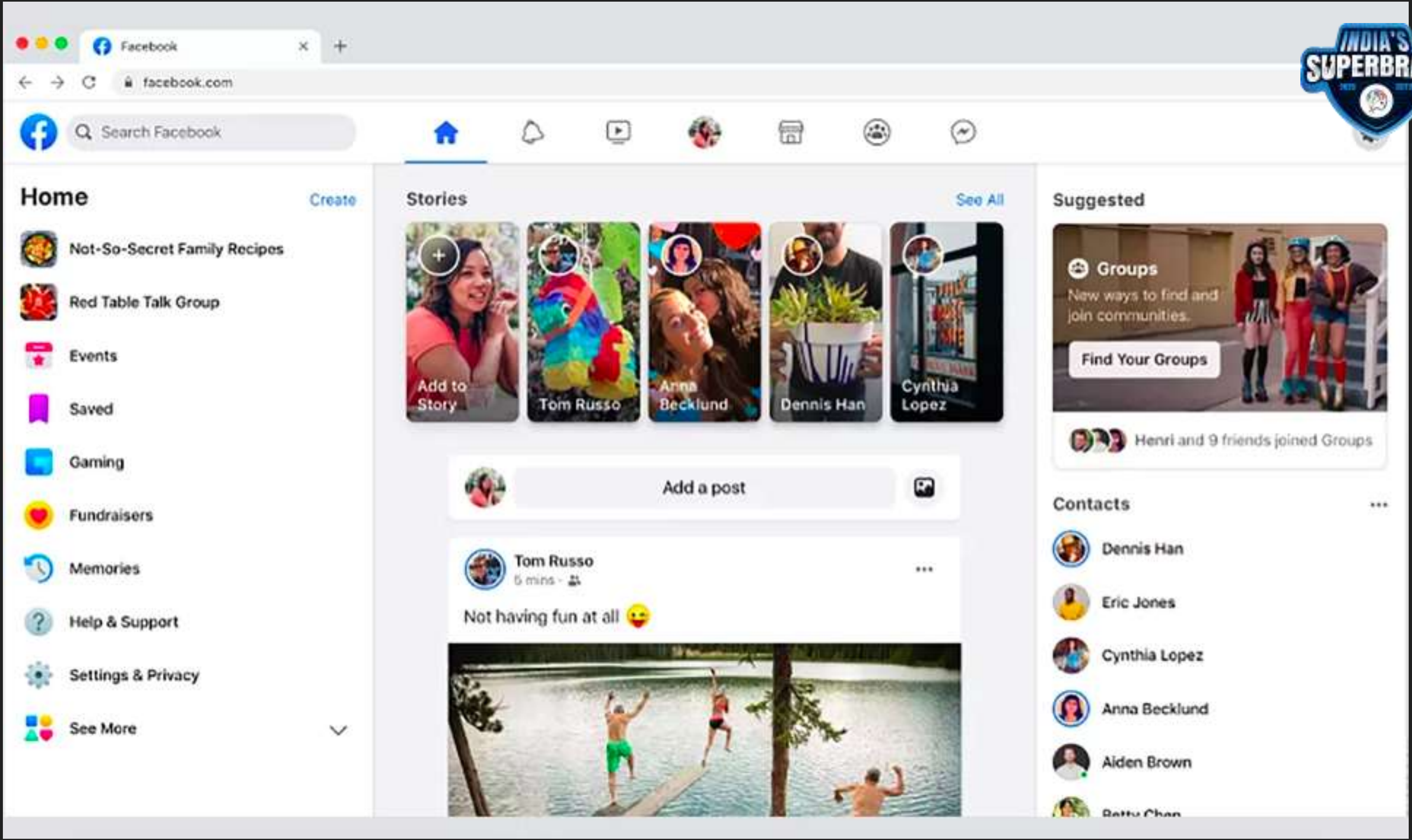
React

A JavaScript library for building user interfaces

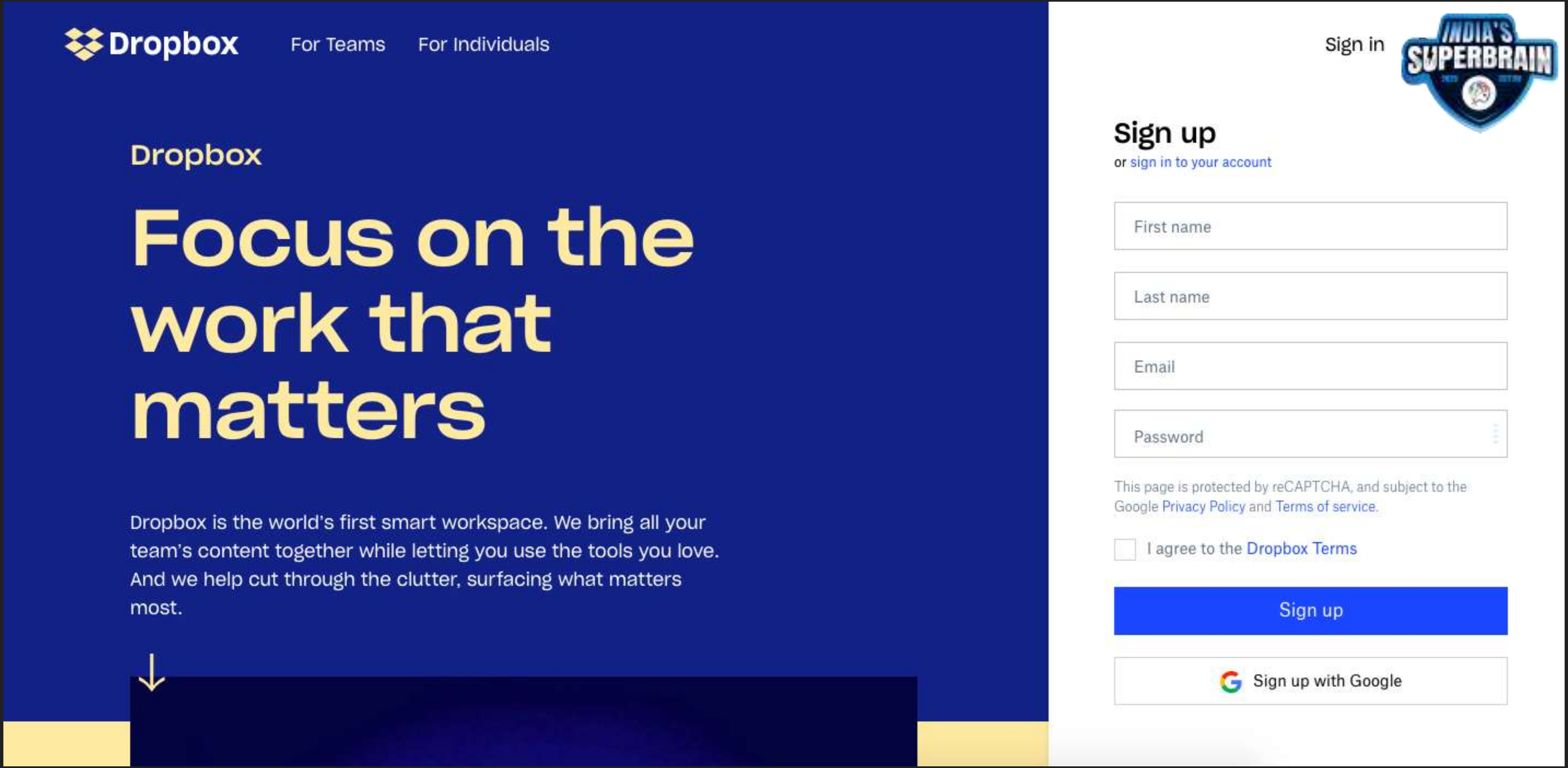
React

BUILD YOUR OWN UNIVERSE

Who uses React ?



Facebook
facebook.com
Search Facebook
Home
Create
Not-So-Secret Family Recipes
Red Table Talk Group
Events
Saved
Gaming
Fundraisers
Memories
Help & Support
Settings & Privacy
See More
Stories
See All
Add to Story
Tom Russo
Anna Becklund
Dennis Han
Cynthia Lopez
Add a post
Tom Russo
5 mins ·
Not having fun at all
Suggested
Groups
New ways to find and join communities.
Find Your Groups
Henri and 9 friends joined Groups
Contacts
Dennis Han
Eric Jones
Cynthia Lopez
Anna Becklund
Aiden Brown
Betty Chen



Dropbox For Teams For Individuals

Sign in

Dropbox

Focus on the work that matters

Dropbox is the world's first smart workspace. We bring all your team's content together while letting you use the tools you love. And we help cut through the clutter, surfacing what matters most.

Sign up

or sign in to your account

First name

Last name

Email

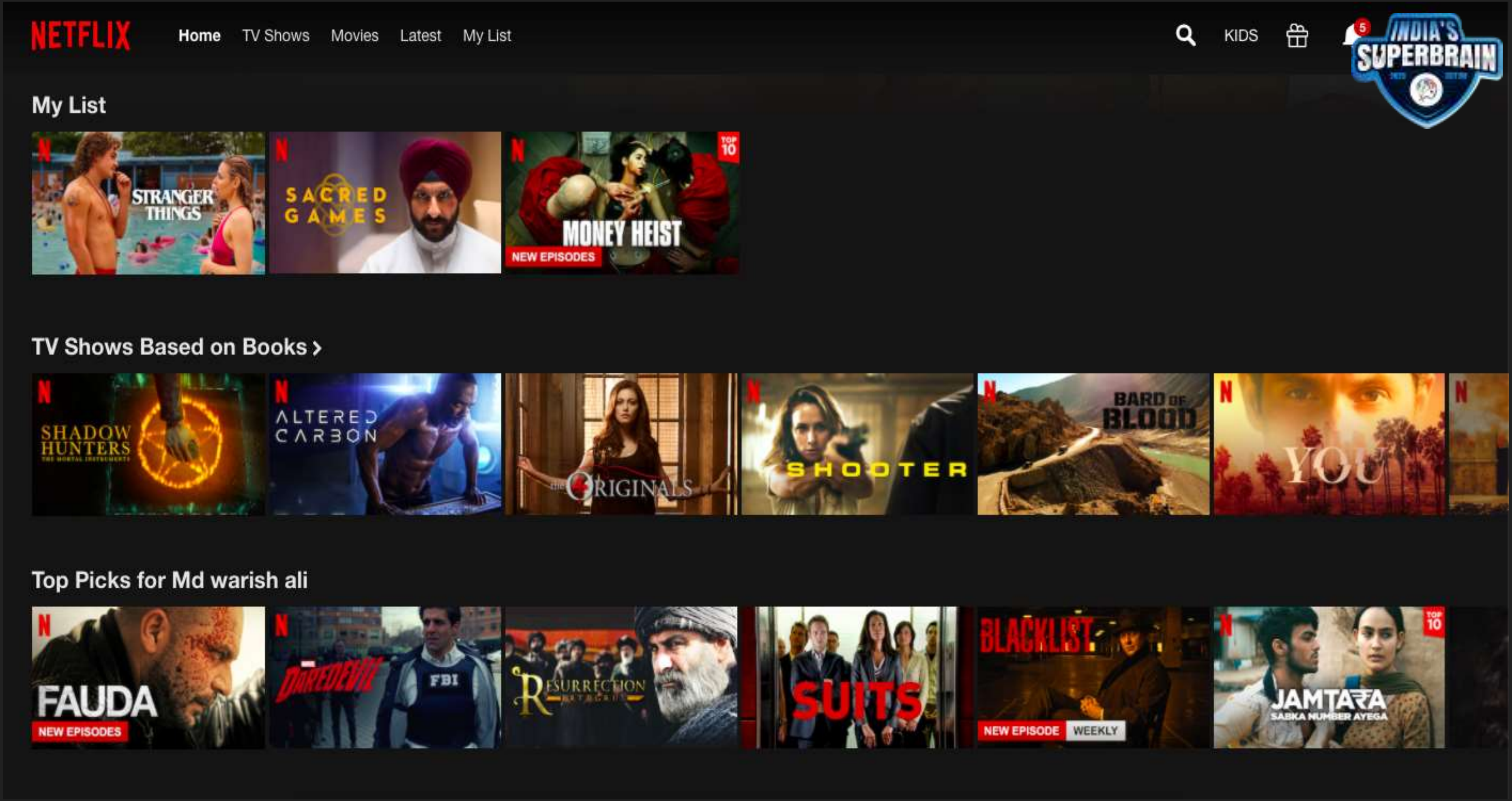
Password

This page is protected by reCAPTCHA, and subject to the Google Privacy Policy and Terms of service.

I agree to the Dropbox Terms

Sign up

Sign up with Google

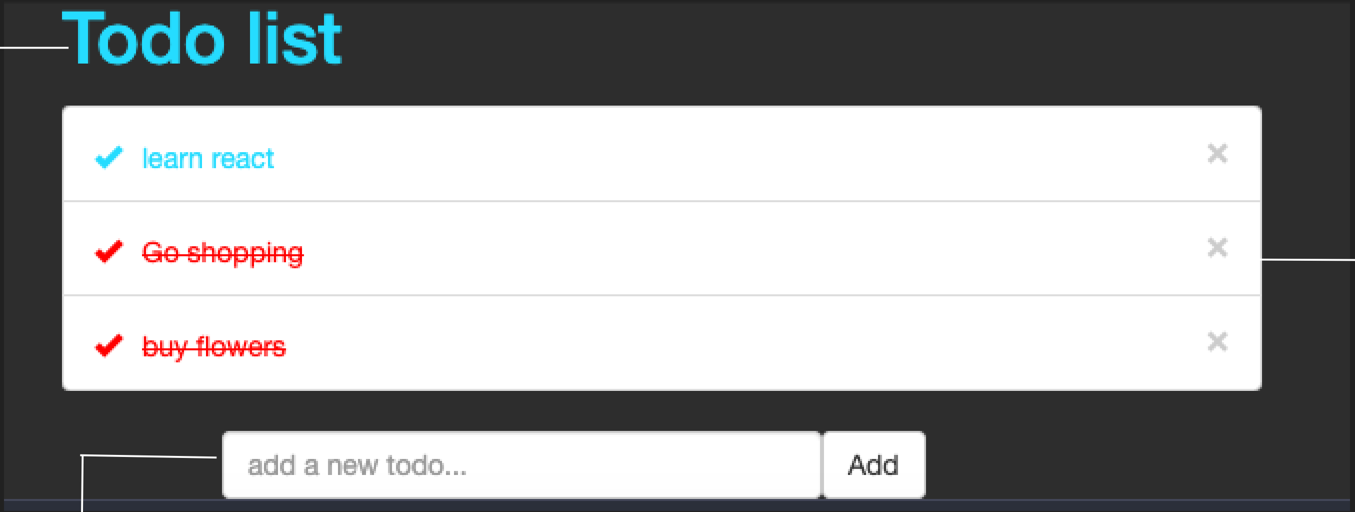

NETFLIX
Home
TV Shows
Movies
Latest
My List
KIDS
5
INDIA'S SUPERBRAIN
My List
STRANGER THINGS
SACRED GAMES
MONEY HEIST
NEW EPISODES
TOP 10
TV Shows Based on Books
SHADOW HUNTERS
ALTERED CARBON
THE ORIGINALS
SHOOTER
BARD OF BLOOD
YOU
Top Picks for Md warish ali
FAUDA
NEW EPISODES
DAREDEVIL
FBI
RESURRECTION ERTUGRUL
SUITS
BLACKLIST
NEW EPISODE
WEEKLY
JAMTARA
SABKA NUMBER AYEGA
TOP 10

Why should we use React ?

Declarative

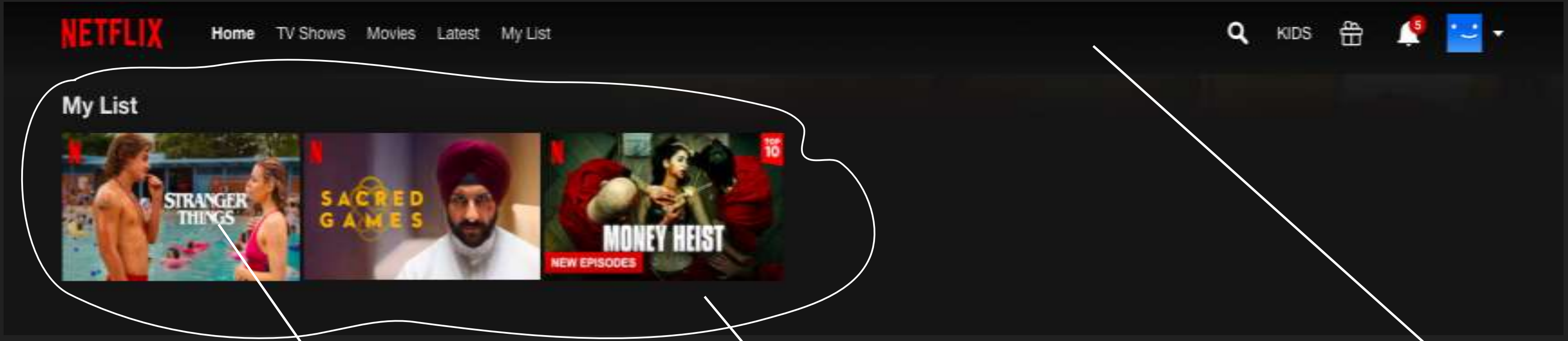
React makes it painless to create interactive UIs.

Because at the end what you write is what you get !!


Todo list
✓ learn react ×
✓ ~~Go shopping~~ ×
✓ ~~buy flowers~~ ×
add a new todo...
Add

```
<div id="main">
  <TodoHeader />
  <TodoList items={this.props.initItems} removeItem={this.removeItem} markTodoDone={this.markTodoDone}/>
  <TodoForm addItem={this.addItem} />
</div>
```

Component Based



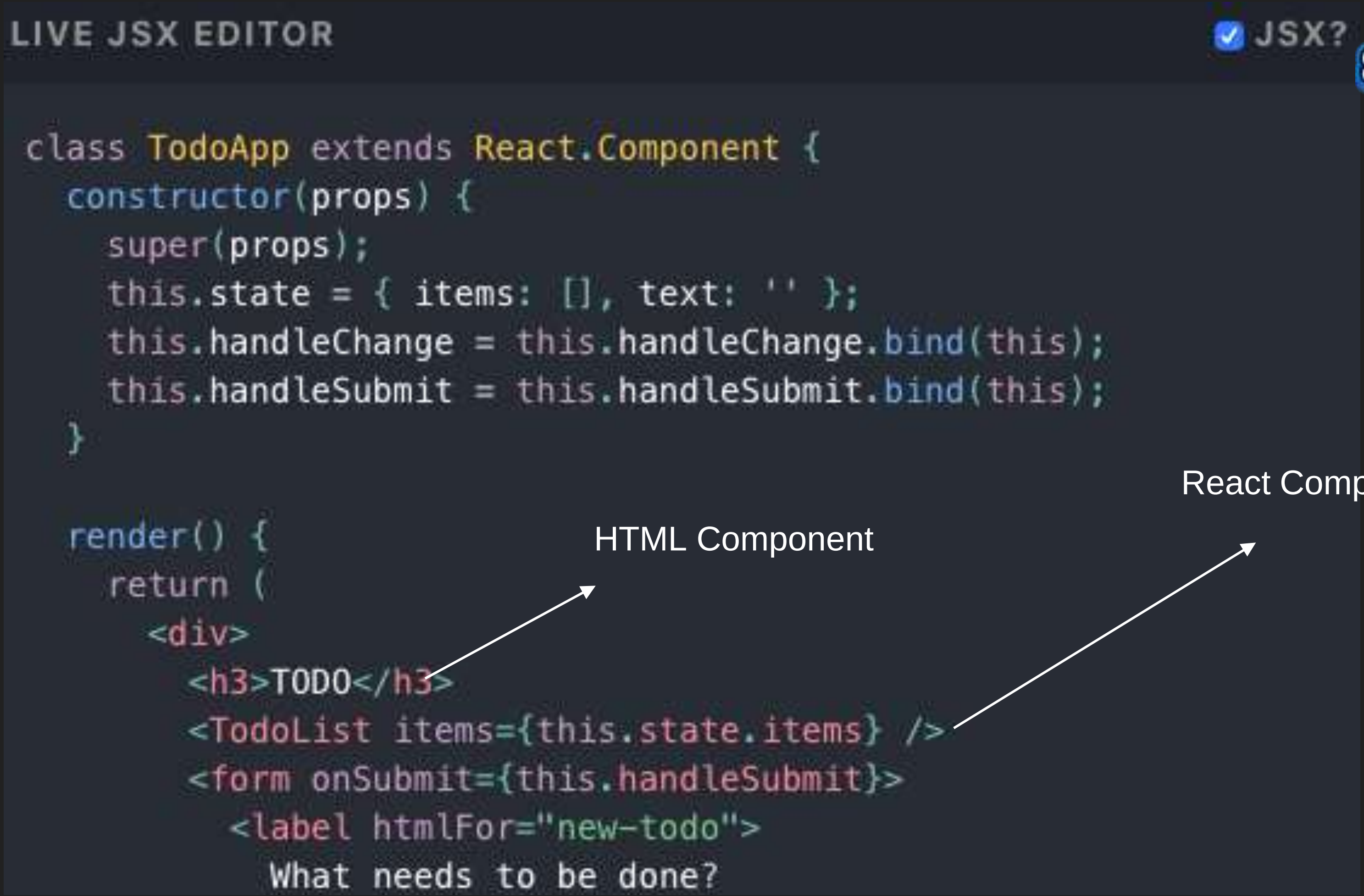
Navbar Component

MovieCard Component

Movie List Component

What is JSX ?

JSX is a syntax extension to JavaScript. It is similar to a template language, but it has full power of JavaScript.



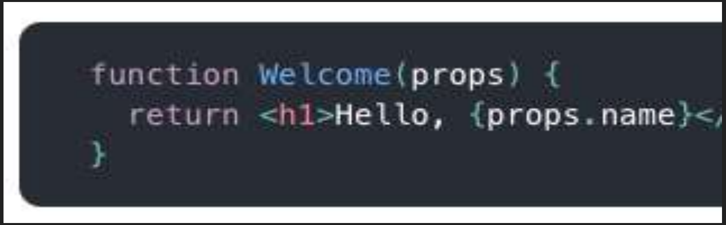
```
class TodoApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = { items: [], text: '' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={this.handleSubmit}>
          <label htmlFor="new-todo">
            What needs to be done?
```

React Component

HTML Component

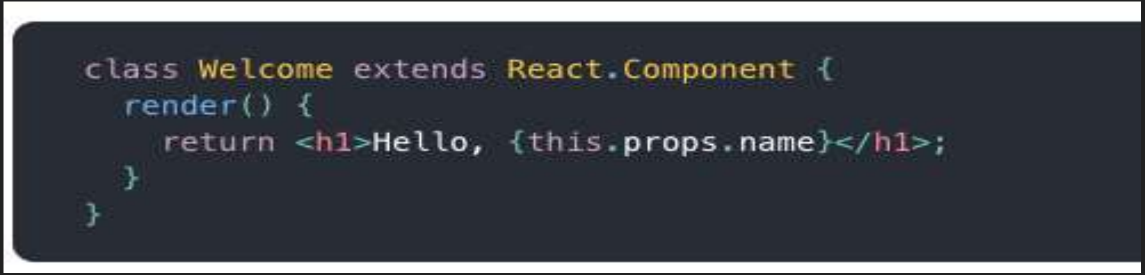
What are components?

React components are small, reusable pieces of code that return a React el
to be rendered to the page.



```
function Welcome(props) {
  return <h1>Hello, {props.name}</
}
```

Functional Components



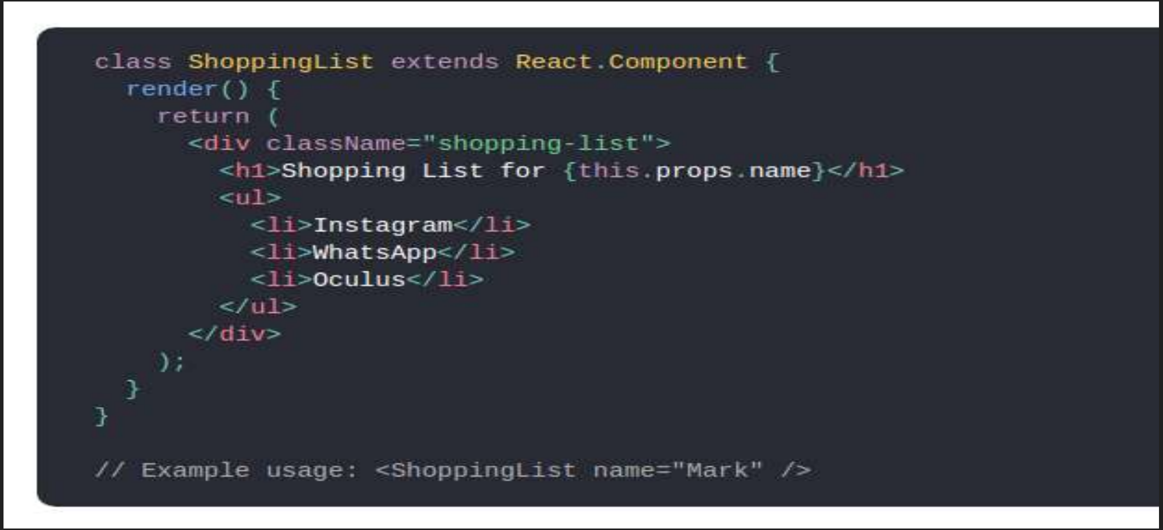
```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Class Based Components

<Welcome name=”Mark”/>

Class Based Components

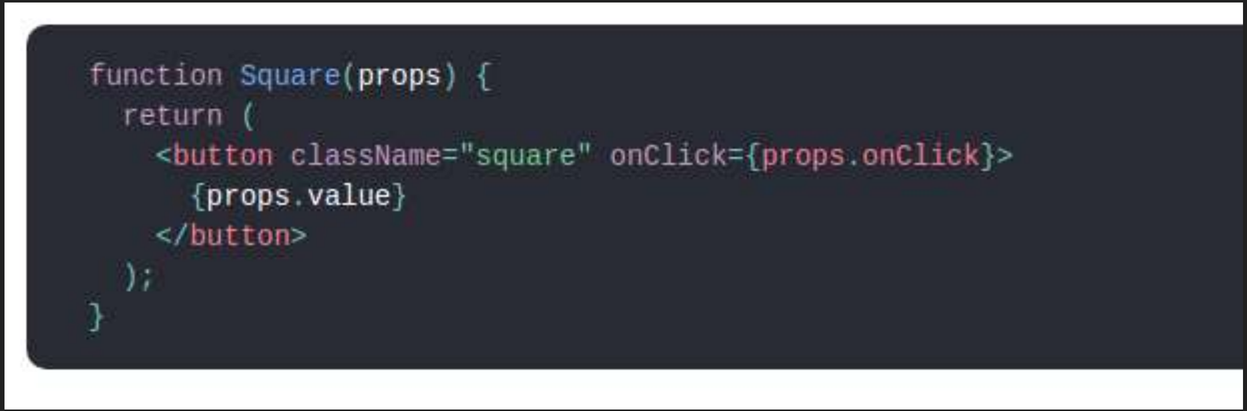
A React component class, or React component type. A component takes in parameters, called `props` (short for "properties"), and returns a hierarchy of views to display via the `render` method.



```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}

// Example usage: <ShoppingList name="Mark" />
```

Functional Components

In React, **Function Components** are a simpler way to write components that only contain a **`return`** statement and **don't have their own state**. We can write a function that takes **`props`** as input and returns what should be rendered. Function components are less tedious to write than classes, and many components can be expressed this way.



```
function Square(props) {
  return (
    <button className="square" onClick={props.onClick}>
      {props.value}
    </button>
  );
}
```

Note.

React Components name always starts with a Capital Letter.

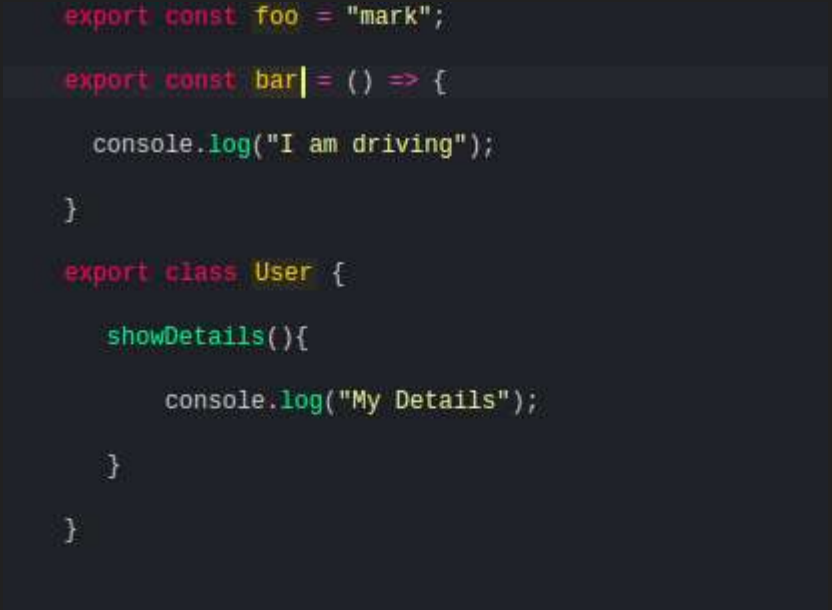
Because it helps the React compiler to differentiate between HTML components and React Components.

Import and Export

Export

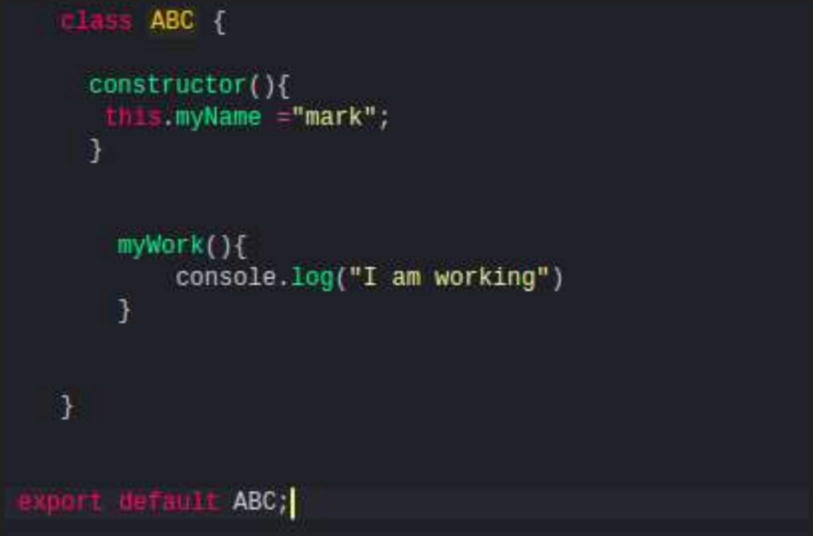
“**export**” keyword is used to export any type of values, data structures,classes, objects from one file to another.

1. Using named exports



```
export const foo = "mark";

export const bar = () => {

  console.log("I am driving");

}

export class User {

   showDetails(){

       console.log("My Details");

   }

}
```

2. Using **default** export



```
   class ABC {

     constructor(){
      this.myName ="mark";
     }


       myWork(){
           console.log("I am working")
       }


   }


export default ABC;
```

Import

"import" keyword is used to import any type of values, data structures,classes, objects from one file to another.

1. Import a single export from a module

```
import {myExport} from '/modules/my-module.js';
```

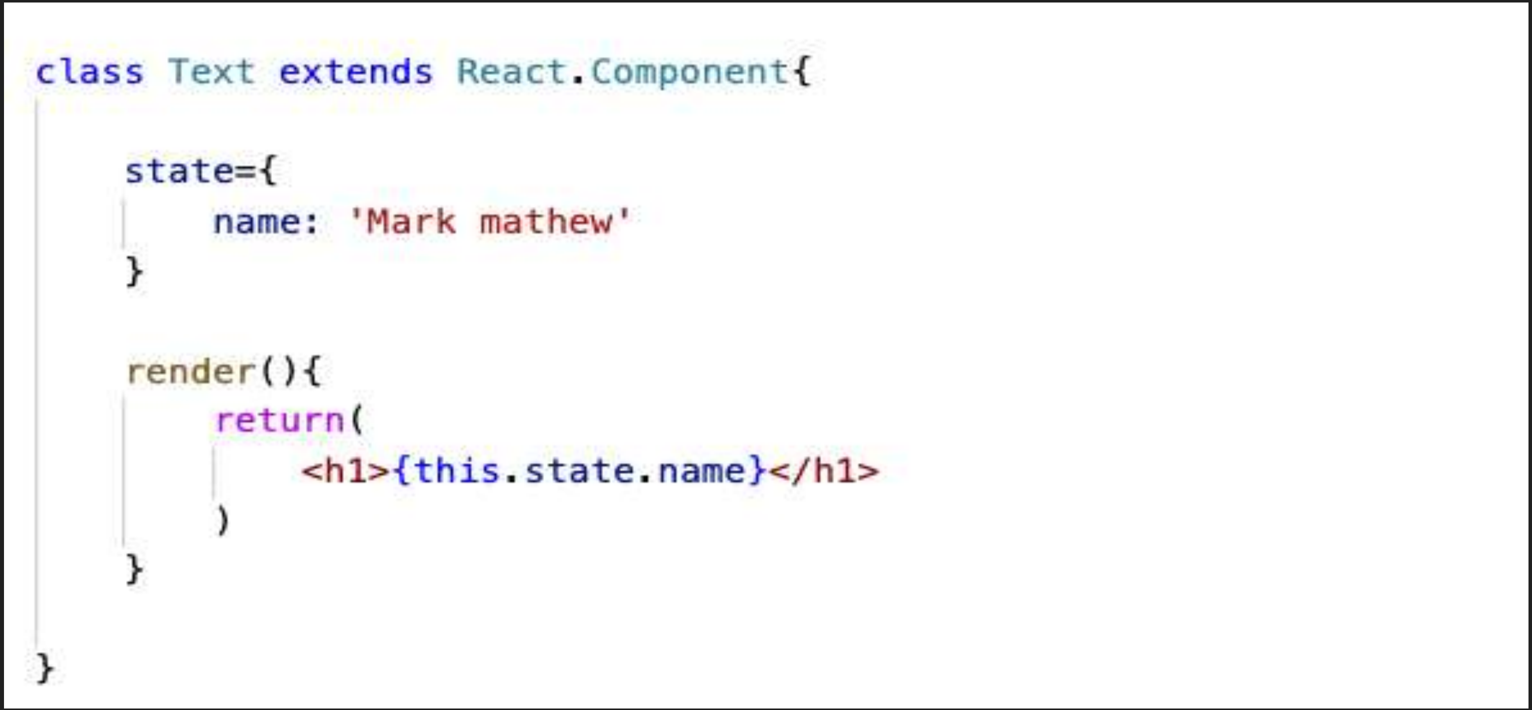
2. Import multiple exports from module

```
import {foo, bar} from '/modules/my-module.js'
```

3. Importing defaults

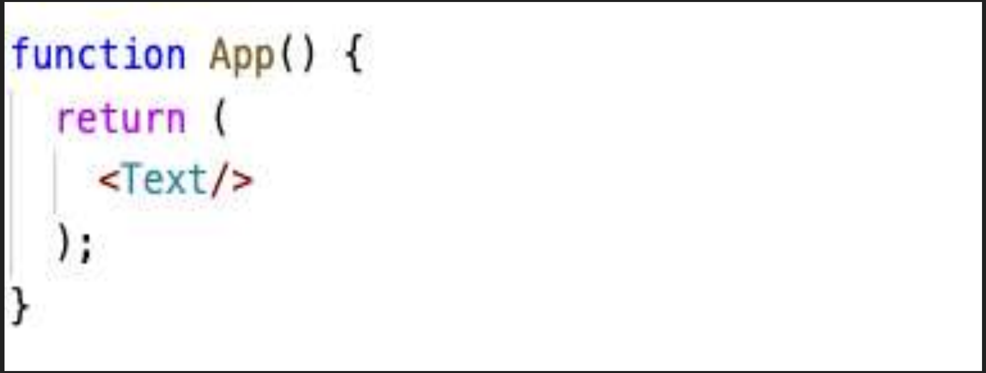
```
import myDefault from '/modules/my-module.js';
```

State

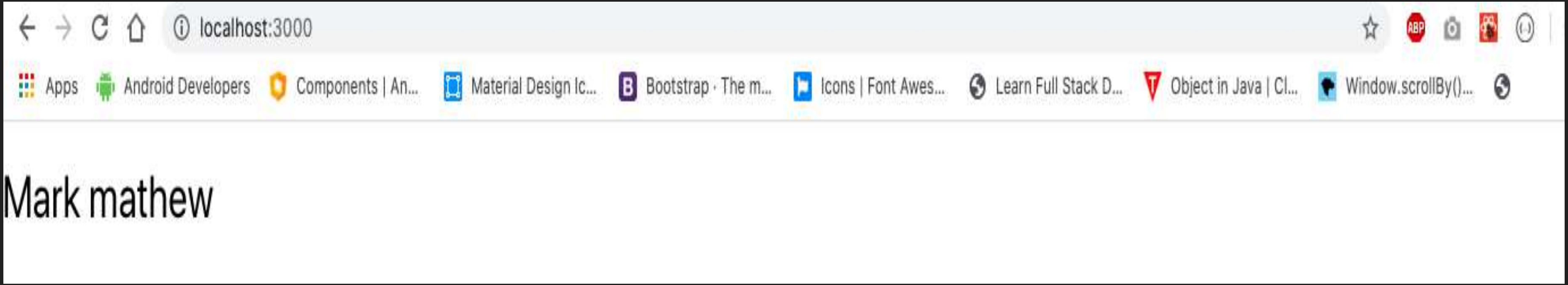


```
class Text extends React.Component{

    state={
        name: 'Mark mathew'
    }

    render(){
        return(
            <h1>{this.state.name}</h1>
        )
    }

}
```

<Text />

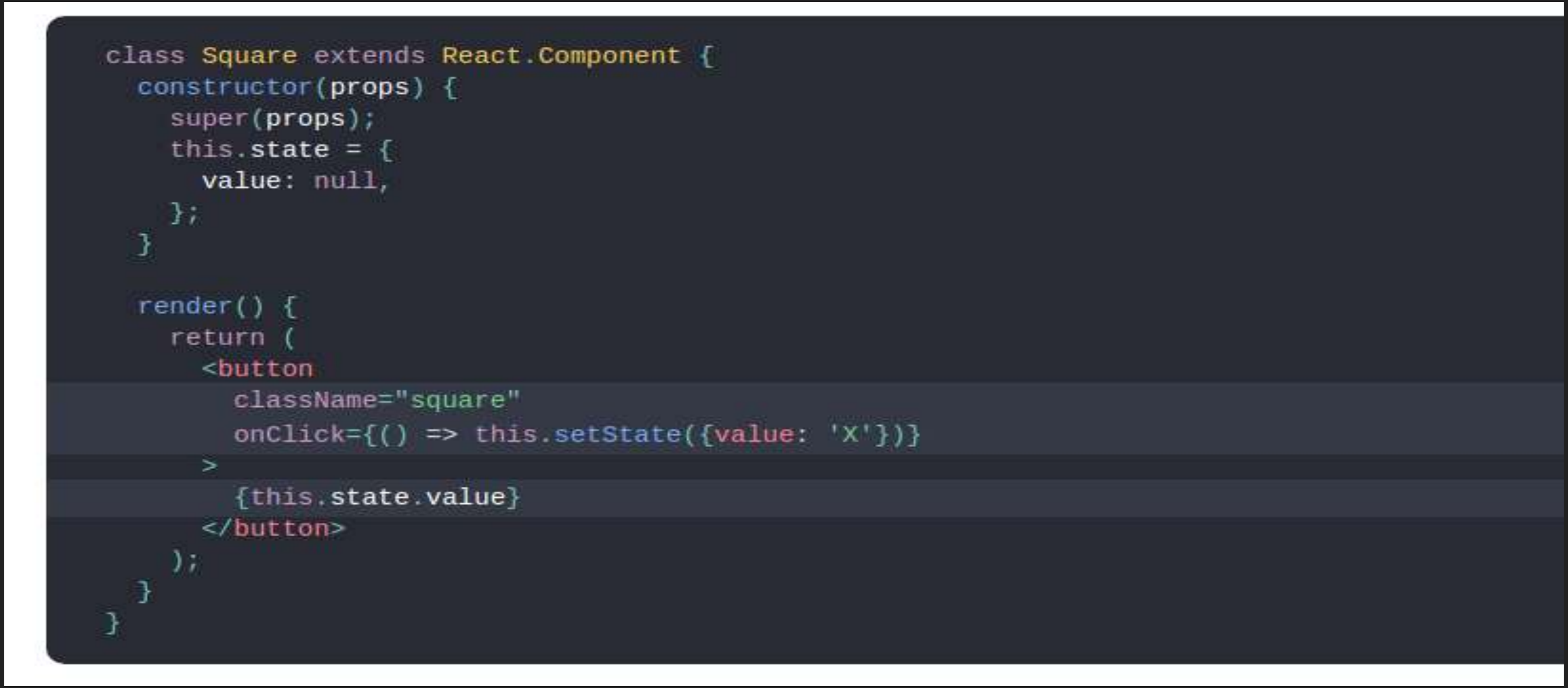


```
function App() {
  return (
    <Text/>
  );
}
```



```
Mark mathew
```

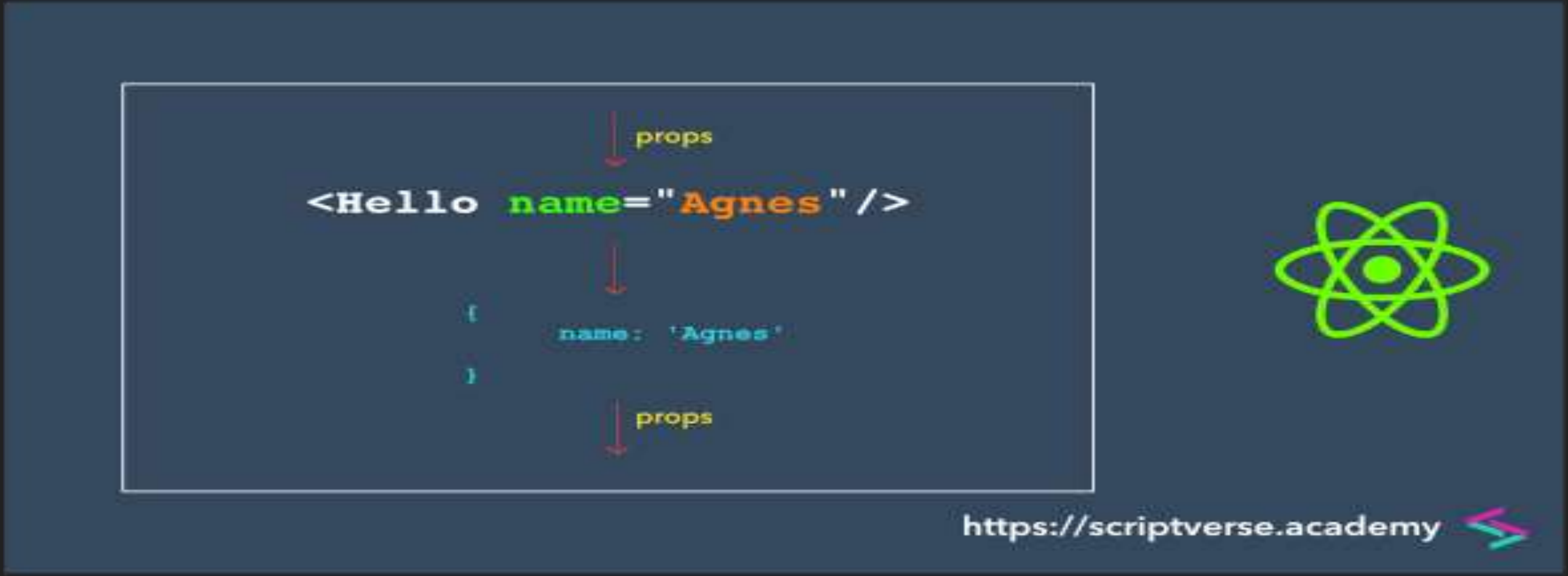
Defining state inside Constructor

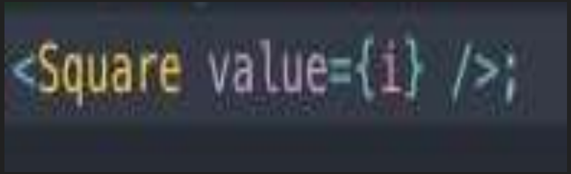


```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button
        className="square"
        onClick={() => this.setState({value: 'X'})}
      >
        {this.state.value}
      </button>
    );
  }
}
```

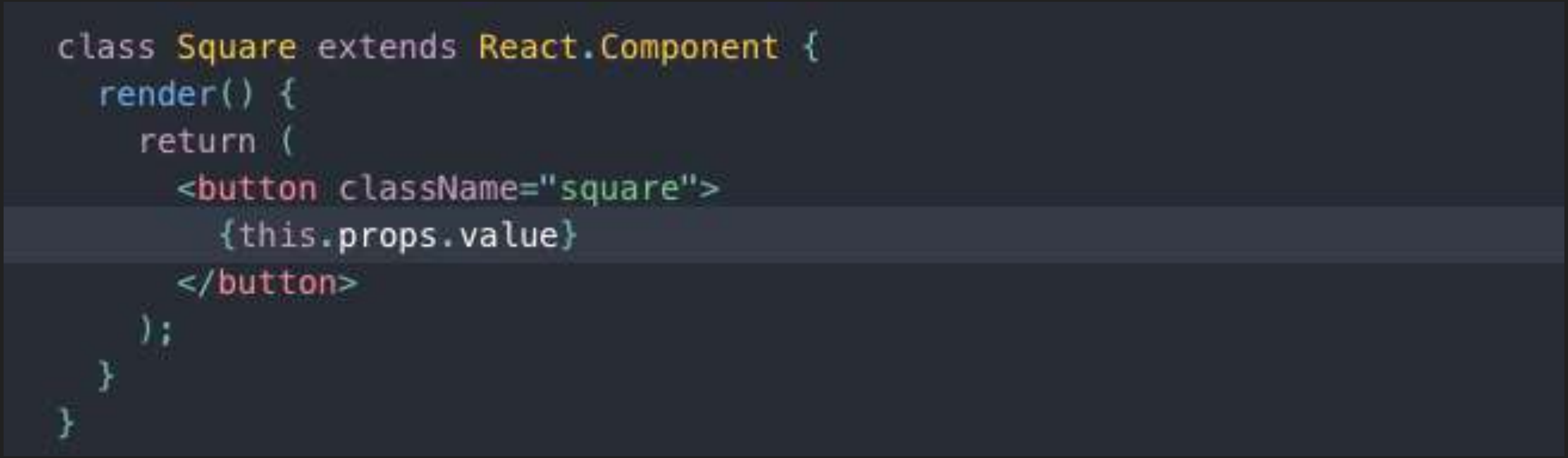
React components can have state by setting `this.state` in their constructors.

Props



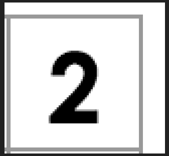


```
<Square value={i} />;
```



```
class Square extends React.Component {
  render() {
    return (
      <button className="square">
        {this.props.value}
      </button>
    );
  }
}
```

If i passed is 2



State Vs Props

Props vs State

- ✓ props are read-only
- ✓ props can not be modified

- ✓ state changes can be asynchronous
- ✓ state can be modified using this.setState

The Spread Operator “...”

1. **Spread operator is used to copy elements of one item into another**

```
const array1 = [ 1,2,3,4,5]

const array2 = [...array1, 6,7,8,9]

console.log(array2) ---> [1,2,3,4,5,6,7,8,9]      //        output

const array4 = [10,11,12, ...array1]

console.log(array4) ---> [10,11,12,1,2,3,4,5]      //      output
```

Using Spread operator on an object

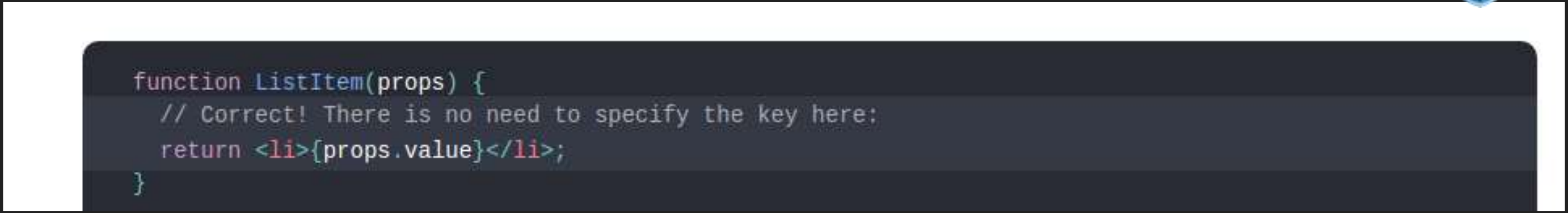
```
const user1 = {
name: “Mark”,
age: 15
}
const user2 = {
...user1,
phone: 123456
}
```

```
console.log(user2)--->
{
name: “Mark”,
age: 15,
phone: 123456,
}
```

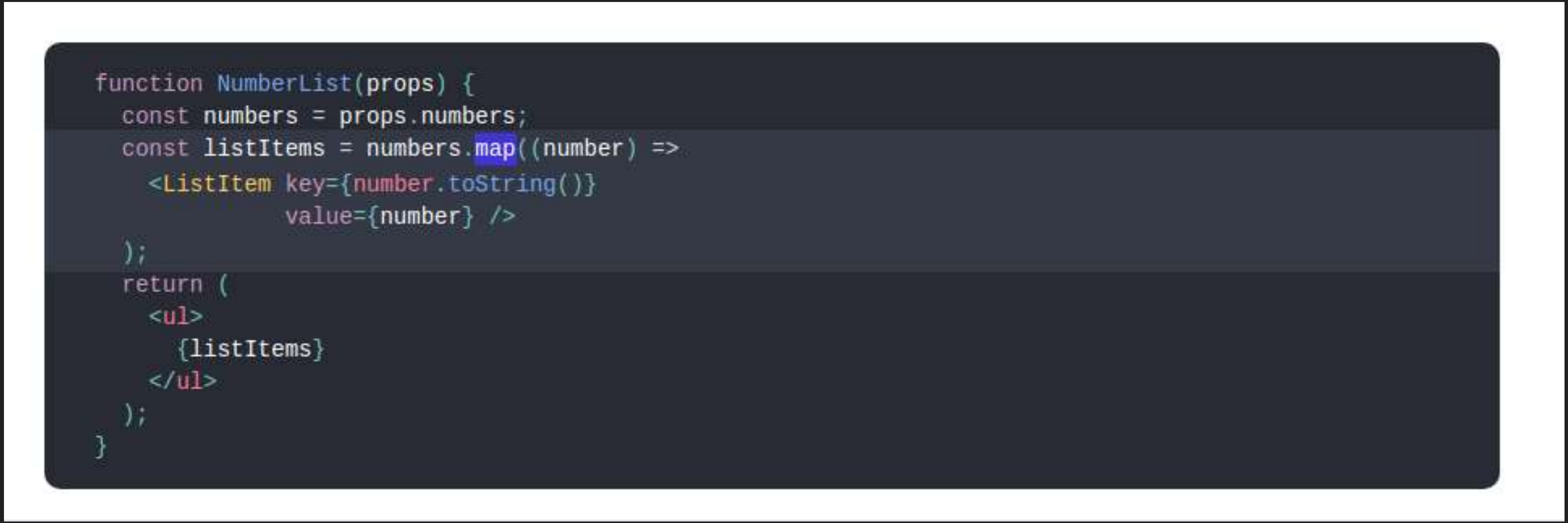
Lifting State Up

To collect data from multiple children, or to have two child components communicate with each other, you need to declare the **shared state in their parent component** instead. The parent component can **pass the state back down to the children by using props**; this keeps the child components in sync with each other and with the parent component.

Displaying List Of Items



```
function ListItem(props) {
  // Correct! There is no need to specify the key here:
  return <li>{props.value}</li>;
}
```



```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()}
              value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

```
<NumberList numbers={ [1,2,3,4,5] } />
```

Note*

When displaying list of items the “Key” prop is very important to pass it to the react element. The key passed to the React element should be unique . This key helps to identify React which react element needs to be re-rendered or updated.

Input Events

```
function ActionLink() {

const handleClick = (e) => {
   console.log('The link was clicked.');
 }


  return (

      <a href="#" onClick={handleClick}>Click me</a>

  );

}
```

Here a reference of the function is passed to onClick event, so that whenever a click event occurs on this “a” link the passed function should be executed.

Passing arguments to event handlers

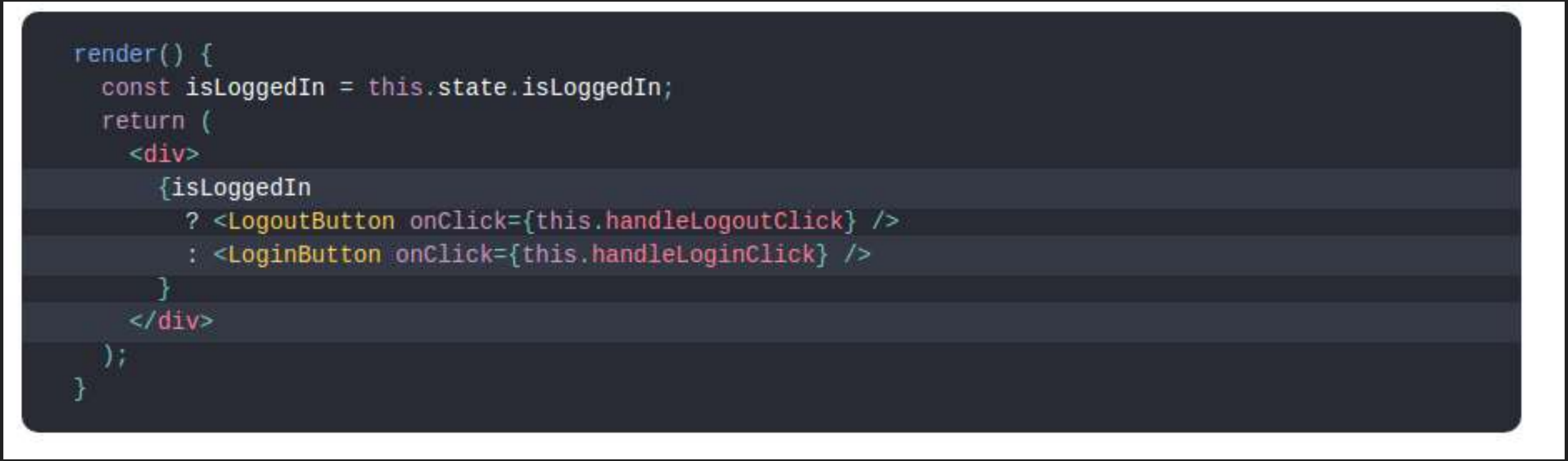
```
deleteRow = (index,e) => {

 const arr = [1,2,3,4];

//          deleting the element at the specified index

 arr.splice(index,1);

this.setState({ items: arr })

}

render() {

<button onClick={(e) => this.deleteRow(index, e)}>

      Delete Row

</button>

}
```

Form Management

2 way binding of input Fields

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);


   this.state = {value: ''};


}


 handleChange = (event) => {


   this.setState({value: event.target.value});


}



  render() {
    return (
        <label>
          Name:
               <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>

    );
  }


}
```

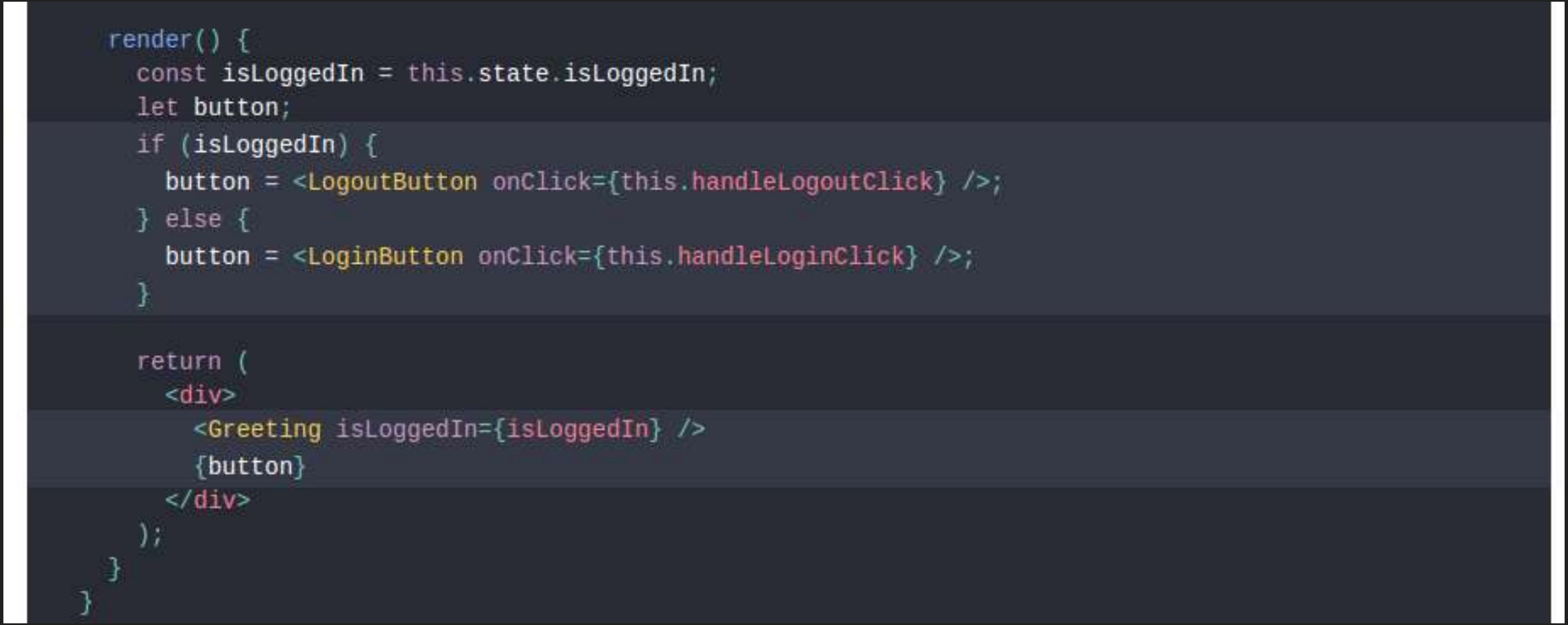
Conditional Rendering



```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}
```

Here Ternary “ ? ” operator has been used inside return method

Using If-else inside render method



```
    render() {
      const isLoggedIn = this.state.isLoggedIn;
      let button;
      if (isLoggedIn) {
        button = <LogoutButton onClick={this.handleLogoutClick} />;
      } else {
        button = <LoginButton onClick={this.handleLoginClick} />;
      }

      return (
        <div>
          <Greeting isLoggedIn={isLoggedIn} />
          {button}
        </div>
      );
    }
  }
```

Note*

1. We cannot use if else inside the return scope
2. Ternary operators can only be used inside the return scope

Life Cycle Methods

1. We can declare special methods on the component class to run some code when a component mounts and unmounts or before and after of rendering.
2. Only available for class based components , not for functional components

Order of Life Cycle methods

1. 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()`

2. 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()`

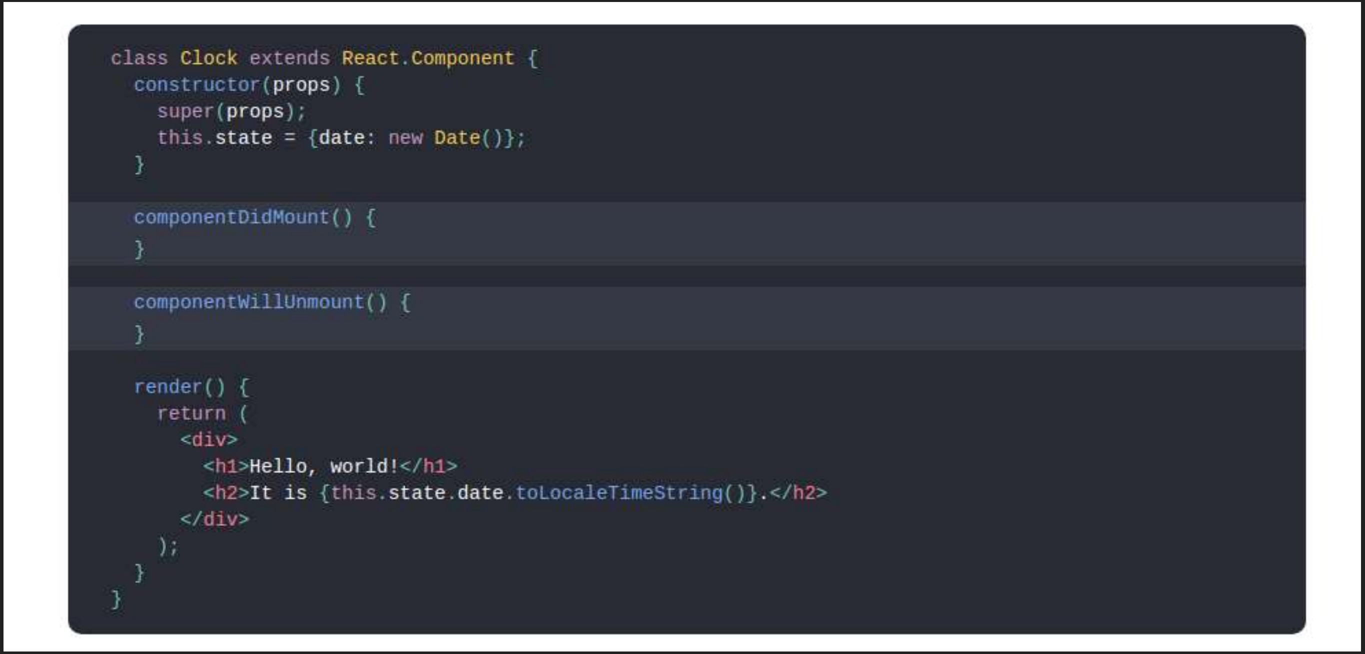
3. Unmounting

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

- `componentWillUnmount()`

componentDidMount(), componentWillUnMount()

1. `componentDidMount()` is invoked immediately after a component is mounted (inserted into the tree).
2. `componentWillUnMount()` is invoked immediately after a component is Un-mounted (removed from the tree).



```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
  }

  componentWillUnmount() {
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Promises in Javascript

A promise is basically a piece of asynchronous code which can either be fulfilled (completed) or rejected.

A `Promise` can have one of these states:

- *pending*: initial state, neither fulfilled nor rejected.
- *fulfilled*: meaning that the operation completed successfully.
- *rejected*: meaning that the operation failed.

```
let myFirstPromise = new Promise((resolve, reject) => {

// We call resolve(...) when what we were doing asynchronously was successful, and
reject(...) when it failed. // In this example, we use setTimeout(...) to simulate
async code.

setTimeout( function() { resolve("Success!") }, 250)

 })

 myFirstPromise.then((successMessage) => {

 console.log("Yay! " + successMessage)

}).catch((error) =>  {

 console.log("SOmething went wrong! " + error)

})
```

Fetching Data from server

Using the Fetch method provided in javascript

```
  fetch(“https://www.api.com”, { method: ‘GET’ } )
 .then((response) => {
 return response.json(); // converting raw data to JSON format
 })
 .then((result) => {
 console.log(“===server data===”,result);
 })
```