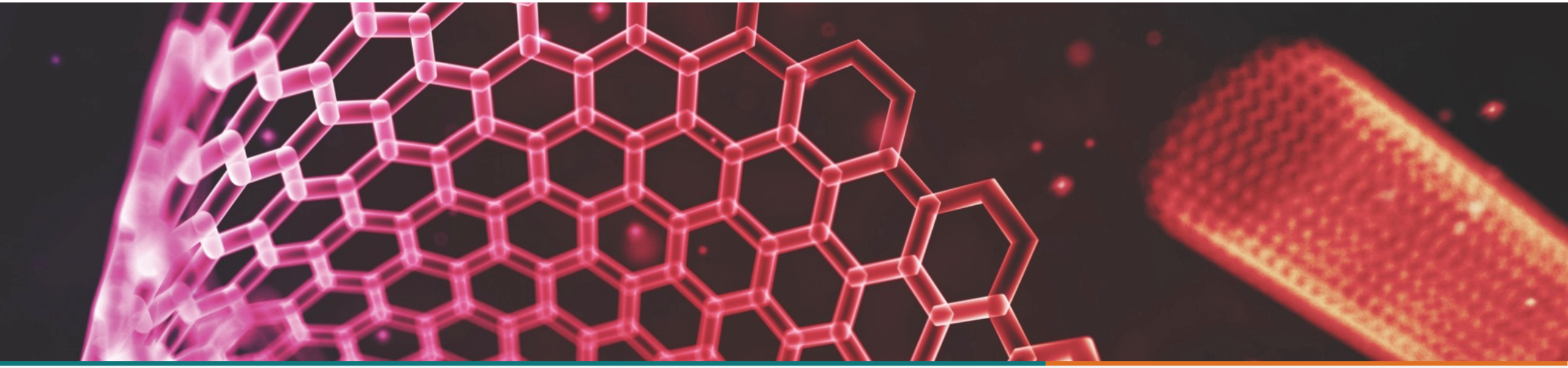
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 554 – Web Programming II

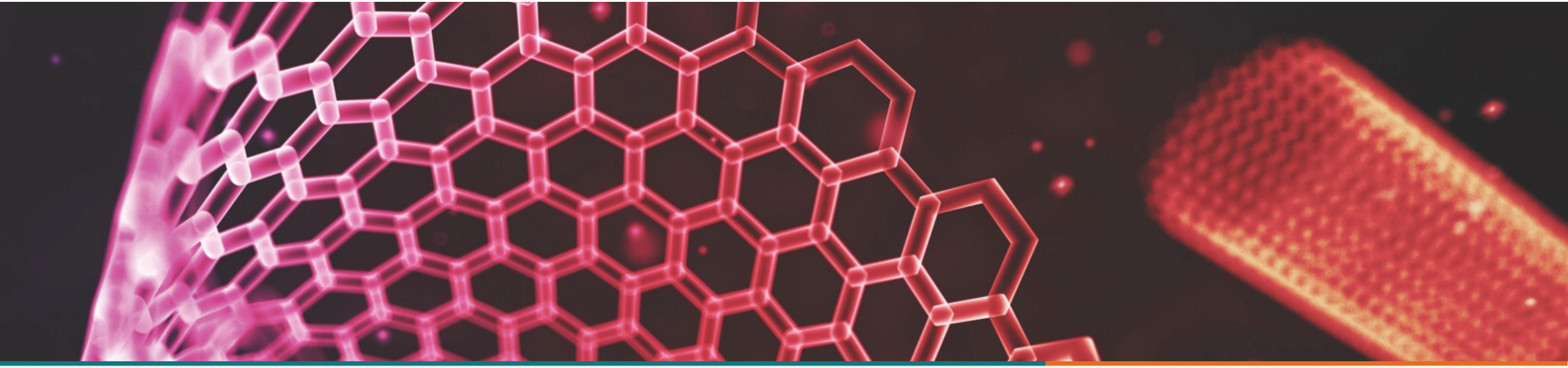
State Management with Context API and useReducer Hook



In this Lecture

We will be looking at how to manage state using the context API, the useContext hook and the useReducer hook.

Context API - Recap



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.

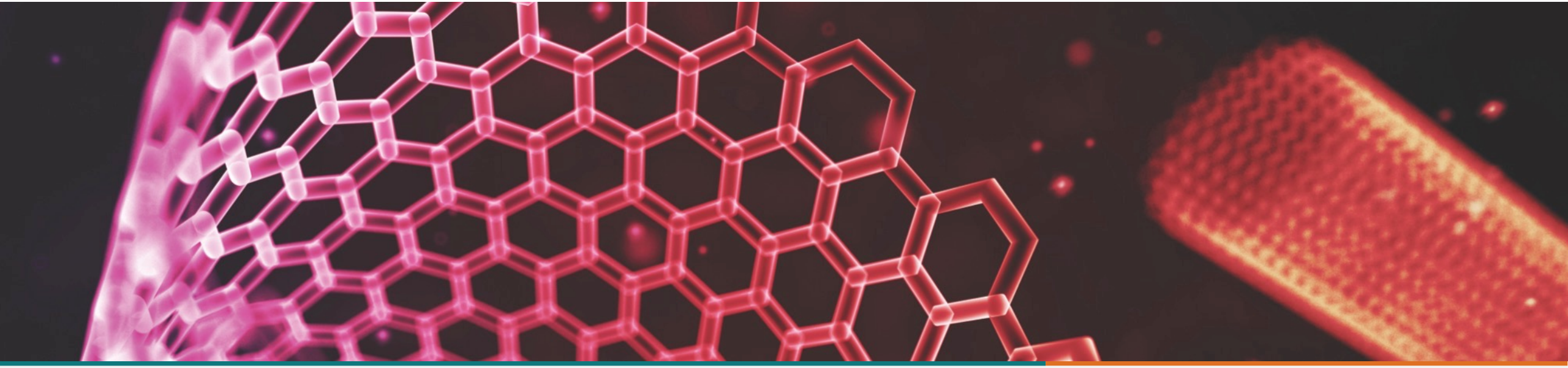
Before the Context API, we could use a module to solve this, which led to the increasing popularity of state management libraries like Redux. Libraries like Redux allows you to get data from the store easily, anywhere in the tree. However, let's focus on the Context API as we will cover Redux in a soon.

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.

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.

ThemeContext.js | App.js | ChildComponent.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

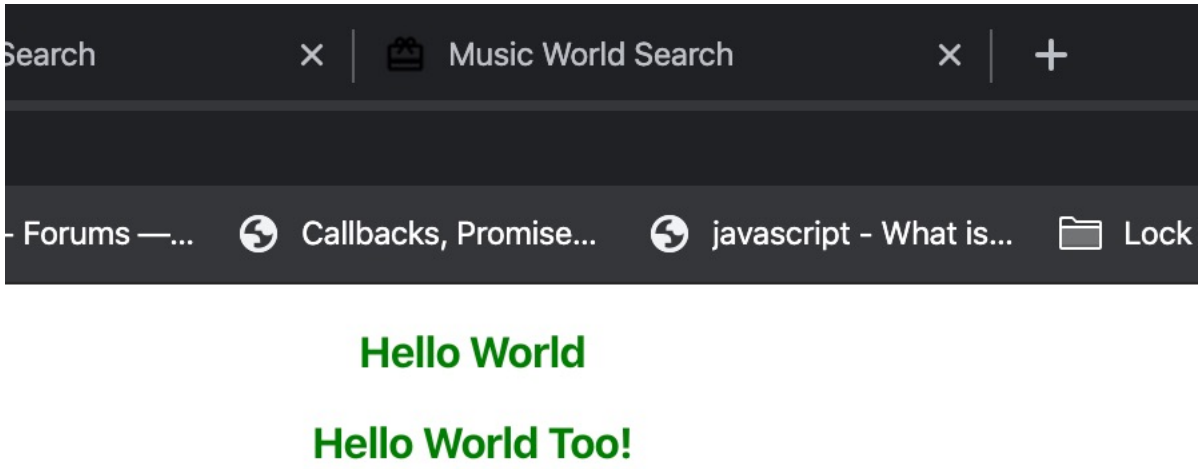
We can do the same for the child's child component.

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

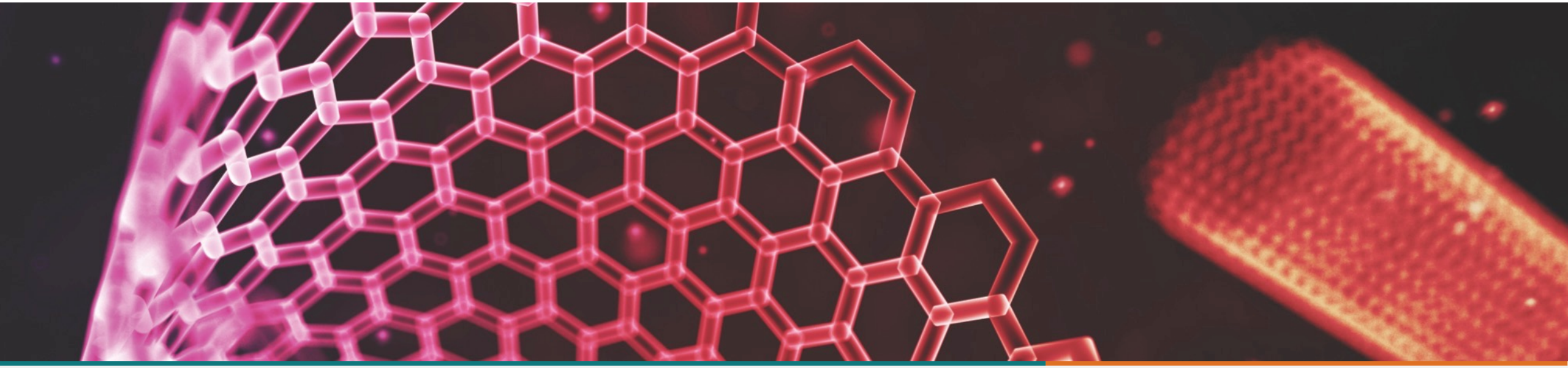
Hello World Too!

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
```

useReducer Hook



useReducer Hook

useReducer

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

An alternative to `useState`. Accepts a reducer of type `(state, action) => newState`, and returns the current state paired with a `dispatch` method. (If you're familiar with Redux, you already know how this works.)

`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 optimize performance for components that trigger deep updates because you can pass `dispatch` down instead of callbacks.

useReducer Hook

Here's the counter example from the `useState` section, rewritten to use a reducer:

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

useReducer Hook

Bailing out of a dispatch

If you return the same value from a Reducer Hook as the current state, React will bail out without rendering the children or firing effects. (React uses the `Object.is` comparison algorithm.)

Note that React may still need to render that specific component again before bailing out. That shouldn't be a concern because React won't unnecessarily go "deeper" into the tree. If you're doing expensive calculations while rendering, you can optimize them with `useMemo`.

useReducer Hook, Using a State Object

```
const intitalState = {
  firstCounter: 0,
  secondCounter: 10
};
const reducer = (state, action) => {
  const { type, payload } = action;
  switch (type) {
    case 'increment':
      return { ...state, firstCounter: state.firstCounter + payload.incBy };
    case 'decrement':
      return { ...state, firstCounter: state.firstCounter - payload.decBy };
    case 'increment2':
      return { ...state, secondCounter: state.secondCounter + payload.incBy };
    case 'decrement2':
      return { ...state, secondCounter: state.secondCounter - payload.decBy };
    case 'reset':
      return intitalState;
    default:
      return state;
  }
};
```

useReducer Hook, Using a State Object

```
<div>Count One: {count.firstCounter}</div>

<button onClick={() => dispatch({ type: 'increment', payload: { incBy: 1 } })}>
  Increment Counter One by 1</button>
<br />
<button onClick={() => dispatch({ type: 'decrement', payload: { decBy: 1 } })}>
  Decrement Counter One by 1</button>
<br />
<button onClick={() => dispatch({ type: 'increment', payload: { incBy: 5 } })}>
  Increment Counter One by 5</button>
<br />
<button onClick={() => dispatch({ type: 'decrement', payload: { decBy: 5 } })}>
  Decrement Counter One by 5</button>
<br />
<div>Count Two: {count.secondCounter}</div>
<button onClick={() => dispatch({ type: 'increment2', payload: { incBy: 1 } })}>
  Increment Counter Two by 1
</button>
<br />
<button onClick={() => dispatch({ type: 'decrement2', payload: { decBy: 1 } })}>
  Decrement Counter Two by 1
</button>
<br />
<button onClick={() => dispatch({ type: 'increment2', payload: { incBy: 5 } })}>
  Increment Counter Two by 5
</button>
<br />
<button onClick={() => dispatch({ type: 'decrement2', payload: { decBy: 5 } })}>
  Decrement Counter Two by 5
</button>
<br />
<button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
</div>
```

```
(property) decBy: number
```

Using useContext & useReducer

useReducer is for dealing with local state (state for a specific component) but if we want to have the same functionality as Redux has where we have a global state that is shared among many other components we can combine useReducer and useContext! We will see an example in our code, It's the same application we saw in the Redux lecture except it uses useReducer and the context API/useContext

Questions?