# 155.https://stackoverflow.com/questions/72649336/referenceerror-process-is-not-defined-while-trying-mint-in-web3-in-reactjs

**T:**The useState set method is not reflecting a change immediately

**Q:**I am trying to learn hooks and the useState method has made me confused. I am assigning an initial value to a state in the form of an array. The set method in useState is not working for me, both with and without the spread syntax.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
I have made an API on another PC that I am calling and fetching the data which I want to set into the state.  
  
Here is my code:  
  
<div id="root"></div><script type="text/babel" defer>// import React, { useState, useEffect } from "react";// import ReactDOM from "react-dom";const { useState, useEffect } = React; // web-browser variantconst StateSelector = () => { const initialValue = [ { category: "", photo: "", description: "", id: 0, name: "", rating: 0 } ]; const [movies, setMovies] = useState(initialValue); useEffect(() => { (async function() { try { // const response = await fetch("http://192.168.1.164:5000/movies/display"); // const json = await response.json(); // const result = json.data.result; const result = [ { category: "cat1", description: "desc1", id: "1546514491119", name: "randomname2", photo: null, rating: "3" }, { category: "cat2", description: "desc1", id: "1546837819818", name: "randomname1", rating: "5" } ]; console.log("result =", result); setMovies(result); console.log("movies =", movies); } catch (e) { console.error(e); } })(); }, []); return <p>hello</p>;};const rootElement = document.getElementById("root");ReactDOM.render(<StateSelector />, rootElement);</script><script src="https://unpkg.com/@babel/standalone@7/babel.min.js"></script><script src="https://unpkg.com/react@17/umd/react.production.min.js"></script><script src="https://unpkg.com/react-dom@17/umd/react-dom.production.min.js"></script>  
  
Neither setMovies(result) nor setMovies(...result) works.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
I expect the result variable to be pushed into the movies array.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**C1:**Are you able to see the changes moving console.log("movies =", movies); outsite the useEffect hook?

**C2:**Good question though

16 **Answer**

**A1:**Much like .setState() in class components created by extending React.Component or React.PureComponent, the state update using the updater provided by useState hook is also asynchronous, and will not be reflected immediately.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Also, the main issue here is not just the asynchronous nature but the fact that state values are used by functions based on their current closures, and state updates will reflect in the next re-render by which the existing closures are not affected, but new ones are created. Now in the current state, the values within hooks are obtained by existing closures, and when a re-render happens, the closures are updated based on whether the function is recreated again or not.  
  
Even if you add a setTimeout the function, though the timeout will run after some time by which the re-render would have happened, the setTimeout will still use the value from its previous closure and not the updated one.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
setMovies(result);console.log(movies) // movies here will not be updated  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
If you want to perform an action on state update, you need to use the useEffect hook, much like using componentDidUpdate in class components since the setter returned by useState doesn't have a callback pattern  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
useEffect(() => { // action on update of movies}, [movies]);  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
As far as the syntax to update state is concerned, setMovies(result) will replace the previous movies value in the state with those available from the async request.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
However, if you want to merge the response with the previously existing values, you must use the callback syntax of state updation along with the correct use of spread syntax like  
  
setMovies(prevMovies => ([...prevMovies, ...result]));  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**C1:**Hi, what about calling useState inside a form submit handler ? I am working on validating a complex form, and I call inside submitHandler useState hooks and unfortunately changes are not immediate !

**C2:**useEffect might not be the best solution though, since it doesn't support asynchronous calls. So, if we would like to make some asynchronous validation on movies state change, we have no control over it.

**C3:**please note that while the advice is very good, the explanation of the cause can be improved - nothing to do with the fact whether or not the updater provided by useState hook is asynchronous, unlike this.state that could have been mutated if this.setState was synchronous, the Closure around const movies would remain the same even if useState provided a synchronous function - see the example in my answer

**C4:**setMovies(prevMovies => ([...prevMovies, ...result])); worked for me

**C5:**It is logging the wrong result because you are logging a stale closure not because the setter is asynchronous. If async was the problem then you could log after a timeout, but you could set a timeout for an hour and still log the wrong result because async isn't what is causing the problem.

**A2:**Additional details to the previous answer:  
  
While React's setState is asynchronous (both classes and hooks), and it's tempting to use that fact to explain the observed behavior, it is not the reason why it happens.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
TLDR: The reason is a closure scope around an immutable const value.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
 ● read the value in render function (not inside nested functions):  
 useEffect(() => { setMovies(result) }, [])  
 console.log(movies)  
  
  
  
 ● add the variable into dependencies (and use the react-hooks/exhaustive-deps eslint rule):  
 useEffect(() => { setMovies(result) }, [])  
 useEffect(() => { console.log(movies) }, [movies])  
  
  
  
 ● use a temporary variable:  
 useEffect(() => {  
 const newMovies = result  
 console.log(newMovies)  
 setMovies(newMovies)  
 }, [])  
  
  
  
 ● use a mutable reference (if we don't need a state and only want to remember the value - updating a ref doesn't trigger re-render):  
 const moviesRef = useRef(initialValue)  
 useEffect(() => {  
 moviesRef.current = result  
 console.log(moviesRef.current)  
 }, [])  
  
  
  
If async was the only reason, it would be possible to await setState().  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
However, both props and state are assumed to be unchanging during 1 render.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Treat this.state as if it were immutable.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
With hooks, this assumption is enhanced by using constant values with the const keyword:  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
const [state, setState] = useState('initial')  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
The value might be different between 2 renders, but remains a constant inside the render itself and inside any closures (functions that live longer even after render is finished, e.g. useEffect, event handlers, inside any Promise or setTimeout).  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Consider following fake, but synchronous, React-like implementation:  
  
// sync implementation:let internalStatelet renderAgainconst setState = (updateFn) => { internalState = updateFn(internalState) renderAgain()}const useState = (defaultState) => { if (!internalState) { internalState = defaultState } return [internalState, setState]}const render = (component, node) => { const {html, handleClick} = component() node.innerHTML = html renderAgain = () => render(component, node) return handleClick}// test:const MyComponent = () => { const [x, setX] = useState(1) console.log('in render:', x) // ✅ const handleClick = () => { setX(current => current + 1) console.log('in handler/effect/Promise/setTimeout:', x) // ❌ NOT updated } return { html: `<button>${x}</button>`, handleClick }}const triggerClick = render(MyComponent, document.getElementById('root'))triggerClick()triggerClick()triggerClick()<div id="root"></div>

**C1:**@AlJoslin at a first glance, that seems like a separate problem, even if it might be caused by closure scope. If you have a concrete question, please create a new stackoverflow question with code example and all...

**C2:**actually I just finished a rewrite with useReducer, following @kentcdobs article (ref below) which really gave me a solid result that suffers not one bit from these closure problems. (ref: kentcdodds.com/blog/how-to-use-react-context-effectively)

**C3:**for some reason solution 2 is not working.. I get the callback, but the value is still empty. useEffect(() => { console.log(movies) }, [movies]) this prints nothing ..

**C4:**@ACV Solution 2 works fine for the original question. If you need to solve a different problem, YMMW, but I am still 100% sure that the quoted code works as documented and the problem is somewhere else.

**C5:**All of these solutions require the use of useEffect. My problem is that my "movies" equivalent is an object I get from a Context provider and can be changed by many other components. I don't want to run the effect every time it's changed because my effect isn't setMovies - it's a different function I need to call only when a particular change is made to movies - a change I'm not seeing when needed because of a stale context.

**A3:**I know that there are already very good answers. But I want to give another idea how to solve the same issue, and access the latest 'movie' state, using my module react-useStateRef.  
  
As you understand by using React state you can render the page every time the state change. But by using React ref, you can always get the latest values.  
  
So the module react-useStateRef let you use state's and ref's together. It's backward compatible with React.useState, so you can just replace the import statement  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
const { useEffect } = Reactimport { useState } from 'react-usestateref' const [movies, setMovies] = useState(initialValue); useEffect(() => { (async function() { try { const result = [ { id: "1546514491119", }, ]; console.log("result =", result); setMovies(result); console.log("movies =", movies.current); // will give you the latest results } catch (e) { console.error(e); } })(); }, []);  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
 ● react-usestsateref

**A4:**I just finished a rewrite with useReducer, following @kentcdobs article (ref below) which really gave me a solid result that suffers not one bit from these closure problems.  
  
See: https://kentcdodds.com/blog/how-to-use-react-context-effectively  
  
I condensed his readable boilerplate to my preferred level of DRYness -- reading his sandbox implementation will show you how it actually works.  
  
import React from 'react'// ref: https://kentcdodds.com/blog/how-to-use-react-context-effectivelyconst ApplicationDispatch = React.createContext()const ApplicationContext = React.createContext()function stateReducer(state, action) { if (state.hasOwnProperty(action.type)) { return { ...state, [action.type]: state[action.type] = action.newValue }; } throw new Error(`Unhandled action type: ${action.type}`);}const initialState = { keyCode: '', testCode: '', testMode: false, phoneNumber: '', resultCode: null, mobileInfo: '', configName: '', appConfig: {},};function DispatchProvider({ children }) { const [state, dispatch] = React.useReducer(stateReducer, initialState); return ( <ApplicationDispatch.Provider value={dispatch}> <ApplicationContext.Provider value={state}> {children} </ApplicationContext.Provider> </ApplicationDispatch.Provider> )}function useDispatchable(stateName) { const context = React.useContext(ApplicationContext); const dispatch = React.useContext(ApplicationDispatch); return [context[stateName], newValue => dispatch({ type: stateName, newValue })];}function useKeyCode() { return useDispatchable('keyCode'); }function useTestCode() { return useDispatchable('testCode'); }function useTestMode() { return useDispatchable('testMode'); }function usePhoneNumber() { return useDispatchable('phoneNumber'); }function useResultCode() { return useDispatchable('resultCode'); }function useMobileInfo() { return useDispatchable('mobileInfo'); }function useConfigName() { return useDispatchable('configName'); }function useAppConfig() { return useDispatchable('appConfig'); }export { DispatchProvider, useKeyCode, useTestCode, useTestMode, usePhoneNumber, useResultCode, useMobileInfo, useConfigName, useAppConfig,}  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
With a usage similar to this:  
  
import { useHistory } from "react-router-dom";// https://react-bootstrap.github.io/components/alertsimport { Container, Row } from 'react-bootstrap';import { useAppConfig, useKeyCode, usePhoneNumber } from '../../ApplicationDispatchProvider';import { ControlSet } from '../../components/control-set';import { keypadClass } from '../../utils/style-utils';import { MaskedEntry } from '../../components/masked-entry';import { Messaging } from '../../components/messaging';import { SimpleKeypad, HandleKeyPress, ALT\_ID } from '../../components/simple-keypad';export const AltIdPage = () => { const history = useHistory(); const [keyCode, setKeyCode] = useKeyCode(); const [phoneNumber, setPhoneNumber] = usePhoneNumber(); const [appConfig, setAppConfig] = useAppConfig(); const keyPressed = btn => { const maxLen = appConfig.phoneNumberEntry.entryLen; const newValue = HandleKeyPress(btn, phoneNumber).slice(0, maxLen); setPhoneNumber(newValue); } const doSubmit = () => { history.push('s'); } const disableBtns = phoneNumber.length < appConfig.phoneNumberEntry.entryLen; return ( <Container fluid className="text-center"> <Row> <Messaging {...{ msgColors: appConfig.pageColors, msgLines: appConfig.entryMsgs.altIdMsgs }} /> </Row> <Row> <MaskedEntry {...{ ...appConfig.phoneNumberEntry, entryColors: appConfig.pageColors, entryLine: phoneNumber }} /> </Row> <Row> <SimpleKeypad {...{ keyboardName: ALT\_ID, themeName: appConfig.keyTheme, keyPressed, styleClass: keypadClass }} /> </Row> <Row> <ControlSet {...{ btnColors: appConfig.buttonColors, disabled: disableBtns, btns: [{ text: 'Submit', click: doSubmit }] }} /> </Row> </Container> );};AltIdPage.propTypes = {};  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Now everything persists smoothly everywhere across all my pages

**C1:**I don't think this answer is particularly helpful in the context of the OP. This answer is not even using useState() which was central to the OP's inquiry.

**C2:**Smooth solution, but not an answer to what was going on

**A5:**React's useEffect has its own state/lifecycle. It's related to mutation of state, and it will not update the state until the effect is destroyed.  
  
Just pass a single argument in parameters state or leave it a black array and it will work perfectly.  
  
React.useEffect(() => { console.log("effect"); (async () => { try { let result = await fetch("/query/countries"); const res = await result.json(); let result1 = await fetch("/query/projects"); const res1 = await result1.json(); let result11 = await fetch("/query/regions"); const res11 = await result11.json(); setData({ countries: res, projects: res1, regions: res11 }); } catch {} })(data)}, [setData])# or use thisuseEffect(() => { (async () => { try { await Promise.all([ fetch("/query/countries").then((response) => response.json()), fetch("/query/projects").then((response) => response.json()), fetch("/query/regions").then((response) => response.json()) ]).then(([country, project, region]) => { // console.log(country, project, region); setData({ countries: country, projects: project, regions: region }); }) } catch { console.log("data fetch error") } })()}, [setData]);  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Alternatively, you can try React.useRef() for instant change in the React hook.  
  
const movies = React.useRef(null);useEffect(() => {movies.current='values';console.log(movies.current)}, [])  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**C1:**The last code example has no need for neither async nor await as you use the Promise API. That's only needed in the first

**A6:**The closure is not the only reason.  
  
Based on the source code of useState (simplified below). Seems to me the value is never assigned right away.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
What happens is that an update action is queued when you invoke setValue. And after the schedule kicks in and only when you get to the next render, these update action then is applied to that state.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Which means even we don't have closure issue, react version of useState is not going to give you the new value right away. The new value doesn't even exist until next render.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
 function useState(initialState) { let hook; ... let baseState = hook.memoizedState; if (hook.queue.pending) { let firstUpdate = hook.queue.pending.next; do { const action = firstUpdate.action; baseState = action(baseState); // setValue HERE firstUpdate = firstUpdate.next; } while (firstUpdate !== hook.queue.pending); hook.queue.pending = null; } hook.memoizedState = baseState; return [baseState, dispatchAction.bind(null, hook.queue)]; }function dispatchAction(queue, action) { const update = { action, next: null }; if (queue.pending === null) { update.next = update; } else { update.next = queue.pending.next; queue.pending.next = update; } queue.pending = update; isMount = false; workInProgressHook = fiber.memoizedState; schedule();}  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
There's also an article explaining the above in the similar way, https://dev.to/adamklein/we-don-t-know-how-react-state-hook-works-1lp8

**A7:**I too was stuck with the same problem. As other answers above have clarified the error here, which is that useState is asynchronous and you are trying to use the value just after setState. It is not updating on the console.log() part because of the asynchronous nature of setState, it lets your further code to execute, while the value updating happens on the background. Thus you are getting the previous value. When the setState is completed on the background it will update the value and you will have access to that value on the next render.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
If anyone is interested to understand this in detail. Here is a really good Conference talk on the topic.  
  
https://www.youtube.com/watch?v=8aGhZQkoFbQ

**C1:**clear & simple thank you!

**A8:**I found this to be good. Instead of defining state (approach 1) as, example,  
  
const initialValue = 1;  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
const [state,setState] = useState(initialValue)  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Try this approach (approach 2),  
  
const [state = initialValue,setState] = useState()  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
This resolved the rerender issue without using useEffect since we are not concerned with its internal closure approach with this case.  
  
P.S.: If you are concerned with using old state for any use case then useState with useEffect needs to be used since it will need to have that state, so approach 1 shall be used in this situation.

**C1:**This answer is not useful. With regard to re-rendering and captured closure values this approach does not make the slightest difference. The only difference it ever makes it when the state value is deliberately set to undefined, in this case you will gain the initialValue again. Which is just a confusing way of doing it, because you could just set it to the initial value without extra steps.

**C2:**Approach 2 pollutes the global space. Approach 1, as mentioned, is an antipattern at best.

**A9:**If we have to update state only, then a better way can be if we use the push method to do so.  
  
Here is my code. I want to store URLs from Firebase in state.  
  
const [imageUrl, setImageUrl] = useState([]);const [reload, setReload] = useState(0);useEffect(() => { if (reload === 4) { downloadUrl1(); }}, [reload]);const downloadUrl = async () => { setImages([]); try { for (let i = 0; i < images.length; i++) { let url = await storage().ref(urls[i].path).getDownloadURL(); imageUrl.push(url); setImageUrl([...imageUrl]); console.log(url, 'check', urls.length, 'length', imageUrl.length); } } catch (e) { console.log(e); }};const handleSubmit = async () => { setReload(4); await downloadUrl(); console.log(imageUrl); console.log('post submitted');};  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
This code works to put URLs in state as an array. This might also work for you.

**C1:**.push mutates the current state, which is a bad practice in React. Here's the proper way to update a state array.

**C2:**Calling setImageUrl in a loop is another bad practice, which will trigger a new render for each time it was called since it's not batched when called asynchronously (outside React's lifecycle). The proper way would be to build the new array and then call setImageUrl only once.

**C3:**Also, using await in a loop like that is inefficient. Something like Promise.all would improve this.

**A10:**I want the upated value of state immediately to save it in DB and I come accros this question. Calling updater just to get the latest value.  
  
const [pin, setPin] = useState(0);setPin(pin ? 0 : 1);setPin((state) => { console.log(state); // 1 //Here I called the DB method and passed the state as param in mehtod return state;});  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**A11:**With custom hooks from my library, you can wait for the state values to update:  
  
 ● useAsyncWatcher(...values):watcherFn(peekPrevValue: boolean)=>Promise - is a promise wrapper around useEffect that can wait for updates and return a new value and possibly a previous one if the optional peekPrevValue argument is set to true.  
  
(Live Demo)  
  
 import React, { useState, useEffect, useCallback } from "react"; import { useAsyncWatcher } from "use-async-effect2"; function TestComponent(props) { const [counter, setCounter] = useState(0); const [text, setText] = useState(""); const textWatcher = useAsyncWatcher(text); useEffect(() => { setText(`Counter: ${counter}`); }, [counter]); const inc = useCallback(() => { (async () => { await new Promise((resolve) => setTimeout(resolve, 1000)); setCounter((counter) => counter + 1); const updatedText = await textWatcher(); console.log(updatedText); })(); }, []); return ( <div className="component"> <div className="caption">useAsyncEffect demo</div> <div>{counter}</div> <button onClick={inc}>Inc counter</button> </div> ); } export default TestComponent;  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
 ● useAsyncDeepState is a deep state implementation (similar to this.setState (patchObject)) whose setter can return a promise synchronized with the internal effect. If the setter is called with no arguments, it does not change the state values, but simply subscribes to state updates. In this case, you can get the state value from anywhere inside your component, since function closures are no longer a hindrance.  
  
(Live Demo)  
  
import React, { useCallback, useEffect } from "react";import { useAsyncDeepState } from "use-async-effect2";function TestComponent(props) { const [state, setState] = useAsyncDeepState({ counter: 0, computedCounter: 0 }); useEffect(() => { setState(({ counter }) => ({ computedCounter: counter \* 2 })); }, [state.counter]); const inc = useCallback(() => { (async () => { await new Promise((resolve) => setTimeout(resolve, 1000)); await setState(({ counter }) => ({ counter: counter + 1 })); console.log("computedCounter=", state.computedCounter); })(); }); return ( <div className="component"> <div className="caption">useAsyncDeepState demo</div> <div>state.counter : {state.counter}</div> <div>state.computedCounter : {state.computedCounter}</div> <button onClick={() => inc()}>Inc counter</button> </div> );}  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**A12:**var [state,setState]=useState(defaultValue)useEffect(()=>{ var updatedState setState(currentState=>{ // Do not change the state by get the updated state updateState=currentState return currentState }) alert(updateState) // the current state.})  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**C1:**Don't do that. setState's setter callback should be pure. Also, here, updatedState would always be undefined.

**C2:**@EmileBergeron do you have a link to the documentation stating that the callbacks should be free of side effects?

**C3:**I dont have the link on hand, but it's documented alongside strict mode, which helps identify unwanted side-effects.

**A13:**Without any addtional NPM package  
  
//...const BackendPageListing = () => { const [ myData, setMyData] = useState( { id: 1, content: "abc" }) const myFunction = ( x ) => { setPagenateInfo({ ...myData, content: x }) console.log(myData) // not reflecting change immediately let myDataNew = {...myData, content: x }; console.log(myDataNew) // Reflecting change immediately } return ( <> <button onClick={()=>{ myFunction("New Content")} }>Update MyData</button> </> )  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**A14:**Not saying to do this, but it isn't hard to do what the OP asked without useEffect.  
  
Use a promise to resolve the new state in the body of the setter function:  
  
const getState = <T>( setState: React.Dispatch<React.SetStateAction<T>>): Promise<T> => { return new Promise((resolve) => { setState((currentState: T) => { resolve(currentState); return currentState; }); });};  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
And this is how you use it (example shows the comparison between count and outOfSyncCount/syncCount in the UI rendering):  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
const App: React.FC = () => { const [count, setCount] = useState(0); const [outOfSyncCount, setOutOfSyncCount] = useState(0); const [syncCount, setSyncCount] = useState(0); const handleOnClick = async () => { setCount(count + 1); // Doesn't work setOutOfSyncCount(count); // Works const newCount = await getState(setCount); setSyncCount(newCount); }; return ( <> <h2>Count = {count}</h2> <h2>Synced count = {syncCount}</h2> <h2>Out of sync count = {outOfSyncCount}</h2> <button onClick={handleOnClick}>Increment</button> </> );};  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**C1:**setState setter callback should be pure.

**C2:**I am in favor of emphasizing pure functions, but that link isn't specific to the useState callback (there are use cases for not using a pure function as the callback). Also, although this isn't "pure", it doesn't actually change any state either. I'm not saying this is the best way (I don't use it), just that it provides an alternative solution to the OPs question

**C3:**Using count and or callbacks hinders the design method behind usestate itself I'm afraid.

**C4:**Again, not saying this is the best way or to use it, just saying it works. Would be more useful to say what the issues would be in using it with a reproducible example as opposed to disregarding it from a design theory level

**C5:**"A design theory level" seems to be a perfectly valid reason to criticize something. Just because something works or is possible doesn't make it worth putting out there. Folks might actually use this pattern in spite of the warnings. There are much better ways to deal with the "problem" (it's not exactly a problem to begin with, just seems like that to folks who are unused to asynchronous code).

**A15:**Use the Background Timer library. It solved my problem.  
  
const timeoutId = BackgroundTimer.setTimeout(() => { // This will be executed once after 1 seconds // even when the application is the background console.log('tac');}, 1000);  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**C1:**Adding a delay is not an actual solution.. its just a workaround.. even then, you dont need a library for that when you can just use a simple setTimeout

**A16:**// replacereturn <p>hello</p>;// withreturn <p>{JSON.stringify(movies)}</p>;  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]   
  
Now you should see, that your code actually does work. What does not work is the console.log(movies). This is because movies points to the old state. If you move your console.log(movies) outside of useEffect, right above the return, you will see the updated movies object.  
  
WARN: THIS PARAGRAPH CONTAINS TAG: [CODE]

**C1:**Not sure why this answer is down voted heavily, it goes tell how to get "expected" console.log value by putting it outside useState function. Simple and sweet, if somebody wants to know why it is happening like that can refer to above detailed clarifications

**C2:**Good try though