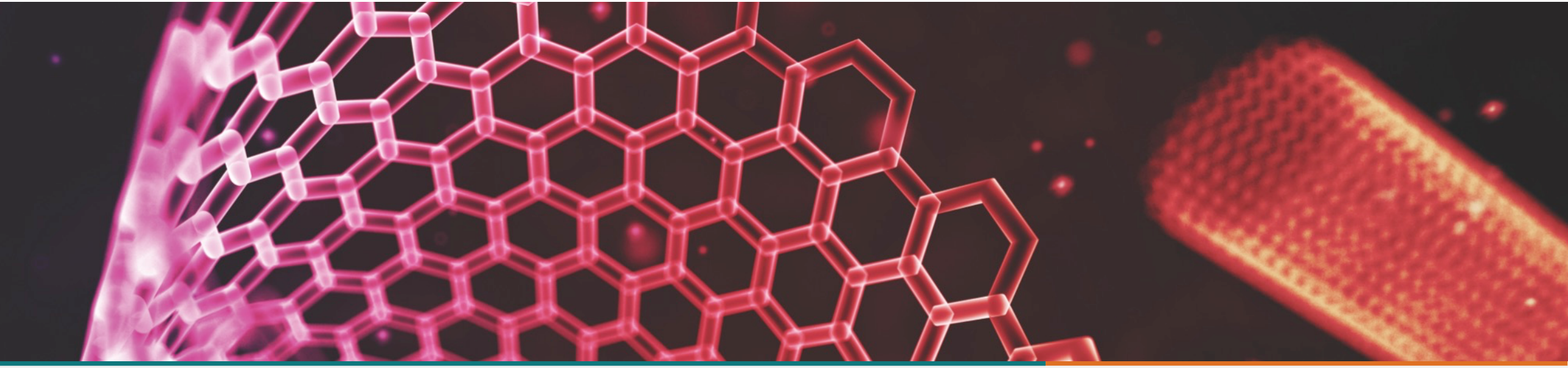
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 554 – Web Programming II

React



What is React?

It's a ***UI component library***. 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.

React UI components are highly self-contained, concern-specific blocks of functionality.

What is React?

A few Examples of some components

- date-picker
- Captcha
- Address
- ZIP code
- Autocomplete
- Contact Form
- User Sign-in/Sign-up

The Problem React Solves

What problem does React solve? Looking at the last few 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

- ***Simpler apps***—React has a CBA with pure JavaScript; a declarative style; and powerful, developer-friendly DOM abstractions (and not just DOM, but also iOS, Android, and so on).

- ***Fast UIs***—React provides outstanding performance thanks to its virtual DOM and smart-reconciliation algorithm, which, as a side benefit, lets you perform testing without spinning up (starting) a headless browser.

- ***Less code to write***—React's great community and vast ecosystem of components provide developers with a variety of libraries and components. This is important when you're considering what framework to use for development.

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.

Component-Based Architecture Using Pure JavaScript.

- Component-based architecture existed before React came on the scene.
- Separation of concerns, loose coupling, and code reuse are at the heart of this approach because it provides many benefits
- A building block of CBA in React is the component class.
- As with other CBAs, it has many benefits, with code reuse being the main one (you can write less code!).

Component-Based Architecture Using Pure JavaScript.

The HTML and JavaScript separation worked well when you had to render HTML on the server, and JavaScript was only used to make your text blink.

Now, single page applications (SPAs) handle complex user input and perform rendering on the browser. This means HTML and JavaScript are closely coupled functionally. For developers, it makes more sense if they don't need to separate between HTML and JavaScript when working on a piece of a project (component).

Powerful Abstractions

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

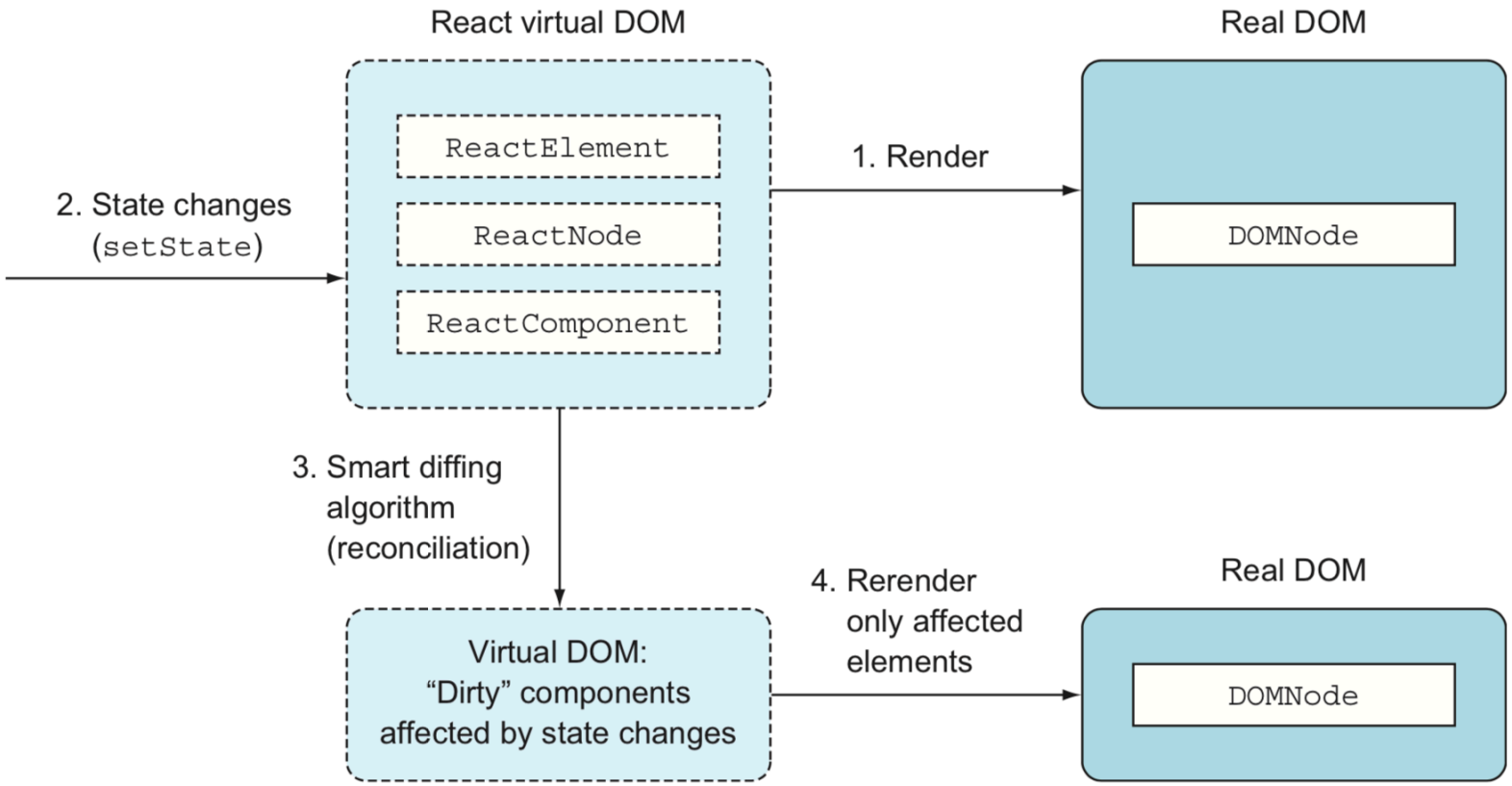
Another example of React's DOM abstraction is that you can render React elements on the server. This can be handy for better search engine optimization (SEO) and/or improving performance.

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.



Ecosystem and Community

- ***The React community is incredible. Most of the time, developers don't even have to implement much of the code. Look at these community resources:***

- ***List of React components***: https://github.com/brillout/awesome-react-components and http://devarchy.com/react-components
- ***Set of React components that implement the Google Material Design specification*** (https://design.google.com): http://react-toolbox.com
- ***Material Design React components:*** www.material-ui.com
- ***Collection of React components for Office and Office 360 experiences*** (http://
- dev.office.com/fabric#/components) using the Office Design Language: https://github.com/OfficeDev/office-ui-fabric-react
- ***Opinionated catalog of open source JS (mostly React) packages***: https://js.coach
- ***Catalog of React components***: https://react.rocks
- ***Khan Academy React components***: https://khan.github.io/react-components
- ***Registry of React components***: www.reactjsx.com

Disadvantages of React

- ***React isn't a full-blown, Swiss Army knife–type of framework*.**
- ***React isn't as mature as other frameworks*.**
- ***React uses a somewhat new approach to web development.***
- ***React only has a one-way binding*.**
- ***React isn't reactive***

Single-Page Applications and React

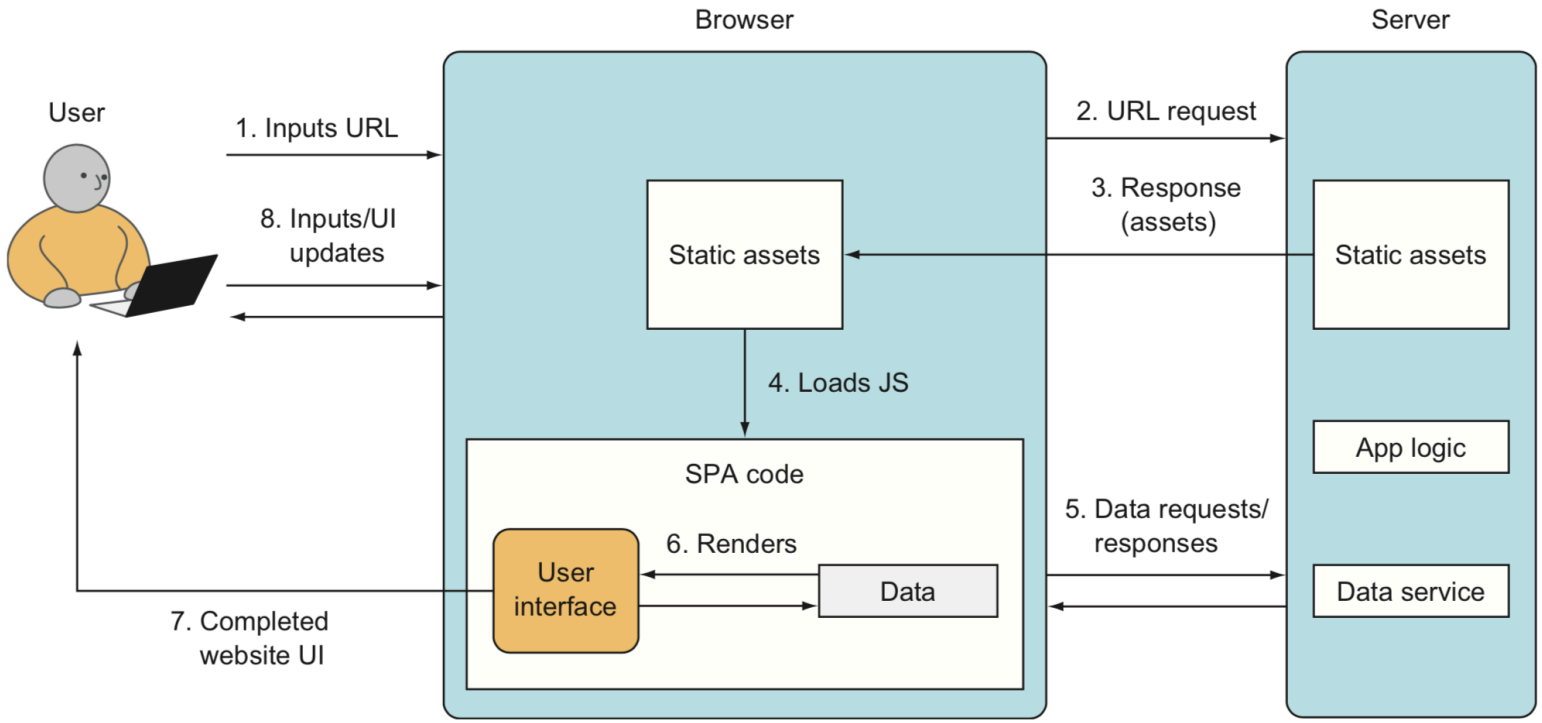
What is a Single-Page Application?

- 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:



Hello World: Rendering an Element

Get the gist on GitHub here: https://goo.gl/iQxWZa

```
<!DOCTYPE HTML>
<html lang="en">
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Hello World</title>
        <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    </head>

    <body>
        <div id="content"></div>
        <script>
            let h1 = React.createElement('h1', null, 'Hello world!')
            ReactDOM.render(h1, document.getElementById('content'))
        </script>
    </body>
</html>
```

Nesting Elements

We saw in the previous example we render one element, the problem is that most UIs have more than one element (such as a link inside a menu).

The solution to creating more-complex structures in a hierarchical manner is nesting elements. In the previous example, we implemented our first React code by creating an h1 React element and rendering it in the DOM with ReactDOM.render():

It's important to note that ReactDOM.render() takes only one element as an argument, which is h1 in the example

Nesting Elements

In this case, you can wrap the elements in a visually neutral element, the container is usually a good choice, as is .

You can pass an unlimited number of parameters to createElement().

All the parameters after the second one become child elements. Those child elements (h1, in this case) are siblings—that is, they're on the same level relative to each other.

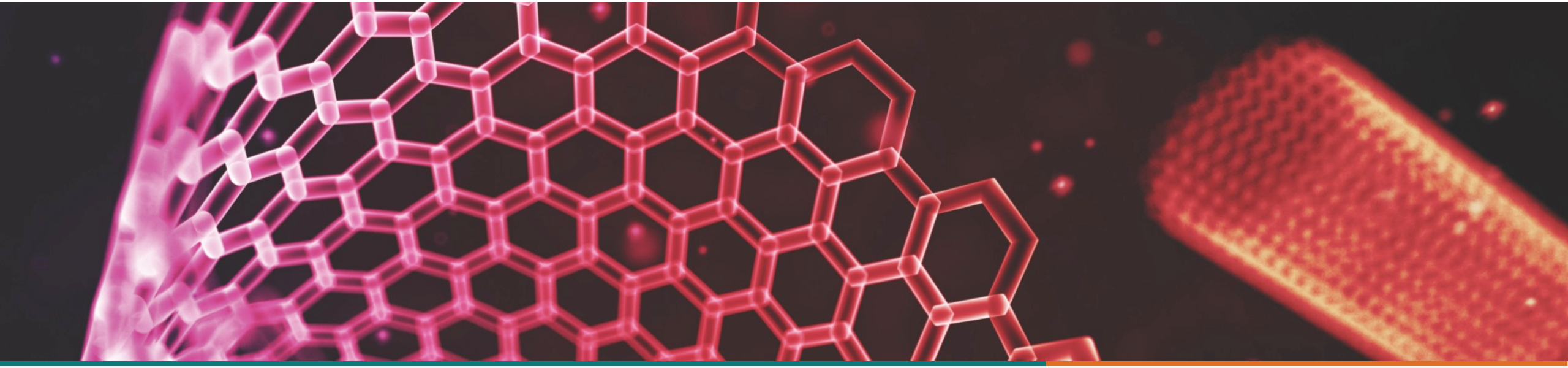
Knowing this, let's use createElement() to create the element with two <h1> child elements

Nesting Elements – Updated Hello World

```
<!DOCTYPE HTML>
<html lang="en">
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Hello World</title>
        <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
        <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
    </head>

    <body>
        <div id="content"></div>
        <script>
            let h1 = React.createElement('h1', null, 'Hello world!')
            let p = React.createElement('p',null, "This is a P tag, it will be placed after the h1")
            let div = React.createElement('div',null,h1,p)
            ReactDOM.render(div, document.getElementById('content'))
        </script>
    </body>
</html>
```

React Components



Two Types of React Components

In React there are two main 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**.

Class Components – Class Components used to be the default component type in React and you really needed class components to build out anything really 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 still be learning class components as there are many existing systems in production in the industry that still have them implemented so they are still useful to know.

In the next lecture we will go through React hooks and adding state to functional components but for this lecture, we will focus on class and stateless function components.

Functional Component Example

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  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;
```

The new default app.js when you use create-react-app that is a function component

Class Component Example

```
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;
```

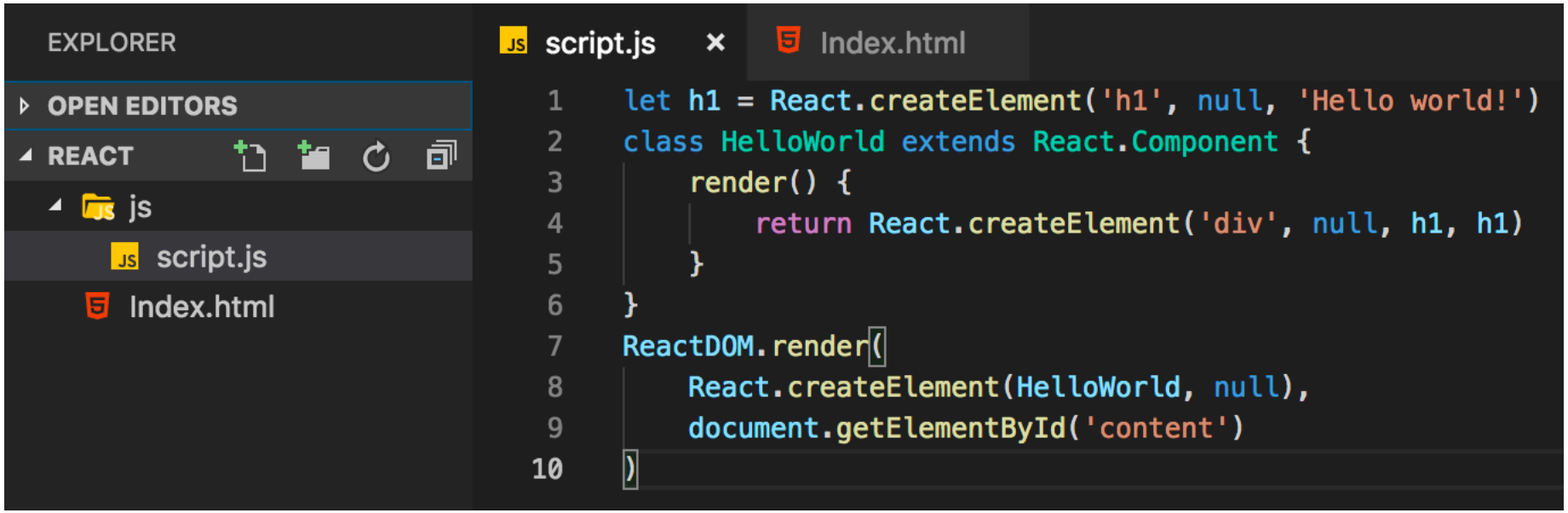
Creating Class Components

- ***Standard HTML tag as a string***; for example, 'h1', 'div', or 'p' (without the angle brackets). The name is lowercase.

- ***React component classes as an object***; for example, HelloWorld. The name is capitalized.

Let's Go Back To Hello World as a Class Component

GitHub gist: https://goo.gl/p1tByg



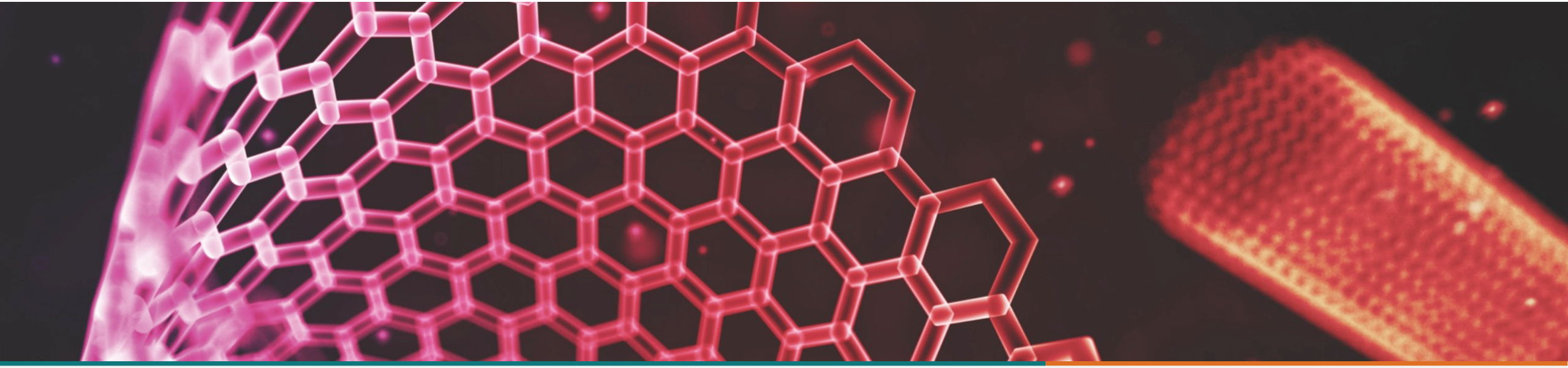
Still using the CDN method of using react, we can create a class in a script.js file and then just include a reference to it in the HTML page.

Let's Go Back To Hello World as a Class Component

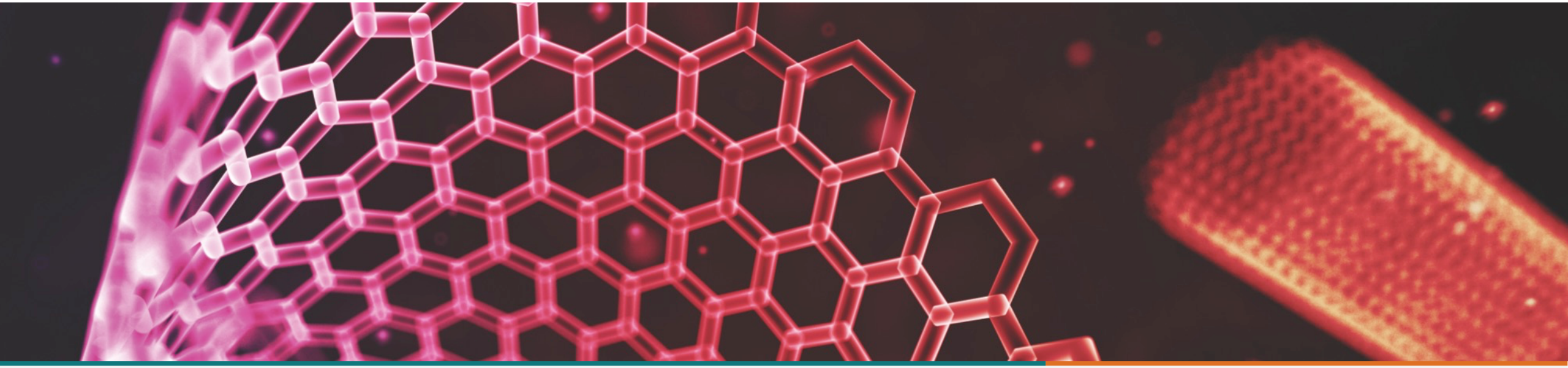
Updated Index.html which includes script.js

```
1  <!DOCTYPE HTML>
2  <html lang="en">
3      <head>
4          <meta http-equiv="content-type" content="text/html; charset=utf-8">
5          <title>Hello World</title>
6          <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
7          <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
8      </head>
9
10     <body>
11         <div id="content"></div>
12         <script src="js/script.js"></script>
13
14     </body>
15 </html>
```

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 – Function Components

```jsx
import React from 'react';
import PropsExample from './PropsExample';
const App = () => {
    const greeting = 'Hello Function Component!';

    const handle_func = () => {
        console.log('Hello');
    };
    return (
        <PropsExample
            greeting={greeting}
            user={{ name: 'Patrick Hill', username: 'graffixnyc' }}
            handleChange={handle_func}
        />
    );
};

export default App;
```

```jsx
import React from 'react';
const PropsExample = (props) => {
    let h1 = null;
    if (props.greeting) {
        h1 = <h1>{props.greeting}</h1>;
    } else {
        h1 = <h1>Hello There</h1>;
    }
    return (
        <div>
            {h1}
            <p>{props.user.name}</p>

            <button onClick={props.handleChange}>{props.user.username}</button>
        </div>
    );
};
export default PropsExample;
```

Properties – Class Components

```jsx
import React, { Component } from 'react';
import './App.css';
import PropsExample from './PropsExample';
class App extends Component {
    render() {
        const greeting = 'Hello Function Component!';
        const handle_func = () => {
            console.log('Hello');
        };
        return (
            <PropsExample
                greeting={greeting}
                user={{ name: 'Patrick Hill', username: 'graffixnyc' }}
                handleChange={handle_func}
            />
        );
    }
}

export default App;
```

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

class PropsExample extends Component {
    render() {
        let h1 = null;
        if (this.props.greeting) {
            h1 = <h1>{this.props.greeting}</h1>;
        } else {
            h1 = <h1>Hello There</h1>;
        }
        return (
            <div>
                {h1}
                <p>{this.props.user.name}</p>
                <button onClick={this.props.handleChange}>{this.props.user.username}</button>
            </div>
        );
    }
}

export default PropsExample;
```

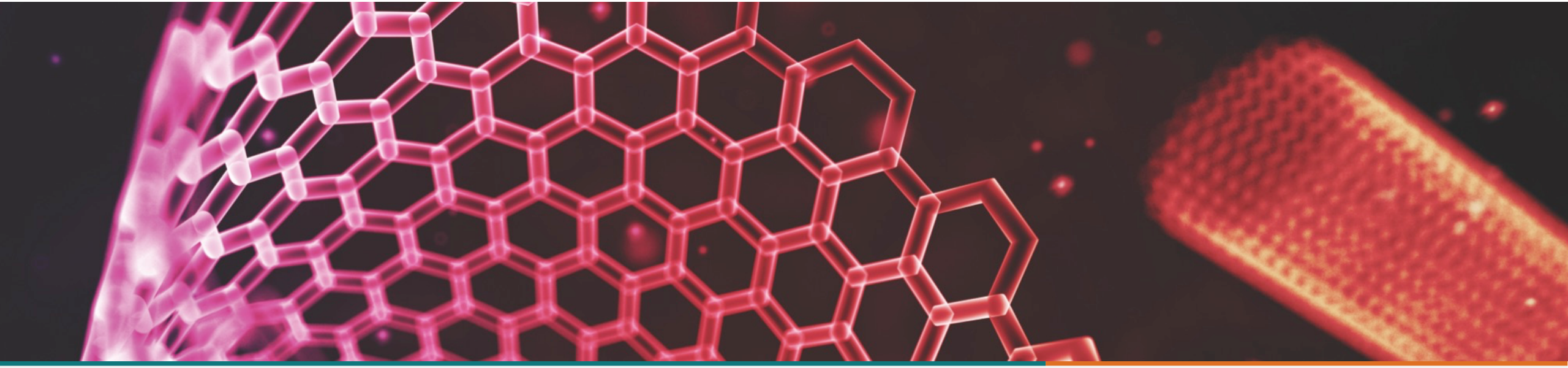
Properties : Back to our CDN Hello Example!

Updated script.js

```
class HelloWorld extends React.Component {
    render() {
        return React.createElement(
            'h1',
            this.props,
            `Welcome to ${this.props.frameworkname},

        ${this.props.message}`
        );
    }
}
ReactDOM.render(
    React.createElement(
        'div',
        null,
        React.createElement(HelloWorld, { id: 'react', frameworkname: 'React.js', message: 'React is awesome!' }),
        React.createElement(HelloWorld, { id: 'vue', frameworkname: 'Vue.js', message: 'Vue JS is nice.' }),
        React.createElement(HelloWorld, { id: 'angular', frameworkname: 'angular', message: 'Angular is angular :P' })
    ),
    document.getElementById('content')
);
```

State



What is State?

The state is an instance of React Component Class can be defined as an object of a set of **observable** properties that control the behavior of the component. In other words, the State of a component is an object that 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 will discuss dealing with state for function components next lecture, so the following slides refer to handling state in class components

State vs Props

State - This is data maintained inside a component. It is local or owned by that specific component. The component itself will update the state using the ***setState*** function (when using class components).

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 Object in Class Components

The ***state object*** is an attribute of a component and can be accessed with a ***this*** reference; for example, ***this.state.name***. You can access and print variables in JSX with curly braces (***{}***). Similarly, you can render ***this.state*** (like any other variable or custom component class attribute) in render(); for example, ***{this.state.inputFieldValue}***. This syntax is similar to the way you access properties with ***this.props.name***.

Setting Initial State in a Class Component

When you want to use state in your render method you need to initialize the state. We do this in the constructor method within your React class like so:

```
constructor(props) {
    super(props);
    this.state = {
        userID: this.props.uid,
        userName: this.props.userName
    };
}
```

Setting the initial state from props that were passed in by the parent

```
constructor(props) {
    super(props);
    this.state = {
        postid: undefinded,
        postauthor: undefined,
        posttext: undefined
    };
}
```

Setting the initial state to be undefined (it will be set by the component later)

Setting State in a Class Component

When we set state we need to do it in a certain way. We can't simply type:

```
this.state.author="Patrick Hill"
```

We need to use the `setState()` method:

```
this.setState({author: 'Patrick Hill'});
```

The only place where you can use the `this.state` function is to set the state is in the constructor

State

State Updates May Be Asynchronous

React may batch multiple ***setState()*** calls into a single update for performance.

Because ***this.props*** and ***this.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:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

State

To fix it, use a second form of ***setState()*** that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

State – Updates are Merged

When you call `setState()`, React merges the object you provide into the current state.

For example, your state may contain several independent variables:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

State – Updates are Merged

Then you can update them independently with separate setState() calls:

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

The merging is shallow, so `this.setState({comments})` leaves `this.state.posts` intact, but completely replaces `this.state.comments`.

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:

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

This also works for user-defined components:

```
<FormattedDate date={this.state.date} />
```

The `FormattedDate` component would receive the `date` in its props and wouldn't know whether it came from the `Clock`'s state, from the `Clock`'s props, or was typed by hand:

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

State vs Props Recap

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

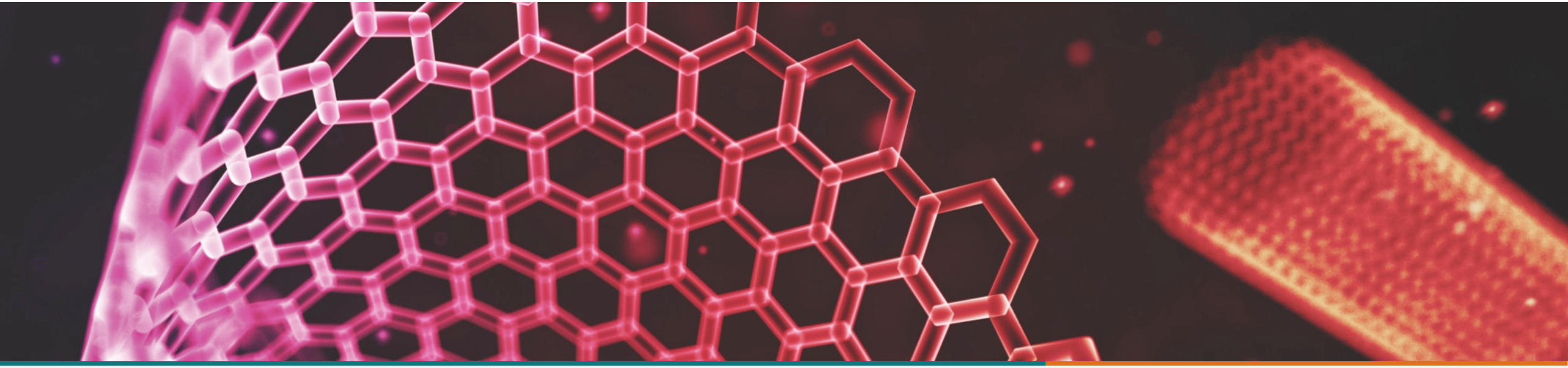
- State is the internal state of your component at a point in time. It is changed based on user action. Your state is the **data of the component at a point in time**. Generally, states revolve around UI updates.
- Props (properties) can be seen as the configuration. **Props cannot be changed**.

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

JSX



What is JSX?

JSX (JavaScript XML) is a preprocessor step that adds HTML like syntax to JavaScript. You can definitely use React without JSX as we have seen 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.

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 as we have seen 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:

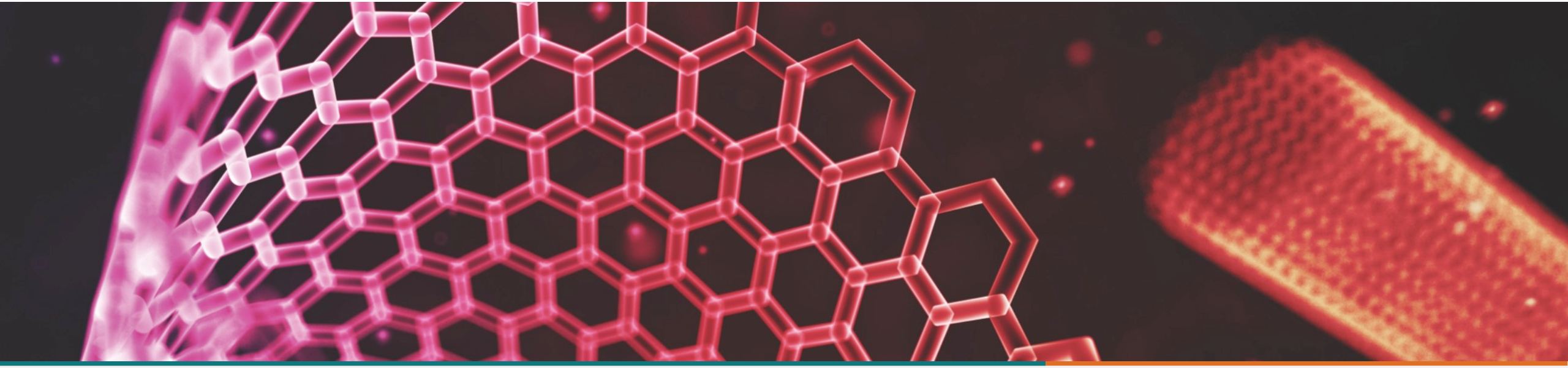
```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
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!'
  }
};
```

Babel



What is Babel?

In the previous slide we said that Babel compiles JSX down. Babel is a JavaScript compiler; it transforms JavaScript JSX from new-style code to old-style code.

- This allows you to write with new syntax that is more descriptive and faster to understand, but output code that runs consistently across all browsers.

Babel also allows you to extend JavaScript syntax, allowing you to do things like write JSX and annotations. These will get compiled down to ES2015, as well.

When we use create-react-app it sets up Babel for us so we can just JSX out of the box without having to configure it

https://babeljs.io

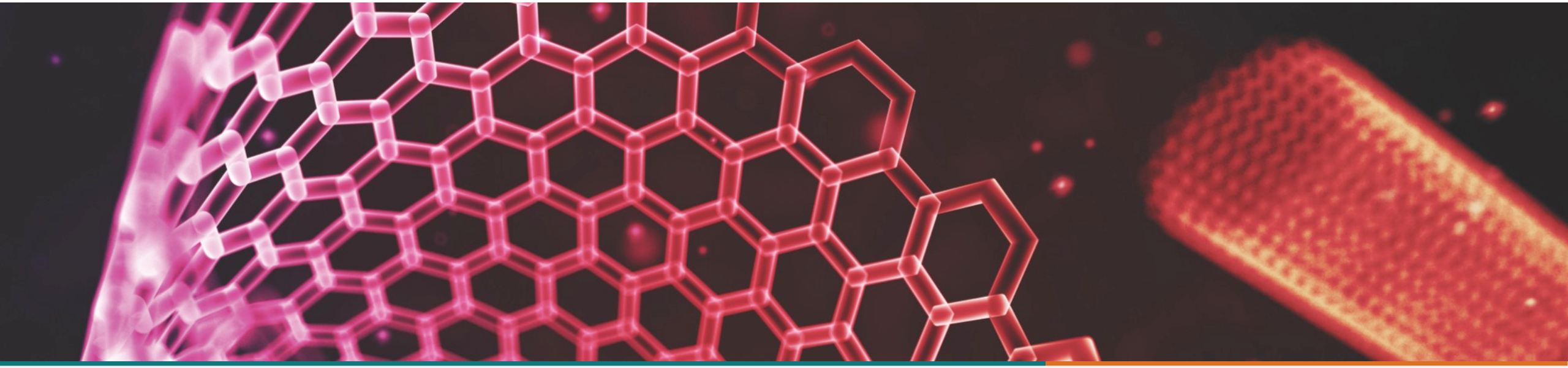
What is Babel?

Babel or **Babel.js** is a free and open-source JavaScript compiler and configurable transpiler used in web development. Babel allows software developers to write source code in a preferred programming language or markup language and have it translated by Babel into JavaScript, a language understood by modern web browsers.

Babel is popular tool for using the newest features of the JavaScript programming language. As a transpiler, or source-to-source compiler, developers can program using new ECMAScript 6 (ES6) language features by using Babel to convert their source code into versions of JavaScript that evolving browsers are able to process. The core version of Babel is downloaded 5 million times a month.

Babel plugins are available to provide specific conversions used in web development. For example, developers working with React.js, can use Babel to convert JSX (JavaScript XML) markup into JavaScript using the Babel preset "react".

Class Component Lifecycle Methods



Class Component Lifecycle Methods

React defines several component events in four categories. Each category can fire events various number of times:

- ***Mounting***—React invokes events only once.
- ***Updating***—React can invoke events many times.
- ***Unmounting***—React invokes events only once.
- ***Error Handling***— React invokes these when there is an error during rendering.

Class Component Lifecycle

"Render Phase"

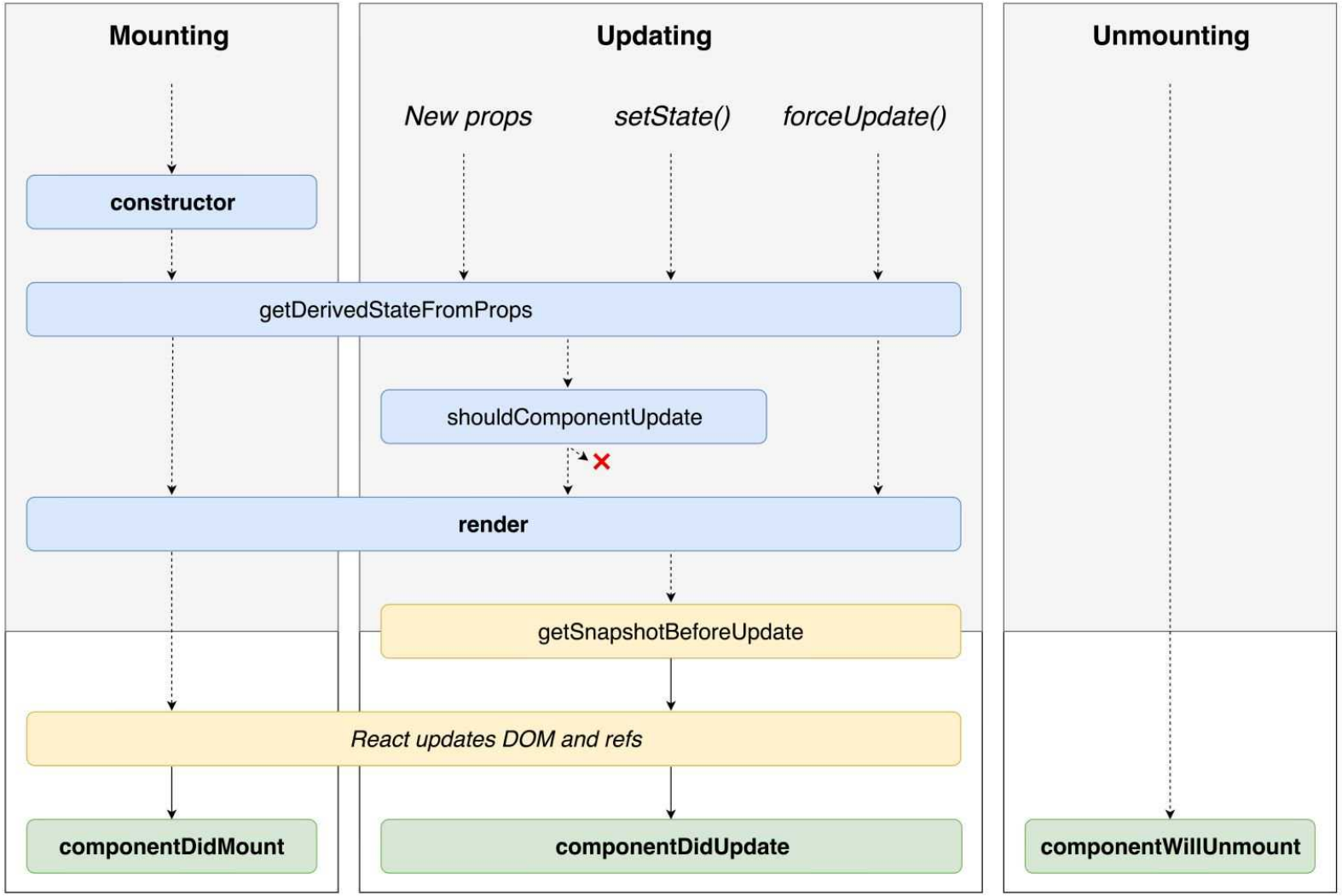
Pure and has no side effects. May be paused, aborted or restarted by React.

"Pre-Commit Phase"

Can read the DOM.

"Commit Phase"

Can work with DOM, run side effects, schedule updates.



Class Component Lifecycle - Mounting

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

- **constructor()**
- **static getDerivedStateFromProps()**
- **render()**
- **componentDidMount()**

Class Component Lifecycle - Updating

An update can be caused by changes to props or state. These methods are called in the following order when a component is being re-rendered:

- **static getDerivedStateFromProps()**
- **shouldComponentUpdate()**
- **render()**
- **getSnapshotBeforeUpdate()**
- **componentDidUpdate()**

Class Component Lifecycle - Unmounting

This method is called when a component is being removed from the DOM:

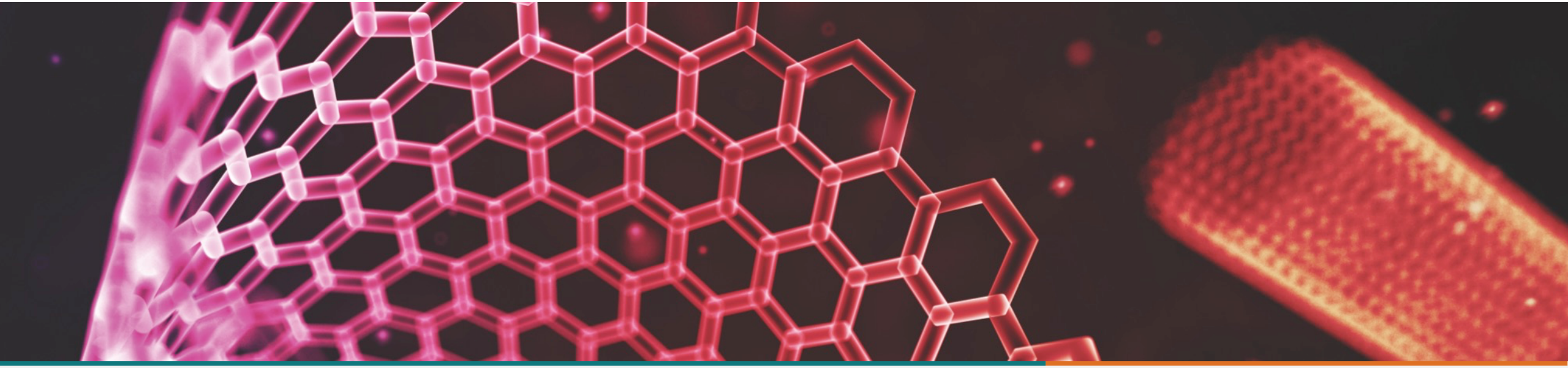
- **componentWillUnmount()**

Class Component Lifecycle – Error Handling

These methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

- **static getDerivedStateFromError()**
- **componentDidCatch()**

React Router



What is React Router?

Many modern websites are actually made up of a single page, they just look like multiple pages because they contain components which render like separate pages.

These are usually referred to as SPAs - **s**ingle-**p**age **a**pplications. At its core, what React Router does is conditionally render certain components to display depending on the *route* being used in the URL (/ for the home page, /about for the about page, etc.).

For example, we can use React Router to connect ***localhost:3000/*** to ***localhost:3000/about*** or ***localhost:3000/shop***

React Router is a npm module that you install in your application:

```
npm install react-router-dom
```

Using React Router

React router gives us some components which help us to implement the routing.

- ***<BrowserRouter>*** **-** The router that keeps the UI in sync with the URL.
- ***<Link>*** **-** Renders a navigation link.
- ***<NavLink>*** **-** A special version of the ***<Link>*** that will add styling attributes to the rendered element when it matches the current URL.
- ***<Redirect>*** **-** Rendering a <Redirect> will navigate to a new location. The new location will override the current location in the history stack, like server-side redirects (HTTP 3xx) do.
- ***<Route>*** **-** Renders a UI component depending on the URL
- ***<Switch>*** **-** Renders the first child <Route> or <Redirect> that matches the location.

Using React Router - Example

So the first thing we need to do is import the router:

```
import { BrowserRouter as Router, Route } from 'react-router-dom';
```

Then we need to wrap our code in app.js with a <Router> component

```
<Router>
   <div className='App'>
      <header className='App-header'>
         <img src={logo} className='App-logo' alt='logo' />
         <h1 className='App-title'>Welcome to React</h1>
      </header>
      <Navigation />
   </div>
</Router>
```

Using React Router - Example

The next thing we need to do is define some routes in app.js

```
<Router>
   <div className='App’>
      <header className='App-header’>
         <img src={logo} className='App-logo' alt='logo' />
         <h1 className='App-title'>Welcome to React</h1>
      </header>
      <Navigation />
      <Route exact path='/' component={Landing} />
      <Route path='/home' component={Home} />
      <Route path='/signin' component={SignIn} />
      <Route path='/account' component={Account} />
      <Route path='/signup' component={SignUpPage} />
      <Route path='/forgotpassword' component={ForgotPassword} />
   </div>
</Router>
```

Using React Router – Example

The next thing we need to do is define some routes in app.js

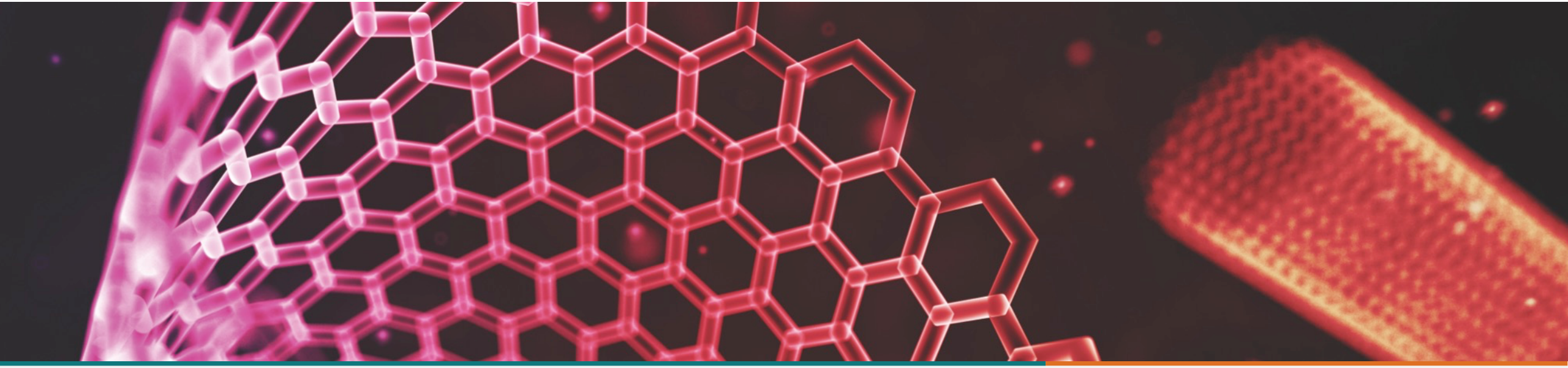
```
<Router>
   <div className='App’>
      <header className='App-header’>
         <img src={logo} className='App-logo' alt='logo' />
         <h1 className='App-title'>Welcome to React</h1>
      </header>
      <Navigation />
      <Route exact path='/' component={Landing} />
      <Route path='/home' component={Home} />
      <Route path='/signin' component={SignIn} />
      <Route path='/account' component={Account} />
      <Route path='/signup' component={SignUpPage} />
      <Route path='/forgotpassword' component={ForgotPassword} />
   </div>
</Router>
```

Using React Router – Our Navigation Component

```
import React from "react";
import { NavLink } from "react-router-dom";
const Navigation = () => (
<nav className="navigation">
   <ul>
      <li>
         <NavLink exact to="/" activeClassName="active">Landing</NavLink>
      </li>
      <li>
         <NavLink to="/home" activeClassName="active">Home</NavLink>
      </li>
      <li>
         <NavLink to="/signin" activeClassName="active">Sign-in</NavLink>
      </li>
      <li>
         <NavLink exact to="/account" activeClassName="active">Account</NavLink>
      </li>
   </ul>
</nav>
);

export default Navigation;
```

Our Next React Lecture



What's Next?

In our next lecture, we will go over:

- Getting data from an API for our component
- React Hooks
- Handling State in Function Components using React Hooks
- Form processing and event handling
- Building out our Firebase Authentication App more
- And more!

Questions?