

CS 4530: Fundamentals of Software Engineering

Module 09: React Hook Patterns

Jon Bell, Adeel Bhutta and Mitch Wand
Khoury College of Computer Sciences

Learning Objectives for this Module

- By the end of this module, you should be able to:
 - Explain the basic use cases for useEffect
 - Explain when a useEffect is executed, and when its return value is executed
 - Construct simple custom hooks and explain why they are useful.
 - Be able to explain the three core steps of a test (assemble, act, assess) can map to UI component testing

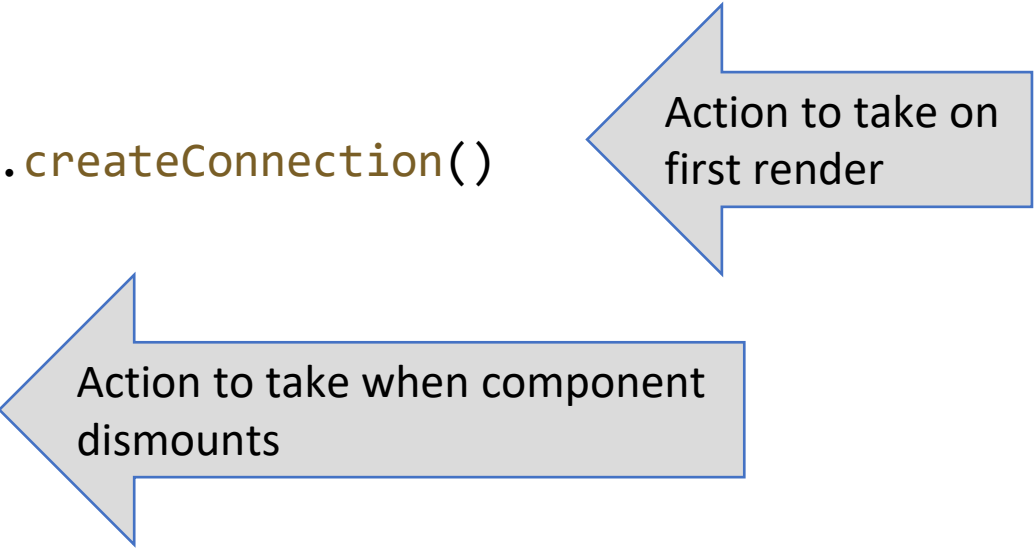
useEffect is a mechanism for synchronizing a component with an external system

```
import { clockServer } from './clock.js';

function ClockClient() {

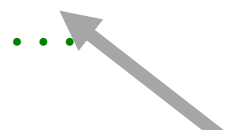
  useEffect(() => {
    const connection = clockServer.createConnection()
    connection.connect();

    return () => {
      connection.disconnect();
    };
  }, []);
  // ...
}
```



Action to take on first render

Action to take when component dismounts



Empty array says: do this on first render only

<https://react.dev/reference/react/useEffect>

An external system means any piece of code that's not inside your React component

- An event in the lifecycle of a component, like `render`.
- A timer managed with `setInterval` and `clearInterval`
- An event subscription like a chat server
- A call to fetch data from an external web site
- An external animation library
- A piece of business logic in an app that is external to your component

A real example: connecting a component to a self-ticking clock

app/Apps/SimpleClockDisplay.tsx

```
export function ClockDisplay(props: {  
  //  
}) {  
  const [localTime, setLocalTime] = useState(0)  
  const incrementLocalTime = () => setLocalTime(localTime => localTime + 1)  
  const listener1 = () => { incrementLocalTime() }  
  const clock = SingletonClockFactory.instance(1000)  
  
  useEffect(() => {  
    clock.addListener(listener1)  
    return () => {  
      clock.removeListener(listener1)  
    }  
  }, [])
```

On first render, add this listener to the clock

On dismount, remove the listener.

Simple example of using an external service: a self-ticking clock

src/app/Apps/SimpleClockDisplayApp.tsx

```
import { ClockDisplay } from './SimpleClockDisplay'

function doNothing() { }

export default function App() {
  return (<VStack>
    <ClockDisplay key={1} name={'Clock A'}
      handleAdd={doNothing} handleDelete={doNothing}/>
    <ClockDisplay key={2} name={'Clock B'}
      handleAdd={doNothing} handleDelete={doNothing} />
    <ClockDisplay key={3} name={'Clock C'}
      handleAdd={doNothing} handleDelete={doNothing}/>
  </VStack>)
}
```

Next, let's look at the clock

```
type Listener = () => void
```

```
class Clock {  
  public time = 0  
  private _listeners: Listener[] = []  
  private _notifyAll() {this._listeners.forEach(eachListener => {eachListener()})}  
  
  public addListener(listener: Listener) {---}  
  public removeListener(listener: Listener) {---}  
}  
  
  public get nListeners () {return this._listeners.length}  
  
  private _timer : NodeJS.Timeout  
  
  public constructor(interval: number) {  
    this._timer = setInterval(() => {  
      this._tick();  
    }, interval);  
  }  
  
  private _tick() {  
    this.time++;  
    this._notifyAll();  
  }  
  
  public _stop() {  
    clearInterval(this._timer);  
  }  
}
```

...and we'll make it a singleton in the usual way

src/app/Classes/ClockWithListeners.ts

```
export default class SingletonClockFactory {  
    private static theClock: Clock | undefined = undefined  
  
    private constructor () {SingletonClockFactory.theClock = undefined}  
  
    public static instance (interval:number) : Clock {  
        if (SingletonClockFactory.theClock === undefined) {  
            SingletonClockFactory.theClock = new Clock(interval)  
        }  
        return SingletonClockFactory.theClock  
    }  
}
```


Next is <ClockDisplay>

```
import SingletonClockFactory from '../Classes/ClockWithListeners';

export function ClockDisplay(props: {
  name: string, key: number,
  handleDelete: () => void, handleAdd: () => void,
}) {
  const [localTime, setLocalTime] = useState(0)
  const incrementLocalTime = () => setLocalTime(localTime => localTime + 1)
  const clock = SingletonClockFactory.instance(1000)

  useEffect(() => {
    const listener1 = () => { incrementLocalTime() }
    clock.addListener(listener1)
    return () => {
      clock.removeListener(listener1)
    }
  }, [])
}
```

ClockDisplay, part 2

```
return (  
  <HStack>  
    <Box>Clock: {props.name}</Box>  
    <Box>Time = {localTime}</Box>  
    <Box>nlisteners = {clock.nListeners}</Box>  
    <IconButton onClick={props.handleDelete}  
      icon={<AiOutlineDelete />} />  
    <IconButton onClick={props.handleAdd}  
      icon={<AiOutlinePlus />} />  
  </HStack>  
)  
}
```

useEffect's Dependencies Control Its Execution

- `useEffect` takes an optional array of dependencies
- The effect is only executed if the values in the dependency change (e.g. by a setter)
- Special Cases:
 - `[]` means run only on first render
 - No argument means run on every render

Example (Part 1)

```
export default function App() {  
  const [n, setN] = useState(0)  
  const [m, setM] = useState(0)  
  
  // runs only on first render.  
  useEffect(() => {  
    console.log('useEffect #1 is run only on first render')}, [])  
  
  useEffect(() => {  
    console.log('useEffect #2N is run only when n changes')}, [n])  
  
  useEffect(() => {  
    console.log('useEffect #2M is run when m changes')}, [m])  
  
  // runs on every render  
  useEffect(() => {  
    console.log('useEffect #3A is called on every render')})  
  
  // runs on every render  
  useEffect(() => {  
    console.log('useEffect #3B is called on every render')})  
}
```

Example (part 2)

```
// runs on every render
useEffect(() => {
  console.log('useEffect #3C is called on every render') })

// observe that effects run in order of definition

return (
  <VStack>
    <Heading>useEffect demo #1</Heading>
    <Text> n is {n} </Text>
    <Button onClick={() => setN((n) => n + 1)}>Increment n</Button>
    <Text> m is {m} </Text>
    <Button onClick={() => setM((m) => m + 1)}>Increment m</Button>
    <Text> count is {count} </Text>
  </VStack>
)
```

When is the cleanup function executed?

- The cleanup function is executed when the page dismounts.
- Demonstrating this takes a little effort:
 - Let's build a list of clock displays!
 - We can add new clock displays
 - We can delete a clock display
 - When we delete a clock display, the display is dismounted, and the cleanup function is run.

The Code (Part 1)

```
type ClockDisplayData = {key:number, name:string, noisyDelete?:boolean}
import { ClockDisplay } from './SimpleClockDisplay'
function makeClockDisplayData(key:number) {
  return {key:key, name:'clock ' + key, noisyDelete: true}
}
export default function App () {
  const [clockDisplays, setClockDisplays] = useState<ClockDisplayData[]>([])
  const [nextKey, setNextKey] = useState(1)

  function handleAdd() {
    const newDisplay = makeClockDisplayData(nextKey)
    setClockDisplays(clockDisplays.concat(newDisplay))
    setNextKey(nextKey+1)
  }
  function handleDelete(targetKey:number) { --- } // not so interesting

  // add a clock display for the first render
  useEffect(() => {handleAdd()}, [])
```

The Code (part 2)

```
function displayOneClock(clockDisplayData:ClockDisplayData) {
  return (
    <Tr key={clockDisplayData.key}>
      <Td>
        <ClockDisplay name={clockDisplayData.name} key={clockDisplayData.key}
          handleDelete={() => handleDelete(clockDisplayData.key)}
          handleAdd={handleAdd}
          noisyDelete={clockDisplayData.noisyDelete}
        />
      </Td>
    </Tr>
  )
}

return (
  <VStack>
    <Heading>Array of Clock Displays</Heading>
    <Table>
      <Tbody>
        {clockDisplays.map((clockDisplayData) => displayOneClock(clockDisplayData))}
      </Tbody>
    </Table>
  </VStack>
)
```


Custom Hooks

- REACT lets us combine `useState` and `useEffect` to build custom hooks.

useFirstRender

```
import * as React from 'react';
import { useState, useEffect } from 'react'

export function useFirstRender(action:() => void) {
  useEffect(() => {
    action()
  }, [])
}
```

Definition

```
import { useFirstRender } from '../Hooks/useFirstRender'
```

```
// illustration of useFirstRender
useFirstRender(() => {
  console.log('useFirstRender #1 is run only on first render')
})
```

Use

A more substantial example: useClock

```
import { useEffect } from 'react';
import SingletonClockFactory, { Clock } from '../Classes/ClockWithListeners';

export function useClock(listener1: () => void): Clock {
  const clock = SingletonClockFactory.instance(1000)
  useEffect(() => {
    clock.addListener(listener1)
    return () => {
      clock.removeListener(listener1)
    }
  }, []);
  return clock
}
```

Using useClock

```
import { useClock } from '../Hooks/useClock';

export function ClockDisplay(props: {
  name: string, key: number,
  handleDelete: () => void, handleAdd: () => void,
  noisyDelete?: boolean
}) {
  const [localTime, setLocalTime] = useState(0)
  const incrementLocalTime = () => setLocalTime(localTime => localTime + 1)

  const clock = useClock(incrementLocalTime)

  return (
    <HStack>
      <Box>Clock: {props.name}</Box>
      <Box>Time = {localTime}</Box>
      <Box>nlisteners = {clock.nListeners}</Box>
      <IconButton --- />
      <IconButton --- />
    </HStack>
  )
}
```

The Rules of Hooks

1. Only call hooks at the top level

- Not within loops, inside conditions, or nested functions
- Rationale: The order of hooks called must always be the same each time a component renders

2. Only call hooks from React Components or Custom Hooks

- Not from any other helper methods or classes
- Rationale: React must know the component that the call to the hook is associated with

```
export function LikeButton() {  
  const [isLiked, setIsLiked] = useState(false);  
  const [count, setCount] = useState(0);  
  ...  
}
```

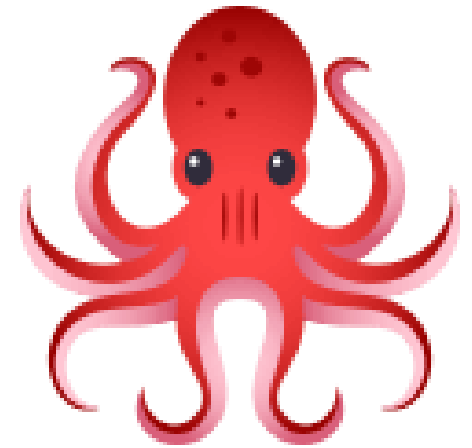
React knows which `useState` is which by tracking calls to them from components in the render tree

We Use Two ESLint Rules for React Hooks

- You should not violate the rules of hooks. These linter plugins help detect violations
- React-hooks/rules-of-hooks
 - Enforces that hooks are only called from React functional components or custom hooks
- React-hooks/exhaustive-deps
 - Enforces that all variables used in useEffects are included as dependencies

Testing React components

- Render components into a “virtual DOM”
 - Just like browser would, but no browser
- Interact with components by “firing events” like a user would
 - Click, enter text, etc. on DOM nodes, just like a user would in a browser
- Inspect components that are rendered
 - Tests specify how to “find” a component in that virtual DOM



“Testing Library”

<https://testing-library.com>

Compatible with many UI libraries
and many testing frameworks

Write UI component tests just like any other test

Follow the generic testing model from Module 2:

- Assemble the situation: —
 - Set up system under test (SUT) to get the state ready
 - [Optional: Prepare collaborators]
- Act - Apply the operation inputs. —
- Assess - Check the outputs, verify the state change, handle the behavior —

1: Render component into a testing DOM tree

2: Interact with the rendered component

3: Check the rendered result

Rendering Components in Virtual DOM

```
let deleteCalled = false;
beforeEach(() => {
  deleteCalled = false;
  render(
    <PersonalizedLikableDeletableHello name="Ripley"
      onDelete={() => { deleteCalled = true; }} /> );
  });
```

- The *render* function prepares our component for testing:
 - Creates a virtual DOM
 - Instantiates our component, mounts it in DOM
 - Mocks all behavior of the core of React
 - Allows us to inspect the rendered result in the screen import

Inspecting Rendered Components: By Text

```
test("It renders the greeting", () => {  
  const greeting = screen.getByText(/Hello, Ripley!/);  
  expect(greeting).toBeInTheDocument();  
})
```

First approach to inspect rendered components: match by text

Acting on Rendered Components: *userEvent*

- Testing Library provides `userEvent.<event>` methods
 - `userEvent.type` (`newItemTextField`, `"Write a better test input"`) ;
`userEvent.click` (`newItemButton`) ;
Also: `change`, `keyDown`, `keyUp`, etc
- These methods **simulate user behavior**:
 - Before clicking: `MouseOver`, `MouseMove`, `MouseDown`, `MouseUp`
 - `type` will click the (virtual) text box, then provide characters one-at-a-time

Inspecting Rendered Components: ARIA label

```
if (isLiked) {
  likeButton = (<IconButton aria-label="unlike"
    icon={<AiFillHeart />} onClick={() => setIsLiked(false)} /> );
} else {
  likeButton = (<IconButton aria-label="like"
    icon={<AiOutlineHeart />} onClick={() => setIsLiked(true)} /> );
}
```

Test

```
test("Like button defaults to not liked, clicking it likes, clicking again unlikes", () => {
  const likeButton = screen.getByLabelText("like");
  fireEvent.click(likeButton);
  const unlikeButton = screen.getByLabelText("unlike");
  fireEvent.click(unlikeButton);
  expect(screen.getByLabelText("like")).toBeInTheDocument();
});
```

3 Tiers for Inspecting Rendered Components

- Queries that reflect how every user interacts with your app
 - byRole – Using accessibility tree
 - byLabelText – Using label on form fields
 - byPlaceholderText – Using placeholder text on form field
 - byText – By exact text in an element
 - byDisplayValue – By current value in a form field
- Queries that reflect how *some* users interact with your app
 - byAltText – By alt text, usually not presented to sighted users
 - byTitle - By a “title” attribute, usually not presented to sighted users
- Queries that have nothing to do with how a user interacts with app
 - byTestId

But wait, there's more...

- You may want different behavior when there are different numbers of matches to a query.
- Testing-library includes a query called Find, which is *async* and will return a promise to wait for all rendering to complete

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
 - Explain the basic use cases for `useEffect`
 - Explain when a `useEffect` is executed, and when its return value is executed
 - Construct simple custom hooks and explain why they are useful.
 - Be able to explain the three core steps of a test (assemble, act, assess) can map to UI component testing