FWP
Semester 2, 2022

Week 04

useEffect hook,

custom hooks,

Routing again,

interaction between components and

form handling yet again

RMIT UNIVERSITY

Segment 1

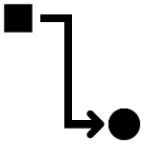
Hooks

- useEffect *again*
- Writing your own (custom)
- React Developer Tools (RDT)

useEffect *hook*

- **The Effect Hook lets you perform side effects in function components.**
- ***Side-effects 'eh!!. .do I have to learn something new?***
  - *"There won't be much to learn. In fact, we'll spend most of our time unlearning." – Dan Abramov*
- **What are effects?**
- **Examples**
  - **Fetching data**
  - **Reading from local storage Registering and deregistering event listeners**
  - **Manually changing DOM elements …**

Effects



- Effects are always executed after render
- Hooks documentation by Facebook states-
  - Mutations, subscriptions, timers, logging, and other side effects are not allowed inside the main body of a function component (referred to as React's render phase).
  - Doing so will lead to confusing bugs and inconsistencies in the UI.
  - Instead, use *useEffect*.
    - The function passed to useEffect will run after the render is committed to the screen.
- To opt out or skip effects, you have to understand basic JavaScript concepts about values

useEffect *recap*

- useEffect runs after every render (*by default*), and can optionally clean up for itself before it runs again.
- If you want your effects to run less often, you can provide a second argument – an array of values. Think of them as the *dependencies* for that effect.
  - If one of the dependencies has changed since the last time, the effect will run again.
- When you call useEffect in your component, this is effectively queuing or scheduling an effect to maybe run, after the render is done.
- After rendering finishes, useEffect will check the list of dependency values against the values from the last render, and will call your effect function if any one of them has changed.

useEffect *on props, state change*

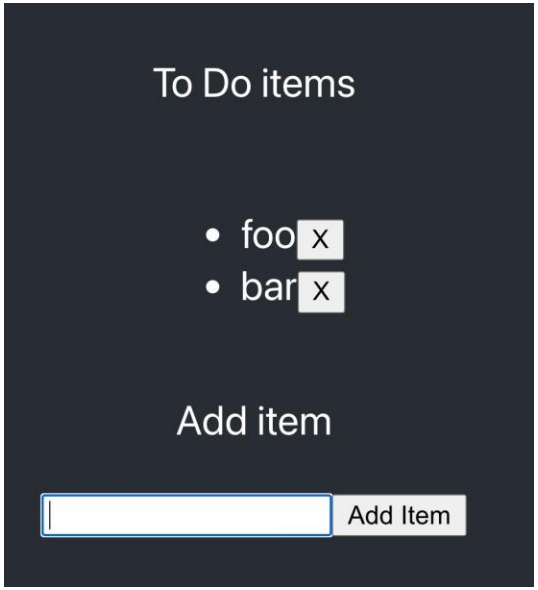
- By default, useEffect runs after every render, but it's also perfect for running some code in response to a state change.
  - You can limit when the effect runs by passing the second argument to useEffect.
- Second argument as an array of "dependencies" – variables that, if changed, the effect should rerun. These can be any kind of variable: props, state, or anything else.
- Time to look at examples:
  - example1 (an example to show when useEffect runs)
  - example2 (state change) &
  - example3 (prop change)

Lectorial Exercise

- **Do you foresee any issues with useEffect hook? *When can its use be problematic?***

useEffect *on fetching data*

- We will now look at an example of data fetching from localStorage and with *useEffect* in the picture.
- The web app allows use to add to-do items in a list
- These items are stored in a localStorage
- The web app allows use to remove the items
- example4
  - check the comment entries



Custom *hook*

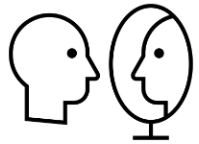
- You could even write your own (*customised*) hook.
- But why!!
- Custom hooks are a handy way to encapsulate hook-related logic that can be re-used across components.
- Think of a custom hook as a super-powered helper function.
- You can call hooks inside custom hooks!
- Custom hooks are-
  - functions whose names begin with 'use...'
  - functions which can call other hooks

Custom *hook*

- We saw one example in the previous week
- But we will keep revisiting the custom hooks
- example5

React Developer Tools: *Chrome*

- **This browser extension is available for Chrome and Firefox and is designed to help you debug React applications.**
- ***Open Extensions page in Chrome***
  - ***Search for React Developer Tools***
  - ***Install and Enable***
- **Here is a good page to learn about these tools**
  - ***https://react-devtools-tutorial.vercel.app/***



Lectorial Exercise

- Can you name few scenarios where you can justify writing a custom hook? Explain your answer.

Rules of hooks

- **Outlined on documentation page:**

  https://reactjs.org/docs/hooks-rules.html

  1. **Only Call Hooks at the Top Level**
  2. **Only Call Hooks from React Functions**
- **Facebook recommends**
  - ***We released an ESLint plugin called eslint-plugin-react-hooks that enforces these two rules. You can add this plugin to your project if you'd like to try it***
- ***Remember we had discussed this during Lectorial 2***

Segment 2

Router again

Interaction between functional components

Router *again*

- We talked about router module during Lectorial 3
- It provides- *routing capabilities, declarative routing* capabilities for react apps
- This week we will look at the following features of router:
  - Nested routes – example8
  - Router hooks – example9
    - We have already seen how useNavigate is utilised
    - We will now look at useLocation hook

Ref: You can read about these at: https://reactrouter.com/docs/en/v6/components/link

Interaction between components

- **Your page will often have multiple components and sometimes**
  1. **Parent component will need to interact with Child component**
  2. **Child component will need to interact with Parent component**
  3. **Children components will need to interact with each other (*siblings*)**

Parent - Child

- We have already seen this- the simplest form of communication between components is via properties — *props*.

```
function App(){
  return <div>
    <AppChild name="JabbaTheHut" />
    </div>
}

function AppChild(props){
  return <span>
      My name is {props.name}
    </span>
}
```

- Props are the parameters passed into child components by parents, similar to arguments to a function.
- You can pass events from parent-child too

example10

Child – Parent

- This is *trickier!*
- When a child needs to update the parent as to changes, they cannot just modify properties. Children cannot update props.
- A parent can pass in a functional prop, and the child can call that function.

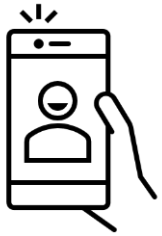
```
function App(){
  const [name, setName] = React.useState("JabbaTheHut");
  return <div>
      <AppChild name={name} onChangeName={()=>{setName("JabbaTheHutt")}}/>
    </div>
}

function AppChild(props){
  return <span>
      My name is {props.name}
      <button onClick={props.onChangeName}>Change Name</button>
    </span>
}
```

Child – Parent

- It often happens that child components need to pass arguments up along with their events.
- This can be achieved by adding arguments to the functional prop ***callback***
- Time for an example
- **example11**

Child - Child



- How can siblings communicate?
- How will a child component communicate with other child component?
- *This is not as easy as others*
  - *We can either use a combination of props and callback*
  - *Use context hook*
  - *Use Redux*
- *We will come back at context and Redux later!*

Lectorial Exercise

- **Why would one write their own file handling and validation React code? There are various 3rd party modules that can be used. What are the advantages and disadvantages of these approaches?**

References

- Reference: *The road to react (2021 edition), by Robin Weiruch; Leanpub*
- The above will be the prescribed reference textbook for the first few week(s) for this course.
- https://reactjs.org/docs/hooks-reference.html

Assignment 1 *first discussion*

- It is online
- Deadline: ***Check Canvas***
- Worth 25%
- To be completed individually OR group of 2
- Partial React website prototype
- DO NOT WRITE CLASS COMPONENTS- you will get a ZERO for the whole assignment.
- Based on exercises covered in week(s) 1-4, you should be able to get started and complete till the CREDIT part.

Next week

- Hooks *yet* again
  - useRef
  - useContext
  - useMemo
  - useLayoutEffect
- Custom hooks *again*
- Interaction between components *again*
- Testing framework