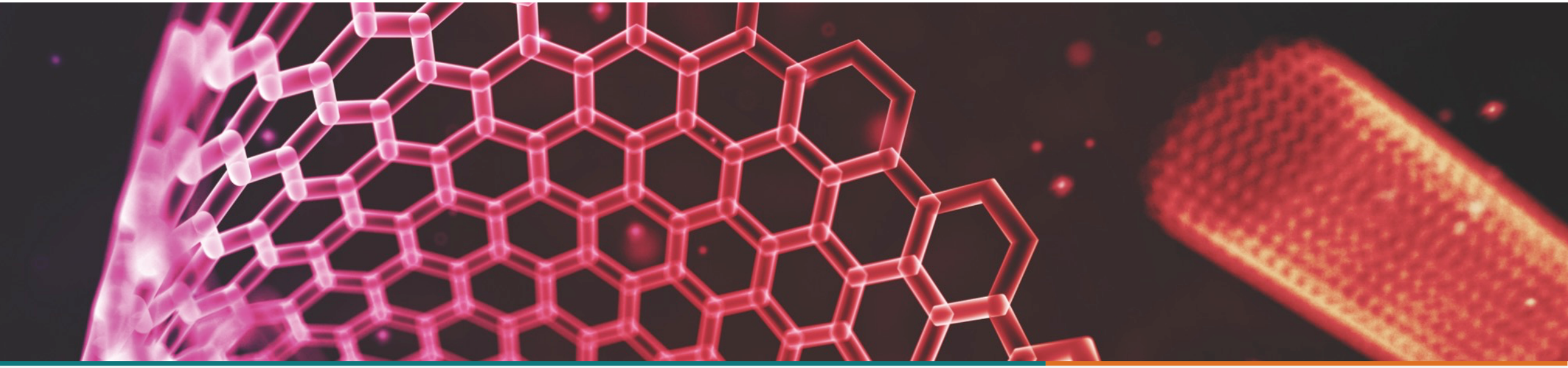
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 554 – Web Programming II

React - Continued



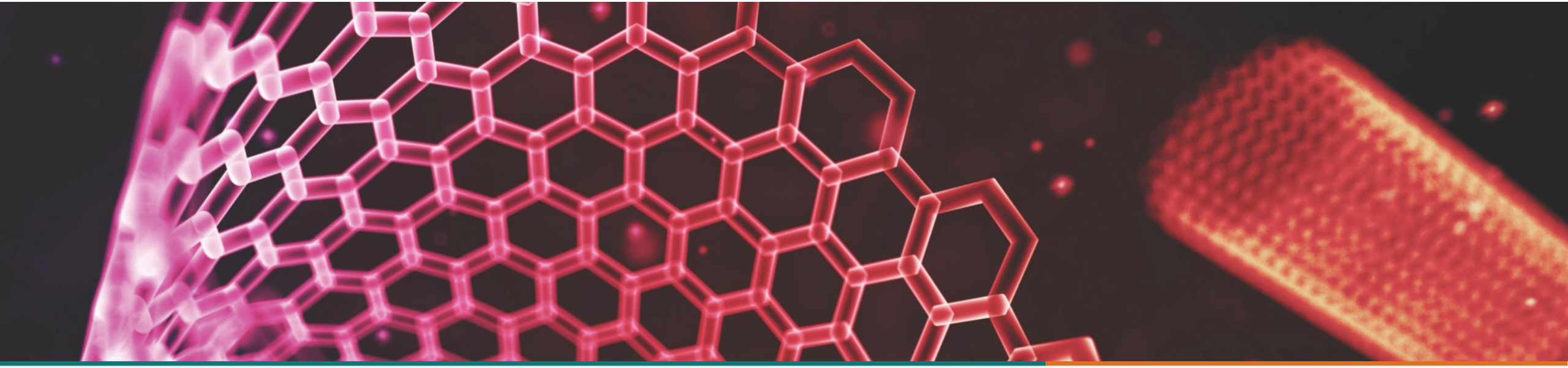
In this Lecture

In this lecture we are going to finish building out the Firebase authentication app.

We will be using Function Components as well as React Hooks. We have already seen the **useState** and **useEffect** hooks but to finish building this out we also need to use the **useContext** hook.

We will also go over the **useMemo** and **useCallback** hooks and example of building your own custom hook.

Context API



What is Context API in React?

Before we go over using the useContext hook, it's important to understand the context API in React.

What is the Context API in React?

The Context API is a component structure provided by the React framework, which enables us to share specific forms of data across all levels of the application. It's aimed at solving the problem of prop drilling.

"Prop drilling (also called "tunneling") refers to the process you have to go through to get data to parts of the React Component tree." – Kent C. Dodds.

Before the Context API, we could use a module to solve this, which led to the increasing popularity of state management libraries like Redux. Libraries like Redux allows you to get data from the store easily, anywhere in the tree. However, let's focus on the Context API as we will cover Redux in a soon.

Context API in React

The Context API has actually always been apart of React for a while but was considered experimental. Moving forward, the API was improved to stability, and as of the release of version 16.3, the feature was moved out of the experimental phase and into the release phase.

Before now many of the tools that have been used like react-redux and react-router all used context to function.

When to Use the Context API

As we mentioned earlier, the Context API is useful for sharing data that can be considered global, such as the currently authenticated user, the theme settings for the application, and more. In situations where we have these types of data, we can use the Context API and we don’t necessarily have to use extra modules.

In fact, any situation where you have to pass a prop through a component so it reaches another component somewhere down the tree is the perfect place where you can use the Context API.

Passing Props Without Using the Context API

Let’s look at an example of passing props from one component, to another, to another. Say we want to pass some theme settings from one component to another, to another.

App.js ● | Child.js | ChildChild.js

```
1   import React from 'react';
2   import './App.css';
3   import Child from './Child';
4   function App() {
5     return (
6       <div className='App'>
7         <Child theme={{ color: 'green', fontWeight: 'bold' }} />
8       </div>
9     );
10  }
11  export default App;
```

App.js

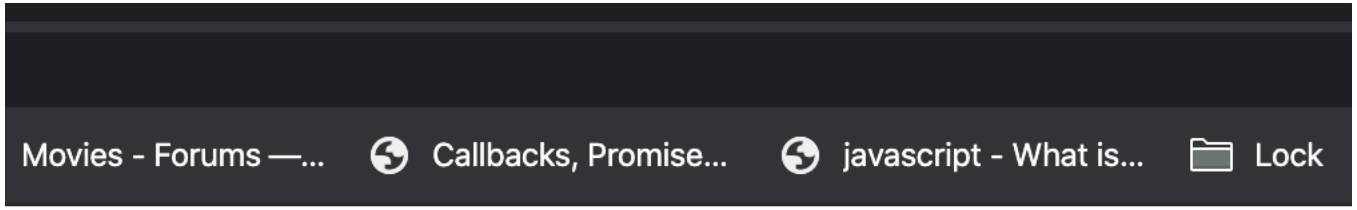
Passing Props Without Using the Context API

Child.js

Passing Props Without Using the Context API

ChildChild.js

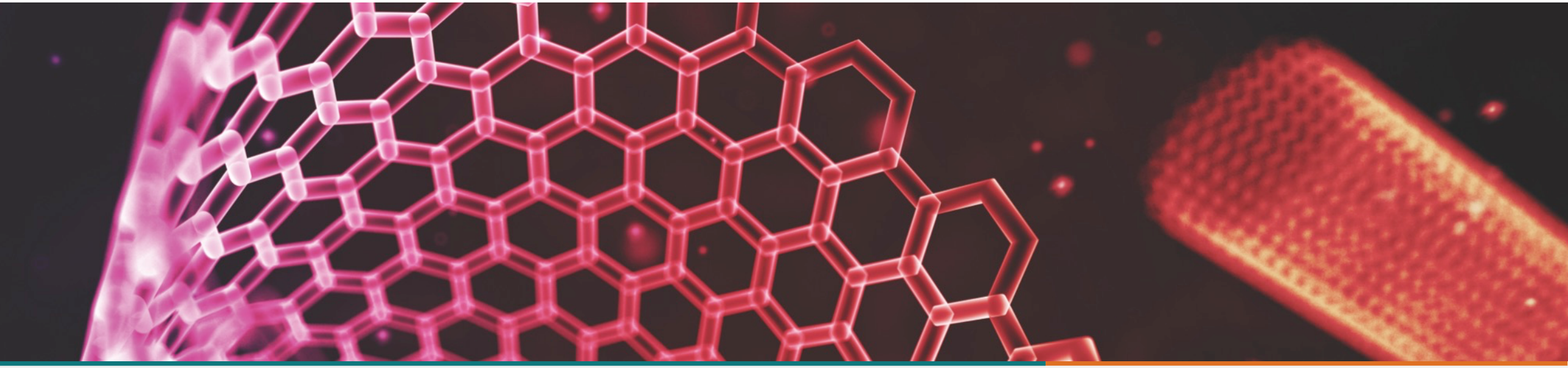
Passing Props Without Using the Context API



I'm the Child

I'm the Child's Child

useContext Hook



Passing Props using Context API and useContext

useContext

```
const value = useContext(MyContext);
```

Accepts a context object (the value returned from `React.createContext`) and returns the current context value for that context. The current context value is determined by the `value` prop of the nearest `<MyContext.Provider>` above the calling component in the tree.

When the nearest `<MyContext.Provider>` above the component updates, this Hook will trigger a rerender with the latest context `value` passed to that `MyContext` provider. Even if an ancestor uses `React.memo` or `shouldComponentUpdate`, a rerender will still happen starting at the component itself using `useContext`.

Passing Props using Context API and useContext

Let’s look at the same previous example but using the Context API and the useContext hook. First, we need to create the context.

ThemeContext.js | App.js | ChildComponent

```
1  import React from 'react';
2  const ThemeContext = React.createContext(null);
3
4  export default ThemeContext;
5
```

ThemeContext.js

React's Context is initialized with React's createContext top-level API. In this case, we are using React's Context for sharing a theme (e.g. color, paddings, margins, font-sizes) across our React components. For the sake of keeping it simple, the theme will only be a color and a font-weight

Passing Props using Context API and useContext

We then import ThemeContext into our app.js and wrap the component in the ThemeContext.Provider component.

ThemeContext.js | App.js | ChildComponent.js | ChildChildComponent.js

```
1   import React from 'react';
2   import './App.css';
3   import ThemeContext from './ThemeContext';
4   import ChildComponent from './ChildComponent';
5   function App() {
6       return (
7           <div className='App'>
8               <ThemeContext.Provider value={{ color: 'green', fontWeight: 'bold' }}>
9                   <ChildComponent />
10              </ThemeContext.Provider>
11          </div>
12      );
13  }
14
15  export default App;
16
```

The Context's Provider component can be used to provide the theme to all React child components below this React top-level component which uses the Provider:

Passing Props using Context API and useContext

React's useContext in our Child component that just uses the Context object which was created before to retrieve the most recent value from it.

JS ThemeContext.js | JS App.js | JS ChildComponent.js × | JS ChildChildComponent.js

```
1   import React, { useContext } from 'react';
2   import ThemeContext from './ThemeContext';
3   import ChildChildComponent from './ChildChildComponent';
4   const ChildComponent = () => {
5       const theme = useContext(ThemeContext);
6       return (
7           <div>
8               <p style={theme}>Hello World</p>
9               <ChildChildComponent />
10          </div>
11      );
12  };
13
14  export default ChildComponent;
15
```

Passing Props using Context API and useContext

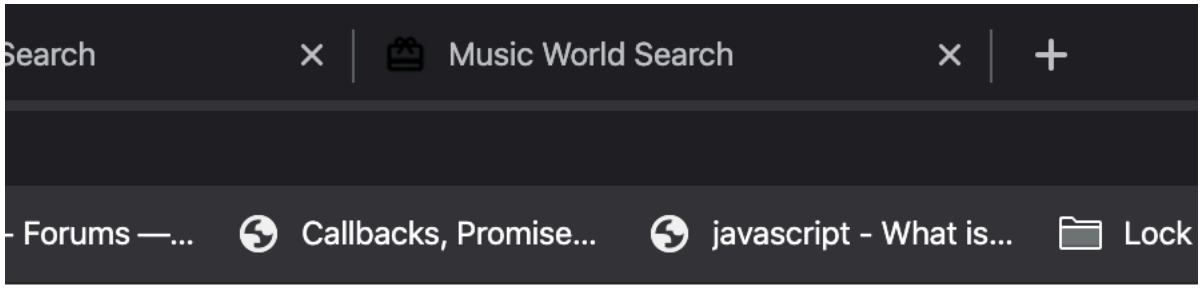
We can do the same for the child's child component.

JS ThemeContext.js | JS App.js | JS ChildComponent.js

```
1   import React, { useContext } from 'react';
2   import ThemeContext from './ThemeContext';
3   const ChildChildComponent = () => {
4       const theme = useContext(ThemeContext);
5       console.log(theme);
6       return <p style={theme}>Hello World Too!</p>;
7   };
8
9   export default ChildChildComponent;
10
```

Passing Props using Context API and useContext

Output:



Passing Props Without Using the Context API

We can also use useState with context

Hello World

Hello World Too!

Toggle Theme

Clicking the button, changes the state which then propagates the changes to the children components

Hello World

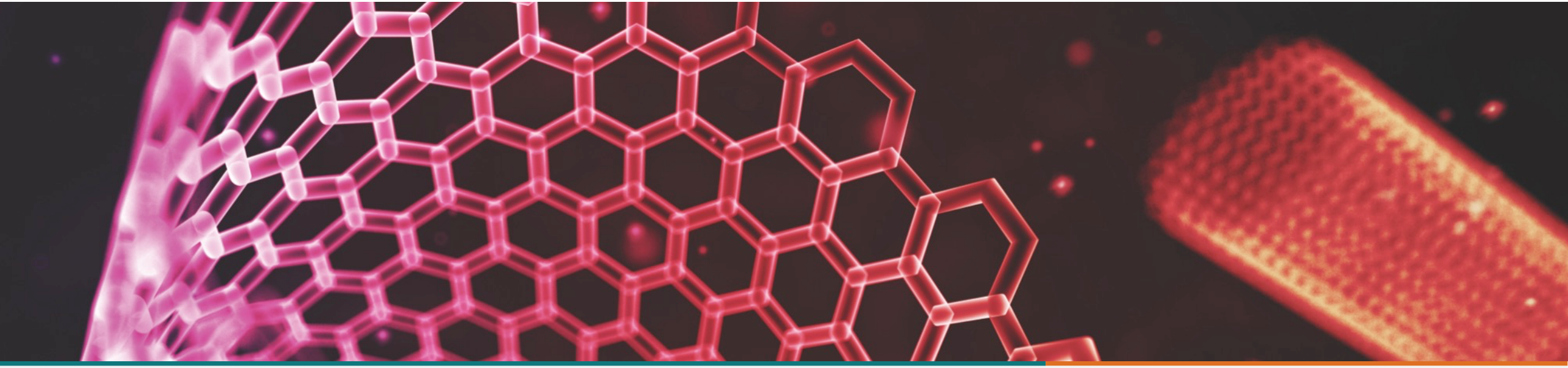
Hello World Too!

Toggle Theme

ThemeContext.js | App.js | ChildComponent.js | ChildChildComponent.js

```
1  import React, { useState } from 'react';
2  import './App.css';
3  import ThemeContext from './ThemeContext';
4  import ChildComponent from './ChildComponent';
5  function App() {
6      const [ theme, setTheme ] = useState({ color: 'red', fontWeight: 'normal' });
7
8      const toggleTheme = () => {
9          if (theme.color === 'red') {
10             setTheme({ color: 'green', fontWeight: 'bold' });
11         } else {
12             setTheme({ color: 'red', fontWeight: 'normal' });
13         }
14     };
15     return (
16         <div className='App'>
17             <ThemeContext.Provider value={theme}>
18                 <ChildComponent />
19             </ThemeContext.Provider>
20
21             <button onClick={toggleTheme}>Toggle Theme </button>
22         </div>
23     );
24 }
25
26 export default App;
27
```

useMemo



useMemo

useMemo

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Returns a memoized value.

Pass a “create” function and an array of dependencies. `useMemo` will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

Remember that the function passed to `useMemo` runs during rendering. Don’t do anything there that you wouldn’t normally do while rendering. For example, side effects belong in `useEffect`, not `useMemo`.

If no array is provided, a new value will be computed on every render.

useMemo

This hook is very similar to **useCallback**, *the difference is that* **useCallback** *returns a memoized callback function instance and* **useMemo** *returns a memoized value as the result of that function call. The use case is different, too.* **useCallback** *is used for callbacks passed to child components. We will look at* ***useCallback*** *next*

Sometimes you have to compute a value, either through a complex calculation or by going to the database to make a costly query or to the network.

Using this hook, this operation is done only once, then the value will be stored in the memorized value and the next time you want to reference it, you'll get it much faster from the cache.

useMemo

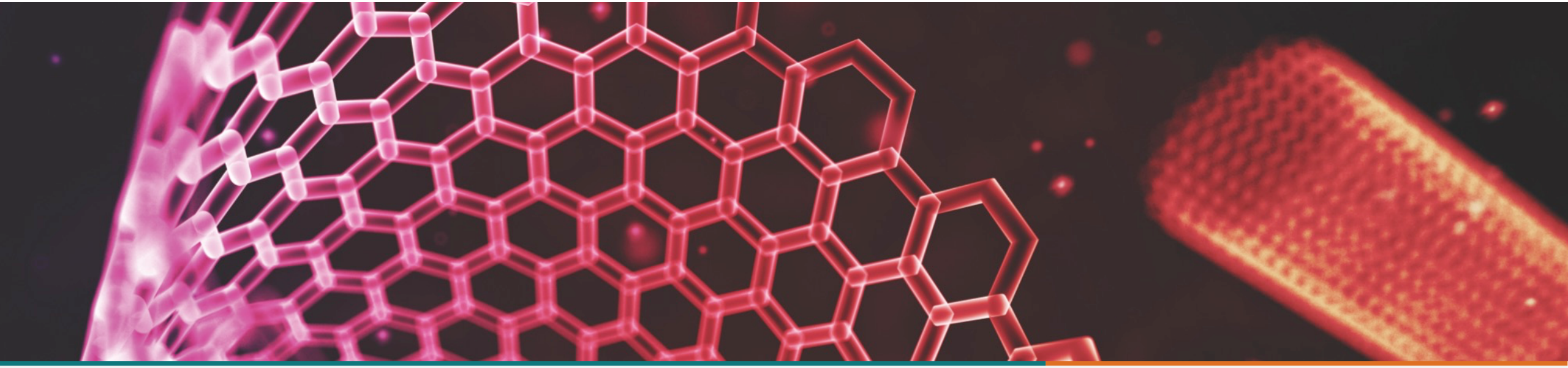
Here is an example of an operation that takes a second or two to complete. This is just a mock operation to simulate an expensive operation.

```
useMemo(
    () => {
        let i = 0;
        while (i < 1000000000) i++;
        return counterOne % 2 === 0;
    },
    [ counterOne ]
);
```

Just like **useEffect**, it takes a 2nd dependency array as mentioned previously. This means this function will only be called when counterOne changes.

Remember that the function passed to **useMemo** runs during rendering. Don't do anything there that you wouldn't normally do while rendering. For example, side effects belong in **useEffect**, not **useMemo**.

useCallback



useCallback

useCallback

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

Returns a memoized callback.

Pass an inline callback and an array of dependencies. `useCallback` will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders (e.g. `shouldComponentUpdate`).

What is the useCallback Hook for?

The useCallback hook is useful for when you do not want a function to be recreated on every render.

Let’s take a look at this code snippet:
Since we are passing a function to Hello,
that is going to get recreated on every render

App.js | Hello.js

```
1   import React, { useState } from 'react';
2   import './App.css';
3   import Hello from './Hello';
4   function App() {
5     const [ count, setCount ] = useState(0);
6     return (
7       <div className='App'>
8         <Hello increment={() => setCount(count + 1)} />
9       </div>
10    );
11  }
12
13  export default App;
```

What is the useCallback Hook for?

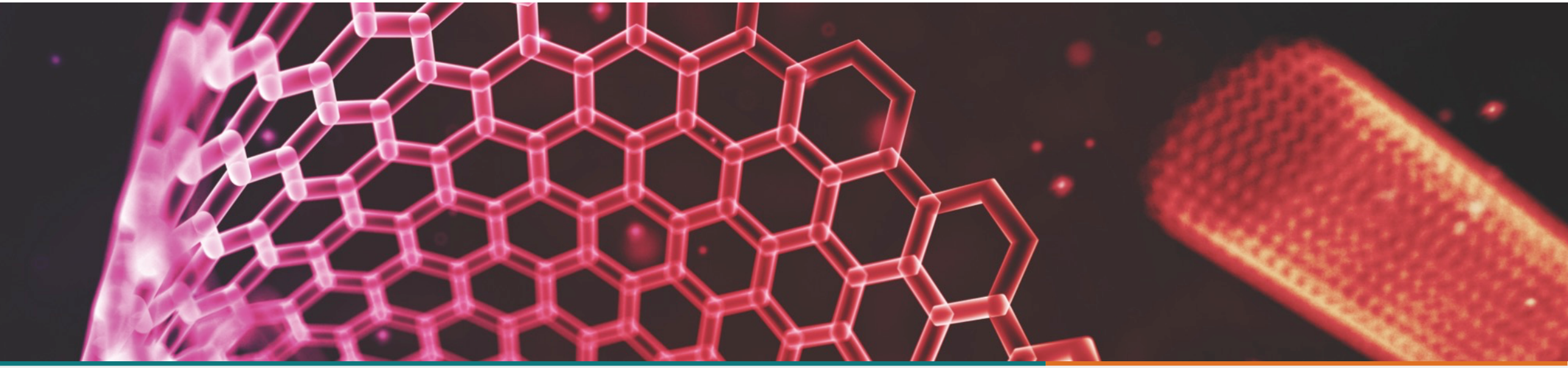
```
    const updateUsername = useCallback(() => {
        setUserName('graffixnyc');
    },[ setUserName ]);

    const updateFirstName = useCallback(() => {
        setFirstName('Patrick');
    },[ setFirstName ]);

    const updateLastName = useCallback(() => {
        setLastName('Hill');
    },[ setLastName ]);

    return (
        <div className='App'>
            <div>
                {username}
                <br />
                {firstName}
                <br />
                {lastName}
                <br />
            </div>
            <button onClick={updateUsername}>Set Username</button>
            <button onClick={updateFirstName}>Set First Name</button>
            <button onClick={updateLastName}>Set Last Name</button>
        </div>
    );
}
```

Create a Custom Hook



Creating a Custom useAxios Hook

useAxios.js × JS App.js

```
1   import { useEffect, useState } from 'react';
2   import axios from 'axios';
3
4   const useAxios = (url) => {
5       const [ state, setState ] = useState({ data: null, loading: true });
6
7       useEffect(
8           () => {
9               setState((state) => ({ data: state.data, loading: true }));
10              axios.get(url).then(({ data }) => {
11                  setState({ data: data, loading: false });
12              });
13          },
14          [ url, setState ]
15      );
16
17      return state;
18  };
19
20  export default useAxios;
```

Creating a Custom useAxios Hook

JS useAxios.js JS App.js

```
1   import React from 'react';
2   import './App.css';
3   import useAxios from './useAxios';
4
5   function App() {
6     let { data, loading } = useAxios('http://api.tvmaze.com/shows');
7     return (
8       <div className='App'>
9         <div>
10          {loading ? (
11            'Loading...'
12          ) : (
13            data.map((show) => {
14              return <div id={show.id}>{show.name}</div>;
15            })
16          )}
17        </div>
18      </div>
19    );
20  }
21
22  export default App;
```

Questions?