FWP
Semester 2, 2023

Week 05

useMemo
useCallback

RMIT UNIVERSITY

Segment 1

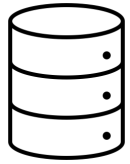
useMemo
useCallback
useRef

useMemo hook

- It returns a *memorized* value!
- *Memoize*
  - In the world of computing, memorization is an optimisation technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

- Now you know what that means…
- The hook accepts 2 arguments — a function compute that computes a result and the dependencies array:
  - ```
    const memoizedResult =
              useMemo(compute, dependencies);
    ```
    ```

useMemo hook- *how does it work?*



- During initial rendering, useMemo(compute, dependencies) invokes compute, memoizes the calculation result, and returns it to the component.
- If during next renderings the dependencies don't change, then useMemo() doesn't invoke compute but returns the memoized value.
- But if dependencies change during re-rendering, then useMemo() invokes compute, memoizes the new value, and returns it.
- Time to look at an example
  - example1

Lectorial Exercise

- If used inappropriately, useMemo could harm the performance? *Why do you think that could happen?*

useCallback hook

- When a component re-renders, every function inside of the component is recreated and so if there are any functions, the references will change between renders.
- `useCallback(callback, dependencies)` will return a memoized instance of the callback that only changes if one of the dependencies has changed.
  - Instead of recreating the function object on every re-render, we can use the same function object between renders.

useCallback hook- *how does it work?*

- It can assist in isolating resource intensive functions so that they will not automatically run on every render.
- The useCallback Hook only runs when one of its dependencies update.
- This can improve performance.
- One reason to use useCallback is to prevent a component from re-rendering unless its props have changed.
- Time for an example
  - example2

Lectorial Exercise

- How do you think useMemo differs from useCallback?

useRef

- While most of the values stored by your component will be directly represented in the user interface of your application, sometimes you'll use a variable only for the mechanics of your app rather than for consumption by users.
- You may want to use setTimeout or setInterval as part of an animation, so you need to keep hold of the IDs they return.
- The useRef React hook is useful in the following two situations:
  - Accessing DOM elements directly inside React
  - Store state values that do not trigger re-renders and are persisted across re-renders

useRef syntax

```
const refContainer = useRef(initialValue);
```

- useRef returns a mutable ref object
  - whose .current property is initialised to the passed argument (initialValue).
  - The returned object will persist for the full lifetime of the component.
- To mutate the value of the object, we can assign the new value to the current property:

```
const App = () => {
   const myRef = useRef("initial value")

   // updating ref
   myRef.current = "updated value"

  // myRef now will be {current: "updated value"}
}
```

useRef v/s useState

- Both preserve their data during render cycles and UI updates, but
  - only the useState Hook with its updater function causes re-renders
- useRef returns an object with a current property holding the actual value.
  - In contrast, useState returns an array with two elements: the first item constitutes the state, and the second item represents the state updater function
- useState and useRef can be considered data Hooks, but
  - only useRef can be used in yet another field of application: to gain direct access to React components or DOM elements

So how is useRef useful- give me an example!

- Consider this component

```
const Component = () => {
    const ref = useRef(null);
    return <div ref={ref}> Hello world </div>;
};
```

- With this reference, you can do lots of useful things like:
  - grabbing an element's height and width
  - seeing whether a scrollbar is present
  - calling focus() on the element at a certain moment
  - …

useRef

- Remember this
  - If you want to update data and cause a UI update, useState is your Hook.
  - If you need some kind of data container throughout the component's lifecycle without causing render cycles on mutating your variable, then useRef is your solution.

useRef

- Time to look at examples

- example3: how useState and useRef behave differently
  - example4: how to use useRef to grab an element

Lectorial Exercise

- What's the difference between useRef and using a variable?
- You could always do something like

```
const Component = () => {
    let renderCount = 0;
    renderCount += 1;

    return <>Hello world</>;
}
```

Segment 2

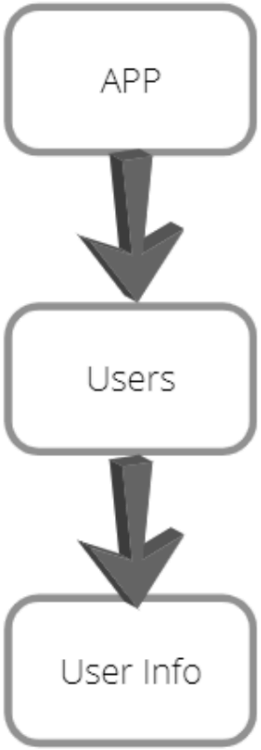
useContext
Custom hooks Asynchronous fetch;
React fragment

useContext

- In a typical React application, data is passed top-down (parent to child) via props, but such usage can be cumbersome for certain types of props- *we talked about in lectorial 4*
- Sometimes multiple components need the information
- When information/data that needs to be passed becomes complex, the use of props becomes frustrating

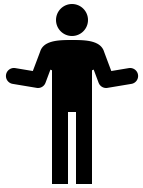
- Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree
- Context is designed to share data that can be considered "*global*" for a tree of React components, such as the current authenticated user, theme, or preferred language.

Without & With context!



useContext

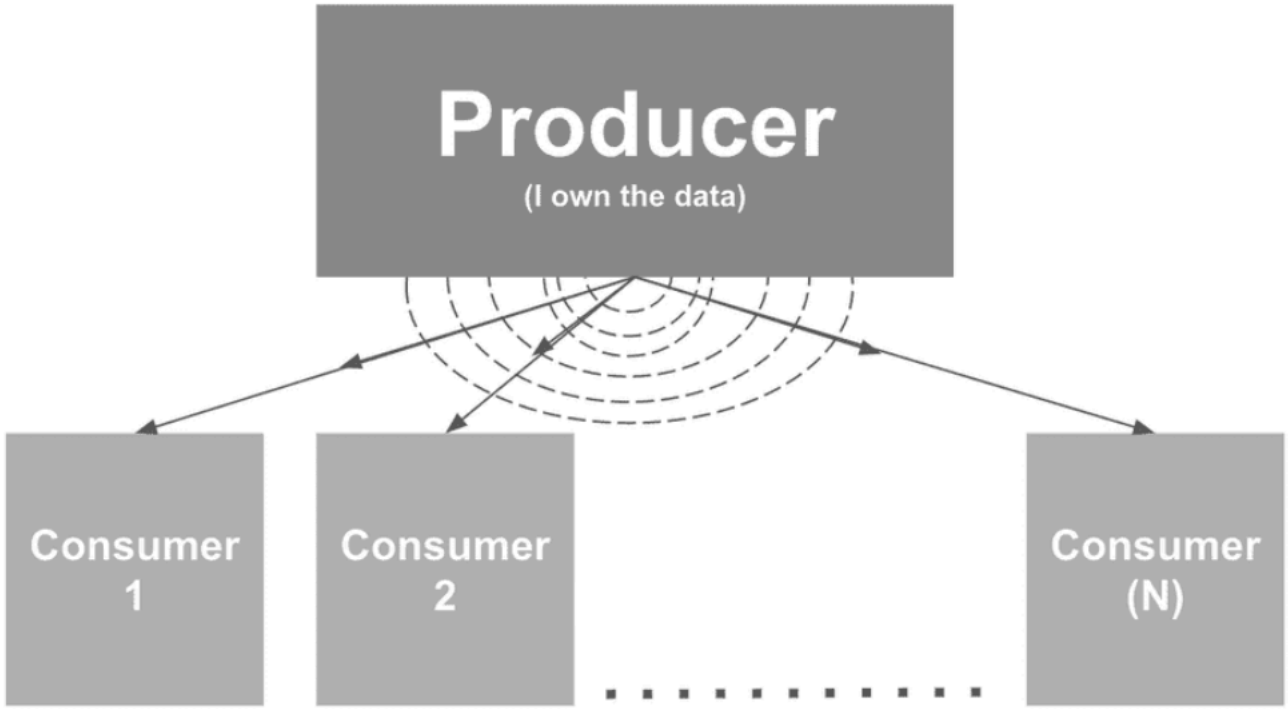
- Context is primarily used when some data needs to be accessible by many components at different nesting levels.
  - Apply it carefully because it makes component reuse more difficult.
- `const MyContext = React.createContext(defaultValue);`
  - Creates a Context object.
  - When React renders a component that subscribes to this Context object it will read the current context value from the closest matching Provider above it in the tree.
  - *What is a provider?*



useContext

- Every Context object comes with a Provider React component that allows consuming components to subscribe to context changes.
- The Provider component accepts a value prop to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers.
  - `<MyContext.Provider value={/* some value */}>`
- All consumers that are descendants of a Provider will re-render whenever the Provider's value prop changes.
  - ***Now, there is a consumer!!***

useContext *producer* and *consumer*



Reference: https://content.breatheco.de/en/lesson/context-api

useContext

- A React component that subscribes to context changes.

```
<MyContext.Consumer>
    {value => /* render something based on the context value */}
</MyContext.Consumer>
```

- Time to now look at an example
  - example5

- We are not yet done with Context- we will revisit it next week!

Custom Hooks *again*

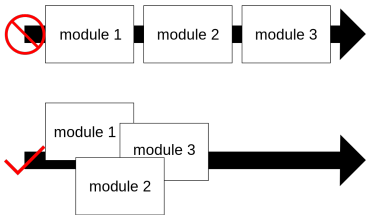
- We have been talking about writing own hooks (*custom hooks*) since week 3
- You remember the rules
  - Start with “use”
  - Call hooks only from React functions
  - Call hooks at top level only
    - *Do not* put hooks inside conditionals.
    - *Do not* put hooks inside loops.
    - *Do not* put hooks inside nested functions.
      - *Each of those three scenarios can lead to you skipping hook calls or changing the number of times you call the hooks for a component.*

Custom Hooks *again*

- One of the very apt uses of a custom hook could be a data fetch scenario
- Imagine an *asynchronous call to an API*- a very typical scenario in a web application
- Let us call a free API- look at these Google APIs
  - https://developers.google.com/apis-explorer
  - We will call the Book API
    - https://developers.google.com/books/docs/v1/reference/
- In a web application say
  - You have text box which can list books from google store based on what you type on it. If no book available on that particular search query than show 'No Book Found'.

But how are asynchronous operations handled in React?

- We touched upon the notion of asynchronous tasks during previous lectorial
- useEffect is a good hook to deal with scenarios such as: calls to an API, fetch data, etc.
  1. It's generally best practice to define a function that you call from your effect and then do your async operations there. It allows you to use React's keywords such as *async/await*
  2. There are other ways to code async operation- use a library known as ***react async***
     - https://docs.react-async.com/getting-started/usage
     - It provides additional hooks that you can play around with



Custom hook that does an asynchronous fetch

- Time to look at an example
- example6

Search : react search

Book Title : React.Js Essentials

Book Title : React Projects

Book Title : React and React Native

Book Title : React and React Native

Book Title : Learning React

Book Title : Fullstack React

Book Title : Learning React

Book Title : React Cookbook

Book Title : React: Up & Running

Book Title : Pro MERN Stack

```
function useAsyncHook(searchBook) {
  const [result, setResult] = React.useState([]);
  const [loading, setLoading] = React.useState("false");

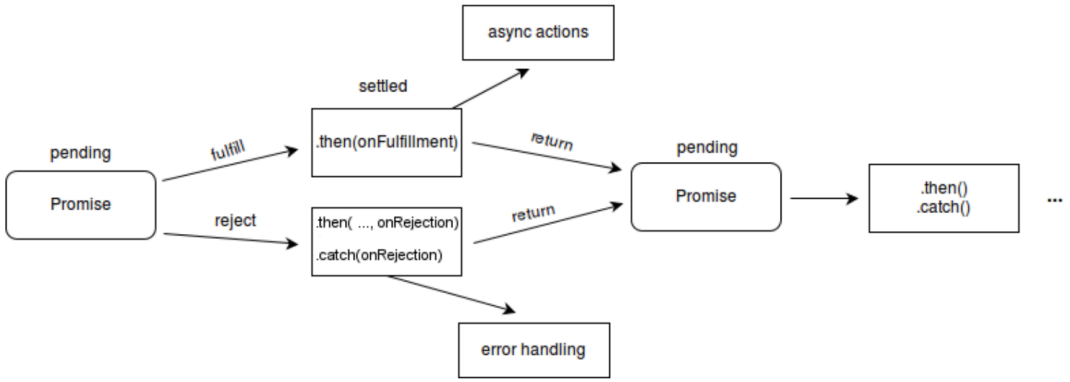
  React.useEffect(() => {
    async function fetchBookList() {
      try {
        setLoading("true");
        const response = await fetch(
          `https://www.googleapis.com/books/v1/volumes?q=${searchBook}
        );
```

How does async/await work in React?

- The async keyword, which you put in front of a function declaration to turn it into an async function.
- An async function is a function that knows how to expect the possibility of the await keyword being used to invoke asynchronous code.
- The advantage of an async function only becomes apparent when you combine it with the await keyword.
- Async functions always return a *promise*. If the return value of an async function is not explicitly a promise, it will be implicitly wrapped in a promise.
- Another new word- *promise!!*

Promise

- The *Promise* object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- A Promise is in one of these states:
  - pending: initial state, neither fulfilled nor rejected.
  - fulfilled: meaning that the operation was completed successfully.
  - rejected: meaning that the operation failed.



Reference: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

React fragments

- React Fragments allow you to wrap or group multiple elements without adding an extra node to the DOM.

```
function Parent () {
  return (
    <React.Fragment>
      <Child1 />
      <Child2 />
    </React.Fragment>
  );
}
```

- Time to look at the last example for today
  - example7

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

Next week

- useContext,
- useLayoutEffect,
- useReducer;
- async programming again