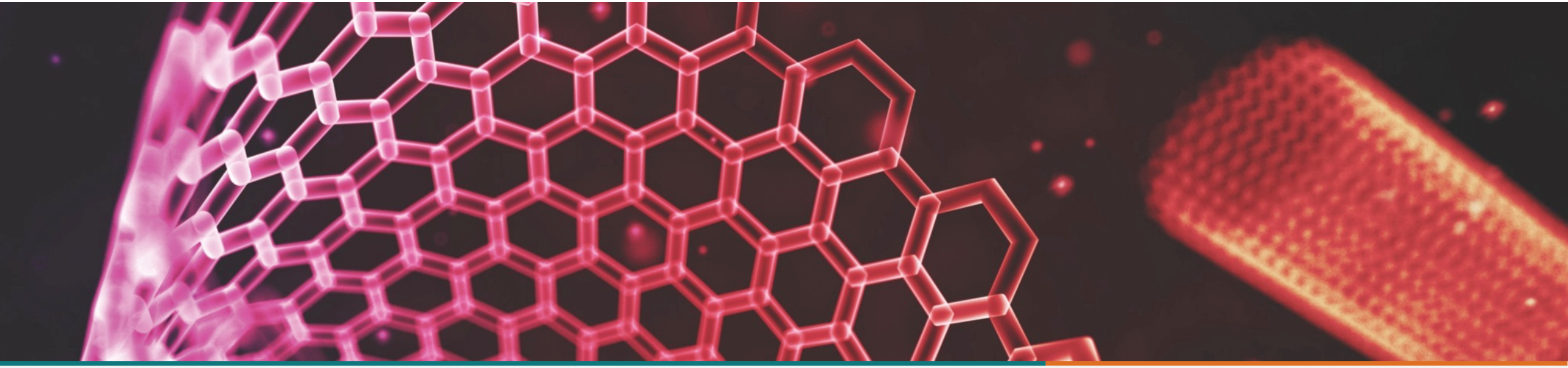
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 554 – Web Programming II

React



What is React?

It's a ***UI component library***. React is an open-source JavaScript library for building user interfaces. It was developed and is maintained by Facebook and a community of individual developers and companies. React is primarily used for developing single-page applications and mobile applications, but it can be used for building dynamic user interfaces in various web applications.

The UI components are created with React using JavaScript, not a special template language. This approach is called ***creating composable UIs***, and it's fundamental to React's philosophy.

The Problem React Solves

What problem does React solve? Looking at the last several years of web development, note the ***problems in building and managing complex web UIs for front-end applications***: React was born primarily to address those.

Benefits of Using React

- **Component-Based** React follows a component-based architecture, which means you can break down your user interface into small, reusable components. This promotes code reusability and maintainability.
- **Virtual DOM Performance** React uses a Virtual DOM to optimize the rendering process. When data changes, React updates a virtual representation of the DOM rather than the actual DOM. This leads to faster updates and better performance. Its diffing algorithm is highly efficient.
- **Strong Community and Ecosystem** React has a large and active community. This means you can find a wealth of open-source libraries, tools, and resources to help with development, which can significantly speed up your projects.
- **Great for Single Page Applications (SPAs)** React is well-suited for building SPAs where most of the application logic is moved to the client side. It provides a smooth user experience by reducing the need for full-page reloads.

Simplicity

In React, this simplicity is achieved with the following features:

- ***Declarative over imperative style***—React embraces declarative style over imperative by updating views automatically.

- ***Component-based architecture using pure JavaScript***—React doesn't use domain-specific languages (DSLs) for its components, just pure JavaScript. And there's no separation when working on the same functionality.

- ***Powerful abstractions***—React has a simplified way of interacting with the DOM, allowing you to normalize event handling and other interfaces that work similarly across browsers.

Declarative vs Imperative Style

Declarative style means developers write ***how it should be, not what to do, step-by-step (imperative)*.**

But why is declarative style a better choice? The benefit is that declarative style reduces complexity and makes your code easier to read and understand.

Declarative vs Imperative Style

```
1   //imperative
2   var arr = [1, 2, 3, 4, 5],
3     arr2 = []
4   for (var i=0; i<arr.length; i++) {
5     arr2[i] = arr[i]*2
6   }
7   console.log('a', arr2)
8
9   //declarative
10  var arr = [1, 2, 3, 4, 5],
11    arr2 = arr.map(function(v,i){ return v*2 })
12  console.log('b', arr2)
```

Output:

```
a [ 2, 4, 6, 8, 10 ]
b [ 2, 4, 6, 8, 10 ]
```

Which code snippet is easier to read and understand?

Declarative vs Imperative Style

The convenience of React's declarative style fully shines when you need to make changes to the view. Those are called changes of the ***internal state***. ***When the state changes, React updates the view accordingly***

React and the Virtual DOM

React uses a ***virtual DOM*** to find differences (the delta) between what's already in the browser and the new view. This process is called ***DOM diffing*** or ***reconciliation of state and view*** (bringing them back to similarity). This means developers don't need to worry about explicitly changing the view; all they need to do is update the state, and the view will be updated automatically as needed.

Powerful Abstractions

React has a powerful abstraction of the document model. It hides the underlying interfaces and provides normalized/synthesized methods and properties.

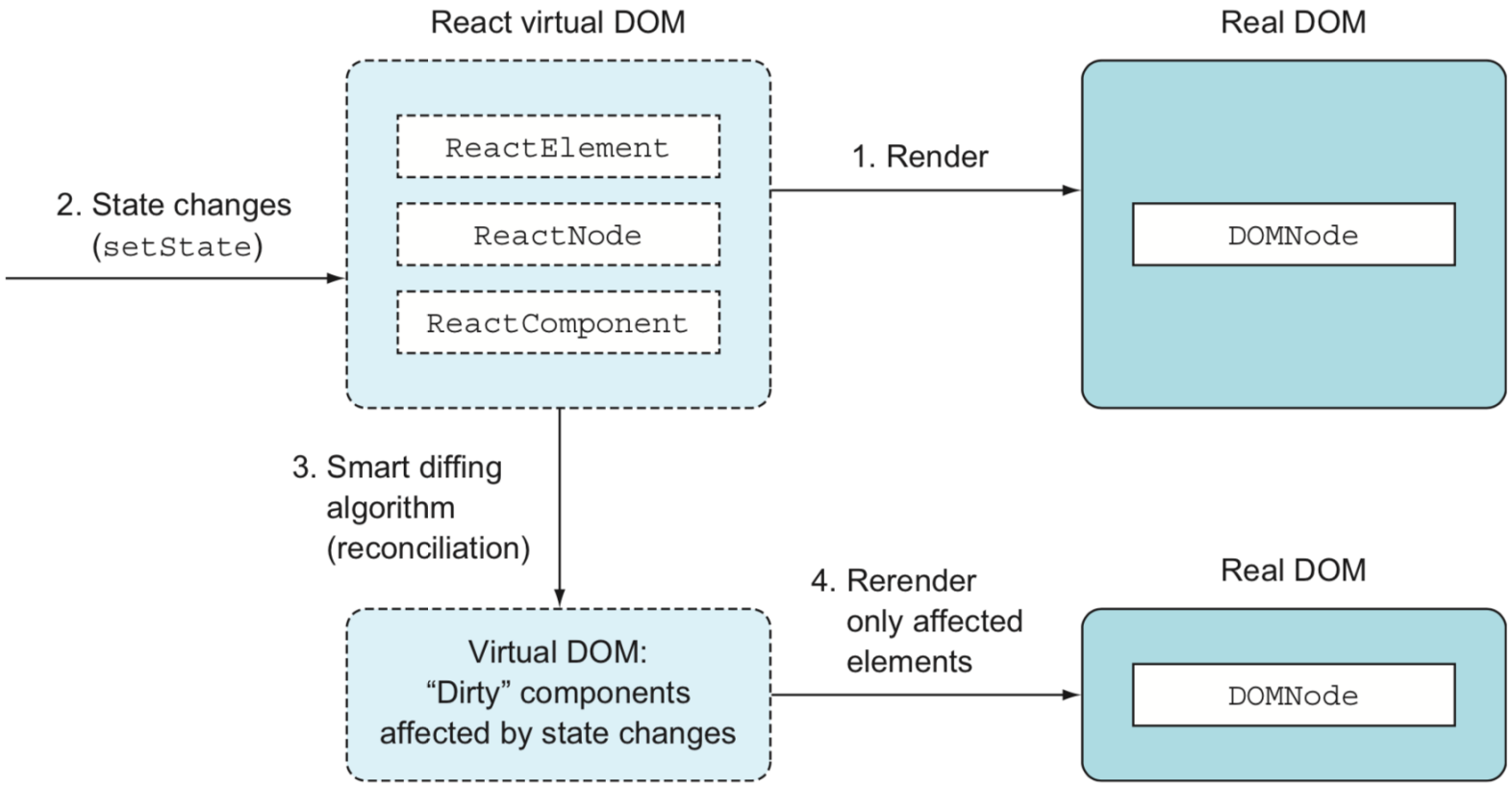
- React applications can be made SEO-friendly by server-side rendering (SSR) and tools like Next.js. This is important for websites that need to be indexed by search engines.

Speed and Testability

React's virtual DOM exists only in the JavaScript memory. ***Every time there's a data change, React first compares the differences using its virtual DOM; only when the library knows there has been a change in the rendering will it update the actual DOM.***

Speed and Testability

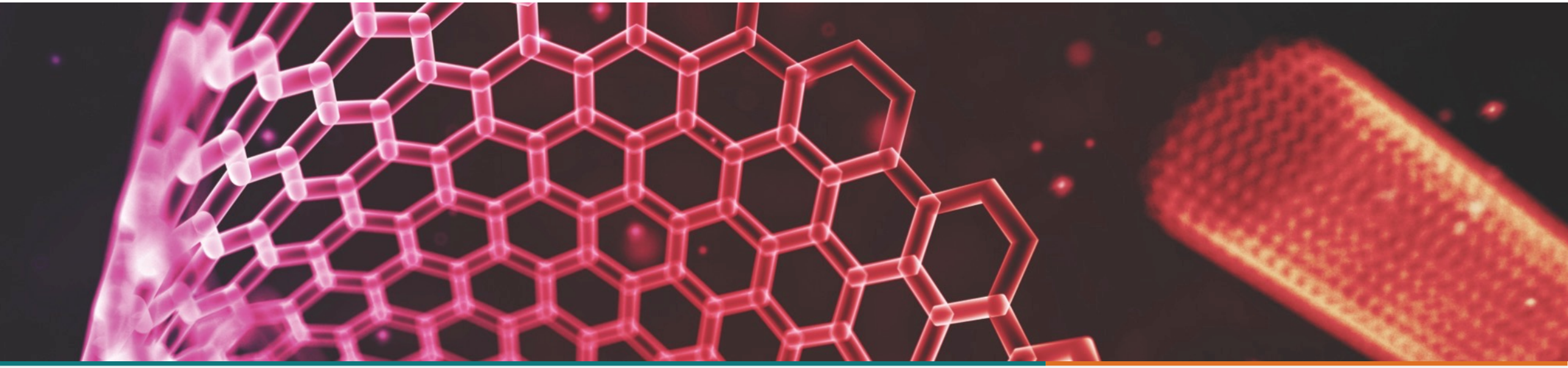
Once a component has been rendered, if its state changes, it's compared to the in-memory virtual DOM and re-rendered if necessary.



Disadvantages of React

- **Learning Curve** React can have a steep learning curve, especially for developers new to the component-based architecture and JSX (JavaScript XML) syntax. This may require more time for teams to become proficient.
- **JSX Complexity** JSX, which is a JavaScript extension for writing HTML within your JavaScript code, can be confusing for developers who are not used to it. It may lead to difficulties in code comprehension for some team members.
- **SEO Challenges** Although React can be made SEO-friendly with server-side rendering (SSR), implementing SSR can add complexity to your project, and not all React applications use this technique.
- **State Management** React doesn't come with a built-in state management solution, and developers often rely on third-party libraries like Redux or the Context API. Choosing the right state management approach and implementing it correctly can be challenging.

Single Page Applications



Single-Page Applications and React

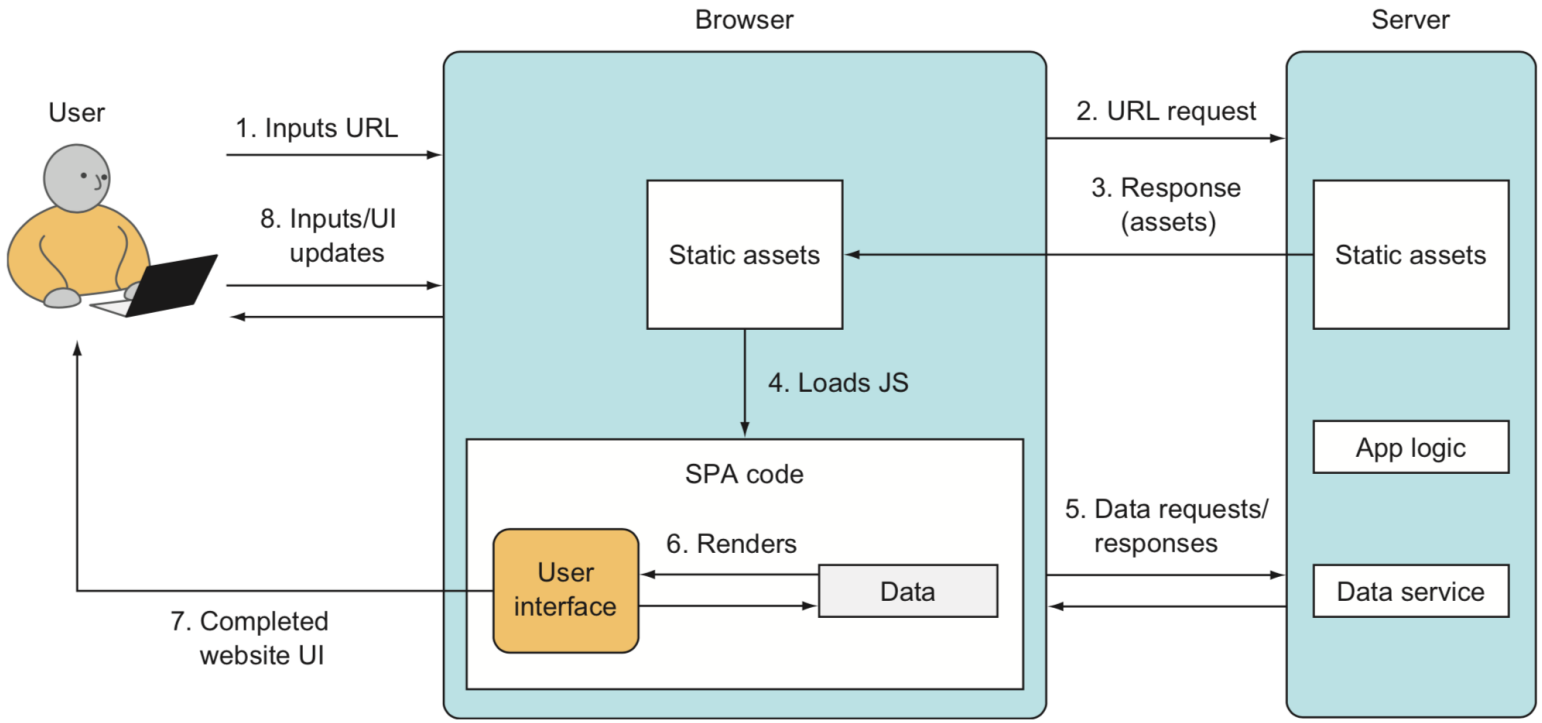
What is a Single-Page Application?

- A **single-page application** (**SPA**) is a web application or website that interacts with the web browser by dynamically rewriting the current web page with new data from the web server, instead of the default method of the browser loading entire new pages. The goal is faster transitions that make the website feel more like a native app.
- In a SPA, all necessary HTML, JavaScript, and CSS code is either retrieved by the browser with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page, although the location hash or the HTML5 History API can be used to provide the perception and navigability of separate logical pages in the application.

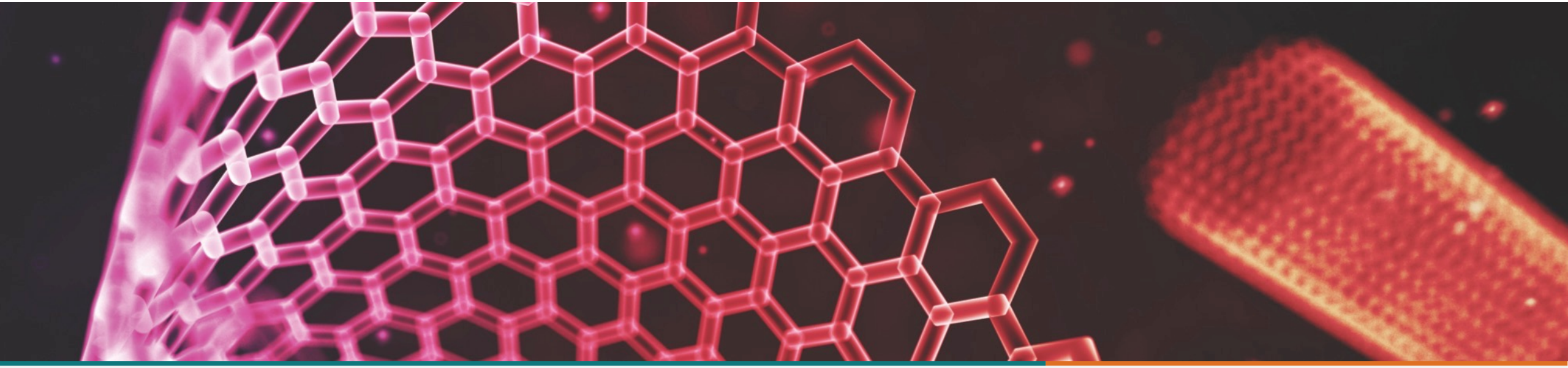
Another name for SPA architecture is ***thick client***, because the browser, being a client, holds more logic and performs functions such as rendering of the HTML, validation, UI changes, and so on. Let's take a bird's-eye view of a typical SPA architecture with a user, a browser, and a server.

Single-Page Applications and React

The figure depicts a user making a request, and input actions like clicking a button, drag-and-drop, mouse hovering, and so on:



JSX



What is JSX?

- JSX stands for JavaScript XML.
- JSX allows us to write HTML in React.
- JSX makes it easier to write and add HTML in React.

JSX is a preprocessor step that adds HTML like syntax to JavaScript. You can definitely use React without JSX but JSX makes React a lot more elegant.

Just like HTML, JSX tags have a tag name, attributes, and children. If an attribute value is enclosed in quotes, the value is a string. Otherwise, wrap the value in braces and the value is the enclosed JavaScript expression.

You can think of it as a very simple way of making an HTML Template that gets compiled into a JS Template Function, in a manner of speaking.

Coding With JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.

JSX converts HTML tags into react elements.

JSX:

```
const myelement = <h1>I Love JSX!</h1>;

ReactDOM.render(myelement, document.getElementById('root'));
```

Without JSX:

```
const myelement = React.createElement('h1', {}, 'I do not use JSX!');

ReactDOM.render(myelement, document.getElementById('root'));
```

To JSX

```
<div className="red">Children Text</div>;
<MyCounter count={3 + 5} />;

// Here, we set the "scores" attribute below to a JavaScript object.
var gameScores = {
  player1: 2,
  player2: 5
};
<DashboardUnit data-index="2">
  <h1>Scores</h1>
  <Scoreboard className="results" scores={gameScores} />
</DashboardUnit>;
```

Or Not to JSX

```
React.createElement("div", { className: "red" }, "Children Text");
React.createElement(MyCounter, { count: 3 + 5 });

React.createElement(
  DashboardUnit,
  { "data-index": "2" },
  React.createElement("h1", null, "Scores"),
  React.createElement(Scoreboard, { className: "results", scores: gameScores })
);
```

The example on the previous slide gets compiled to the following without JSX. I hope you will agree JSX syntax reads more naturally.

JSX

In the example below, we declare a variable called `name` and then use it inside JSX by wrapping it in curly braces:

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

You can put any valid JavaScript expression inside the curly braces in JSX. For example, `2 + 2`, `user.firstName`, or `formatName(user)` are all valid JavaScript expressions.

JSX

In the example below, we embed the result of calling a JavaScript function, `formatName(user)`, into an `<h1>` element.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

JSX - Expressions

JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of `if` statements and `for` loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

JSX - Children

Specifying Children with JSX

If a tag is empty, you may close it immediately with `/>`, like XML:

```
const element = <img src={user.avatarUrl} />;
```

JSX tags may contain children:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

JSX – Prevents Injection Attacks

JSX Prevents Injection Attacks

It is safe to embed user input in JSX:

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

JSX – Represents Objects

Babel compiles JSX down to `React.createElement()` calls.

These two examples are identical:

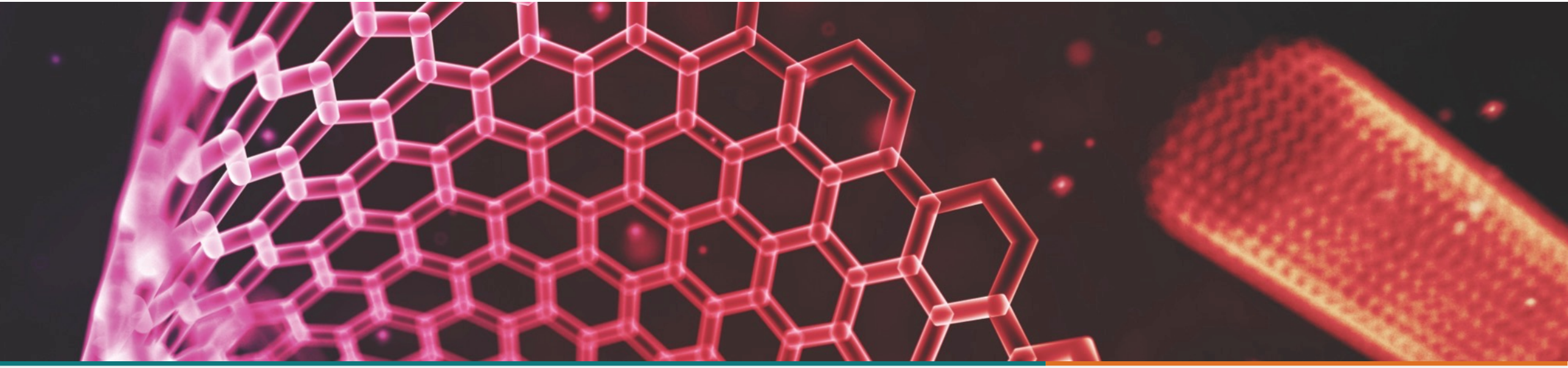
```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

`React.createElement()` performs a few checks to help you write bug-free code but essentially it creates an object like this:

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

React Components



Two Types of React Components

In React there are two types of components

Function Components – Function components were named Functional Components previously. Functional components were React components that do not contain state. They were often referred to as "**stateless**" components If we wanted our component to contain state, we needed to create a Class Component but now with the use of React Hooks, we can have state for Functional Components! And the fine folks at React have rebranded them **Function Components**. Function components are now the preferred and default way to create React components

Class Components – Class Components used to be the default component type in React and you really needed class components to build out anything useful in React, but now that function components can handle state using React hooks, function components have now become the default component type in React and class components are not being used for modern React apps.

We will be covering Function Components in the course.

Functional Component Example

```jsx
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.jsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}

export default App
```

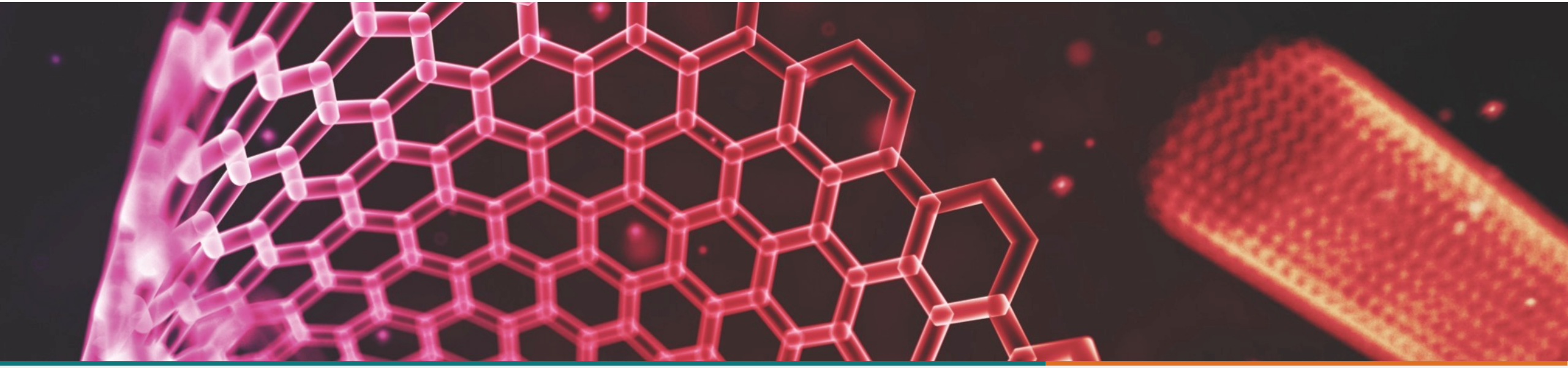
The default App.jsx when you use Vite that is a function component

Class Component Example

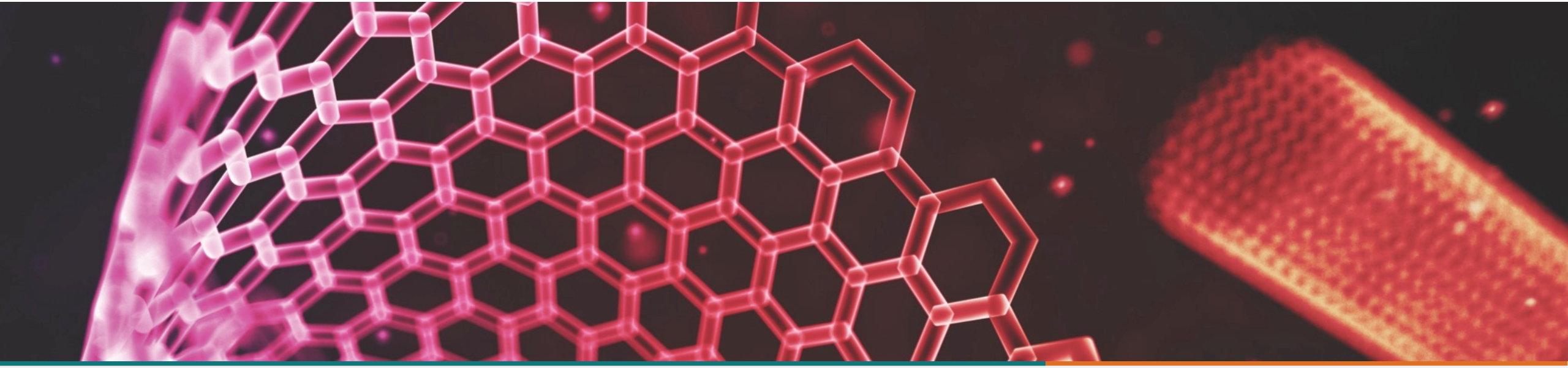
```jsx
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
    render() {
        return (
            <div className='App'>
                <header className='App-header'>
                    <img src={logo} className='App-logo' alt='logo' />
                    <p>
                        Edit <code>src/App.js</code> and save to reload.
                    </p>
                    <a className='App-link' href='https://reactjs.org' target='_blank' rel='noopener noreferrer'>
                        Learn React
                    </a>
                </header>
            </div>
        );
    }
}

export default App;
```

No longer covered in the course since everything is going Function based now

Component Properties and State



Properties



Properties

Properties are a cornerstone of the declarative style that React uses. Think of properties as ***unchangeable values within an element***. One component (the parent component) passes data to its child components as properties or what it's more known as in the React world, "props".

One thing to remember is that ***properties are immutable within their components***. A parent assigns properties to its children upon their creation. ***The child element isn't supposed to modify its properties.*** (A *child* is an element nested inside another element; for example, <h1/> is a child of <HelloWorld/>.)

You can pass a property PROPERTY_NAME with the value VALUE, like this:

<TAG PROPERTY_NAME=VALUE/>

Properties closely resemble HTML attributes.

You can pass any type datatype as a prop: Objects, Booleans, Numbers etc.. You can even pass functions in as properties!

Properties

You can use these properties for a number of things such as:

- Rendering different elements based on the properties passed in
- Passing in initial data for the component to consume (like a user object perhaps with user data, or data with the results of an API call)
- Passing in a function to lift state.

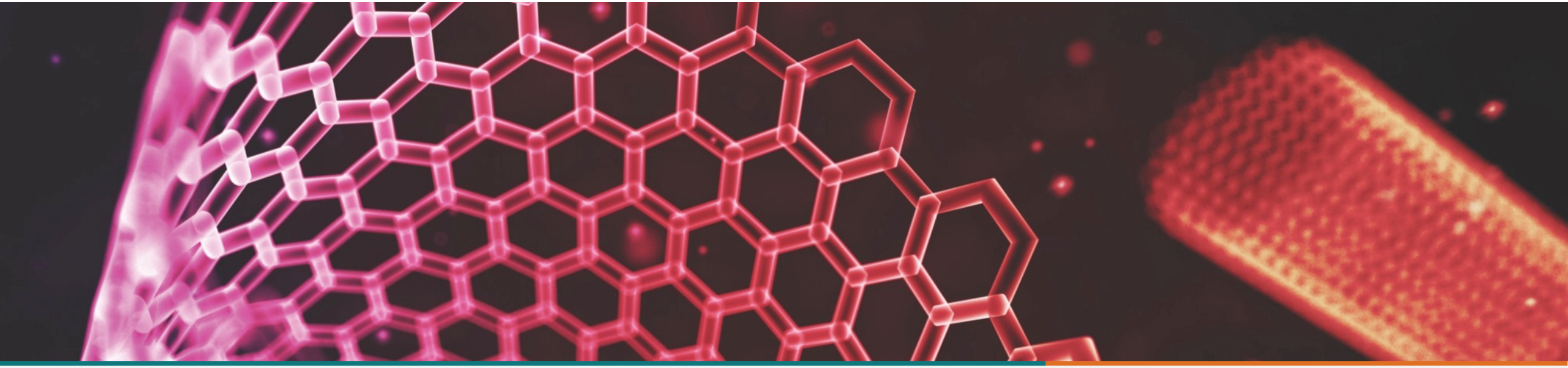
Properties

```jsx
import PropsExample from './PropsExample.jsx';

function App() {
  const greeting = 'Hello Function Component!';

  const handle_func = (name) => {
    console.log(`Hello ${name} from within handle_func in app.js`);
  };
  return (
    <div className='App'>
      <PropsExample
        user={{name: 'Patrick Hill', username: 'graffixnyc'}}
        handleClick={handle_func}
        greeting={greeting}
      />
    </div>
  );
}

export default App;
```

Properties

```
import './App.css';
import Child from './Child';

function PropsExample(props) {
  let h1 = null;

  const btnClick = ()=>{
    props.handleClick(props.user.name)
  }
  if (props.greeting) {
    h1 = <h1>{props.greeting}</h1>;
  } else {
    h1 = <h1>Hello there!</h1>;
  }
  return (
    <div>
      {h1}
      <h2>{props.user.name}</h2>
      <button onClick={btnClick}>{props.user.username}</button>
      <Child greeting={props.greeting} />
    </div>
  );
}

export default PropsExample;
```

State



What is State?

The state can be defined as an object or a single variable or a set of **observable** properties that control the behavior of the component. In other words, the State of a component holds some information that may change over the lifetime of the component. You can think of state as the components data that can change.

For example, let us think of a clock component, if we store the current time in state, then every second we should update the state of the time and the UI will re-render the element to show the current time.

We deal with state in React Function components using the **useState** React hook

State vs Props

State - This is data maintained inside a component. It is local or owned by that specific component.

Props - Data passed in from a parent component. props are read-only in the child component that receives them. However, callback functions can also be passed, which can be executed inside the child to initiate an update.

The difference is all about which component owns the data. State is owned locally and updated by the component itself. Props are owned by a parent component and are read-only. Props can only be updated if a callback function is passed to the child to trigger an upstream change.

The state of a parent component can be passed a prop to the child. They are referencing the same value, but only the parent component can update it.

State

State Updates May Be Asynchronous

Because *`state`* may be updated asynchronously, you should not rely on their values for calculating the next state.
For example, this code may fail to update the counter:

```
setCount(count + 1)
```

To fix it, use a second form of setting state that accepts a function rather than an object. That function will receive the previous state as the first argument.

```
setCount((count) => count + 1)
```

State – The Data Flows Down

Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn't care whether it is defined as a function or a class.

This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.
A component may choose to pass its state down as props to its child components:

```
function Clock(props) {
  const [clockData, setClock] = useState(props.date);
  const [counter, setCounter] = useState(0);
```

Here we are setting two pieces of state. The initial state for our clock is set by a property being passed into our Clock component from the parent component (App.jsx in this case). We are setting the initial state of the counter to 0

We then consume the state when we render the component:

```
return (
  <div className='App'>
    <h1>Counter: {counter}</h1>
    <h2>the current time is: {clockData.toLocaleTimeString()}</h2>
  </div>
);
```

State vs Props Recap

React introduces a concept of *state* and *properties*

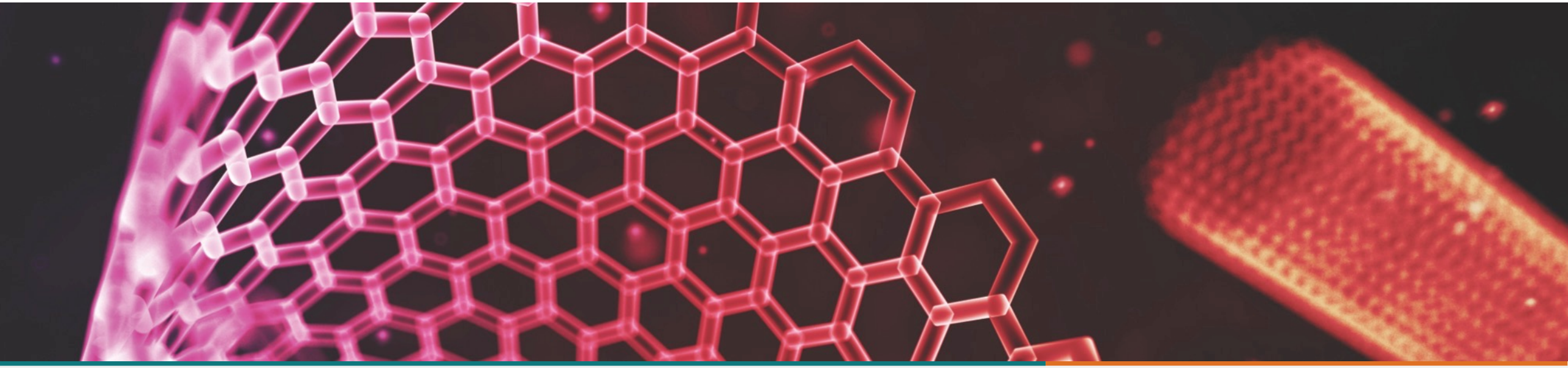
- State is the internal state of your component at a point in time. It is changed based on user action. Your state is the **data of the component at a point in time**. Generally, states revolve around UI updates.
- Props (properties) can be seen as the configuration. **Props cannot be changed**.

We can build components with or without states. Components without states are called *stateless components*.

You can read some details and opinions online:

- https://github.com/uberVU/react-guide/blob/master/props-vs-state.md

React Hooks



What Are React Hooks?

React hooks are nothing more than functions that “hook” into React’s state and lifecycle features from a function component.

- Hooks allow us to easily manipulate the state of our functional component without needing to convert them into class components.

So no longer do we need to use class-based components if we want our component to deal with state.

React Hooks do not work in class-based components, they can only be used in function components.

Rules of Hooks

- Don’t call Hooks inside loops, conditions, or nested functions—*Only call Hooks **at the top level**.*
- Don’t call Hooks from regular JavaScript functions—Only call Hooks **from React function components**.

Some Basic React Hooks

Today we will cover two of the most used React Hooks. We will look at other hooks in future lectures.

- **useState**
- **useEffect**

useState

Using the React Hook useState, we can more easily handle state.

First, we need to import the hook:

```
import React, { useState} from 'react';
```

Once we import the hook, then we can set the initial state of the component:

```
const ShowList = () => {
    const [ searchData, setSearchData ] = useState(undefined);
    const [ showsData, setShowsData ] = useState(undefined);
    const [ searchTerm, setSearchTerm ] = useState('');
```

For each piece of state that our component uses, we have a separate statement, no more having to have a state object.

useState

```
const ShowList = () => {
    const [ searchData, setSearchData ] = useState(undefined);
    const [ showsData, setShowsData ] = useState(undefined);
    const [ searchTerm, setSearchTerm ] = useState('');
```

useState is actually an array, the first element is the value of the state, the second element is a function that sets the state, so we destructure those two elements out into useful names like searchData for the value, and setSearchData as the update function.

Let’s see an example without using destructuring and then using it.

```jsx
import React, { useState } from 'react';

const App = (props) => {
    const count = useState(0);
    const text = useState('');

    return (
        <div>
            <p>
                The current {text[0] || 'count'} is: {count[0]}
            </p>
            <button onClick={() => count[1](count[0] + 1)}>Increment</button>
            <button onClick={() => (count[0] <= 0 ? count[1](0) : count[1](count[0] - 1))}>Decrement</button>
            <button onClick={() => count[1](0)}>Reset</button>
            <br />
            <br />
            <label>
                Enter Text:
                <input value={text[0]} onChange={(e) => text[1](e.target.value)} />
            </label>
        </div>
    );
};

export default App;
```

```jsx
import React, { useState } from 'react';

const App = (props) => {
    const [ count, setCount ] = useState(0);
    const [ text, setText ] = useState('');

    return (
        <div>
            <p>
                The current {text || 'count'} is: {count}
            </p>
            <button onClick={() => setCount(count + 1)}>Increment</button>
            <button onClick={() => (count <= 0 ? setCount(0) : setCount(count - 1))}>Decrement</button>
            <button onClick={() => setCount(0)}>Reset</button>
            <br />
            <br />
            <label>
                Enter Text:
                <input value={text} onChange={(e) => setText(e.target.value)} />
            </label>
        </div>
    );
};

export default App;
```

useEffect

React Class Components have lifecycle methods:

componentDidMount: componentDidMount() is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

componentDidUpdate: componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

componentWillUnmount: componentWillUnmount() is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

The useEffect Hook functions **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount** combined.

useEffect

Using the React Hook useEffect, we can more have lifecycle method like behavior in function components.

First, we need to import the hook:

```
import React, { useState, useEffect } from 'react';
```

Once we import the hook, then we can call the useEffect method:

useEffect() takes a function as an input and returns nothing. The function it takes will be executed for you **after every** render cycle.

```
useEffect(() => {
  console.log('useEffect has been called');
  setUser({ firstName: 'Patrick', lastName: 'Hill', username: 'graffixnyc' });
});
```

useEffect

The function inside of useEffect() gets executed unnecessarily (i.e. whenever the component re-renders)

We have an infinite loop because setUser() causes the function to re-render

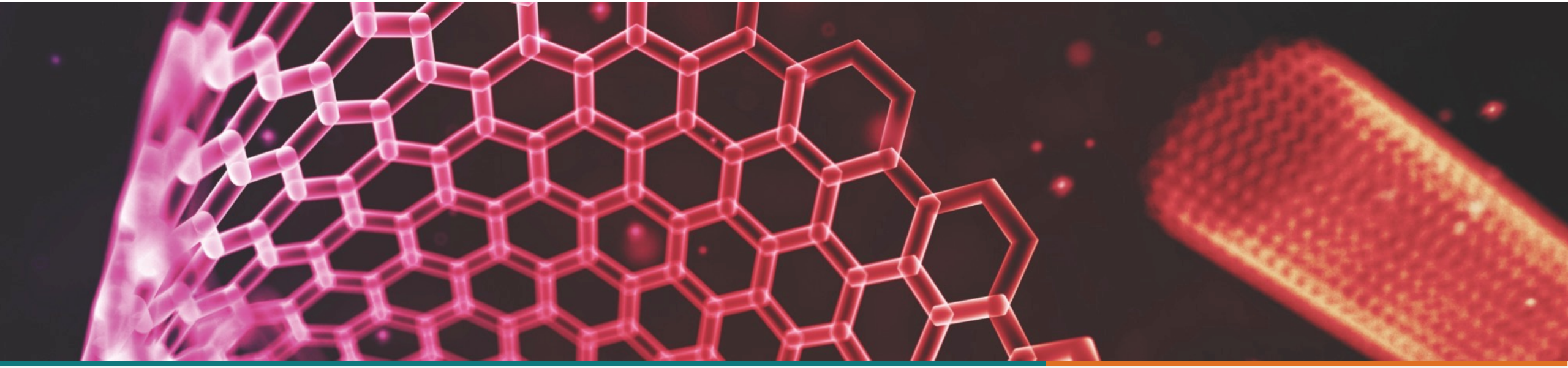
```
useEffect(() => {
  console.log('useEffect has been called');
  setUser({ firstName: 'Patrick', lastName: 'Hill', username: 'graffixnyc' });
});
```

useEffect() takes a second argument which allows us to control when the function that was passed in as the first argument will be executed.

```
useEffect(() => {
  console.log('useEffect has been called');
  setUser({ firstName: 'Patrick', lastName: 'Hill', username: 'graffixnyc' });
}, []);
```

Here, we pass an empty array ([]) as the second argument.
This leads React to only execute the function passed as the first argument when the component is rendered for the first time. Effectively, it now behaves like componentDidMount. We will see a case where the array is not empty in our code demo!

Installing and Running a React App



Using React with Vite

We will be using Vite to create the boilerplate React Application for us:

You can run: ***npm create vite@latest my-react-app -- --template react-swc***

This will create a react application using JavaScript and the SWC.

SWC (stands for Speedy Web Compiler) is a super-fast TypeScript / JavaScript compiler written in Rust. It's a library for Rust and JavaScript at the same time. If you are using SWC from Rust, see rustdoc and for most users, your entry point for using the library will be parser. In the past, this was done with Babel which is a JavaScript transpiler.

After we create our application, we CD into that directory (my-react-app in the above example), run **npm install**, and to run the app run **npm run dev**

Note: You can also run ***npm create vite@latest*** and it will walk you through which template you'd like to use.

Next Lecture

In our next lecture, We will learn about form processing and event handling, we will also learn about how to fetch data from an API using the useEffect hook. We will also cover React Router Dom so we can perform routing in our React Applications.

Questions?