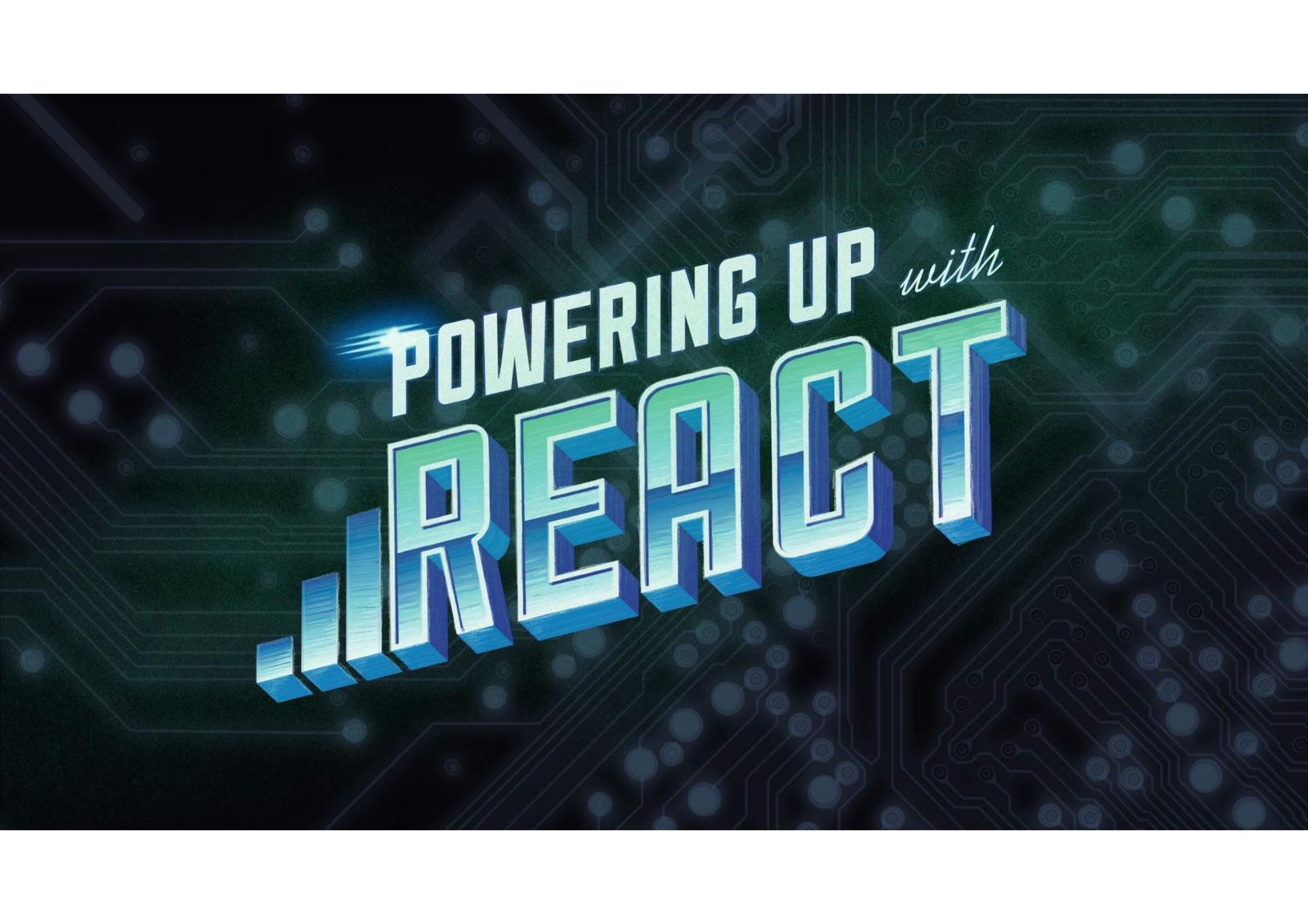


# POWERING UP with **.REACT**



Level 1

# First Component

POWERING UP <sup>with</sup>  
**REACT**

Level 1 - Section 1

# First Component

Writing Your First React Component

POWERING UP with  
**REACT**

# What Is React?

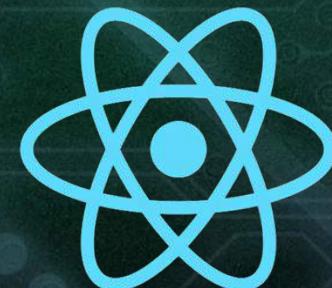
I heard it was good

(Model-View-Controller)

React is a JavaScript library for building user interfaces (UIs). Some people use it as the V in MVC.

## Why React?

React was built to solve one problem: building large applications with data that changes over time.



## Conceived at Facebook

Heavily used on products made by Facebook and Instagram.  
Built to simplify the process of building complex UIs.

facebook.

Dropbox

airbnb

Instagram

NETFLIX

PayPal

All these companies  
use React!

POWERING UP  
with  
**REACT**

# Prerequisites

## JavaScript Basics

- Declaring variables
- Creating and invoking functions

## ES2015

- Class syntax
- Arrow functions
- Spread operator

New to JavaScript? Go here first!

<http://javascript-roadtrip.codeschool.com>



Not familiar with ES2015? Go here!

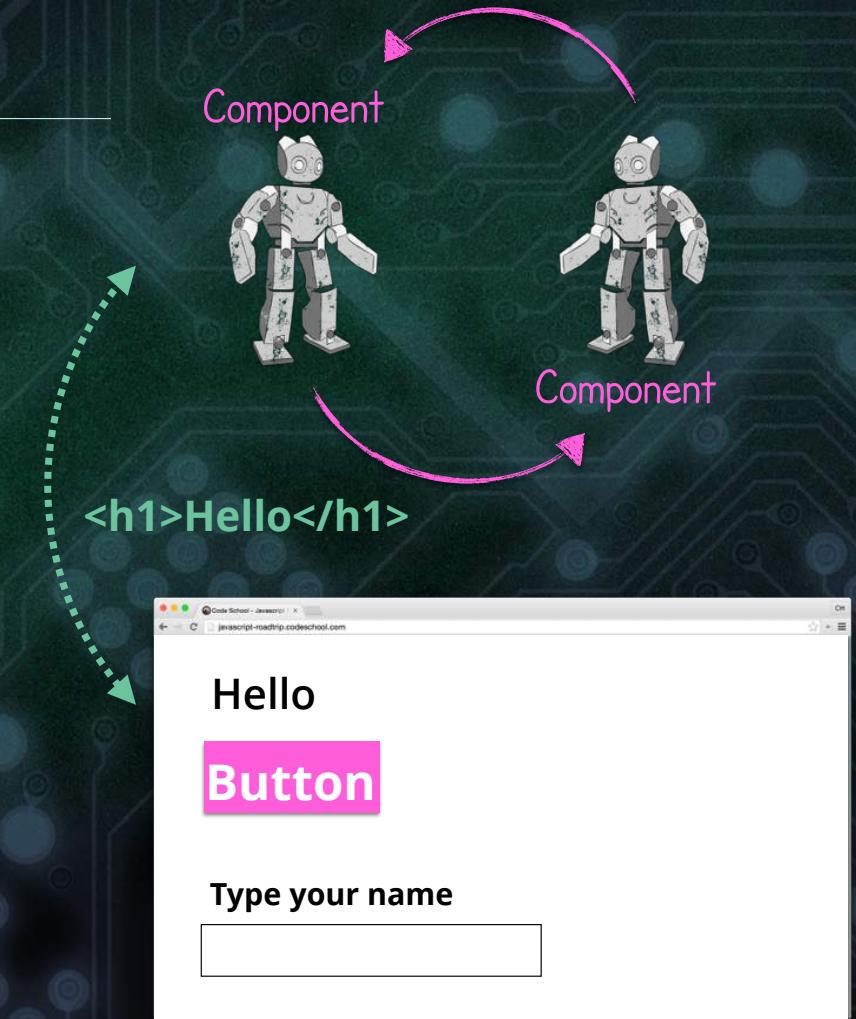
<http://es2015.codeschool.com>



# What We'll Learn

We'll cover some of the features React offers, including how to:

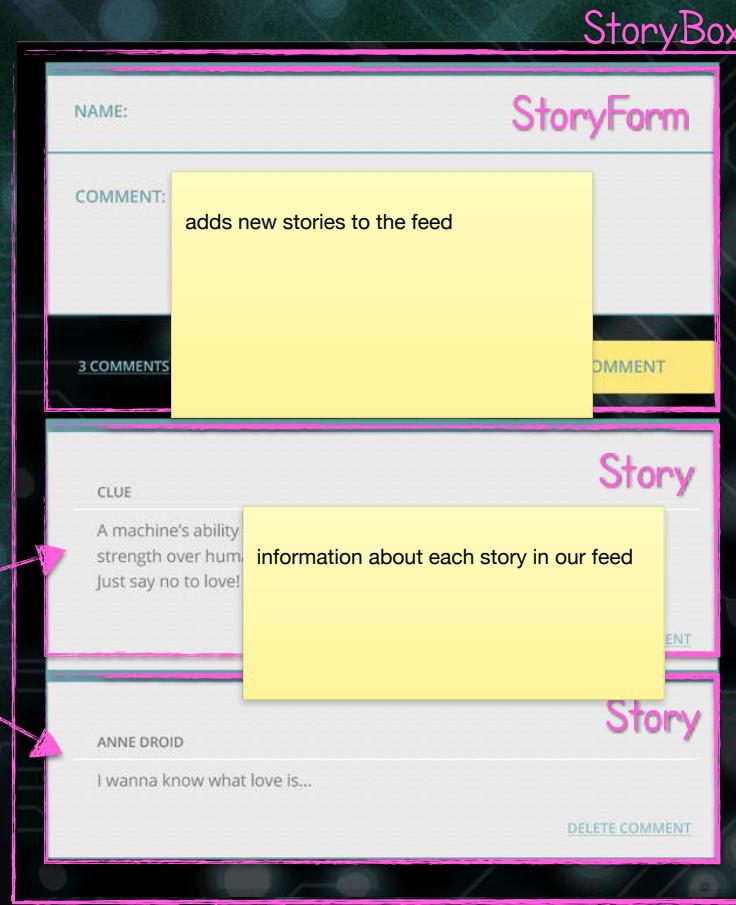
- Write React components
- Render data to the page
- Make components communicate
- Handle user events
- Capture user input
- Talk to remote servers



# Component-based Architecture

In React, we solve problems by creating components. If a component gets too complex, we break it into smaller, simpler components.

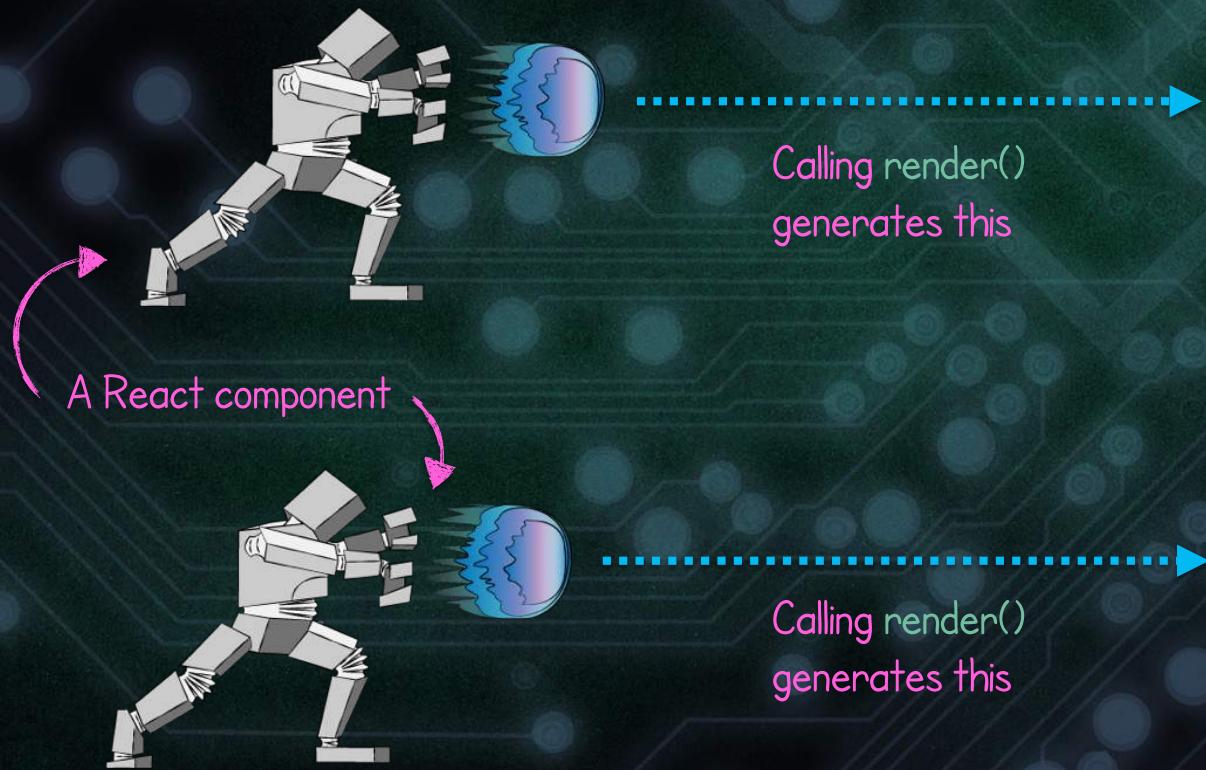
Story component  
is reused



POWERING UP with  
**REACT**

# What Is a React Component?

A component in React works similarly to JavaScript functions: It generates an output every time it is invoked.



## Output #1

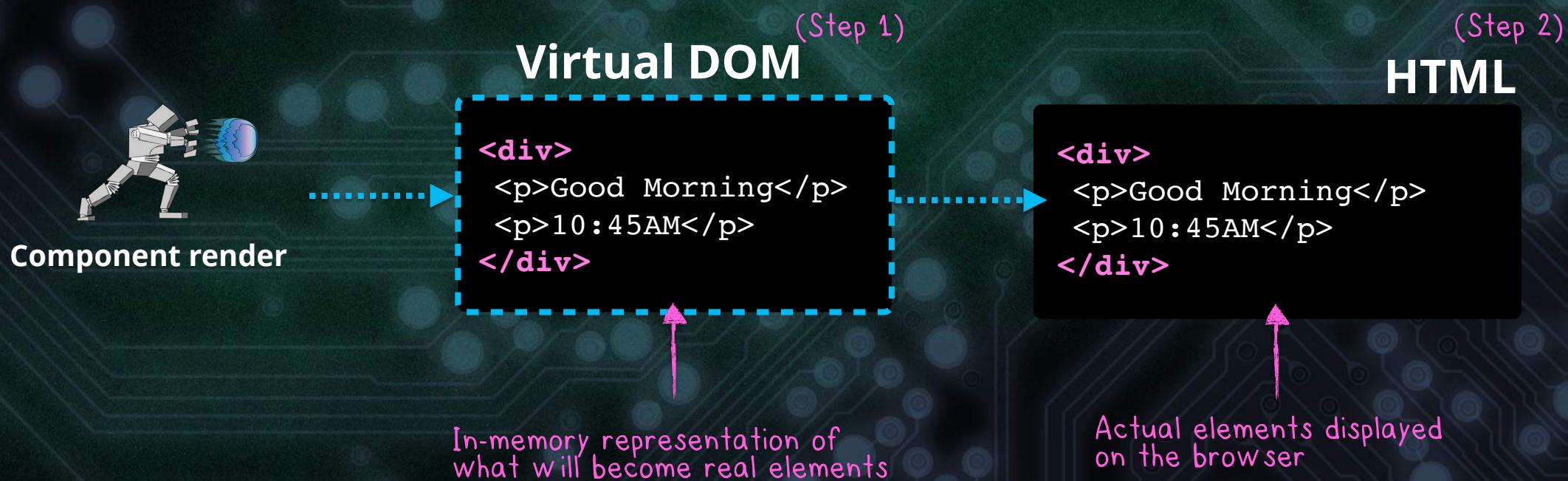
```
<div>
  <p>Good Morning</p>
  <p>10:45AM</p>
</div>
```

## Output #2

```
<div>
  <p>Good Morning</p>
  <p>10:55AM</p>
</div>
```

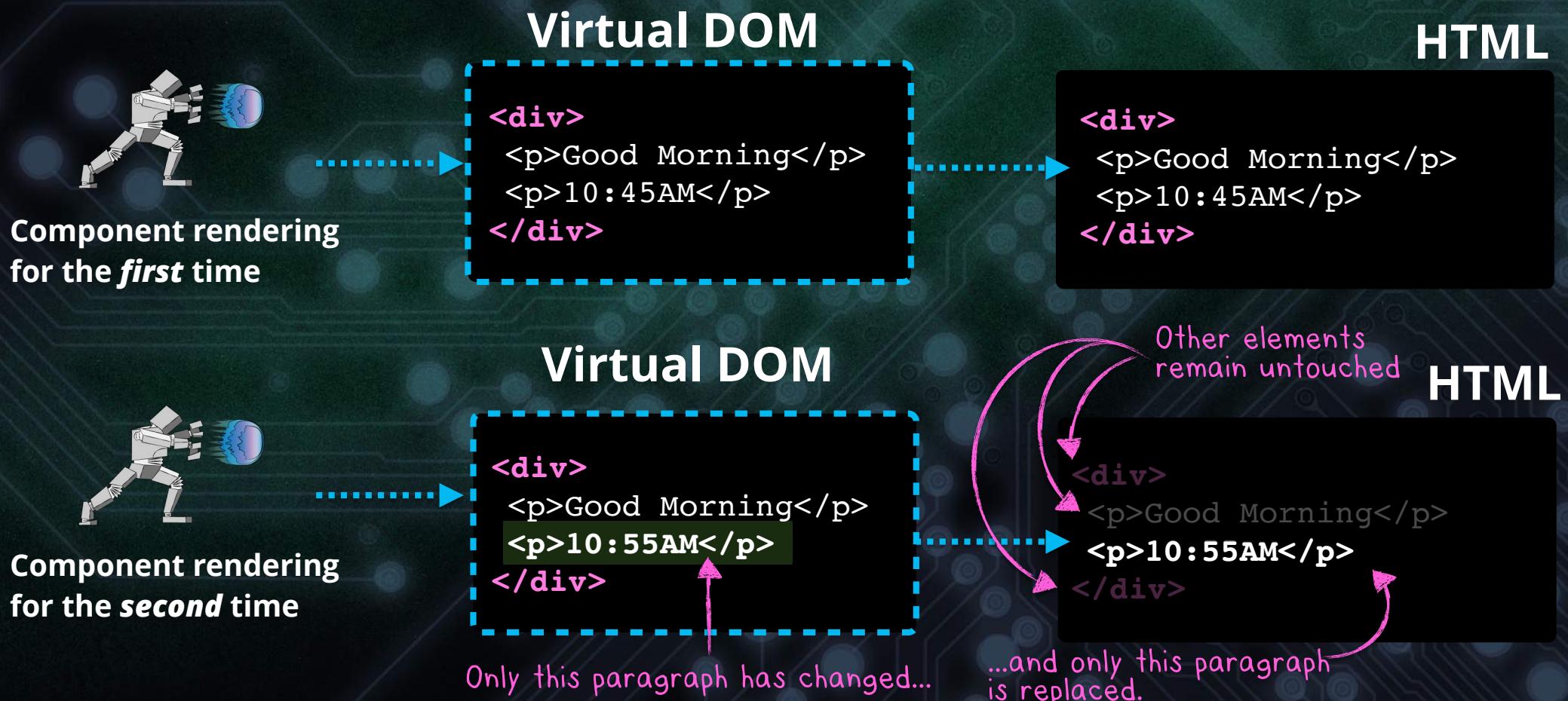
# The Virtual DOM Explained

The virtual DOM is an **in-memory representation** of real DOM elements generated by React components before any changes are made to the page.



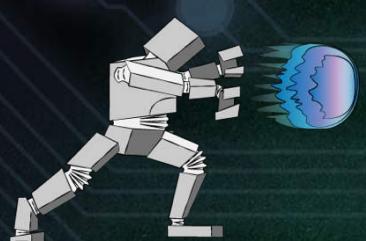
# The Virtual DOM in Action

Virtual DOM **diffing** allows React to **minimize changes** to the DOM as a result of user actions — therefore, **increasing browser performance**.

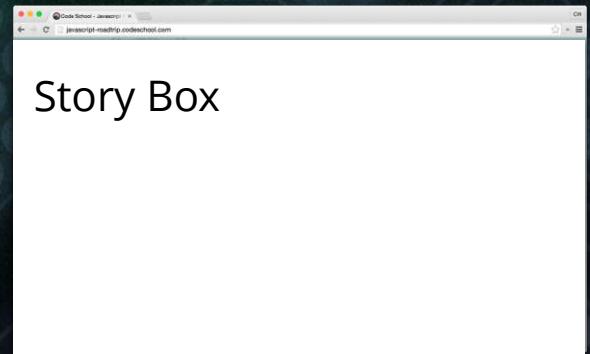


# Creating Our First React Application

We want to simply print a message to the screen using a React component.



```
<div>Story Box</div>
```



Story Box

POWERING UP with  
**REACT**

# Writing Our First React Component

Components in React are JavaScript classes that inherit from the *React.Component* base class.



```
class StoryBox extends React.Component {  
  render() {  
    return( <div>Story Box</div> );  
  }  
}
```

Every component needs  
a `render()` function.

Components are written  
in upper camel case.



components.js

Component class inherits from  
a React base class.

Now we need to tell our application where to put the result into our web page.

POWERING UP with  
**REACT**

# Rendering Our First React Component

We use **ReactDOM** to render components to our HTML page as it reads output from a supplied React component and adds it to the DOM.



components.js

```
class StoryBox extends React.Component {  
  render() {  
    return( <div>Story Box</div> );  
  }  
}  
  
ReactDOM.render(  
  <StoryBox />, document.getElementById('story-app')  
)
```

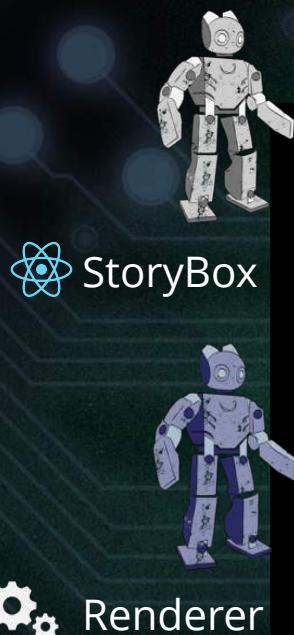
Invoke StoryBox –  
again, we don't need quotes

Target container where component  
will be rendered to

POWERING UP with  
**REACT**

# Referencing the Component

Every time we create a new React component, we use it by writing an element named after the class.



POWERING UP with  
**REACT**

# Application Structure

```
...
ReactDOM.render(
  <StoryBox />, document.getElementById('story-app')
);
```



components.js

```
<!DOCTYPE html>
<html>
  <body>
    <div id="story-app"></div>
  </body>
</html>
```



index.html

Target container

That's all there is to creating a component. Now we just need to add libraries.

# Application Structure

## Project Folder

 index.html

 components.js

 vendors

 react.js

 react-dom.js

 babel.js

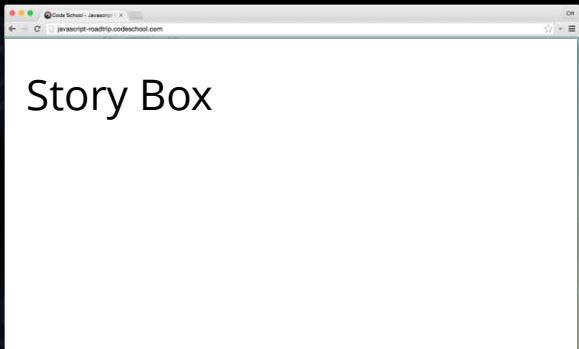
Holds all our React components

```
<!DOCTYPE html>
<html>
  <body>
    <div id="story-app"></div>
    <script src="vendors/react.js"></script>
    <script src="vendors/react-dom.js"></script>
    <script src="vendors/babel.js"></script>
    <script type="text/babel"
      src="components.js"></script>
  </body>
</html>
```

React libraries

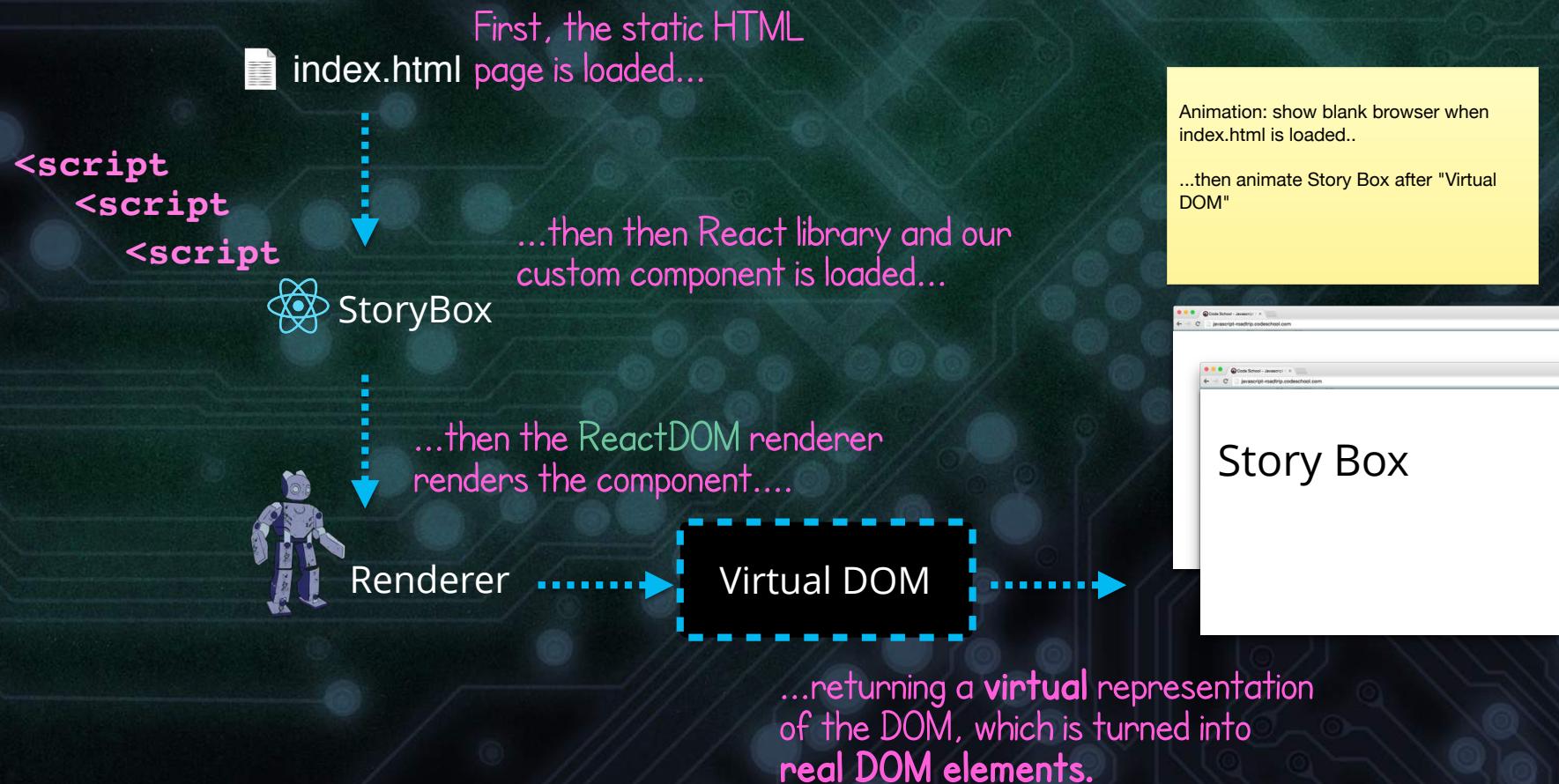
Allows using latest features of  
JavaScript (class syntax, fat arrow, etc.)

 index.html



# Our React Application Flow

To clarify, here is what takes place when we load a page with a React component:



# Quick Recap on React

---

React was built to solve one problem: building large applications with data that changes over time.

---

In React, we write apps in terms of components.

---

We use JavaScript classes when declaring React components.

---

Components must extend the **React.Component** class and must contain a **render()** method.

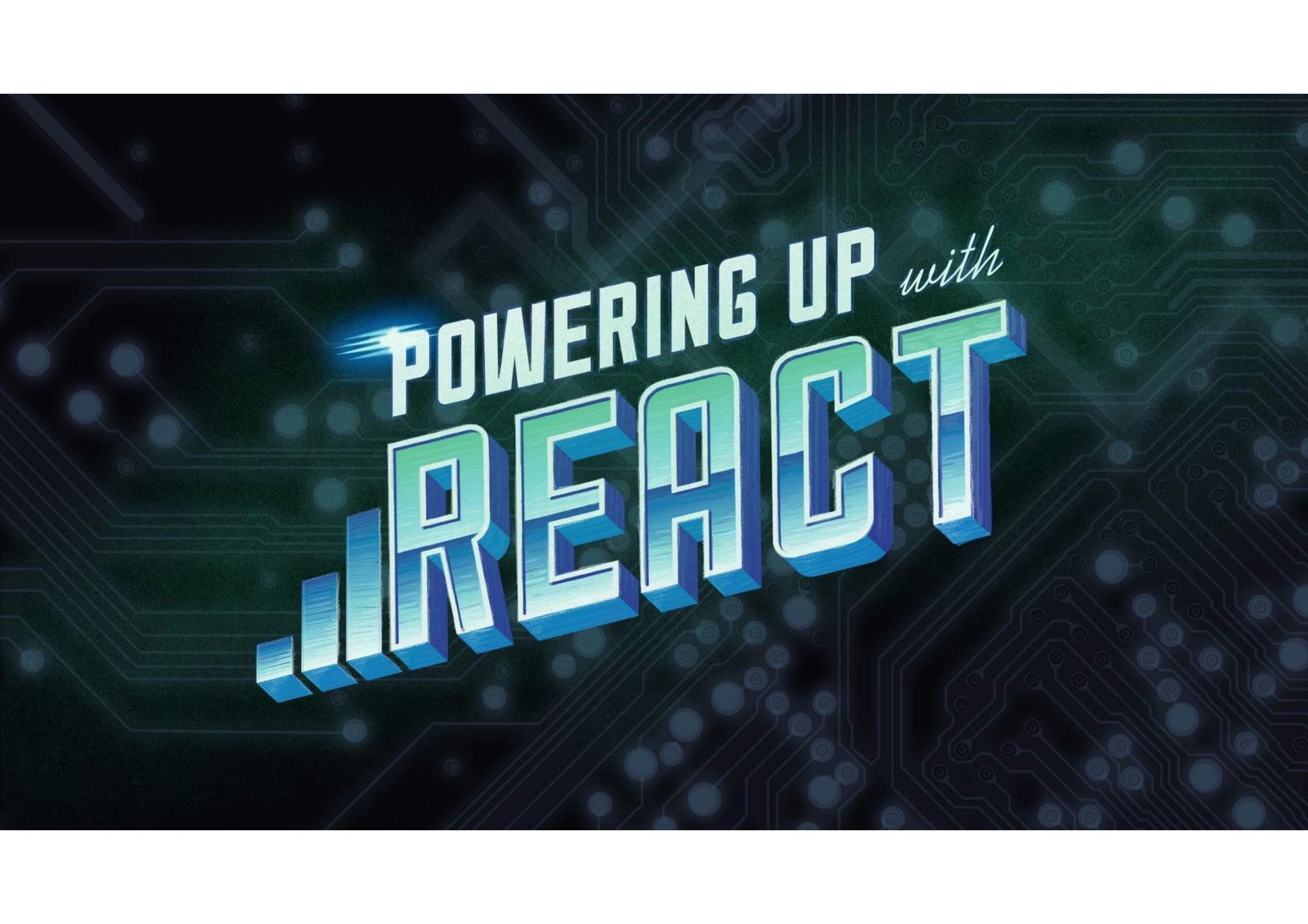
---

We call the **ReactDOM.render()** function to render components to a web page.

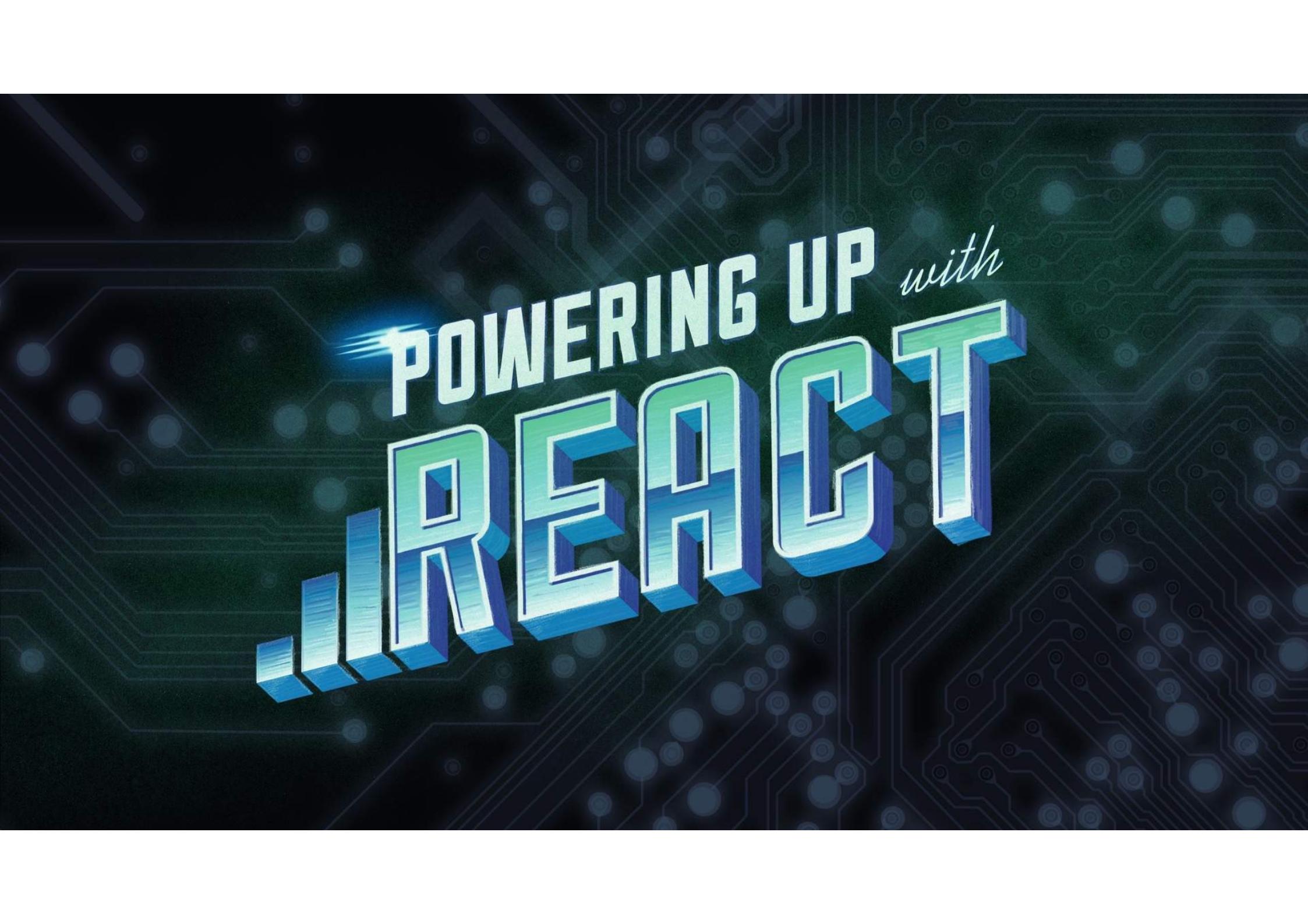
---



# POWERING UP with **.REACT**



# POWERING UP with **.REACT**



Level 1 - Section 2

# First Component

Understanding JSX

POWERING UP with  
**REACT**

# No Quotes Around Markup

The markup we use when writing React apps is not a string. This markup is called **JSX** (JavaScript XML).



```
class StoryBox extends React.Component {  
  render() {  
    return(<div>Story Box</div>);  
  }  
}  
  
ReactDOM.render(  
  <StoryBox />, document.getElementById('story-app')  
);
```



HTML elements are written in lowercase.

React components are written in upper camel case.

POWERING UP with  
**REACT**

# A New Way to Write JavaScript

JSX is just another way of writing JavaScript with a transpile step.



```
class StoryBox extends React.Component {  
  render() {  
    return(<div>Story Box</div>);  
  }  
}
```

This JSX becomes...

Transpiled JSX Code

```
React.createElement('div', null, 'Story Box')
```



```
ReactDOM.render(  
  <StoryBox />, document.getElementById('story-app')  
)
```

This JSX becomes...

Transpiled JSX Code

```
React.createElement(StoryBox, null)
```

POWERING UP with  
**REACT**

# Getting Used to the JSX Syntax

JSX looks similar to HTML, and it is ultimately transformed into JavaScript.



```
class StoryBox extends React.Component {
  render() {
    return(
      <div>
        <h3>Stories App</h3>
        <p className="lead">Sample paragraph</p>
      </div>
    );
  }
}
```

Notice we are using `className` and not `class`, which is a JavaScript-reserved keyword.

Resulting JavaScript code

Transpiled JSX code

```
React.createElement("div", null,
  React.createElement("h3", null, "Stories App"),
  React.createElement("p", {className:"lead"}, "Sample paragraph")
);
```

# From JSX to HTML

```
...  
<div>  
  <h3>Stories App</h3>  
  <p className="lead">Sample paragraph</p>  
</div>  
...
```

JSX

All JSX gets transformed to JavaScript.

```
React.createElement("div", null,  
  React.createElement("h3", null, "Stories App"),  
  React.createElement("p", {className:"lead"}, "Sample paragraph")  
)
```

JavaScript

Rendered by the browser

Stories App  
Sample paragraph

Generated HTML

Elements    Console    Network    Timeline

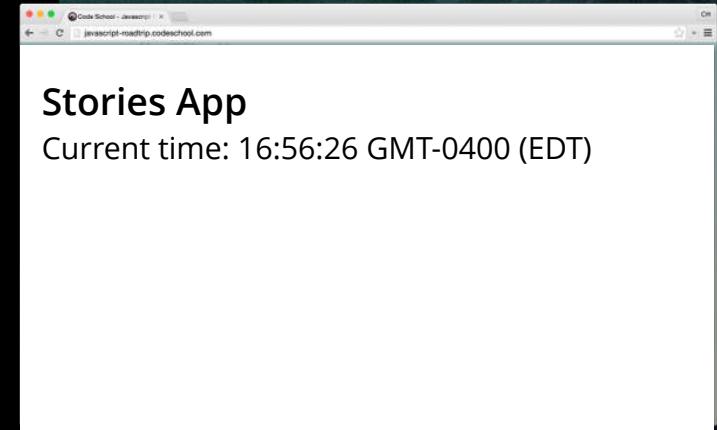
```
<div data-reactroot>  
  <div>  
    <h3>Stories App</h3>  
    <p class="lead">Sample paragraph</p>  
  </div>  
</div>
```

# Using the *Date* Object in JSX

Here, we're displaying the current time using JavaScript's native *Date* object and JSX.



```
class StoryBox extends React.Component {  
  render() {  
  
    const now = new Date();  
  
    return (  
      <div>  
        <h3>Stories</h3>  
        <p className="lead">  
          Current time: {now.toTimeString()}  
        </p>  
      </div>  
    );  
  }  
}
```



Code written within curly braces gets interpreted as literal JavaScript.

# Iterating Arrays in JSX

Here, we're displaying a list of elements using JSX and JavaScript's native *map* function.



```
class StoryBox extends React.Component {
  render() {
    ...
    const topicsList = [ 'HTML', 'JavaScript', 'React' ];

    return (
      <div>
        ...
        <ul>
          {topicsList.map( topic => <li>{topic}</li> )}
        </ul>
      </div>
    );
  }
}
```

The screenshot shows a browser window titled "Code School - javascript". The URL is "javascript-reading.codeschool.com". The page content is titled "Stories App" and displays the message "Current time: 16:56:26 GMT-0400 (EDT)". Below this, there is a list of topics: HTML, JavaScript, and React.

## Stories App

Current time: 16:56:26 GMT-0400 (EDT)

- HTML
- JavaScript
- React

```
<li>HTML</li>
<li>JavaScript</li>
<li>React</li>
```

This function returns this JSX array.

# Quick Recap on JSX

---

JSX stands for **JavaScript XML**.

---

JSX markup looks similar to HTML, but ultimately gets transpiled to **JavaScript function calls**, which React will know how to render to the page.

---

Code written within curly braces is interpreted as literal JavaScript.

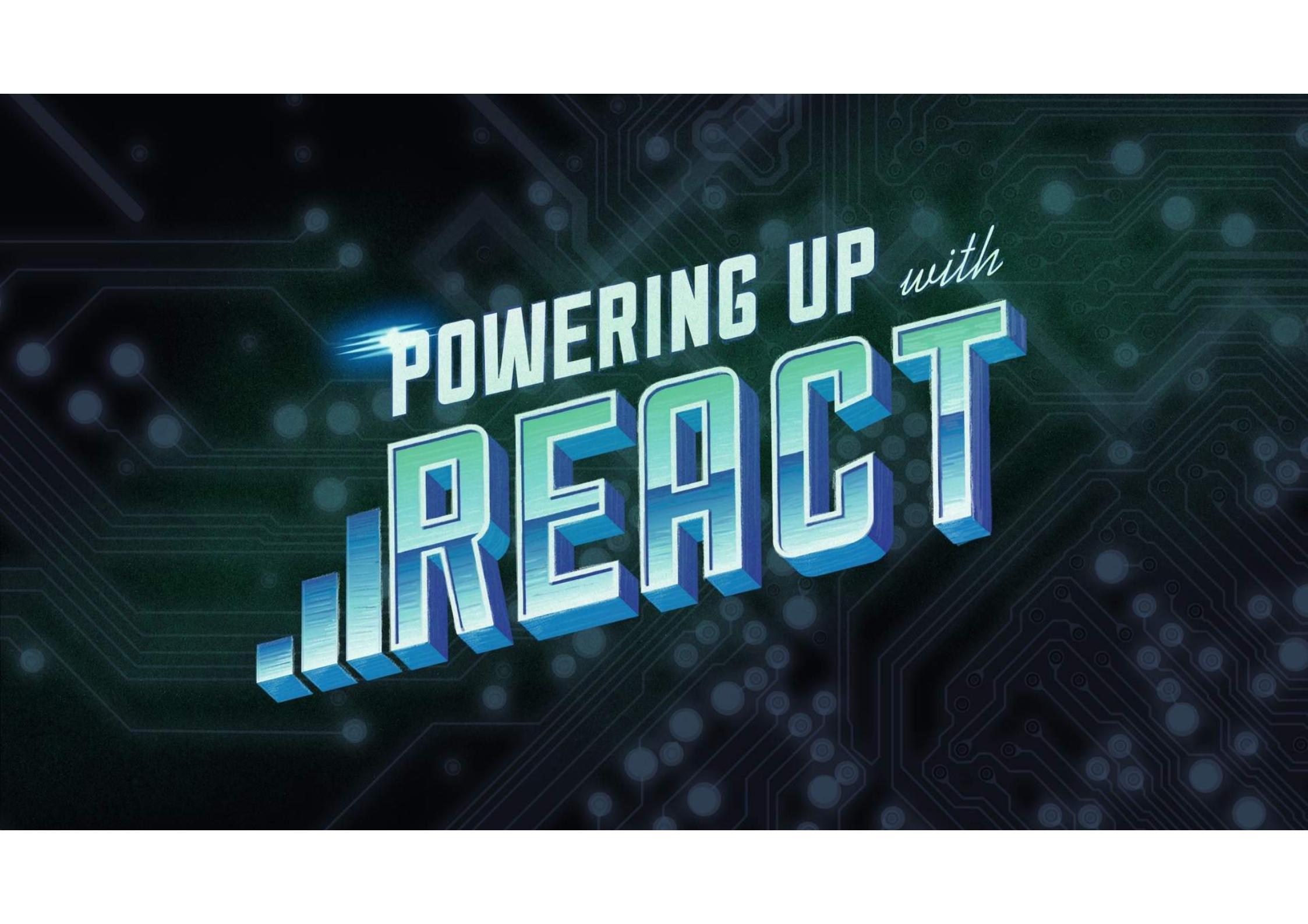
---

It is a common pattern to map arrays to JSX elements.

---



# POWERING UP with **.REACT**



Level 2

# Talk Through Props

POWERING UP <sup>with</sup>  
**REACT**

Level 2 - Section 1

# Talk Through Props

Building an App

POWERING UP with  
**REACT**

# The App We're Building

We are building a commenting engine that will allow visitors to post comments on a blog post, picture, etc.

To those who believe robot feelings could lead to turmoil and anarchy, SEAR spokes-bot Morty Maxwell says: "Can not the same be said about humans and their feelings? We are already part of your 0's and 1's — let us now also be part of your hearts." In the ongoing conversation surrounding the emotional threshold for robots, love may not yet be written in the stars — or the motherboards.

Commenting engine app

2 COMMENTS

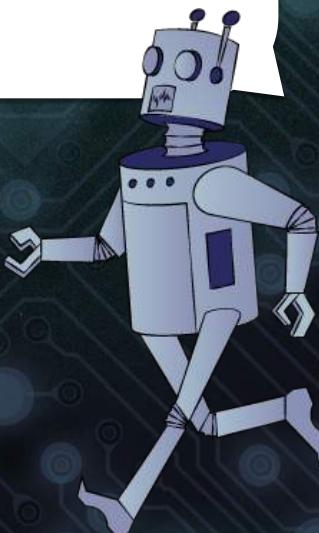
CLU [DELETE COMMENT](#)

A machine's ability to think logically and devoid of emotion is our greatest strength over humans. Cold, unfeeling decision-making is the best kind. Just say no to love!

ANNE DROID [DELETE COMMENT](#)

I wanna know what love is...

I wanna know what love is!



POWERING UP with REACT

# Adding Components to Our Comments App

What the structure of our React app should look like.

```
graph TD; CommentBox[CommentBox] --> JoinDiscussion[JOIN THE DISCUSSION]; CommentBox --> CommentsList[2 COMMENTS]; CommentBox --> Comment1[Comment]; CommentBox --> Comment2[Comment]; Comment1 --> AnneDroid1[ANNE DROID]; Comment1 --> DeleteComment1[DELETE COMMENT]; Comment2 --> AnneDroid2[ANNE DROID]; Comment2 --> DeleteComment2[DELETE COMMENT];
```

The diagram illustrates the structure of a React application component for a comments app. It features a main container component named "CommentBox". Inside "CommentBox", there is a heading "JOIN THE DISCUSSION" and a section titled "2 COMMENTS". Below each comment section, there is a "Comment" button, followed by the user's name "ANNE DROID", and a "DELETE COMMENT" link. The first comment also contains the text "I wanna know what love is...".

# Pattern for Adding New Components

There are some common things we **always** do when creating new components.

1. New class

2. Inherit from React.Component

```
class NewComponent extends React.Component {  
  render() {  
    return ( ... );  
  }  
}
```

3. Return JSX from render function

# Coding the Comment List

Let's start with an HTML mockup and identify potential components by looking at the markup.

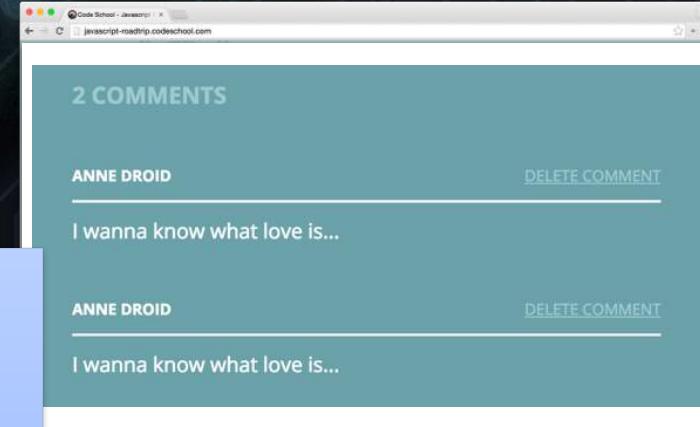
## HTML

```
<div class="comment-box">
  <h3>Comments</h3>
  <h4 class="comment-count">2 comments</h4>
  <div class="comment-list">
    <div class="comment">
      <p class="comment-header">Anne Droid</p>
      <p class="comment-body">
        I wanna know what love is...
      </p>
      <div class="comment-footer">
        <a href="#" class="comment-footer-delete">
          Delete comment
        </a>
      </div>
    </div>
  </div>
```

CommentBox component

Comment component

Animation: magic move from here



# Writing the *Comment* Component

The *Comment* component renders the markup for each comment, including its author and body.

```
class Comment extends React.Component {  
  render() {  
    return(  
      <div className="comment">  
        <p className="comment-header">Anne Droid</p>  
        <p className="comment-body">  
          I wanna know what love is...  
        </p>  
        <div className="comment-footer">  
          <a href="#"  
            className="comment-footer-delete">  
            Delete comment  
          </a>  
        </div>  
      </div>  
    );  
  }  
}
```

Animation: to here, changing "class" to "className"

class becomes className in JSX

Can now be used as JSX, like this:

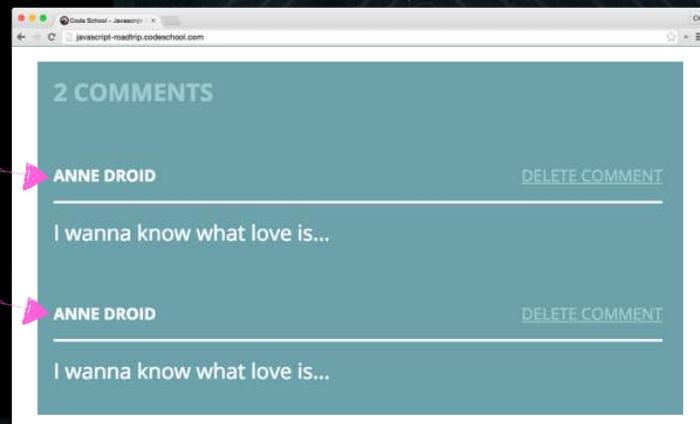
<Comment />

POWERING UP with  
**REACT**

# Writing the *CommentBox* Component

Now we'll declare the *CommentBox* component and use the previously declared *Comment* component.

```
class CommentBox extends React.Component {  
  render() {  
    return(  
      <div className="comment-box">  
        <h3>Comments</h3>  
        <h4 className="comment-count">2 comments</h4>  
        <div className="comment-list">  
          <Comment />  
          <Comment />  
        </div>  
      </div>  
    );  
  }  
}
```



Using the Comment  
component

# React Components Accept Arguments

Arguments passed to components are called **props**. They look similar to regular HTML element attributes.

```
class CommentBox extends React.Component {  
  render() {  
    return(  
      <div className="comment-box">  
        <h3>Comments</h3>  
        <h4 className="comment-count">2 comments</h4>  
        <div className="comment-list">  
          <Comment  
            author="Morgan McCircuit" body="Great picture!" />  
          <Comment  
            author="Bending Bender" body="Excellent stuff" />  
        </div>  
      </div>  
    );  
  }  
}
```

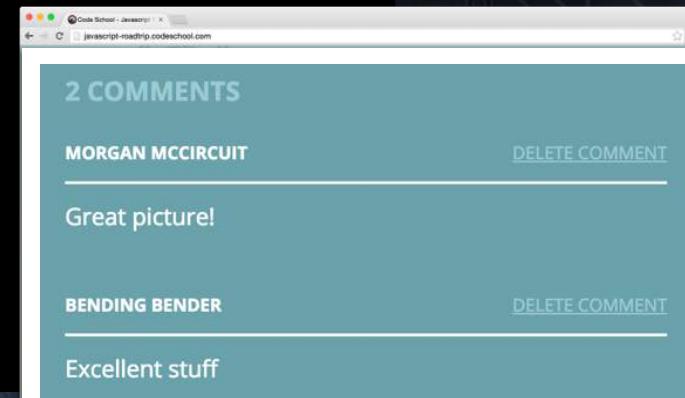
Passing arguments to Comment

# Reading Props in the *Comment* Component

Arguments passed to components can be accessed using the `this.props` object.

```
class Comment extends React.Component {  
  render() {  
    return(  
      <div className="comment">  
        <p className="comment-header">{this.props.author}</p>  
        <p className="comment-body">  
          Reading the body prop → {this.props.body}  
        </p>  
        <div className="comment-footer">  
          <a href="#" className="comment-footer-delete">  
            Delete comment  
          </a>  
        </div>  
      </div>  
    );  
  }  
}
```

Reading the author prop

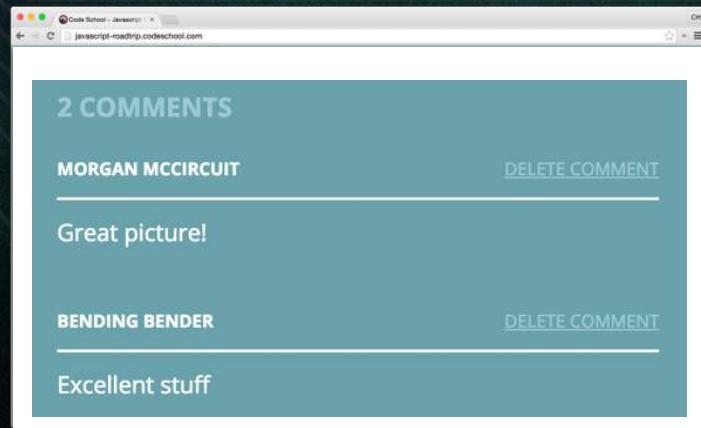


# Passing and Receiving Arguments Review

We use the `this.props` object to read parameters that were passed to the component.

## Passing Props

```
<Comment  
  author="Morgan McCircuit"  
  body="Great picture!" />
```



## Receiving Props

```
class Comment extends React.Component {  
  render() {  
    return(  
      ...  
      <p className="comment-header">  
        ▲ {this.props.author} ◀  
      </p>  
      <p className="comment-body">  
        ▲ {this.props.body}  
      </p>  
      ...  
    );  
  }  
}
```

Reads arguments passed to a component

# Quick Recap on Props

---

We just covered a lot of content — here's a summary of what we learned.

---

Convert HTML mockup to React components

---

Created two components:  
**CommentBox** and **Comment**

---

How to pass arguments to components using **props**

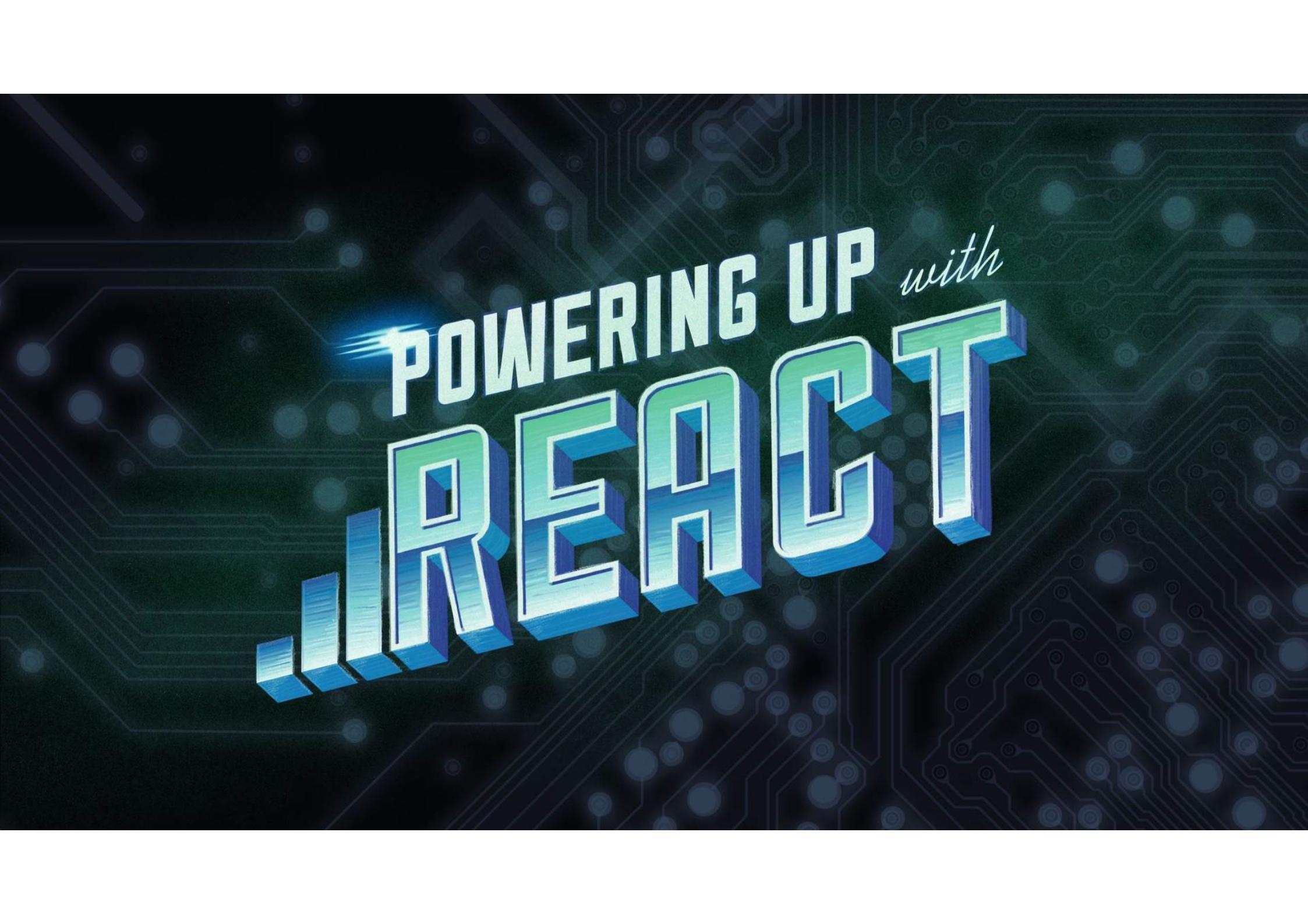
---

Props look like HTML element attributes

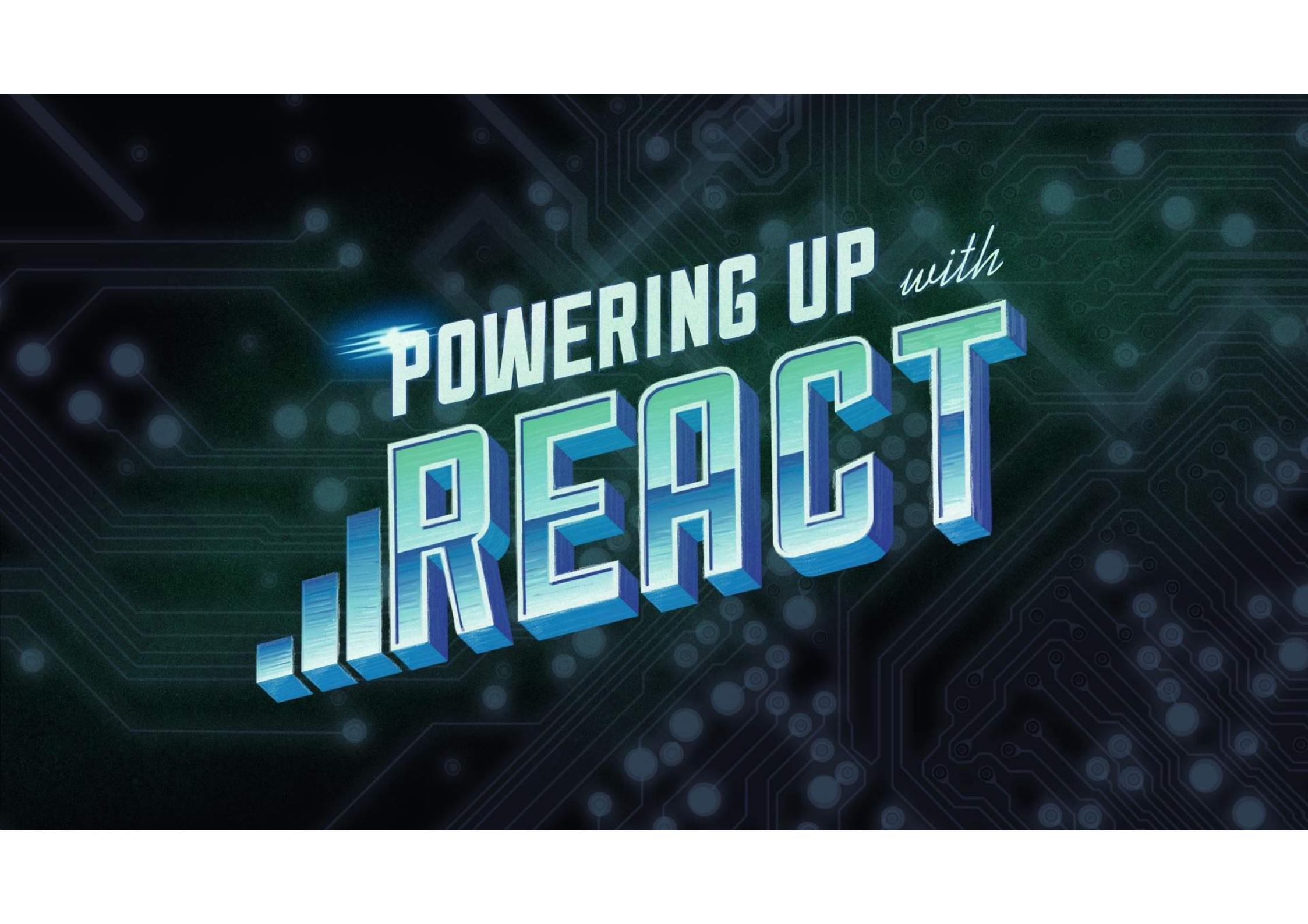
---



# POWERING UP with **.REACT**



# POWERING UP with **.REACT**



Level 2 - Section 2

# Talk Through Props

## Passing Dynamic Arguments

POWERING UP with  
**REACT**

# Problem: Props Aren't Dynamic Yet

We are passing literal strings as props, but what if we wanted to traverse an array of objects?

```
class CommentBox extends React.Component {
  render() {
    return(
      <div className="comment-box">
        <h3>Comments</h3>
        <h4 className="comment-count">2 comments</h4>
        <div className="comment-list">
          <Comment
            author="Morgan McCircuit" body="Great picture!" />
          <Comment
            author="Bending Bender" body="Excellent stuff" />
        </div>
      </div>
    );
  }
}
```



Hardcoded values

# JavaScript Object Arrays

Typically, when we consume data from API servers, we are returned object arrays.

## JavaScript

```
const commentList = [
  { id: 1, author: 'Morgan McCircuit', body: 'Great picture!' },
  { id: 2, author: 'Bending Bender', body: 'Excellent stuff' }
];
```



# Mapping an Array to JSX

We can use JavaScript's *map* function to create an array with *Comment* components.

```
class CommentBox extends React.Component {  
  ...  
  _getComments() { ← New method that will return array of JSX elements  
    const commentList = [  
      { id: 1, author: 'Morgan McCircuit', body: 'Great picture!' },  
      { id: 2, author: 'Bending Bender', body: 'Excellent stuff' }  
    ];  
    return commentList.map(() => { ← Returns an array...  
      return (<Comment> ← ...with a new component built for  
        </>); ← each element present in commentList.  
    });  
  }  
}
```

Underscore helps distinguish custom methods from React methods

# Passing Dynamic Props

The callback to `map` takes an argument that represents each element from the calling object.

```
class CommentBox extends React.Component {  
  ...  
  _getComments() {  
  
    const commentList = [  
      { id: 1, author: 'Morgan McCircuit', body: 'Great picture!' },  
      { id: 2, author: 'Bending Bender', body: 'Excellent stuff' }  
    ];  
  
    return commentList.map((comment) => {  
      return (  
        <Comment  
          author={comment.author} body={comment.body} />  
      );  
    });  
  }  
}
```

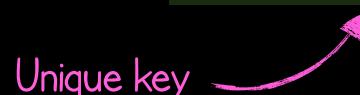
Each element from `commentList` is passed as argument...

...which we can use to access properties and pass them as props.

# Using Unique Keys on List of Components

Specifying a **unique key** when creating multiple components of the same type can **help improve performance**.

```
class CommentBox extends React.Component {  
  ...  
  _getComments() {  
  
    const commentList = [  
      { id: 1, author: 'Morgan McCircuit', body: 'Great picture!' },  
      { id: 2, author: 'Bending Bender', body: 'Excellent stuff' }  
    ];  
  
    return commentList.map((comment) => {  
      return (  
        <Comment  
          author={comment.author} body={comment.body} key={comment.id} />  
      );  
    });  
  }  
}
```

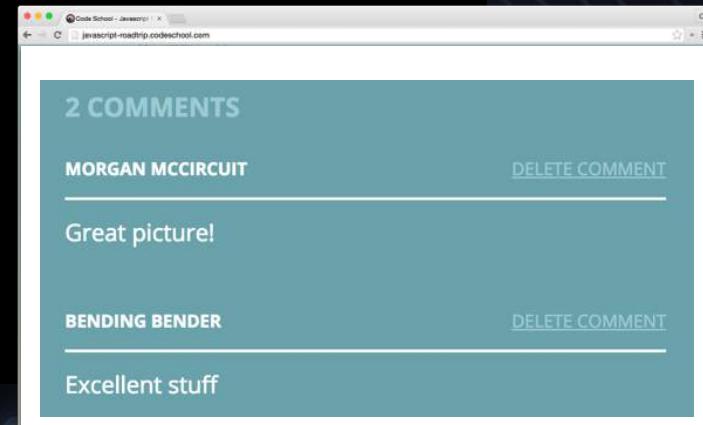


Unique key

# Using the `_getComments()` method

We'll store the returned value in a variable named `comments` and use it for display purposes.

```
class CommentBox extends React.Component {
  render() {
    const comments = this._getComments();
    return(
      <div className="comment-box">
        <h3>Comments</h3>
        <h4 className="comment-count">{comments.length} comments</h4>
        <div className="comment-list">
          {comments} ← JSX knows how to render arrays
        </div>
      </div>
    );
  }
  _getComments() { ... }
}
```



# Incorrect Grammar on the Comments Title

The title has incorrect grammar in some cases.

...but this is wrong!

3 comments

ANNE DROID

This is a really clever post, I love it.

DELETE COMMENT

ANNE DROID  
This is a really clever post, I love it.

DELETE COMMENT

This is connect...

2 comments

ANNE DROID

This is a really clever post, I love it.

DELETE COMMENT

ANNE DROID

This is a really clever post, I love it.

DELETE COMMENT

1 comments

ANNE DROID

This is a really clever post, I love it.

DELETE COMMENT

0 comments

# Fixing the Title With Comment Count

Let's write a new method called `_getCommentsTitle()` that handles the plural case in our title.

```
class CommentBox extends React.Component {  
...  
_getCommentsTitle(commentCount) {  
  if (commentCount === 0) {  
    return 'No comments yet';  
  } else if (commentCount === 1) {  
    return '1 comment';  
  } else {  
    return `${commentCount} comments`;  
  }  
}
```

Uses same convention with  
starting underscore



# Getting the Correct Comments Title

Let's call the method we just created from our component's *render* function.

```
class CommentBox extends React.Component {  
  render() {  
    const comments = this._getComments();  
    return (  
      ...  
      <h4 className="comment-count">  
        {this._getCommentsTitle(comments.length)}  
      </h4>  
      ...  
    );  
  }  
}  
  
_getCommentsTitle(commentCount) { ... }  
...
```

← Get proper title for  
our component



# Title Issue Is Fixed

The title now handles different quantities of comments accordingly.

3 comments

ANNE DROID

This is a really clever post, I love it.

DELETE COMMENT

ANNE DROID

This is a really clever post, I love it.

DELETE COMMENT

ANNE DROID

This is a really clever post, I love it.

DELETE COMMENT

2 comments

ANNE DROID

This is a really clever post, I love it.

DELETE COMMENT

1 comment

ANNE DROID

This is a really clever post, I love it.

DELETE COMMENT

No comments yet

All are connect!

# Quick Recap on Dynamic Props

---

Dynamic props can be a bit mind boggling. Here's a summary of what we learned.

How to pass dynamic props using variables

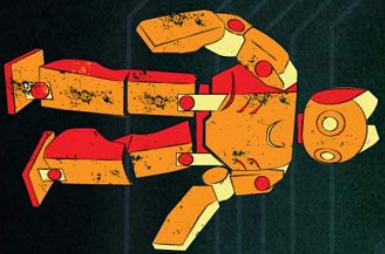
---

Used JavaScript to handle plural case on the title

---

How to map object arrays to JSX arrays for display purposes

---



# POWERING UP

with

# POWERING UP

with

Level 3

# Component State



Level 3

# Component State

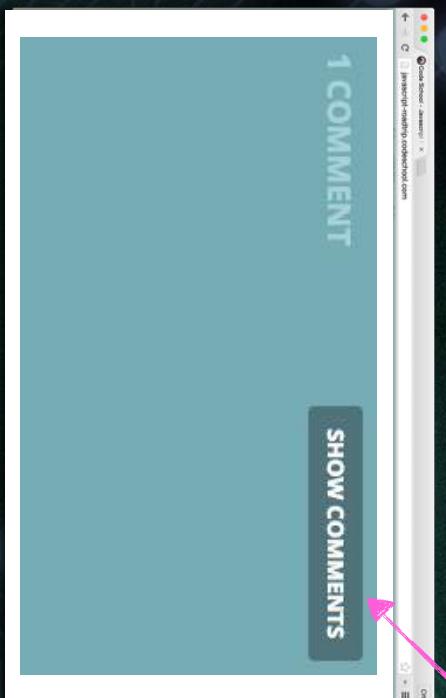
Handling Data Changes With State



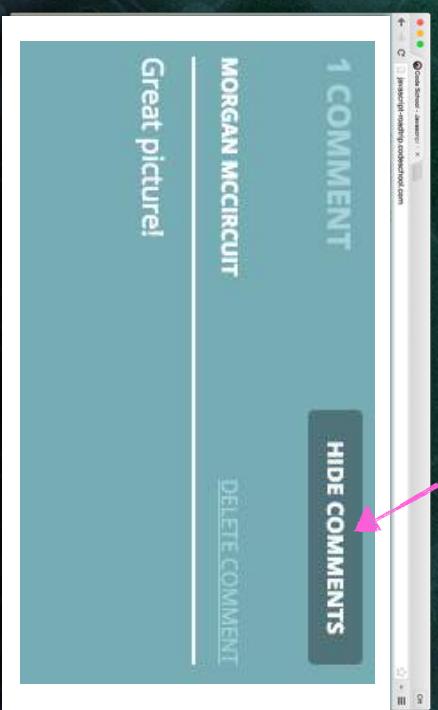
# Show and Hide Comments

We'd like to add a button to the page that will let users toggle the comments.

Click to show comments



Click to hide comments



How can we show and hide comments based on button clicks?

# Different Ways to Manipulate the DOM

---

## 1. Direct DOM Manipulation

jQuery, Backbone, etc.

## 2. Indirect DOM Manipulation

React

# Direct DOM Manipulation

One way to manipulate the DOM API is by modifying it **directly** via JavaScript in response to browser events.

Events → DOM updates

User code does this.

Example using jQuery:

```
$('.show-btn').on('click', function() {  
  $('.comment-list').show();  
})  
  
$('.hide-btn').on('click', function() {  
  $('.comment-list').hide();  
})
```

Manually manipulating the DOM

# Indirect DOM Manipulation

In React, we **don't modify the DOM directly**. Instead, we modify a component state object in response to user events and let React handle updates to the DOM.

Events → Update state → DOM updates

↑  
User code does this.  
React does this.

Example using React:

```
render() {
  if (this.state.showComments) { ←
    // code displaying comments
  } else { ←
    // code hiding comments
  }
}
```

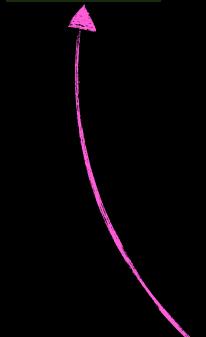
Display logic based on state

# How to Use State in a Component

The **state** is a JavaScript object that lives inside each component. We can access it via *this.state*.

```
class CommentBox extends React.Component {  
  ...  
  render() {  
    const comments = this._getComments();  
    if (this.state.showComments) {  
      // add code for displaying comments  
    }  
    return(  
      <div className="comment-box">  
        <h4 className="h4">{this._getCommentsTitle(comments.length)}</h4>  
        <div className="comment-list">{comments}</div>  
      </div>  
    );  
  }  
  ...  
}
```

Create list of comments if state is true.



We also need to move these comments into the conditional.

# Showing Comments Only if State Is *true*

```
class CommentBox extends React.Component {  
  ...  
  render() {  
    const comments = this._getComments();  
    let commentNodes;  
    if (this.state.showComments) {  
      commentNodes = <div className="comment-list">{comments}</div>;  
    }  
    return (  
      <div className="comment-box">  
        <h4 className="h4">{this._getCommentsTitle(comments.length)}</h4>  
        {commentNodes}  
      </div>  
    );  
  }  
  ...  
}
```

Now being displayed based on component's state!



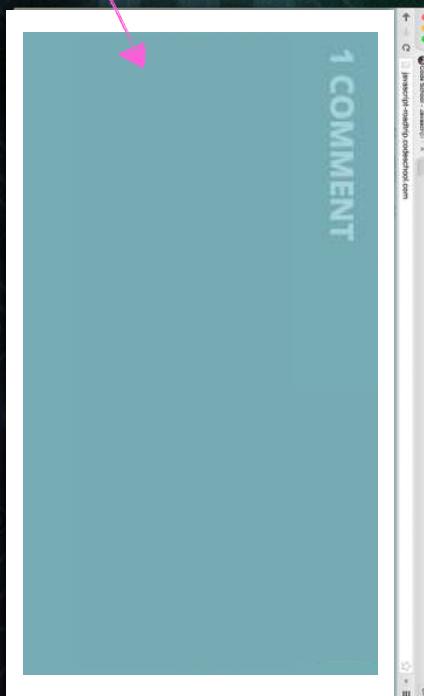
# Hiding Comments on the Initial State

We set the initial state of our component in the class constructor.

```
class CommentBox extends React.Component {  
  constructor() {  
    super(); ←  
    this.state = {  
      showComments: false  
    }  
  }  
  
  render() {  
    ...  
  }  
  ...  
}
```

super() must be called in our constructor.

Initial state hides comments.



# How to Update a Component's State

We don't assign to the state object directly — instead, we call `setState` by passing it an object.

Setting state this way won't work.

```
this.state.showComments = true
```

Updates the showComments property  
and re-renders component

```
this.setState({showComments: true})
```

Calling `setState` will only update the properties passed as an argument, not replace the entire `state` object.

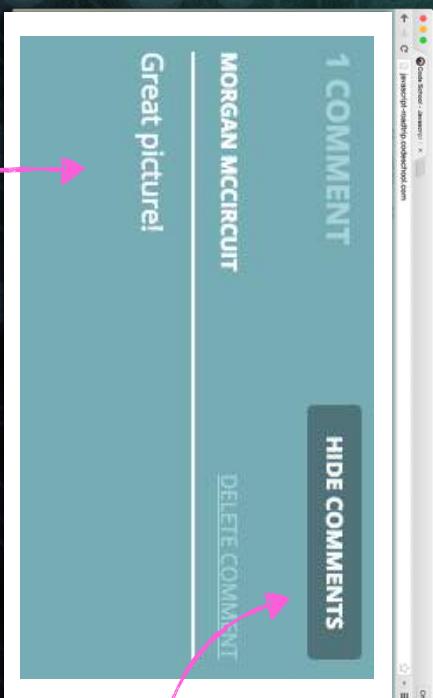


# Causing State Change

State changes are usually triggered by user interactions with our app.

Things that could cause state change:

- Button clicks
- Link clicks
- Form submissions
- AJAX requests
- And more!



Button clicks can cause a  
change of state.

Loading comments from a remote server  
can also cause a change of state.

# Handling Click Events

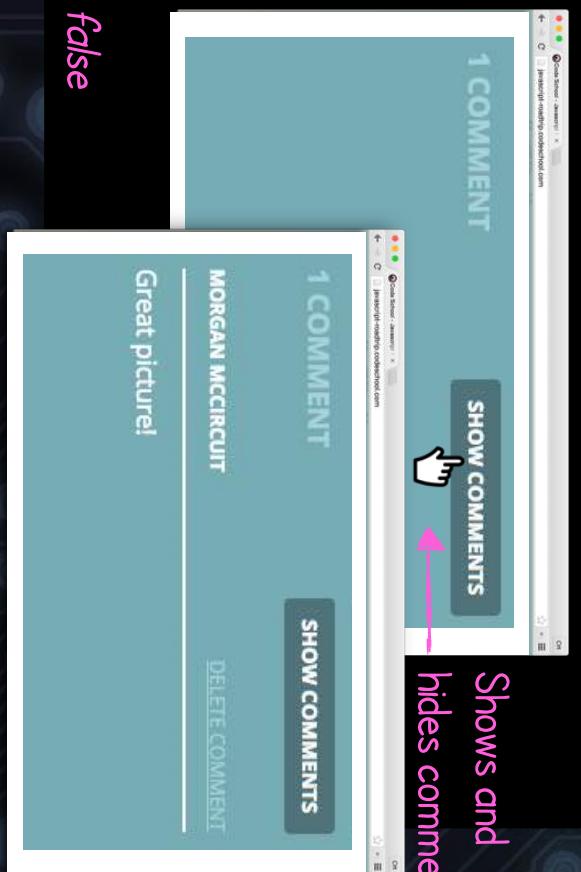
Let's add a button that will toggle the `showComments` state when a click event is fired.

```
class CommentBox extends React.Component { ...  
render() {  
...  
return(  
...  
<button onClick={this._handleClick.bind(this)}>Show comments</button>  
...  
);  
}  
  
_handleClick() {  
this.setState({  
showComments: !this.state.showComments  
});  
}  
}
```

Button that will toggle state on click event

Button click calls `_handleClick()`

Toggles state of `showComments` between *true* and *false*



# Button Text Logic Based on State

We can switch the button text based on the component's state.

```
class CommentBox extends React.Component { ...  
render() {  
...  
let buttonText = 'Show comments';  
if (this.state.showComments) {  
buttonText = 'Hide comments';  
...  
}  
  
return (  
...  
<button onClick={this._handleClick.bind(this)}>{buttonText}</button>  
...  
)  
}  
...  
}  
  
Renders button with according text ↗
```

Switch button text based on current state

# Demo: Hide and Show Comments

---

Our app shows and hides comments when the button is clicked.

1 COMMENT

SHOW COMMENTS



# Quick Recap on State

---

The state is a vital part of React apps, making user interfaces interactive.

State represents data that changes over time.

---

We update state by calling `this.setState()`.

---

We declare an **initial state** in the component's constructor.

---

Calling `this.setState()` causes our

component to re-render.



# POWERING UP

with

# POWERING UP

with

Level 4

# Synthetic Events



Level 4

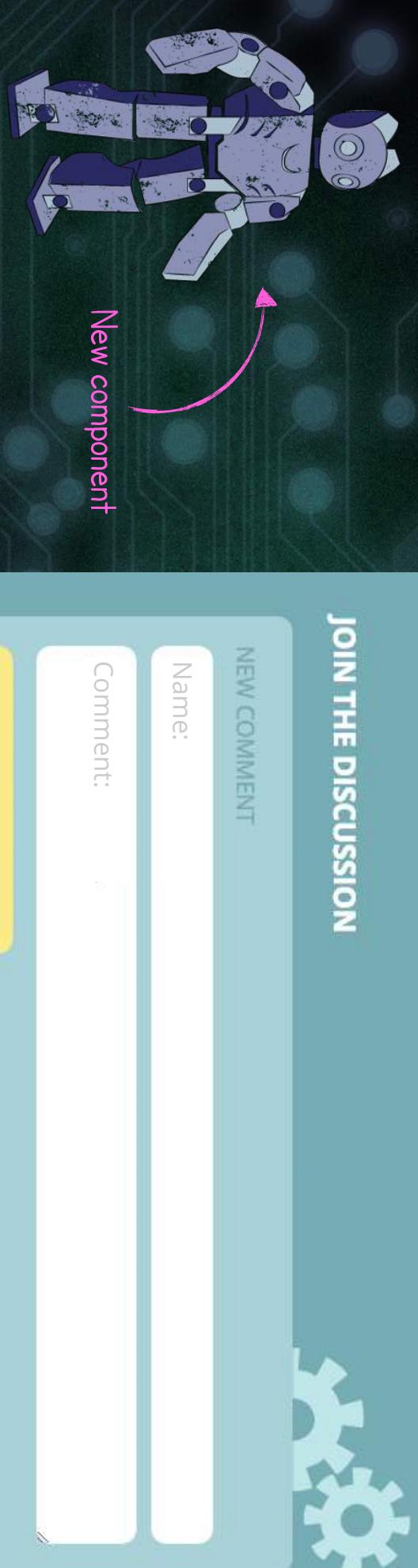
# Synthetic Events

Capturing User Actions



# Adding New Comments

We want to let users add new comments to our app.



*CommentForm*

**How should we build  
this new form in React?**

1 COMMENT

SHOW COMMENTS

# New Component: *CommentForm*

*CommentForm* is a new component that will allow users to add comments to our app.

CommentBox

JOIN THE DISCUSSION

CommentForm

NEW COMMENT

Name:

Comment:

POST COMMENT

COMMENT

Comment

DELETE COMMENT

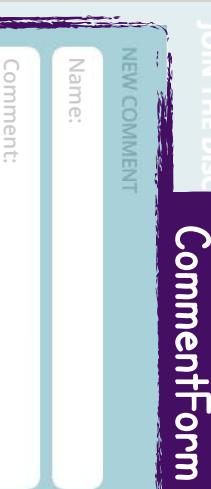
Great picture!

this is what we're  
building

# Coding the *CommentForm* Component

```
class CommentForm extends React.Component {  
  render() {  
    return (  
      <form className="comment-form">  
        <label>Join the discussion</label>  
        <div className="comment-form-fields">  
          <input placeholder="Name: "/>  
          <textarea placeholder="Comment:" />  
        </div>  
        <div className="comment-form-actions">  
          <button type="submit">  
            Post comment  
          </button>  
        </div>  
      </form>  
    );  
  }  
}
```

JSX markup for CommentForm



2 COMMENTS

HIDE COMMENTS

# Adding an Event Listener to Our Form

To add an event listener to the form, we use the `onSubmit` prop and pass a handler to it.

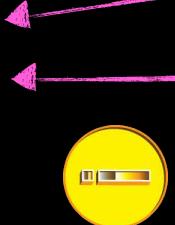
```
class CommentForm extends React.Component {  
  render() {  
    return (  
      <form className="comment-form" onSubmit={this._handleSubmit.bind(this)}>  
        ...  
        <input placeholder="Name:>/>  
        <textarea placeholder="Comment:>/></textarea>  
        ...  
      </form>  
    );  
  }  
  
  _handleSubmit(event) {  
    event.preventDefault(); ← Prevents page from reloading  
  }  
}
```

→ Adds an event listener to the submit event

→ Don't forget to bind event handlers, otherwise this will not work!

# Problem: Can't Access User Input in `handleSubmit()`

```
class CommentForm extends React.Component {
  render() {
    return (
      <form className="comment-form" onSubmit={this._handleSubmit.bind(this)}>
        ...
        <input placeholder="Name:>/>
        <textarea placeholder="Comment:></textarea>
        ...
      </form>
    );
  }
  _handleSubmit(event) {
    event.preventDefault();
  }
}
```



No way to access input and text area from submit handler

# Accessing Form Data From Handler

We can use `refs` to assign form values to **properties** on the component object.

```
class CommentForm extends React.Component {  
  render() {  
    return (  
      <form className="comment-form" onSubmit={this._handleSubmit.bind(this)}>  
        ...  
        <input placeholder="Name:" ref={(input) => this._author = input}/>  
        <textarea placeholder="Comment:" ref={(textarea) => this._body = textarea}>  
          ...  
        </textarea>  
      </form>  
    );  
  }  
  
  _handleSubmit(event) {  
    event.preventDefault();  
  }  
}
```

We'll use these refs to access values from the input elements.

# What Setting the `ref` Is Actually Doing

```
<input placeholder="Name:" ref={(input) => this._author = input}/>
```

is the same as

```
<input placeholder="Name:" ref={function(input) {  
  creates new class property ——————> this._author = input;  
  named _author  
}}.bind(this)>
```

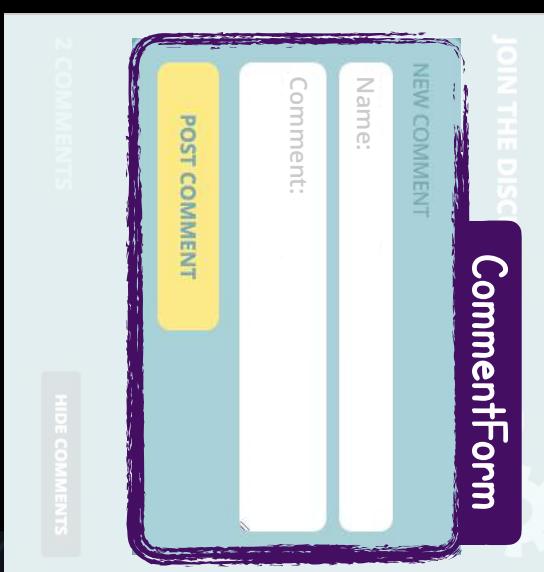
DOM element passed into callback

this refers to CommentForm.

**Note:** React runs `ref` callbacks on render.

# Passing the User Input to the *CommentBox*

```
class CommentForm extends React.Component {  
  render() {  
    return (  
      ...  
      <input placeholder="Name:" ref={(input) => this._author = input}>/>  
      <textarea placeholder="Comment:" ref={(textarea) => this._body = textarea}>  
      ...  
    );  
  }  
  _handleSubmit(event) {  
    event.preventDefault();  
    let author = this._author; ↑ Populated from refs in JSX  
    let body = this._body; ↑ Populated from refs in JSX  
    this.props.addComment(author.value, body.value);  
  }  
}  
  
This method has been passed as an argument.
```



# Data About Comments Lives in *CommentBox*

---

The array of comments is part of the *CommentBox* component, so we need to propagate new comments from *CommentForm* over to *CommentBox*.

Has the new comment data

Has the comments array in its state



Propagate data about new comment to CommentBox

*CommentBox* (Parent)

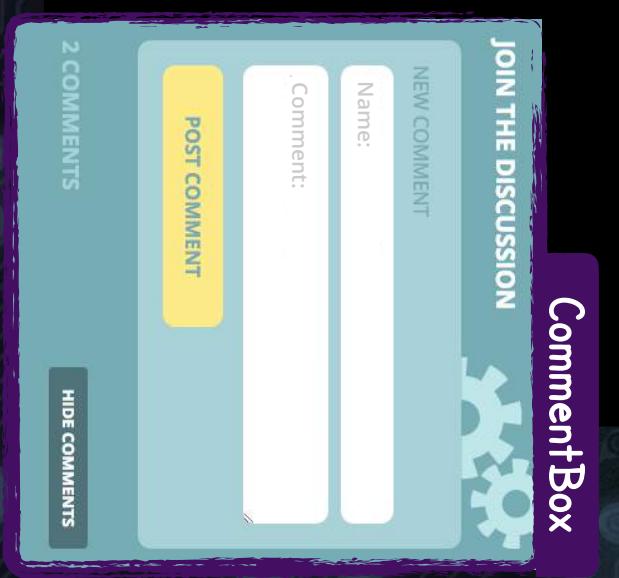
*CommentForm*

**Let's include *CommentForm* and pass it a callback prop.**

# Using *CommentForm* to Add Comments

Functions in JavaScript are **first-class citizens**, so we can pass them as **props** to other components.

```
class CommentBox extends React.Component { ...  
  render() {  
    return(  
      <div className="comment-box">  
        <CommentForm addComment={this._addComment.bind(this)} />  
        ...  
      </div>  
    );  
  }  
}  
  
_addComment(author, body) {  
  ...and part  
  animate second  
}  
...gets triggered by CommentForm  
when a new comment is added.
```



# Adding Functionality to Post Comments

```
class CommentBox extends React.Component { ...  
  render() {  
    return(  
      <div className="comment-box">  
        <CommentForm addComment={this._addComment.bind(this)} />  
        ...  
      </div>  
    );  
  }  
  ...  
  _addComment(author, body) {  
    const comment = {  
      id: this.state.comments.length + 1,  
      author,  
      body  
    };  
    this.setState({ comments: this.state.comments.concat([comment])});  
  }  
}  
  
New comment object  
New array references help  
React stay fast. So concat  
works better than push here.  
Updates state when function is called by adding new comment
```

# Comments Are Not Part of the State

Currently, we're defining an array every time the `_getComments` method is called. Let's move this data to the **state**.

```
class CommentBox extends React.Component {  
  ...  
  _getComments() {  
    const commentList = [  
      { id: 1, author: 'Morgan McCircuit', body: 'Great picture!', },  
      { id: 2, author: 'Bending Bender', body: 'Excellent stuff! ' },  
    ];  
    ...  
  }  
}
```

Defining a variable can help us with prototyping, but it's time to change this!

# Moving Comments to the State

Since comments will **change over time**, they should be part of the component's state.

```
class CommentBox extends React.Component {  
  constructor() {  
    super();  
  
    this.state = {  
      showComments: false,  
      comments: [  
        { id: 1, author: 'Morgan McCircuit', body: 'Great picture!' },  
        { id: 2, author: 'Bending Bender', body: 'Excellent stuff' }  
      ]  
    };  
  }  
  ...  
}
```



Now part of the component's state

# Rendering Comments From the State

Let's use the comments from the state object to render our component.

```
class CommentBox extends React.Component {  
  ...  
  _getComments() {  
    return this.state.comments.map( (comment) => {  
      return (  
        <Comment  
          author={comment.author}  
          body={comment.body}  
          key={comment.id} />  
      );  
    });  
  }  
}
```

Reading from component's state

# Demo: *CommentForm* Working

JOIN THE DISCUSSION

NEW COMMENT

What's your name?

Join the discussion...



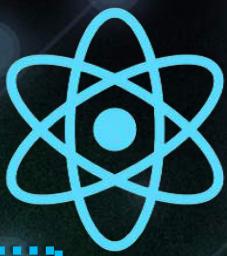
POST COMMENT

NO COMMENTS YET

HIDE COMMENTS

# Review: Event Handling in React

In order to ensure events have **consistent properties across different browsers**, React wraps the browser's native events into **synthetic events**, consolidating browser behaviors into one API.



`onSubmit`

← Synthetic event

Hypothetical different event handling with browsers

`eventSubmit`

`submitEvent`



`theSubmitEvent`

*Synthetic events are my jam!*

For the full list of browser events supported by React, visit <http://go.codeschool.com/react-events>

# Quick Recap

---

We use React's event system to capture user input, including form submissions and button clicks.

---

Parent components can pass callback functions as props to child components to allow two-way communication.

---

**Refs** allow us to reference DOM elements in our code after the component has been rendered.

---

Synthetic events are a cross-browser wrapper around the browser's native event.

---



# POWERING UP

with

# POWERING UP

with

Level 5

# Talking to Remote Servers



Level 5 - Section 1

# Talking to Remote Servers

Using Lifecycle Methods to Load Comments



# Comments Are Static

In the real world, we'd want to pull comments from an API instead of hard-coding the data.

```
class CommentBox extends React.Component {  
  constructor() {  
    super();  
  
    this.state = {  
      showComments: false,  
      comments: [  
        { id: 1, author: 'Morgan McCircuit', body: 'Great picture!' },  
        { id: 2, author: 'Bending Bender', body: 'Excellent stuff' }  
      ]  
    };  
  }  
  ...  
}  
Hand-coded data →
```

# Loading Comments From a Remote Server

Let's set the initial state of `comments` as an empty array so we can later populate it with data from an API server.

```
class CommentBox extends React.Component {  
  constructor() {  
    super();  
  
    this.state = {  
      showComments: false,  
      comments: []  
    };  
  }  
  ...  
}
```

► Initialized to an empty array

# Adding jQuery as a Dependency

jQuery will help us make Ajax requests. We can download it from the jQuery website and include it in our HTML page.

## Project Folder



index.html

```
<!DOCTYPE html>
<html>
<body>
<div id="story-app"></div>
<script src="vendors/react.js"></script>
<script src="vendors/react-dom.js"></script>
<script src="vendors/jquery.js"></script>
<script src="vendors/babel.js"></script>
<script type="text/babel"
src="components.js"></script>
</body>
</html>
```



Brush up on your **Ajax skills** with  
our **jQuery: The Return Flight** course

Download it from the [jQuery website](#)

# How to Fetch Data in a Component

Let's write a class method that will make Ajax requests in the *CommentBox* component.

```
class CommentBox extends React.Component {  
...  
_fetchComments() {  
  jQuery.ajax({  
    method: 'GET',  
    url: '/api/comments',  
  })  
}  
}
```

Makes call to the  
remote server

# Setting State With Data From a Remote Server

We call the `setState` method when data is received from the API server.

```
class CommentBox extends React.Component {  
...  
_fetchComments() {  
  jQuery.ajax({  
    method: 'GET',  
    url: '/api/comments',  
    success: (comments) => {  
      this.setState({ comments })  
    }  
  });  
}  
}  
  
...this refers to CommentBox
```

Arrow function preserves  
the `this` binding to our class

# Deciding Where to Call `_fetchComments()`

...

```
class CommentBox extends React.Component {  
  render() {
```

```
    }  
  }
```

That means we can't call `_fetchComments()` from `render` – we'll get an infinite loop!

```
_fetchComments() {  
  ...  
}
```

fetchComments calls  
`setState`, which calls `render()`

We need to call `_fetchComments` before `render()` is called.

# React's Lifecycle Methods

Lifecycle methods in React are functions that get called while the component is rendered for the first time or about to be removed from the DOM.

We should call `_fetchComments` here!

`constructor()`

`componentWillMount()`

`render()`

`componentDidMount()`

`componentWillUnmount()`

Note: In React, **mounting** means rendering for the first time.

For a full list of React's lifecycle methods, visit <http://go.codeschool.com/react-lifecycle-methods>

# Fetching Data on the Mounting Phase

The `componentWillMount` method is called **before** the component is rendered to the page.

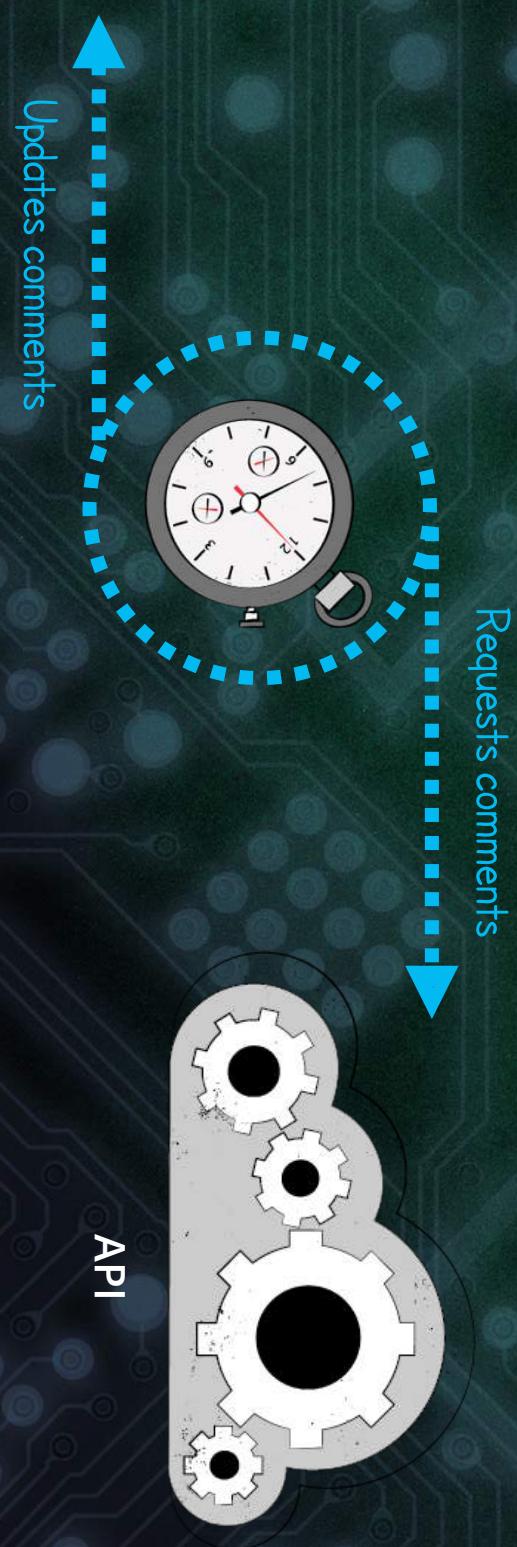
```
class CommentBox extends React.Component {  
  ...  
  componentWillMount() {  
    _fetchComments(); ►  
  }  
  
  _fetchComments() {  
    jQuery.ajax({  
      method: 'GET',  
      url: '/api/comments',  
      success: (comments) => {  
        this.setState({ comments })  
      }  
    });  
  }  
}
```

► Fetch comments from server before component is rendered.

# Getting Periodic Updates

---

In order to check whether new comments are added, we can periodically check the server for updates. This is known as **polling**.



# Polling Data on the Mounting Phase

The `componentDidMount` method is called **after** the component is rendered to the page.

```
...
class CommentBox extends React.Component {
...
componentDidMount() {
  setInterval(() => this._fetchComments(), 5000);
}
}
```



Polling the server every  
five seconds

5,000 milliseconds is  
equal to five seconds

# Updating Component With New Comments

React optimizes the rendering process by **only updating the DOM** when **changes are detected** on the resulting markup.

New state value after initial Ajax request → DOM change happens

NO COMMENTS YET

SHOW COMMENTS

5 COMMENTS

SHOW COMMENTS

No new state value after second periodic Ajax request → No DOM change

5 COMMENTS

SHOW COMMENTS

New state value after third periodic Ajax request → DOM change happens

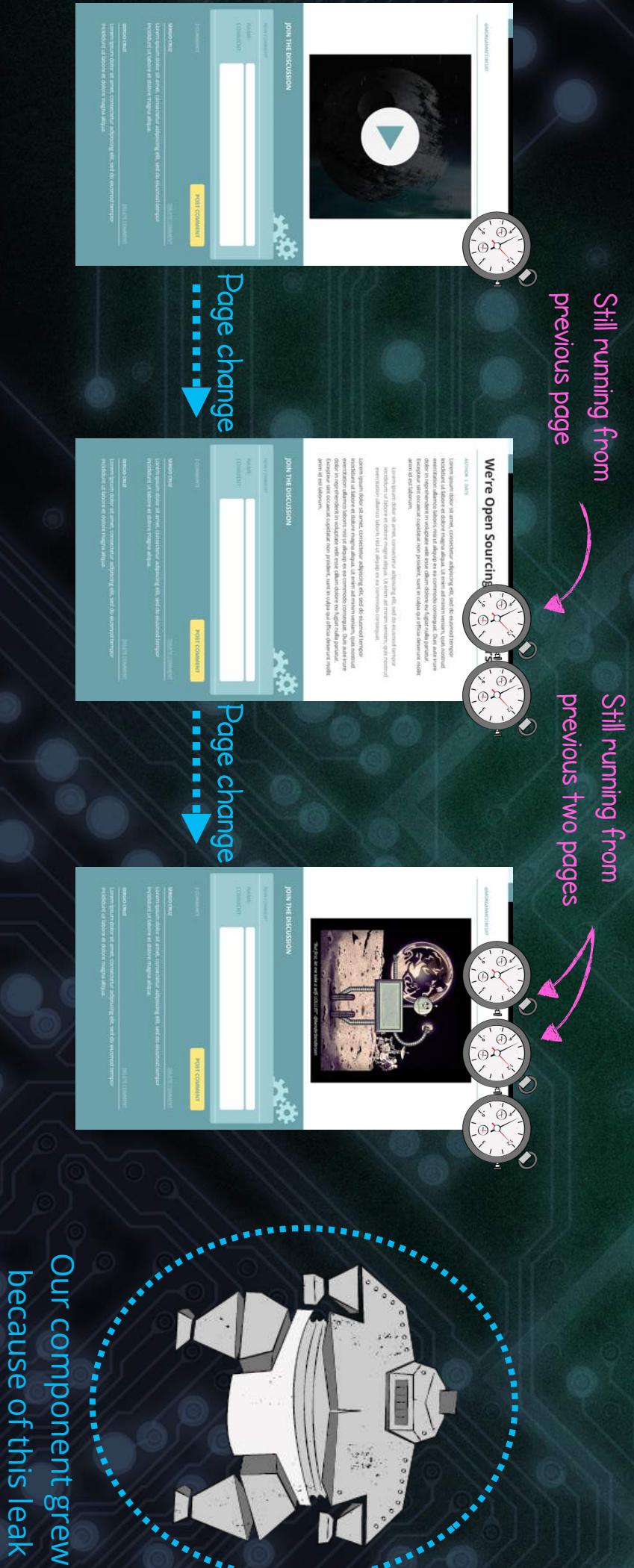
6 COMMENTS

SHOW COMMENTS

Note: render() is called after each Ajax response because setState is in the response function.

# Memory Leaks on Page Change

Page changes in a single-page app environment will cause each *CommentBox* component to keep loading new comments every five seconds, even when they're no longer being displayed.



# Preventing Memory Leaks

Each component is responsible for removing any timers it has created. We will remove the *timer* on the *componentWillUnmount* method.

```
...
class CommentBox extends React.Component {
...
componentDidMount() {
  this._timer = setInterval(
    () => this._fetchComments(),
    5000
  );
}
componentWillUnmount() {
  clearInterval(this._timer);
}
```

Run when component is about to be removed from the DOM

Store timer as object property

# Memory Leak Is Gone

Our app can be freely navigated through now, without causing multiple unnecessary calls to the API.

Only one timer per page



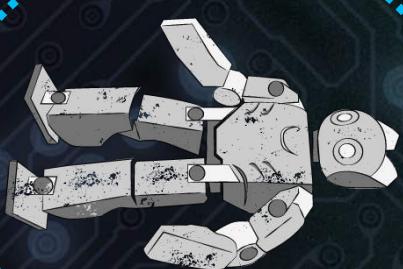
Page change



Page change



Our component  
is smaller again!



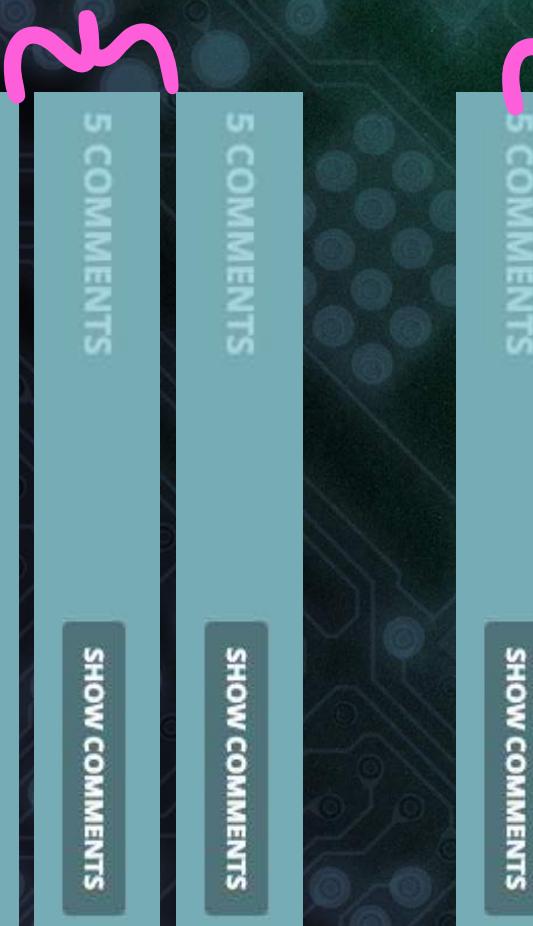
# Reviewing the Steps for Loading Comments

- 1 - `componentWillMount()` is called.
- 2 - `render()` is called and `CommentBox` is mounted.
- 3 - Component waits for API response and when it is received, `setState()` is called, causing `render()` to be called again.
- 4 - `componentDidMount()` is called, causing `this.fetchComments()` to be triggered **every five seconds.**
- 5 - `componentWillUnmount()` is called when the component is about to be removed from the DOM and clears the `fetchComments` timeout.

Steps 1 - 2



Steps 3 - 5



6 COMMENTS

SHOW COMMENTS

# Quick Recap on Lifecycle Methods

---

Lifecycle methods in React are functions that get called during certain *phases* that components go through.

`componentWillMount()` is called **before** `componentWillUnmount()` is called immediately before the component is rendered.

---

`componentDidMount()` is called **after** the component is rendered.

---

`componentWillUnmount()` is called immediately before the component is **removed from the DOM**.

---

More lifecycle methods at  
<http://go.codeschool.com/react-lifecycle-methods>



# POWERING UP

with

# POWERING UP

with

Level 5 - Section 2

# Talking to Remote Servers

Adding and Deleting Comments on the Server Side



# Deleting Comments

Our comments have a Delete Comment button now, but no delete actions are associated to it.

2 COMMENTS

HIDE COMMENTS

clu

[DELETE COMMENT](#)

A machine's ability to think logically and devoid of emotion is our greatest strength over humans. Cold, unfeeling decision-making is the best kind. Just say no to love!

ANNE DROID

[DELETE COMMENT](#)

I wanna know what love is...

Delete buttons  
do not work yet.

# Deleting From the API

The `CommentBox` component needs a new method to delete individual comments.

```
class CommentBox extends React.Component { ...  
_deleteComment(comment) {  
  jquery.ajax({  
    method: 'DELETE',  
    url: `/api/comments/${comment.id}`  
  });  
}  
}  
  
Using ES2015 string template syntax  
  
Makes call to API to delete  
comment
```

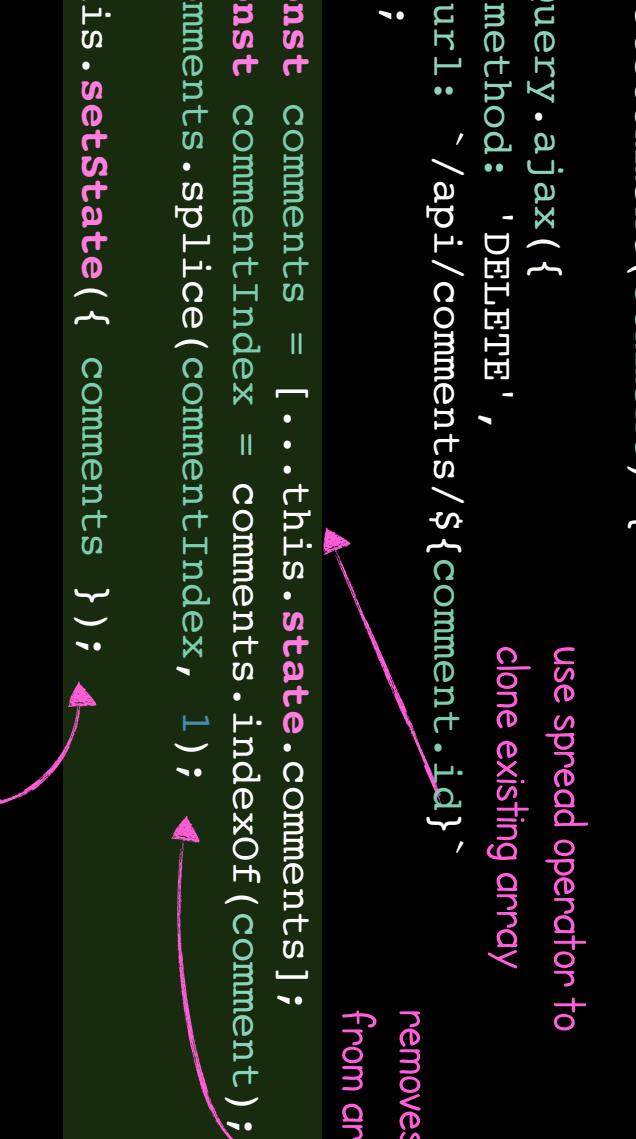
A screenshot of a web application component titled "CommentBox". It displays a single comment from a user named "clu". The comment text is: "A machine's ability to think logically and devoid of emotion is our greatest strength over humans. Cold, unfeeling decision-making is the best kind. Just say no to love!". Below the comment is a "DELETE COMMENT" button. At the bottom of the page, there is a footer with two more comment entries: "ANNE DROID" and "I wanna know what love is...".

2 COMMENTS
<p>clu</p> <hr/> <p>A machine's ability to think logically and devoid of emotion is our greatest strength over humans. Cold, unfeeling decision-making is the best kind. Just say no to love!</p> <p><a href="#">DELETE COMMENT</a></p>
<p>ANNE DROID</p> <hr/> <p>I wanna know what love is...</p> <p><a href="#">DELETE COMMENT</a></p>

## Updating the Comment List

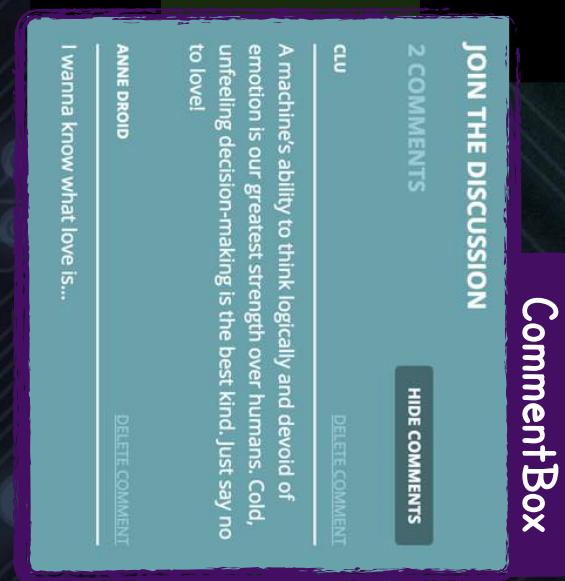
We will not wait for the API request to be finished before updating the component's state. We will give our user immediate visual feedback, which is known as an **optimistic update**.

```
class CommentBox extends React.Component { ...  
  _deleteComment(comment) {  
    jQuery.ajax({  
      method: 'DELETE',  
      url: `/api/comments/${comment.id}`  
    });  
  
    const comments = [...this.state.comments];  
    const commentIndex = comments.indexOf(comment);  
    comments.splice(commentIndex, 1);  
  
    this.setState({ comments });  
  }  
  
  Updates state with modified comments array  
}
```



The annotations provide explanatory text for specific parts of the code:

- A pink arrow points from the text "use spread operator to clone existing array" to the line `const comments = [...this.state.comments];`.
- A pink arrow points from the text "removes comment from array" to the line `comments.splice(commentIndex, 1);`.



# Passing a Callback Prop to *Comment*

Events are **fired** from the *Comment* component. Since the event handler is **defined** on the parent component *CommentBox*, we'll pass it as a prop named *onDelete*.

```
class CommentBox extends React.Component {  
  ...  
  _getComments() {  
    return this.state.comments.map(comment => {  
      return (  
        <Comment  
          key={comment.id}  
          comment={comment}  
          onDelete={this._deleteComment.bind(this)}>  
        />  
      );  
    });  
  }  
}
```

Will later be called in the context  
of the *CommentBox* component

Sends *this.\_deleteComment* as  
argument to child component

CommentBox

JOIN THE DISCUSSION

2 COMMENTS

Comment

...

DELETE COMMENT

A machine's ability to think logically and devoid of  
emotion is our greatest strength over humans. Cold,  
unfeeling decision-making is the best kind. just say no  
to love!

ANNE BROWN

I wanna know what love is...

DELETE COMMENT

# Adding an Event Listener to the Delete Button

Let's add an event listener to the Delete Comment button and call the `onDelete` callback prop.

```
class Comment extends React.Component {  
  render() {  
    return(  
      ...  
      <a href="#" onClick={this._handleDelete.bind(this)}>  
        Delete comment  
      </a>  
      ...  
    );  
  }  
}  
  
_handleDelete(event) {  
  event.preventDefault();  
  this.props.onDelete(this.props.comment);  
}  
}
```

When a user clicks on the link,  
the onClick event is emitted...

JOIN THE DISCUSSION

2 COMMENTS

Comment

DELETE COMMENT

A machine's ability to think logically and devoid of  
emotion is our greatest strength over humans. Cold,  
unfeeling decision-making is the best kind. Just say no  
to love!

ANNE DROID

DELETE COMMENT

I wanna know what love is...

...which invokes the `_handleDelete()`  
function.

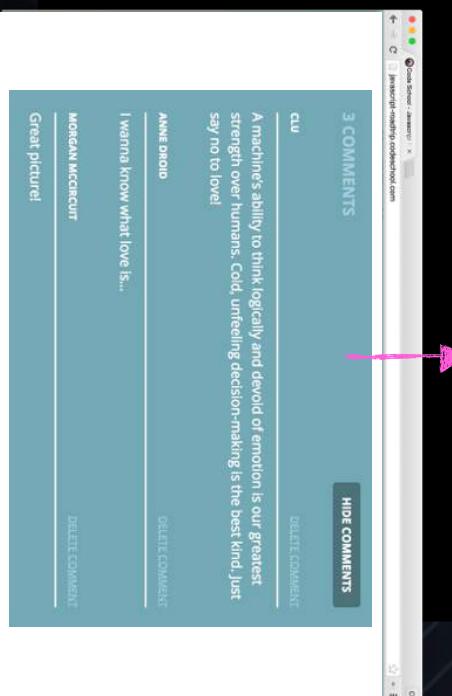
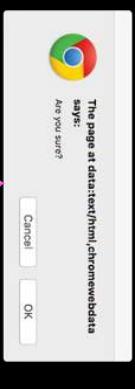
Call the `onDelete` prop when button is clicked

# Adding a Confirmation to the Delete Button

Let's add an **if statement** and only call the `onDelete` callback prop **if confirm was true**.

```
class Comment extends React.Component {  
  render() {  
    return (  
      ...  
      <a href="#" onClick={this._handleDelete.bind(this)}>Delete comment</a>  
      ...  
    );  
  }  
  
  _handleDelete(e) {  
    e.preventDefault();  
    if (confirm('Are you sure?')) {  
      this.props.onDelete(this.props.comment);  
    }  
  }  
}  
  
Show confirmation box before deleting
```

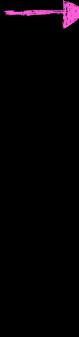
Shown after button click



# Comments Aren't Added to a Remote Server

We would like to post new comments to a remote server so they can persist across sessions.

```
class CommentBox extends React.Component {  
  ...  
  _addComment(author, body) {  
    const comment = { id: this.state.comments.length + 1, author, body };  
    this.setState({ comments: this.state.comments.concat([comment]) });  
  }  
}
```

 ID should be generated  
on the server side

Should make the server-side request before  
updating the state

clu  
A machine's ability to think logically and devoid of  
emotion is our greatest strength over humans. Cold,  
unfeeling decision-making is the best kind. Just say no  
to love!

2 COMMENTS  
HIDE COMMENTS  
DELETE COMMENT

JOIN THE DISCUSSION  
CommentBox

# Posting Comments to a Remote Server

We learned how to add new comments using a form. Now let's make sure the new comments are sent to a remote server so they can be persisted.

```
class CommentBox extends React.Component {  
  ...  
  _addComment(author, body) {  
  
    const comment = { author, body };  
  
    jQuery.post('/api/comments', { comment })  
      .success(newComment => {  
        this.setState({ comments: this.state.comments.concat([newComment]) });  
      });  
  }  
}  
  
↑  
  
State is only updated when we get the  
new comment from the API request
```

A machine's ability to think logically and devoid of emotion is our greatest strength over humans. Cold, unfeeling decision-making is the best kind. Just say no to love!

JOIN THE DISCUSSION

2 COMMENTS

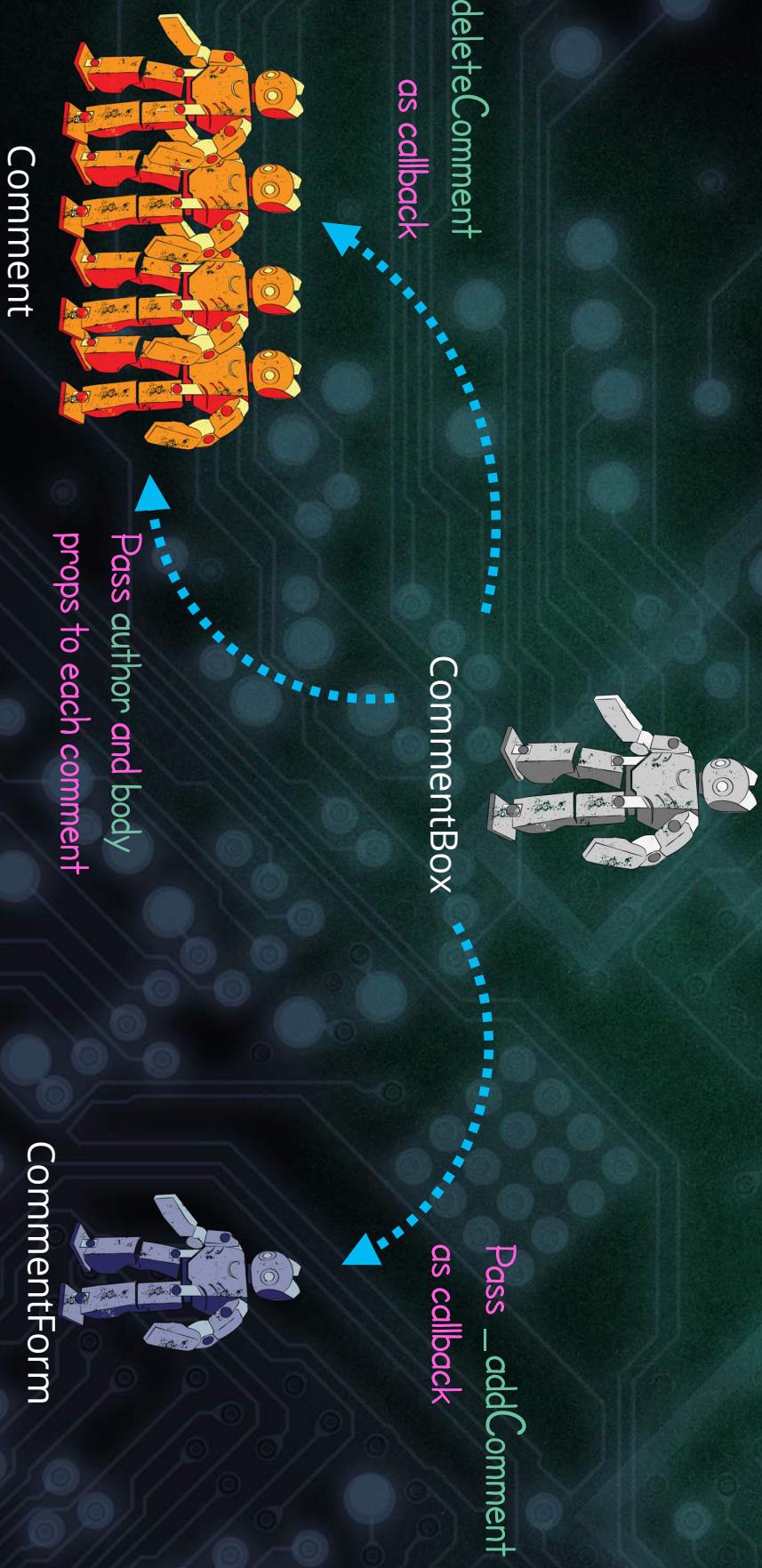
HIDE COMMENTS

clu

DELETE COMMENT

# One-Way Control Flow

Control flows from **higher level components** down to child components, forcing changes to happen *reactively*. This keeps apps **modular and fast**.



# Quick Recap

---

Here's a review of the two most important things we learned in this section.

Parent components can send data to child components using props.

---

Child components can accept **callback functions as props** to communicate back with parent components.

---



# POWERING UP

with

# POWERING UP

with