CS 4530: Fundamentals of Software Engineering

Module 8: Patterns of React

Adeel Bhutta, Jan Vitek and Mitch Wand

Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - Recognize and apply common patterns in functional React components (useState, useEffect, useContext, useCustomHook)
  - Understand Rules of React
  - Understand how React functional components allow behaviors to be reused

React “Hooks” Solve Common Problems

- How to keep track of state that can be re-used across multiple renders?
- How to define some aspects of our component that should change when some data changes?
- How to share data from one component to many, without passing lots of props?
- Broadly: How to define common behaviors that can be reused by other components?

React “Hooks”

To solve common problems, we will discuss the following:

- useState
- useEffect
- useContext
- Custom Hooks

useState Tracks Mutable State

Problem: How to keep track of state that can be re-used across multiple renders, and How to tell React that state has changed, so component should re-render?

```
const [state, setState] = useState<TypeOfState>(initialValue);
```

useState returns an array of length 2: the first value is the current state value, second is a setter we can call to update that value.

initialValue is the value that state should take before the first call to setState

<TypeOfState> is an optional generic type parameter to declare the type of state

Problem 1 - where to store this?

Problem 2 - How to tell React?

```
export function LikeButton(){
  const [isLiked, setIsLiked] = useState(false);
  if (isLiked) {
    return (<IconButton aria-label="unlike"
                        icon={<AiFillHeart />} onClick={() => setIsLiked(false)} /> );
  } else {
    return (<IconButton aria-label="like"
                        icon={<AiOutlineHeart />} onClick={() => setIsLiked(true)} /> );
  }
}
```

Pattern: Create one useState hook for each state variable

- To have multiple state variables, call *useState* for each one
- Example: Track how many times the “like” button has been clicked

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

  if (isLiked) {
    return (<IconButton aria-label="unlike"
                        icon={<AiFillHeart />} onClick={() => setIsLiked(false)} /> );
  } else {
    return (<IconButton aria-label="like" icon={<AiOutlineHeart />} onClick={() => {
      setCount(count + 1)
      setIsLiked(true)
    }} /> );
  }
}
```

State Setters are Asynchronous

- Recall from Module 7: When setter is called, React uses carefully optimized approach *to* re-render our component and the state variable is updated
- Components are *not* re-rendered immediately upon calling a state setter

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

  if (isLiked) {
    ...
  } else {
    return (<IconButton aria-label="like" icon={<AiOutlineHeart />}
onClick={() => {
      console.log(`Pre-setCount, count=${count}`)
      setCount(count + 1)
      setIsLiked(true)
      console.log(`Post-setCount, count=${count}`)
    }} /> );
  }
}
```

Output:
(Click like)
1. Pre-setCount, count=0
2. Post-setCount, count=0

Pattern: use useEffect to invoke side-effects after rendering

- Problem: How to define side-effects that run in response to the data changing (and in turn, the component re-rendering)?
- Solution: React's *useEffect* hook

```
useEffect(()=>{
  // Code that runs after each render
  return () => {
    // Code that runs after the component is removed from the page OR before hook runs again
  }
})
```

useEffect invokes Side-Effects after rendering

- React's *useEffect* hook accepts a function that is *always* called *after* the component is updated

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

  useEffect(()=>{
    console.log(`Like has been clicked ${count} times`)
  })
  if (isLiked) {
    return (<IconButton aria-label="unlike"
             icon={<AiFillHeart />} onClick={() => setIsLiked(false)} /> );
  } else {
    return (<IconButton aria-label="like" icon={<AiOutlineHeart />}
    onClick={() => {
      console.log(`Pre-setCount, count=${count}`)
      setCount(count + 1)
      setIsLiked(true)
      console.log(`Post-setCount, count=${count}`)
    }} /> );
  }
}
```

Output:
Like has been clicked 0 times
(Click like)
1. Pre-setCount, count=0
2. Post-setCount, count=0
Like has been clicked 1 times

(Click un-like)
Like has been clicked 1 times
(Click like)
Like has been clicked 2 times
(Click un-like)
Like has been clicked 2 times

useEffect Dependencies Limit Their Execution

- *useEffect* takes an optional array of dependencies
- The effect is only executed if the values in the dependency array change (by reference equality)

```
useEffect(()=>{
  // Code that runs after each render
  return () => {
    // Code that runs after the component is removed from the page OR before hook runs again
  }
})
useEffect(()=>{
  // Code that runs after each render if dependency or anotherDependency change
  return () => {
    // Code that runs after the component is removed from the page OR before hook runs again
  }
}, [dependency, anotherDependency])
```

Only run the effect if dependency or anotherDependency change to point to a different thing

```
useEffect(()=>{
  // Code that runs after each render if dependency or anotherDependency change
}, [])
```

Only run the effect on the very first render

useEffect Dependencies Limit Their Execution

- If we add “count” to the dependencies array, then the effect is only executed when the value of “count” changes

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

  useEffect(()=>{
    console.log(`Like has been clicked ${count} times`)
  }, [count])
  if (isLiked) {
    return (<IconButton aria-label="unlike"
            icon={<AiFillHeart />} onClick={() => setIsLiked(false)} /> );
  } else {
    return (<IconButton aria-label="like" icon={<AiOutlineHeart />}
    onClick={() => {
      console.log(`Pre-setCount, count=${count}`)
      setCount(count + 1)
      setIsLiked(true)
      console.log(`Post-setCount, count=${count}`)
    }} /> );
  }
}
```

Output:
Like has been clicked 0 times
(Click like)
1. Pre-setCount, count=0
2. Post-setCount, count=0
Like has been clicked 1 times

(Click un-like)
(Click like)
Like has been clicked 2 times
(Click un-like)

useEffect + useState: Maintaining state for side-effects

- An *extremely* common pattern is to combine useEffect and useState
- Often requires using a “state updater” instead of concrete value

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

  useEffect(()=>{
    if(isLiked){
      setCount((prevCount) => prevCount + 1)
    }
  }, [isLiked])
  useEffect(()=>{
    console.log(`Like has been clicked ${count + 1} times`)
  }, [count])

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

Alternate call pattern for state setter: pass a function that returns the new state based on the old state

Run this effect only when isLiked changes

Run this effect only when count changes

Pattern: useContext and shared state

- Problem: Applications often hold some data that changes very infrequently, and is needed by many components. Passing that data as properties is cumbersome
- Example: Covey.Town's frontend has a TownController. Any component that needs to access data about the town needs a reference to it

```
export function CoveyTown(){
  const [townController, setTownController] = useState<TownController>();
  if(townController){ //Logged in
    return <TownMap townController={townController} />
  }
  return <Login />
}
export function TownMap(props: {townController: TownController}){
  return <div>
    <NewConversationModal townController={props.townController} />
    <SocialSidebar townController={props.townController} />
  </div>
}
```

Global state: once we are logged in, every component will need this

We need to pass this to EVERY component!?

useContext Accesses Shared State

- *React.createContext* creates a "context" - a pointer to shared state
- A *provider* for that context sets the value
- *useContext* returns the current value for that context
- A custom hook makes it easy for client components to access the shared value

```
export const TownControllerContext = React.createContext<TownController | null>(null);
export function CoveyTown(){
  const [townController, setTownController] = useState<TownController>();
  if(townController){ //Logged in
    return (<TownControllerContext.Provider value={townController}>
              <TownMap />
            </TownControllerContext.Provider>)
  }
  return <Login />
}
export default function useTownController(): TownController {
  const ctx = useContext(TownControllerContext);
  assert(ctx, 'TownController context should be defined in order to use this hook.');
  return ctx;
}
```

Create a context to store our shared state

Shared state: Every component nested within the provider can access

This hook will always return the TownController

The Rules of Hooks

- Hooks are APIs provided by React that let components “hook” into React’s internal behavior
- Each time that a component is rendered, the hooks will be called again
- React needs to be able to correlate the same calls to the same hook, e.g. to differentiate between two useState calls
- The rules of hooks ensure consistent behavior

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

How does React keep track of which state variable is which?

(The Rules of Hooks say how)

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

Pattern: use<HookName> For Custom Hooks

- Problem: How to compose and reuse “behaviors” that might involve storing state and performing side-effects?
- Solution: Create a “custom hook” - a function that starts with “use” and calls other hooks
- By convention, all custom React hooks should start with the prefix “use”

use<HookName>: Write Custom Hooks

- Calls to multiple hooks can be composed into a “custom” hook
- By convention, all custom React hooks should start with the prefix “use”

```
export function useLogCountOfProp(propertyName: string, propertyValue: boolean){
  const [count, setCount] = useState(0);
  useEffect(()=>{
    if(propertyValue){
      setCount((prevCount) => prevCount + 1)
    }
  }, [propertyValue])
  useEffect(()=>{
    console.log(`Property ${propertyName} was set to true ${count} times`);
  }, [count, propertyName])
}
export function LikeButton(){
  const [isLiked, setIsLiked] = useState(false);
  useLogCountOfProp('isLiked', isLiked);
  // No 'count' here, just the original like button
}
```

React Functional Components are More Modular than Class Components

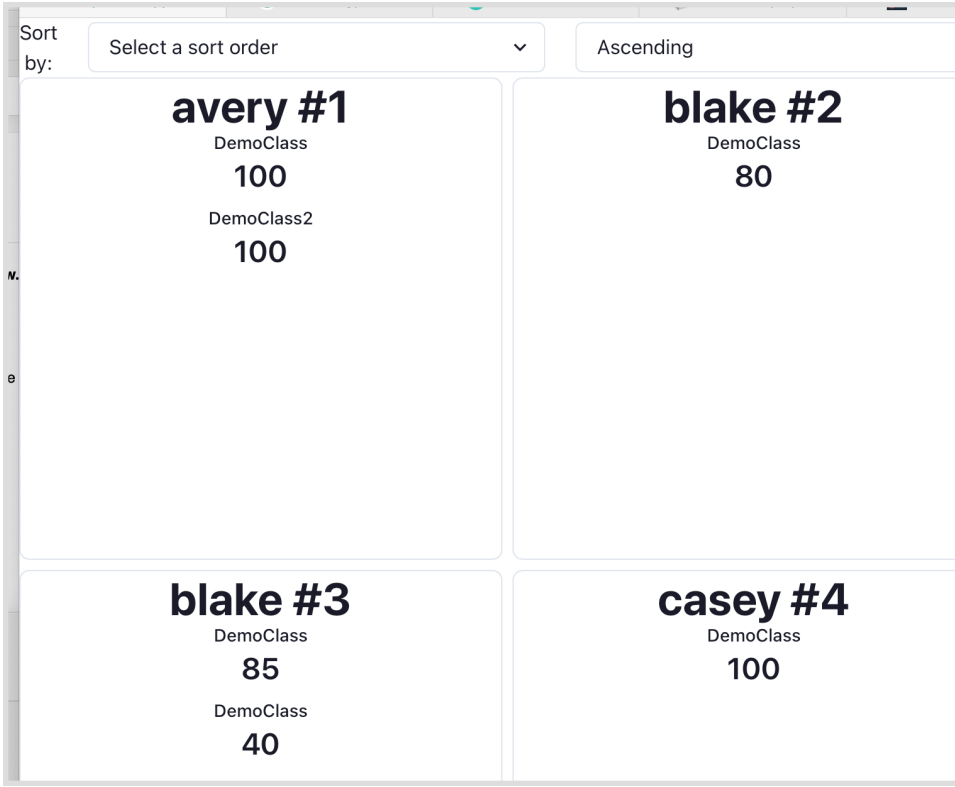
- Functional components
  - Create a useEffect for each behavior
  - Each useEffect can have its own cleanup callback
  - Compose multiple hooks into custom hooks for reusable behaviors
- Class components
  - Implement side-effects in componentDidMount, componentDidUpdate, componentWillUnmount
  - Each side-effect is spread between all three methods
  - All side-effects are mixed together
  - Can not easily reuse effects between components

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

A Bigger Example: Transcript App

- Fetches student transcripts from our REST API
  - Uses useEffect to fetch data when page is first loaded
  - Stores transcripts as state in component
  - Has not yet fully implemented “edit” or “add” functionality



Review

- Now that you've studied this lesson, you should be able to:
  - Recognize and apply four common patterns in functional React components (useState, useEffect, useContext, useCustomHook)
  - Understand Rules of Hooks
  - Understand how React functional components allow behaviors to be reused