValue(1);
updateValue(previousValue => previousValue + 1);
```

This is ideal when the state update depends on some previous value of state.

A counter with functional setState updates.

Here's the code for the screencast above:

```
const CounterFnSetState = () => {
  const [count, setCount] = useState(0);
  return (
    <>
      <p>Count value is: {count}</p>
      <button onClick={() => setCount(0)}>Reset</button>
      <button
        onClick={() => setCount(prevCount => prevCount + 1)}>
        Plus (+)
      </button>
      <button
        onClick={() => setCount(prevCount => prevCount - 1)}>
       Minus (-)
      </button>
    </>
  );
}
```

Here's a live, editable `useState` cheat sheet if you want to dive deeper on your own.

`useEffect`

With `useEffect`, you invoke side effects from within functional components, which is an important concept to understand in the React Hooks era.

Basic side effect

> ***Watch the title of the document update.***

Here's the code responsible for the screencast above:

```
const BasicEffect = () => {
  const [age, setAge] = useState(0)
  const handleClick = () => setAge(age + 1)

  useEffect(() => {
    document.title = 'You are ' + age + ' years old!'
  })

  return <div>
    <p> Look at the title of the current tab in your browser </p>
    <button onClick={handleClick}>Update Title!! </button>
  </div>
}
```

Effect with cleanup

It's pretty common to clean up an effect after some time. This is possible by returning a function from within the effect function passed to `useEffect`. Below is an example with `addEventListener`.

```
const EffectCleanup = () => {
  useEffect(() => {
    const clicked = () => console.log('window clicked')
    window.addEventListener('click', clicked)

    // return a clean-up function
    return () => {
      window.removeEventListener('click', clicked)
    }
  }, [])

  return <div>
    When you click the window you'll
    find a message logged to the console
  </div>
}
```

Multiple effects

Multiple `useEffect` calls can happen within a functional component, as shown below:

```
const MultipleEffects = () => {
  // 🍟
  useEffect(() => {
    const clicked = () => console.log('window clicked')
    window.addEventListener('click', clicked)

    return () => {
      window.removeEventListener('click', clicked)
    }
  }, [])

  // 🍟 another useEffect hook
  useEffect(() => {
    console.log("another useEffect call");
  })

  return <div>
    Check your console logs
  </div>
}
```

Note that `useEffect` calls can be skipped — i.e., not invoked on every render. This is done by passing a second array argument to the effect function.

Skipping effects (array dependency)

```
  )

  return (
    <div>
      <h1>{randomNumber}</h1>
      <button
        onClick={() => {
          setRandomNumber(Math.random())
        }}
      >
        Generate random number!
      </button>
      <div>
        {effectLogs.map((effect, index) => (
          <div key={index}>{'🍔'.repeat(index) + effect}</div>
        ))}
      </div>
    </div>
  )
}
```

In the example above, `useEffect` is passed an array of one value: `[randomNumber]`.

Thus, the effect function will be called on mount *and* whenever a new random number is generated.

Here's the **Generate random number** button being clicked and the effect function being rerun upon generating a new random number:

Skipping effects (empty array dependency)

In this example, `useEffect` is passed an empty array, `[]`. Therefore, the effect function will be called only on mount.

```
  )

  return (
    <div>
      <h1>{randomNumber}</h1>
      <button
        onClick={() => {
          setRandomNumber(Math.random())
        }}
      >
        Generate random number!
      </button>
      <div>
        {effectLogs.map((effect, index) => (
          <div key={index}>{'🍔'.repeat(index) + effect}</div>
        ))}
      </div>
    </div>
  )
}
```

Here's the button being clicked and the effect function not invoked:

Skipping effects (no array dependency)

Without an array dependency, the effect function will be run after every single render.

```
useEffect(() => {
console.log("This will be logged after every render!")
})
```

Here's a live, editable `useEffect` cheat sheet if you'd like to explore further.

`useContext`

`useContext` saves you the stress of having to rely on a Context consumer. React Context has a simpler API when compared to `MyContext.Consumer` and the render props API it exposes.

Context is React's way of handling shared data between multiple components.

The following example highlights the difference between consuming a context object value via `useContext` or `Context.Consumer`:

```
// example Context object
const ThemeContext = React.createContext("dark");

// usage with context Consumer
function Button() {
  return <ThemeContext.Consumer>
        {theme => <button className={theme}> Amazing button </button>}
  </ThemeContext.Consumer>
}


// usage with useContext hook
import {useContext} from 'react';

function ButtonHooks() {
 const theme = useContext(ThemeContext)
 return <button className={theme}>Amazing button</button>
}
```

Here's a live example with `useContext`:

And here's the code responsible for the example above:

```
const ThemeContext = React.createContext('light');

const Display = () => {
 const theme = useContext(ThemeContext);
 return <div
        style={{
        background: theme === 'dark' ? 'black' : 'papayawhip',
        color: theme === 'dark' ? 'white' : 'palevioletred',
        width: '100%',
        minHeight: '200px'
        }}
    >
        {'The theme here is ' + theme}
    </div>
}
```

Here's a live, editable React Context cheat sheet if you'd like to tinker around yourself.

`useLayoutEffect`

`useLayoutEffect` has the very same signature as `useEffect`. We'll discuss the difference between `useLayoutEffect` and `useEffect` below.

```
useLayoutEffect(() => {
//do something
}, [arrayDependency])
```

Similar usage as `useEffect`

Here's the same example for `useEffect` built with `useLayoutEffect`:

And here's the code:

```
const ArrayDep = () => {
    const [randomNumber, setRandomNumber] = useState(0)
    const [effectLogs, setEffectLogs] = useState([])

    useLayoutEffect(
      () => {
        setEffectLogs(prevEffectLogs => [...prevEffectLogs, 'effect fn has
been invoked'])
      },
      [randomNumber]
    )

    return (
      <div>
        <h1>{randomNumber}</h1>
        <button
          onClick={() => {
            setRandomNumber(Math.random())
          }}
        >
```

`useLayoutEffect` vs. `useEffect`

What's the difference between `useEffect` and `useLayoutEffect`? The function passed to `useEffect` fires after layout and paint — i.e., after the render has been committed to the screen. This is OK for most side effects that shouldn't block the browser from updating the screen.

There are cases where you may not want the behavior `useEffect` provides, though; for example, if you need to make a visual change to the DOM as a side effect, `useEffect` won't be the best choice.

To prevent the user from seeing flickers of changes, you can use `useLayoutEffect`. The function passed to `useLayoutEffect` will be run before the browser updates the screen.

You can read my follow-up piece for a deep dive on the differences between `useEffect` and `useLayoutEffect`.

Here's a live, editable `useLayoutEffect` cheat sheet.

`useReducer`

`useReducer` may be used as an alternative to `useState`. It's ideal for complex state logic where there's a dependency on previous state values or a lot of state sub-values.

Depending on your use case, you may find `useReducer` quite testable.

Basic usage

As opposed to calling `useState`, call `useReducer` with a `reducer` and `initialState`, as shown below. The `useReducer` call returns the state property and a `dispatch` function.

Increase/decrease bar size by managing state with useReducer.

Here's the code responsible for the above screencast:

```
const initialState = { width: 15 };

const reducer = (state, action) => {
  switch (action) {
    case 'plus':
      return { width: state.width + 15 }
    case 'minus':
      return { width: Math.max(state.width - 15, 2) }
    default:
      throw new Error("what's going on?" )
  }
}

const Bar = () => {
  const [state, dispatch] = useReducer(reducer, initialState)
  return <>
    <div style={{ background: 'teal', height: '30px', width: state.width }}>
</div>
    <div style={{marginTop: '3rem'}}>
        <button onClick={() => dispatch('plus')}>Increase bar size</button>
```

Initialize state lazily

`useReducer` takes a third function parameter. You may initialize state from this function, and whatever's returned from this function is returned as the state object. This function will be called with `initialState` — the second parameter.

Same increase/decrease bar size, with state initialized lazily.

Here's the code for the example above:

```
const initializeState = () => ({
  width: 100
})

// ✅ note how the value returned from the fn above overrides initialState
below:
const initialState = { width: 15 }
const reducer = (state, action) => {
  switch (action) {
    case 'plus':
      return { width: state.width + 15 }
    case 'minus':
      return { width: Math.max(state.width - 15, 2) }
    default:
      throw new Error("what's going on?" )
  }
}

const Bar = () => {
  const [state, dispatch] = useReducer(reducer, initialState,
```

Imitate the behavior of `this.setState`

`useReducer` uses a reducer that isn't as strict as Redux's. For example, the second parameter passed to the reducer, `action`, doesn't need to have a `type` property.

This allows for interesting manipulations, such as renaming the second parameter and doing the following:

```
const initialState = { width: 15 };

const reducer = (state, newState) => ({
  ...state,
  width: newState.width
})

const Bar = () => {
  const [state, setState] = useReducer(reducer, initialState)
  return <>
    <div style={{ background: 'teal', height: '30px', width: state.width }}>
</div>
    <div style={{marginTop: '3rem'}}>
        <button onClick={() => setState({width: 100})}>Increase bar
size</button>
        <button onClick={() => setState({width: 3})}>Decrease bar
size</button>
    </div>
    </>
}
```

The results remain the same with a setState-like API imitated.

Here's an editable `useReducer` cheat sheet.

`useCallback`

`useCallback` returns a memoized callback. Wrapping a component with `React.Memo()` signals the intent to reuse code. This does not automatically extend to functions passed as parameters.

React saves a reference to the function when wrapped with `useCallback`. Pass this reference as a property to new components to reduce rendering time.

`useCallback` example

The following example will form the basis of the explanations and code snippets that follow.

And here's the code:

```
const App = () => {
    const [age, setAge] = useState(99)
    const handleClick = () => setAge(age + 1)
    const someValue = "someValue"
    const doSomething = () => {
      return someValue
    }

    return (
      <div>
        <Age age={age} handleClick={handleClick}/>
        <Instructions doSomething={doSomething} />
      </div>
    )
}

const Age = ({ age, handleClick }) => {
  return (
    <div>
      <div style={{ border: '2px', background: "papayawhip", padding: "1rem"
```

In the example above, the parent component, `<Age />`, is updated (and re-rendered) whenever the **Get older** button is clicked.

Consequently, the `<Instructions />` child component is also re-rendered because the `doSomething` prop is passed a new callback with a new reference.

Note that even though the `Instructions` child component uses `React.memo` to optimize performance, it is still re-rendered.

How can this be fixed to prevent `<Instructions />` from re-rendering needlessly?

`useCallback` with referenced function

```
const App = () => {
  const [age, setAge] = useState(99)
  const handleClick = () => setAge(age + 1)
  const someValue = "someValue"
  const doSomething = useCallback(() => {
    return someValue
  }, [someValue])

  return (
    <div>
      <Age age={age} handleClick={handleClick} />
      <Instructions doSomething={doSomething} />
    </div>
  )
}

const Age = ({ age, handleClick }) => {
  return (
    <div>
      <div style={{ border: '2px', background: "papayawhip", padding: "1rem"
```

`useCallback` with inline function

`useCallback` also works with an inline function as well. Here's the same solution with an inline `useCallback` call:

```
const App = () => {
  const [age, setAge] = useState(99)
  const handleClick = () => setAge(age + 1)
  const someValue = "someValue"

  return (
    <div>
      <Age age={age} handleClick={handleClick} />
      <Instructions doSomething={useCallback(() => {
        return someValue
      }, [someValue])} />
    </div>
  )
}

const Age = ({ age, handleClick }) => {
  return (
    <div>
      <div style={{ border: '2px', background: "papayawhip", padding: "1rem"
}}>
```

Here's live, editable `useCallback` cheat sheet.

`useMemo`

The `useMemo` function returns a memoized value. `useMemo` is different from `useCallback` in that it internalizes return values instead of entire functions. Rather than passing a handle to the same function, React skips the function and returns the previous result, until the parameters change.

This allows you to avoid repeatedly performing potentially costly operations until necessary. Use this method with care, as any changing variables defined in the function do not affect the behavior of `useMemo`. If you're performing timestamp additions, for instance, this method does not care that the time changes, only that the function parameters differ.

`useMemo` example

The following example will form the basis of the explanations and code snippets that follow.

Here's the code responsible for the screenshot above:

```
const App = () => {
    const [age, setAge] = useState(99)
    const handleClick = () => setAge(age + 1)
    const someValue = { value: "someValue" }
    const doSomething = () => {
      return someValue
    }

    return (
      <div>
        <Age age={age} handleClick={handleClick}/>
        <Instructions doSomething={doSomething} />
      </div>
    )
}

const Age = ({ age, handleClick }) => {
  return (
    <div>
      <div style={{ border: '2px', background: "papayawhip", padding: "1rem"
```

The example above is similar to the one for `useCallback`. The only difference here is that `someValue` is an object, *not* a string. Owing to this, the `Instructions` component still re-renders despite the use of `React.memo`.

Why? Objects are compared by reference, and the reference to `someValue` changes whenever `<App />` re-renders.

Any solutions?

Basic usage

The object `someValue` may be memoized using `useMemo`. This prevents the needless re-render.

```
const App = () => {
    const [age, setAge] = useState(99)
    const handleClick = () => setAge(age + 1)
    const someValue = useMemo(() => ({ value: "someValue" }))
    const doSomething = () => {
      return someValue
    }

    return (
      <div>
        <Age age={age} handleClick={handleClick}/>
        <Instructions doSomething={doSomething} />
      </div>
    )
}

const Age = ({ age, handleClick }) => {
  return (
    <div>
      <div style={{ border: '2px', background: "papayawhip", padding: "1rem"
```

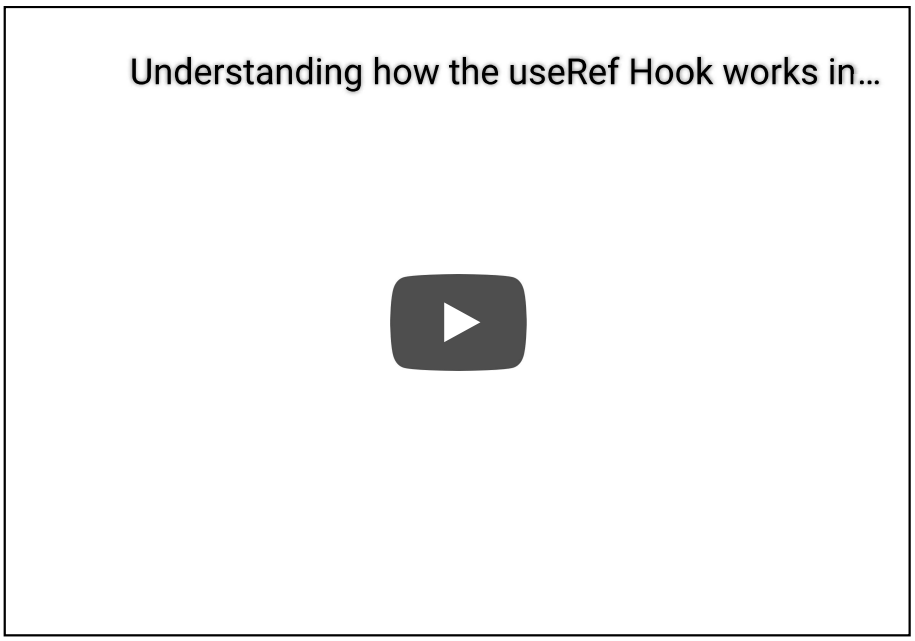
Here's a live, editable `useMemo` demo.

`useRef`

`useRef` returns a "ref" object. Values are accessed from the `.current` property of the returned object. The `.current` property could be initialized to an initial value — `useRef(initialValue)`, for example. The object is persisted for the entire

lifetime of the component.

Learn more in this comprehensive `useRefs` guide or check out our `useRefs` video tutorial:



Accessing the DOM

Consider the sample application below:

Accessing the DOM via useRef.

Here's the code responsible for the screencast above:

```
const AccessDOM = () => {
  const textAreaEl = useRef(null);
  const handleBtnClick = () => {
    textAreaEl.current.value =
    "The is the story of your life. You are an human being, and you're on a
website about React Hooks";
    textAreaEl.current.focus();
  };
  return (
    <section style={{ textAlign: "center" }}>
      <div>
        <button onClick={handleBtnClick}>Focus and Populate Text
Field</button>
      </div>
      <label
        htmlFor="story"
        style={{
          display: "block",
          background: "olive",
          margin: "1em",
```

Instance-like variables (generic container)

Other than just holding DOM refs, the “ref” object can hold any value. Consider a similar application below, where the ref object holds a string value:

Here’s the code:

```
const HoldStringVal = () => {
    const textAreaEl = useRef(null);
    const stringVal = useRef("This is a string saved via the ref object ---
")
    const handleBtnClick = () => {
      textAreaEl.current.value =
      stringVal.current + "The is the story of your life. You are an human
being, and you're on a website about React Hooks";
      textAreaEl.current.focus();
    };
    return (
      <section style={{ textAlign: "center" }}>
        <div>
          <button onClick={handleBtnClick}>Focus and Populate Text
Field</button>
        </div>
        <label
          htmlFor="story"
          style={{
            display: "block",
```

You could do the same as storing the return value from a `setInterval` for cleanup.

```
function TimerWithRefID() {
  const setIntervalRef = useRef();

  useEffect(() => {
    const intervalID = setInterval(() => {
      // something to be done every 100ms
    }, 100);

    // this is where the interval ID is saved in the ref object
    setIntervalRef.current = intervalID;
    return () => {
      clearInterval(setIntervalRef.current);
    };
  });
}
```

Other examples

Working on a near-real-world example can help bring your knowledge of Hooks to life. Until data fetching with React Suspense is released, fetching data via Hooks proves to be a good exercise for more Hooks practice.

Below's an example of fetching data with a loading indicator:

The code appears below:

```
const fetchData = () => {
  const stringifyData = data => JSON.stringify(data, null, 2)
  const initialData = stringifyData({ data: null })
  const loadingData = stringifyData({ data: 'loading...' })
  const [data, setData] = useState(initialData)

  const [gender, setGender] = useState('female')
  const [loading, setLoading] = useState(false)

  useEffect(
    () => {
      const fetchData = () => {
        setLoading(true)
        const uri = 'https://randomuser.me/api/?gender=' + gender
        fetch(uri)
          .then(res => res.json())
          .then(({ results }) => {
            setLoading(false)
            const { name, gender, dob } = results[0]
            const dataVal = stringifyData({
```

Here' a live, editable `useRef` cheat sheet.

Conclusion

Hooks give a lot of power to functional components. I hope this cheat sheet proves useful in your day-to-day use of React Hooks.

> ***Thanks to Hooks and a couple other new React features. Illustration by me*** 🙂

Cheers!

Full visibility into production React apps

Debugging React applications can be difficult, especially when users experience issues that are hard to reproduce. If you're interested in monitoring and tracking Redux state, automatically surfacing JavaScript errors, and tracking slow network requests and component load time, try LogRocket.

LogRocket is like a DVR for web apps, recording literally everything that happens on your React app. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when an issue occurred. LogRocket also monitors your app's performance, reporting with metrics like client CPU load, client memory usage, and more.

The LogRocket Redux middleware package adds an extra layer of visibility into your user sessions. LogRocket logs all actions and state from your Redux stores.

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

Ohans Emmanuel Follow

Author of *Understanding Redux*. I love God. I love GF a little too much 💕🤣 Read The Redux.js Books.

#react

8 Replies to "React Hooks cheat sheet: Best practices with examples"

Dan Says:
July 20, 2019 at 3:44 pm

Reply

Thanks, some interesting points on here. I'm currently building a single page app using React and WordPress and the hooks are proving very useful. I'm having problems persisting useState data with a route change, still looking for clues..!

Ohans E. Says:
July 24, 2019 at 9:35 am

Reply

Nice! Typically, you'd have to centralize the data you want to share across routes – either via a centra store like redux', or a central context object, or perhaps via the browser's LocalStroage. You've got many options and the best for your specific use case depends on the application you're building.

Rick Says:
July 28, 2019 at 2:43 pm

Reply

I have a question: The official docs (and every blog post I've seen about hooks) says that fetching data should be done in useEffect. Changing the DOM "manually" with a reference to an element should be done in useLayoutEffect to avoid flicker. This seems like a contradiction to me. When you fetch data, 99% of the time you're going to display some of that data in the UI. So you are indirectly (not manually with a reference to an element) changing the DOM. So, you'll have a flicker if you do the fetch/state change in useEffect. So, why don't all the docs say that fetching data should be standardly done in useLayoutEffect?

Grdpr Says:
September 3, 2019 at 3:25 pm

Reply

Great article! I'm trying to set a random number to a color state using hooks:
const COLOR = function() {
return '#' + Math.floor(Math.random() * 16777215).toString(16);
};
const [bgColor, setBgColor] = useState(COLOR);
The value should be different every time the page is refreshed. In dev mode it's working but when I build the app, the value become static. Would use "useEffect" for that case?

Samuel Karani (@Samuel_karani) Says:
September 17, 2019 at 6:58 pm

Reply

Great article, but shouldn't useMemo have second parameter [] to prevent rerenders?

Nick Says:
May 27, 2020 at 7:44 am

Reply

Really good article! Thanks for that! Just noticed that in the Skipping effects (array dependency) section, the array that is passed to useEffect doesn't have the randomNumber in the code example.

Nibin John Says:
June 28, 2020 at 4:54 am

Reply

Appreciate your effort to make this

Jerry Says:
April 13, 2021 at 12:07 am

Reply

probably the best hooks tutorial out there. simple easy explanation.

Leave a Reply

Enter your comment here...