CS 4530: Fundamentals of Software Engineering
Module 09: React Hook Patterns

Adeel Bhutta, Joydeep Mitra 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

Lesson 9.1 useEffect

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 redisplay.
- 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: a display that connects to a self-ticking clock

src/app/Components/SimpleClockDisplay.tsx

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

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

The parent provides the clock

On first render, add this listener to the clock

On dismount, remove the listener.

Display logic will come later…

src/Apps/ThreeClocks/App.tsx

Our app will have three displays of the clock

```
import * as React from 'react'; import { useState } from 'react';
import ClockDisplay from '../../Components/ClockDisplay'
import SingletonClock from '../../Classes/SingletonClockFactory'
function doNothing() { }

export default function App() {
  const [clock, _] = useState(SingletonClock.getInstance(1000));

  return (
    <VStack>
      <ClockDisplay key={1} name={"Clock A"} clock={clock}
        handleAdd={doNothing}handleDelete={doNothing}
      />
      <ClockDisplay key={2} name={"Clock B"} clock={clock}
        handleAdd={doNothing} handleDelete={doNothing}
      />
      <ClockDisplay key={3} name={"Clock C"} clock={clock}
        handleAdd={doNothing} handleDelete={doNothing}
      />
    </VStack>
  );
}
```

src/Classes/SingletonClockFactory.ts

Next, let's look at the clock

```
type Listener = () => void

class Clock implements IClock{

    private _listeners: Listener[] = []
    private _notifyAll() {this._listeners
       .forEach(eachListener => {eachListener()})}

    public addListener(listener: Listener) {---}
    public removeListener(listener: Listener) {---}

    get nListeners () {return this._listeners.length}

    private _timer : NodeJS.Timeout
    private _interval : number
    public id : string

    public constructor(interval: number) {
        this.id = nanoid(4)
        this._interval = interval;
        this.start()
    }

    public start() {
        console.log(`Clock ${this.id} starting`)
        this._timer = setInterval(() => {
            this._tick();
        }, this._interval);

    }

    private _tick() {
        this._notifyAll();
        }

    public stop() {
        console.log(`Clock ${this.id} stopping`)
        clearInterval(this._timer);

    }

}
```

We'll make the clock a singleton in the usual way

src/Classes/SingletonClockFactory.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
    }

}
```

src/Components/SimpleClockDisplay.tsx

Let's look at <ClockDisplay> again

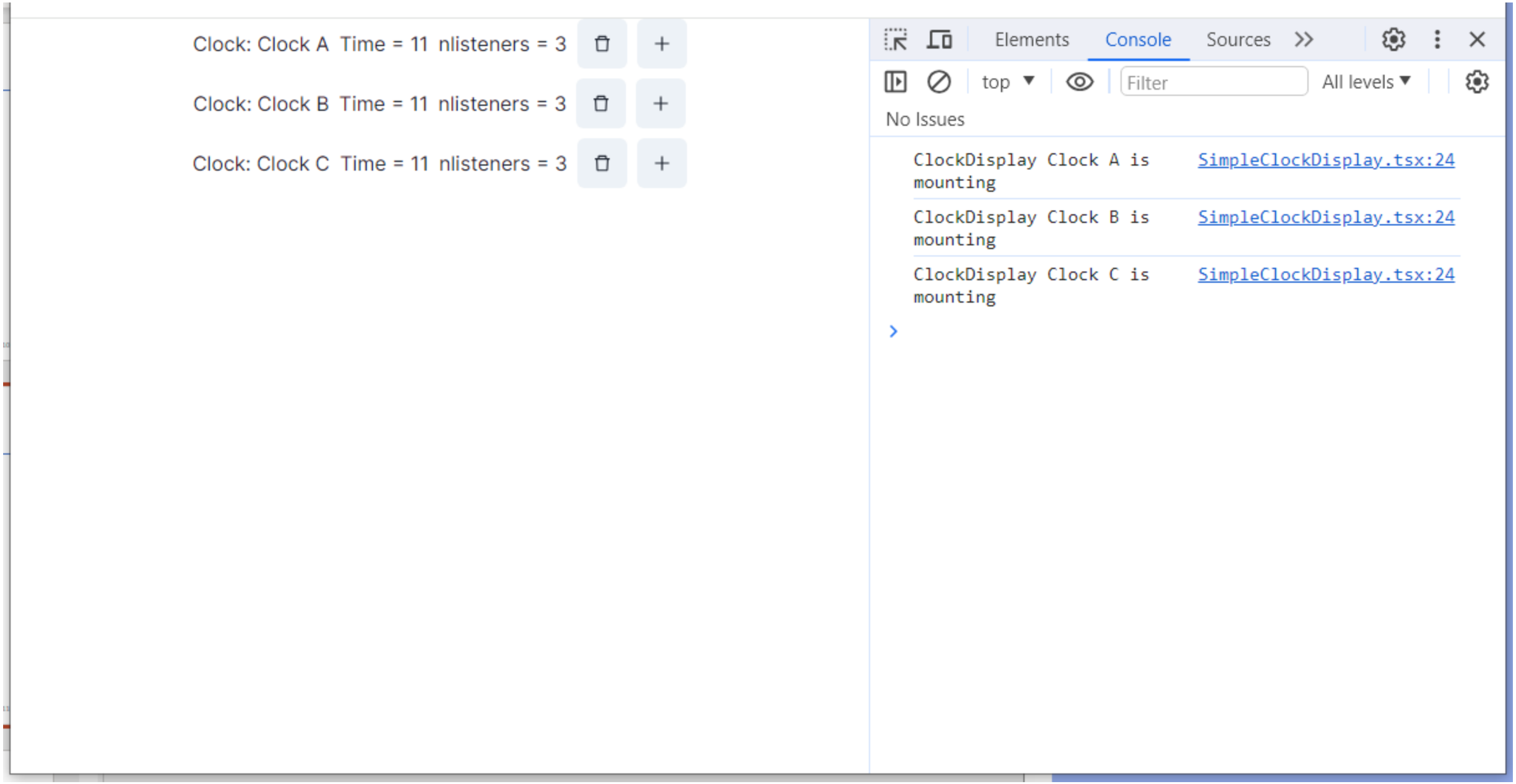
```
export default function ClockDisplay(props: {
  name: string; key: number;  clock: IClock;
  handleDelete: () => void;  handleAdd: () => void;
}): JSX.Element {
  const [localTime, setLocalTime] = useState(0);
  const incrementLocalTime = () => { setLocalTime((localTime) => localTime + 1); };

  const listener1 = () => { incrementLocalTime(); };
  const clock = props.clock;

  useEffect(() => {
    clock.addListener(listener1);
    console.log(`ClockDisplay ${props.name} is mounting`);
    return () => {
      console.log("ClockDisplay " + props.name + " is unmounting");
      clock.removeListener(listener1);
    };
  }, []);
```

business logic

ClockDisplay, part 2: the display logic

```
function handleStop() { clock.stop(); }
function handleStart() { clock.start(); }
```

```
  return (
    <HStack>
      <Box>Clock: {props.name}</Box>
      <Box>Clock ID: {clock.id} </Box>
      <Box>Time = {localTime}</Box>
      <Box>nlisteners = {clock.nListeners}</Box>
      <Button aria-label={"start"} onClick={handleStart}>Start</Button>
      <Button aria-label={"stop"} onClick={handleStop}>Stop</Button>
      <IconButton aria-label={"delete"} onClick={props.handleDelete}
                  icon={<AiOutlineDelete />}
      />
      <IconButton aria-label={"add"} onClick={props.handleAdd}
                  icon={<AiOutlinePlus />}
      />
    </HStack>
  );
```


Clock: Clock A Time = 11 nlisteners = 3
Clock: Clock B Time = 11 nlisteners = 3
Clock: Clock C Time = 11 nlisteners = 3
Elements
Console
Sources
top
Filter
All levels
No Issues
ClockDisplay Clock A is mounting
SimpleClockDisplay.tsx:24
ClockDisplay Clock B is mounting
SimpleClockDisplay.tsx:24
ClockDisplay Clock C is mounting
SimpleClockDisplay.tsx:24

useEffect's Dependencies Control Its Execution

- useEffect takes an optional array of dependencies
- The effect is only executed if one or more of 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

src/app/Apps/useEffect-demo.tsx

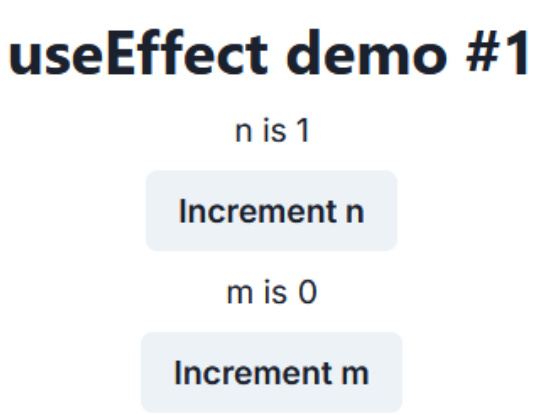
Example (Part 1)

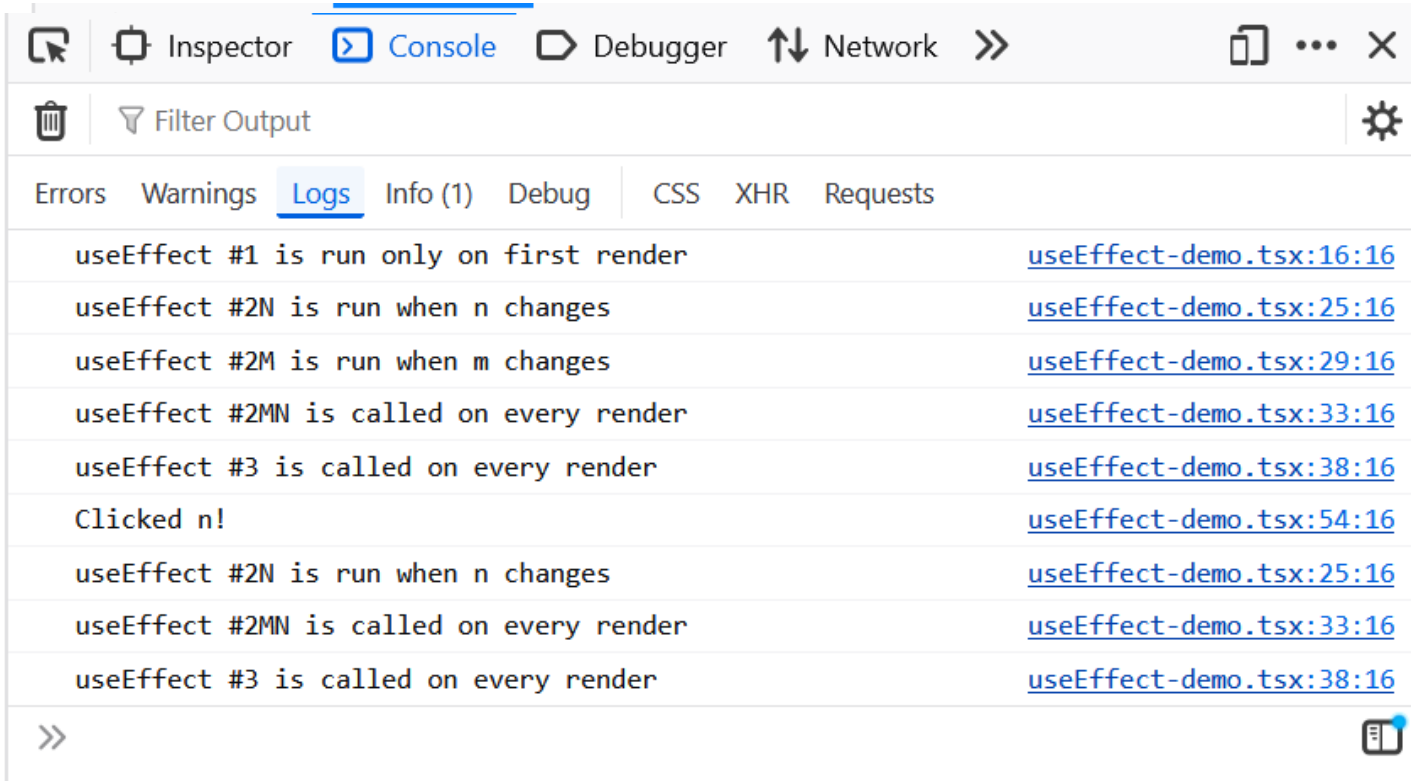
```
export default function App() {
    const [n, setN] = useState(0)
    const [m, setM] = useState(0)

    // 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])

    useEffect(() => {
        console.log('useEffect #2MN is run when m or n changes')
    }, [m,n]

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

    // observe that effects run in order of definition
```

Example (part 2)

```
    function onClickN() {
        console.log('Clicked n!');
        setN(n => n + 1);
    }

    function onClickM() {
        console.log('Clicked m!');
        setM(m => m + 1);
    }

    return (
        <VStack>
            <Heading>useEffect demo #1</Heading>
            <Text> n is {n} </Text>
            <Button onClick={onClickN}>Increment n</Button>
            <Text> m is {m} </Text>
            <Button onClick={onClickM}>Increment m</Button>
        </VStack>

}
```

Demo


useEffect demo #1
n is 1
Increment n
m is 0
Increment m


Inspector
Console
Debugger
Network
Filter Output
Errors
Warnings
Logs
Info (1)
Debug
CSS
XHR
Requests
useEffect #1 is run only on first render useEffect-demo.tsx:16:16
useEffect #2N is run when n changes useEffect-demo.tsx:25:16
useEffect #2M is run when m changes useEffect-demo.tsx:29:16
useEffect #2MN is called on every render useEffect-demo.tsx:33:16
useEffect #3 is called on every render useEffect-demo.tsx:38:16
Clicked n! useEffect-demo.tsx:54:16
useEffect #2N is run when n changes useEffect-demo.tsx:25:16
useEffect #2MN is called on every render useEffect-demo.tsx:33:16
useEffect #3 is called on every render useEffect-demo.tsx:38:16

When is the cleanup function executed?

- In general, the cleanup function is executed sometime before the next time the hook is run.
- For the first-time-only case, this means when the component is dismounted.
- Let's look at useEffect demo again, this time with noisy cleanups.

src/Apps/useEffect-demoWithCleanUps.tsx

```
function cleanup(message: string) {return () => {console.log('cleanup: ' + message)}}

export default function App() {
    const [n, setN] = useState(0)
    const [m, setM] = useState(0)

    useEffect(() => {
        console.log('useEffect #1 is run only on first render')
        return cleanup('useEffect #1')
    }, [])

    useEffect(() => {
        console.log('useEffect #2N is run only when n changes')
        return cleanup('useEffect #2N')
    }, [n])

    ... // other effects
```

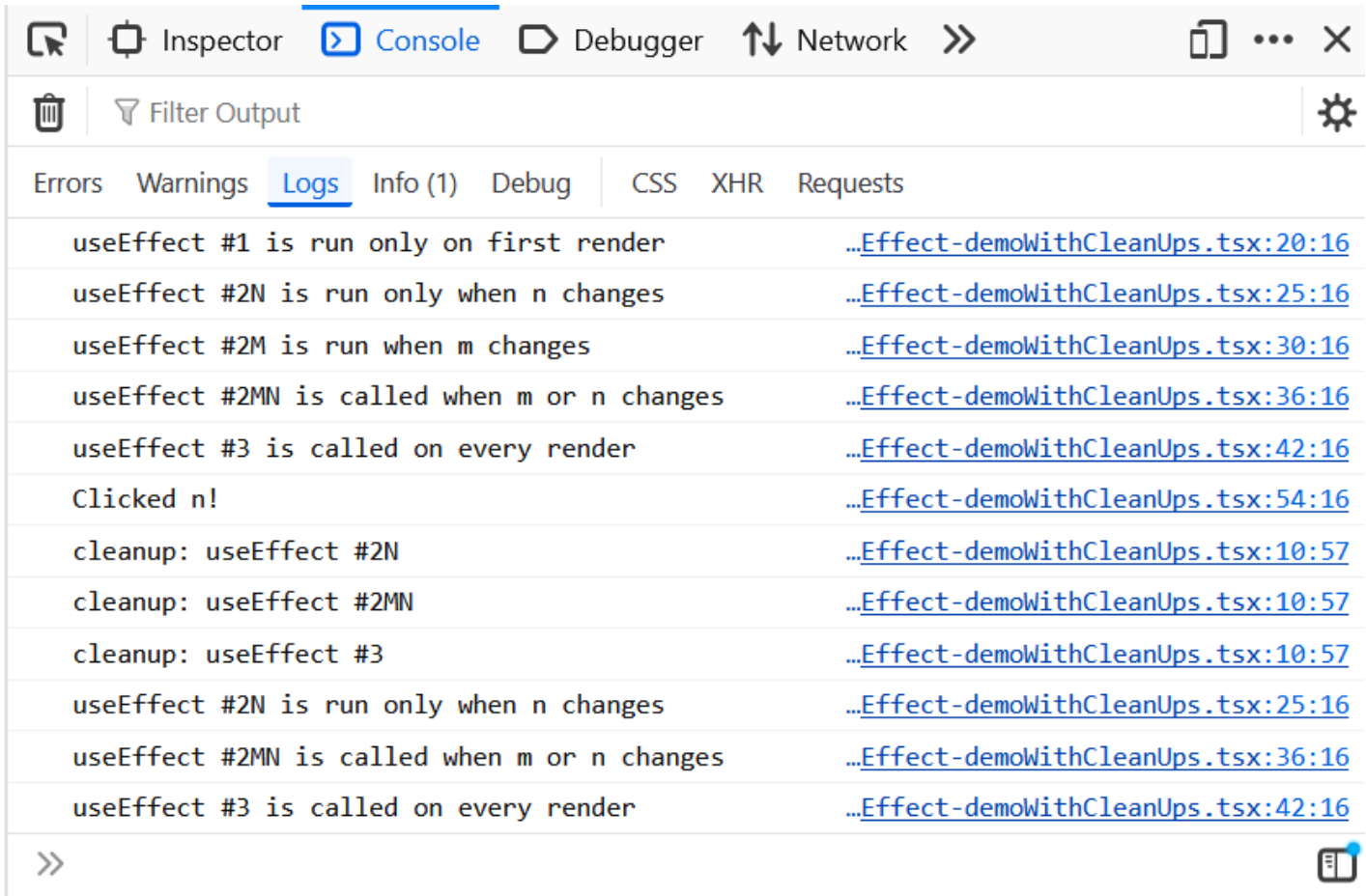
useEffect demo with CleanUps

n is 1

Increment n

m is 0

Increment m



Communication with an (Actual) External System

- In the previous example, everything was present in the one codebase, including the clock.
- More common situation is that the user interface components need to communicate with a remote system, not necessarily in the same codebase.

useEffect and and fetch working together

```
// the value of globalCount will be passed to the children
const [globalCount, setGlobalCount] = useState(0);
const [sumLocalCounts, setSumLocalCounts] = useState(0);
const [localCountA, setLocalCountA] = useState(0);
const [localCountB, setLocalCountB] = useState(0);


// when either local count changes, recalulate the sum
useEffect(() => {
  recalculateGlobalCount();
}, [localCountA, localCountB]);

async function recalculateGlobalCount() {
  console.log("recalculating");
  const response = await fetch(`/sum/${localCountA}/${localCountB}`);
  const sum = await response.json();
  console.log("sum", sum);
  setGlobalCount(sum.sum);
  setSumLocalCounts(localCountA + localCountB);
  return;
}
```

Lesson 9.2 Custom Hooks

Custom Hooks

- REACT lets us combine useState and useEffect to build custom hooks.
- Custom Hooks let us separate business logic from display logic

Example: useClock

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

src/Components/SimpleClockDisplayWithUseClock.tsx

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:IClock = useClock(incrementLocalTime)

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

}
```

src/Apps/ToDoApp/App.tsx

A somewhat larger example: ToDoList

```
export default function ToDoApp () {
  const [todoList,setTodolist] = useState<ToDoItem[]>([])
  const [itemKey,setItemKey] = useState<number>(0)   // first unused key

  function handleAdd (title:string, priority:string) {
    if (title === '') {return}   // ignore blank button presses
    setTodolist(todoList.concat({title: title, priority: priority, key: itemKey}))
    setItemKey(itemKey + 1)
  }

  function handleDelete(targetKey:number) {
    const newList = todoList.filter(item => item.key != targetKey)
    setTodolist(newList)
  }
```

business logic

```
  return (
  <VStack>
    <Heading>TODO List</Heading>
    <ToDoItemEntryForm onAdd={handleAdd}/>
    <ToDoListDisplay items={todoList} onDelete={handleDelete}/>
  </VStack>
  )
}
```

display logic

src/Apps/ToDoAppWithCustomHooks/App.tsx

Refactoring ToDoList

```
export default function ToDoApp () {

  const {todoList, handleAdd, handleDelete} = useToDoItemList()


  return (
  <VStack>
    <Heading>TODO List</Heading>
    <ToDoItemEntryForm onAdd={handleAdd}/>
    <ToDoListDisplay items={todoList} onDelete={handleDelete}/>
  </VStack>
  )
}
```

business logic is encapsulated

src/Apps/ToDoAppWithCustomHooks/useToDoItemList.tsx

The hook encapsulates the business logic

```
export default function useToDoItemList () {
  const [todoList,setTodolist] = useState<ToDoItem[]>([])
  const [itemKey,setItemKey] = useState<number>(0)   // first unused key

  function handleAdd (title:string, priority:string) {
    if (title === '') {return}   // ignore blank button presses
    setTodolist(todoList.concat({title: title, priority: priority, key: itemKey}))
    setItemKey(itemKey + 1)
  }

  function handleDelete(targetKey:number) {
    const newList = todoList.filter(item => item.key != targetKey)
    setTodolist(newList)
  }

  return {todoList: todoList, handleAdd: handleAdd, handleDelete: handleDelete}
}
```

The hook is like a class managing a piece of state

```
export default function useToDoItemList () {
  const [todoList,setTodolist] = useState<ToDoItem[]>([])
  const [itemKey,setItemKey] = useState<number>(0)   // first unused key

  function handleAdd (title:string, priority:string) {
    if (title === '') {return}   // ignore blank button presses
    setTodolist(todoList.concat({title: title, priority: priority, key: itemKey}))
    setItemKey(itemKey + 1)
  }

  function handleDelete(targetKey:number) {
    const newList = todoList.filter(item => item.key != targetKey)
    setTodolist(newList)
  }

  return {todoList: todoList, handleAdd: handleAdd, handleDelete: handleDelete}
}
```

handleAdd and handleDelete are the only methods for manipulating the state

The hook's state becomes part of its user's state.

```
export default function useToDoItemList () {
  const [todoList,setTodolist] = useState<ToDoItem[]>([])
  const [itemKey,setItemKey] = useState<number>(0)   // first unused key

  function handleAdd (title:string, priority:string) {
    if (title === '') {return}   // ignore blank button presses
    setTodolist(todoList.concat({title: title, priority: priority, key: itemKey}))
    setItemKey(itemKey + 1)
  }

  function handleDelete(targetKey:number) {
    const newList = todoList.filter(item => item.key != targetKey)
    setTodolist(newList)
  }

  return {todoList: todoList, handleAdd: handleAdd, handleDelete: handleDelete}
}
```

calling these setters redisplays the whole component

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

Putting it All Together

- In the previous examples, we learned
  - how to introduce side effects
  - how to create our own hooks
- We can use these concepts to make React components interact with a remote system (e.g., REST services).

client/src/hooks/useAnswerPage.ts in ip2 codebase

Interacting With a REST-based Server

```
/**
* Custom hook for managing the answer page's state, navigation, and real-time updates.
*
*/
const useAnswerPage = () => {
const { qid } = useParams();
const navigate = useNavigate();


const { user, socket } = useUserContext();
const [questionID, setQuestionID] = useState<string>(qid || '');
const [question, setQuestion] = useState<PopulatedDatabaseQuestion |
  null>(null);
  . . .
```

Interacting With a REST-based Server

```
useEffect(() => {
/**
* Function to fetch the question data based on the question ID.
*/
const fetchData = async () => {
   try {
            const res = await getQuestionById(questionID, user.username);
            setQuestion(res || null);
     } catch (error) {
            console.error('Error fetching question:', error);
     }
  };
   fetchData().catch(e => console.log(e));
}, [questionID, user.username]);
```

Async call to REST service

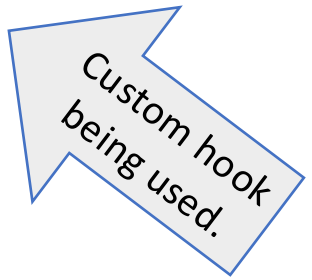
Run effect only when questionID and username changes

Interacting With a REST-based Server

```
/**
* Function to get a question by its ID.
*
* @param qid - The ID of the question to retrieve.
* @param username - The username of the user requesting the question.
* @throws Error if there is an issue fetching the question by ID.
*/
const getQuestionById = async (
qid: string,
username: string,
): Promise<PopulatedDatabaseQuestion> => {
const res = await api.get(`${QUESTION_API_URL}/getQuestionById/${qid}?username=${username}`);
if (res.status !== 200) {
  throw new Error('Error when fetching question by id');
}
return res.data;
};
```

Interacting With a REST-based Server

```
/**
* AnswerPage component that displays the full content of a question along with its answers.
* It also includes the functionality to vote, ask a new question, and post a new answer.
*/
const AnswerPage = () => {
const { questionID, question, handleNewComment, handleNewAnswer } = useAnswerPage();


if (!question) {
  return null;
}


return (
<>
<VoteComponent question={question} />
<AnswerHeader ansCount={question.answers.length} title={question.title} />
  . . .
```



Lesson 9.3 Testing your REACT components

Testing React components

- The AAA pattern ("Assemble/Act/Assess") still applies
- Need a test double for the React system
  - render components into a "virtual dom" or into a captive web browser
- The FakeStackOverflow codebase uses Cypress, a popular tool for end-to-end testing.

"Testing Library" https://testing-library.com is another test system for React. It is compatible with many UI libraries and many testing frameworks

https://docs.cypress.io/guides/end-to-end-testing/writing-your-first-end-to-end-test

Cypress commands work on a "virtual DOM"

`.visit()`	Visit a remote URL. Many tests begin with this command.
`.contains()`	Select a DOM element by text content.
`.get()`	Find DOM elements by selector
`.click()`	Click a DOM element.
`.type()`	Type into a DOM element.

These will fail if the specified element does not exist

https://docs.cypress.io/api/table-of-contents#Cypress-API

Recall: Most tests are in AAA form: Assemble/Act/Assess

```
test('addStudent should add a student to the database', () => {
    // const db = new DataBase ()
    expect(db.nameToIDs('blair')).toEqual([])

    const id1 = db.addStudent('blair');

    expect(db.nameToIDs('blair')).toEqual([id1])
});
```

Assemble (and check that you've assembled it

Act (do the action that you are trying to test)

Assess: check to see that the response is correct

A typical cypress test

testing/cypress/e2e/addAnswer.cy.ts
(from IP2 starter)

```
it("5.1 | Created new answer should be displayed at the top of the answers page",
() => {
    const answers = [
      "Test Answer 1",
      A1_TXT,
      A2_TXT,
    ];
    cy.visit("http://localhost:3000");
    cy.contains(Q1_DESC).click();
    cy.contains("Answer Question").click();
    cy.get("#answerUsernameInput").type("joym");
    cy.get("#answerTextInput").type(answers[0]);
    cy.contains("Post Answer").click();
    cy.get(".answerText").each(($el, index) => {
      cy.contains(answers[index]);
    });
    cy.contains("joym");
    cy.contains("0 seconds ago");
  });
```

Assemble (and check that you've assembled it correctly)

Act (do the action that you are trying to test)

Assess: check to see that the response is correct

run with: npx cypress run

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