FWP
Semester 2, 2023

Week 06

useContext *again*

useLayoutEffect

useImperativeHandle

useReducer

Asynchronous call *again*

RMIT UNIVERSITY

Segment 1

useContext *again*

useLayoutEffect

useContext *revisited*

- Looking back at what we have covered re this hook
  - Use this hook when you need to share data globally like current authenticated user, theme, or preferred language, *etc*.
  - Context is primarily used when some data needs to be accessible by many components at different nesting levels.
  - For example, consider a Page component that passes a user and avatarSize prop several levels down so that deeply nested Link and Avatar components can read it-

```
<Page user={user} avatarSize={avatarSize} />
// ... which renders ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... which renders ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... which renders ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

useContext *revisited*

- You're not limited to a single child for a component. You may pass to multiple children
- This pattern is sufficient for many cases when you need to decouple a child from its immediate parents.
- So, remember
  - If same data needs to be accessible by many components in the tree, and at different nesting levels.
    - Context lets you "broadcast" such data, and changes to it, to all components below.
- According to Context API we need
    - A Context object.
    - A Context provider.
    - A Context consumer.

useContext- updating context

- It is often necessary to update the context from a component that is nested somewhere deeply in the component tree.
- In this case you can pass a function down through the context to allow consumers to update the context.
- Time to look at an example where we will update context
  - example1: emp- salary example

useContext- *practical use*

- A very good use of this hook is to *change themes* on a web page
- You could create *light* and *dark themes* and switch between them
- Have a look at these interesting implementations in your *free time*:
  - [https://www.nicknish.co/blog/react-hooks-deep-dive-usecontext]
  - [https://www.codewithlinda.com/blog/dark-mode-with-react-context/]

useContext- *consuming multiple contexts*

- In some situations, you will need to use two or more contexts together.
- Each time you want to use both the contexts together, you will have to wrap the components with both of the providers and consumers.
  - This would get complex if we needed to use more than two contexts
- A good practice would be to
  - wrap the consumers and providers to create a combined consumer and provider
  - use the combined consumer and provider in almost the same way we use a regular context

useContext- *consuming multiple contexts*

- There are even third party libraries present to for dealing with multiple contexts, such as
  - consuming-multiple-contexts
  - https://github.com/mjancarik/consume-multiple-contexts
- Let us look at some code snippets from React useContext documentation website and see how it can be simplified?
- Please look at the next slide →

consuming multiple contexts without library

```
const ThemeContext = React.createContext('light');
const UserContext = React.createContext();

class App extends React.Component {
  render() {
    const {signedInUser, theme} = this.props;

    return (
      <ThemeContext.Provider value={theme}>
        <UserContext.Provider value={signedInUser}>
          <Layout />
        </UserContext.Provider>
      </ThemeContext.Provider>
    );
  }
}

function Layout() {
  return (
    <div>
      <Sidebar />
      <Content />
    </div>
  );
}

function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}

function Sidebar() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfileSidebar user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}
```

← **Raw code without library**

Using library →

```
import { createNamedContext, createMultipleContexts } from 'consume-multiple-contexts';

const ThemeContext = React.createContext('light');
const UserContext = React.createContext();

const withContext = createMultipleContexts(
    createNamedContext('theme', ThemeContext),
    createNamedContext('user', UserContext)
);

.
.

function Content() {
  return withContext(({ theme, user }) => (
    <ProfilePage theme={theme} user={user} />
  ));
}

function Sidebar() {
  return withContext(({ theme, user }) => (
    <ProfileSidebar theme={theme} user={user} />
  ));
}
```

useContext

- Some apt applications of Context hook can be
  - Theming—Pass down app theme
  - Internationalization—Pass down translation messages
  - Authentication—Pass down current authenticated user
- *Gotchas*
  - You should not be reaching for context to solve every state sharing problem
  - If your state is frequently updated, React Context may not be an efficient choice
  - Context does NOT have to be global to the whole app, but can be applied to one part of your tree
  - You can (and probably should) have multiple logically separated contexts in your app.
  - React Context is an excellent API for apps with infrequent state changes, but it can quickly turn into a developer's nightmare if not used correctly.

useLayoutEffect

- **useLayoutEffect and useEffect appear to do the same except here are the differences:**

useLayoutEffect	useEffect
runs synchronously after a render but before the screen is updated	runs asynchronously and after a render is painted to the screen.
• You cause a render somehow (change state, or the parent re-renders) • React renders your component (calls it) • useLayoutEffect runs and React waits for it to finish. • Screen is visually updated	• You cause a render somehow (change state, or the parent re-renders) • React renders your component (calls it) • The screen is visually updated • useEffect runs

useLayoutEffect

- As per React's documentation on hooks
- 99% of the times you want to/will utilise useEffect
- *So when to useLayoutEffect?*
- If your component is flickering when state is updated – hmm, what is *flickering…*
  - it renders in a partially-ready state first →
  - then immediately re-renders in its final state.

Reference: Obtained from free animation gifs website: https://giphy.com

useLayoutEffect

- useLayoutEffect is synchronous which means that
  - the app won't visually update until your effect finishes running…
  - it could cause performance issues like stuttering if you have slow code in your effect.
- Time to look an examples

  - example2: useEffect versus useLayoutEffect
  - example3: here we will make change to a DOM node directly before browser has a chance to paint

Lectorial Exercise

- Can you think of some scenarios where useLayoutEffect will come handy?

Segment 2

useReducer hook

Asynchronous *again*

Retaining values upon refresh;

useReducer *hook*

- React documentation states that - useReducer is an alternative to useState when state logic is complex.
- useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one.
- useReducer also lets you optimise performance for components that trigger deep updates because you can pass dispatch down instead of callbacks.
- In turn, hooks like useContext and useReducer eliminate the dependency on Redux for many React apps

Reducer is a JS concept!

- JavaScript *reducer* is one of the most useful array methods
- It is a cleaner way of iterating over and processing the data stored in an array.
- The reduce method accepts two arguments: a reducer function for the array that is used as a callback and an optional initialValue argument. The reducer function takes four arguments: accumulator, currentValue, currentIndex, and array.
- Look at the example at:
  - https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_reduce2

useReducer *hook*

How it is used?

```
const [state, dispatch] = useReducer(reducer,
                                     initialState);
```

- **It accepts a reducer function with the application initial state, returns the current application state, then dispatches a function.**
- **An alternative to useState**
  - **Accepts a reducer of type**
    - **(state, action) => newState**
  - **and returns the current state paired with a dispatch method.**

useReducer *hook*

- **Time to look at an example**
  - **example4: *a to do list example with useReducer & useContext***

- **You are advised to pay undivided attention while this example is being discussed**
  - **That is where you will realise the power of these hooks.**

Lectorial Exercise

- Can you think of some scenarios where useReducer will be a much better choice than useState?

useReducer *versus* Redux: *future discussion*

- Something we will come back in the forthcoming lectorials
- Redux creates one global state container which is above your whole application and is called a store WHILE useReducer creates an independent component co-located state container within your component.
- useReducer is often co-used with useContext to deal with complex state management
- According to React documentation-
  - having all your state handled by React and not using *third party library like Redux* will make your code easier to understand and work with React Hooks will make building components much faster with less code.

Asynchronous *again*

- We are back to asynchronous operations
- In week 5, we looked at an example of custom hook that made an asynchronous call to Google's Book API using JavaScript keywords *async-await*
- In JavaScript, there is yet another way to make an asynchronous call
  - Using *then/catch* and *finally*
  - This is left for your self exploration
- According to JS documentation
  - then and catch and finally are methods of the Promise object, and they are chained one after the other.
  - Each takes a callback function as its argument and returns a Promise.

Asynchronous *again*

- Often, using *chained then* methods can require difficult alterations
- By contrast, async/await lent itself to a more readable solution
- Adding many then methods can easily force us further down the path towards callback issues
- When you have a choice, always use async/await as compared to then/catch

Asynchronous *again*

- There is a third-party library *react async* that you can use in react web apps
- React async
  - https://docs.react-async.com/
- It makes it easy to handle asynchronous UI states
- React Async consists of a React component and several additional hooks
- Let us look at one simple example
  - example5: retrieve movie info from SWAPI - Star Wars API (https://pipedream.com/apps/swapi)

How to persist state upon page refresh?: *a common malady*

- Sometimes you may want to keep the state of a React component persistent after a browser refresh.
- A simple way to accomplish this without having to rely on any third-party library, is to use the localStorage API together with useEffect hook.
  - *Note: there are other ways too*
  - If you want to reuse a component across an application, this is not a good approach!!
- Time for an example
  - example6

Hooks Hooks Hooks

- **Looking back we have covered**
  1. **useState**
  2. **useEffect**
  3. **Custom hooks**
  4. **useRef**
  5. **useContext**
  5. **useLayoutEffect**
  7. **useMemo**
  8. **useCallback**
  9. **useImperativeHandle:** ***lab this week***
  10. **useReducer:** ***we will revisit***

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
- https://redux.js.org/introduction/getting-started

Next week

- Assignment 1 due
- Mandatory demo: Following week
- MID-SEMESTER BREAK
- No classes or consultation sessions