

## Web Front-End Development

### Week 10: React Interactivity

Today we'll finally add some interactivity.

Components get data through props that are passed in, but these can never be changed. In order to store data in components that can be changed React uses state.

#### State

<https://reactjs.org/docs/state-and-lifecycle.html>

Props are used for data that doesn't change, state is used for data that does change.

State is how we store dynamic data, data that can change.

State has some similarities to props, but it is private and fully controlled by the component.

State is not passed into a component. State is defined in a component and that component is the only place that state can be changed. This makes the component the "single source of truth" for the data.

State is not accessible to any component other than the one that owns and sets it, which makes it encapsulated.

Functions create stateless components so if a component has state it must be a class that extends `React.Component`.

`function Character(props) {}`

becomes

`class Character extends React.Component {}`

Add a single empty method to the class called `render()`.

Move the body of the function into the `render()` method.

Replace `props` with `this.props` in the `render()` body.

In classes we need to use "this" to access props or state in the current class.

Example:

<https://repl.it/@aileenjp/react-favorites-state>

I've converted my Character component to a class called Favorite.

```
class Favorite extends React.Component{
  render() {
    var nameStyle = {
      padding: 10,
      margin: 20,
      marginLeft: "33%",
      backgroundColor: this.props.bgcolor,
      color: "#333",
      align: "center",
      width: "33%",
      fontFamily: "monospace",
      fontSize: 28,
      textAlign: "left"
    };
  }
}
```

```

        return (
          <div style={nameStyle}>{this.props.name}</div>
        );
      }
    }
  }
}

```

## Adding State

State is defined as an object with key/value pairs. Each key represents a piece of dynamic data you want to store and the value is its initial value.

You set up state in the component's constructor method after calling its super(props) method.

```

constructor(props) {
  super(props);
  this.state = {
    mood: 'great',
    hungry: false
}

```

To access a component's state property using the expression {this.state.*propertyname*}  
 {this.state.mood}

A component can do more than just read its own state. A component can also *change* its own state by calling the function `setState()`.

`setState()` takes as a parameter an *object* that will update the component's state

```
this.setState({ hungry: true });
```

The mood part of the state remains unaffected.

To change the state values you must call `setState()` you can't update the state directly.

`setState()` takes an object, and *merges* that object with the component's current state. It then tells React that the state has changed. React merges in the state update and then calls the `render()` method again to update the DOM. If there are properties in the current state that aren't part of that object, then those properties remain unchanged.

React may batch multiple `setState()` calls into a single update for performance so the state may be updated asynchronously, you should not rely on their values for calculating the next state.

A component may choose to pass its state down to its child components as props but the child components won't know or care where it came from, and since they're props, they won't be changed.

This is commonly called a "top-down" or "unidirectional" data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components "below" them in the tree.

The most common way to call `setState()` is to call it in a function, which we'll look at in a bit.

Example:

<https://repl.it/@aileenjp/react-favorites-state>

- Add a class constructor to Favorite
  - Constructor methods always take in props
  - Constructor methods must always first call the base super(props) constructor
- Add state to Favorite
  - The constructor() method defines state to include the number of votes
- Add a method to increase the number of votes by calling setState()

Now let's look at events so we can figure out how to call this method and update state.

## Events

<https://reactjs.org/docs/handling-events.html>

React's approach to handling events is aimed at improved performance and browser compatibility.

Instead of using addEventListener() to handle events, React handles events within its components by assigning event listeners to DOM elements.

Syntax differences:

- React events are named using camelCase, rather than lowercase.
  - onChange rather than onchange

## Event Handlers

Functions are often used as event handlers. The function is often a class method but can also be passed from another component as a prop. Today let's focus on using a class method.

```
handleClick() { }
```

With JSX you assign a function as the event handler, rather than a string.

- onClick={handleClick} rather than onchange="handleClick()"
- Naming convention suggests that we use "handle" and the event type as the name of our event handler

```
<button onClick={this.handleClick} />
```

Names like onClick only create event listeners if they're used on HTML-like JSX elements. Otherwise, they're just ordinary prop names.

In JavaScript, class methods are not bound by default so in JSX callbacks "this" does not remain in context. bind() wraps the function in a new bound function and therefore "this" remains in context. In React, whenever you define an event handler that uses this, you need to bind that method in the constructor.

```
this.methodName = this.methodName.bind(this)
```

Therefore we have to bind this.handleClick() to this in the constructor method.  
`this.handleClick = this.handleClick.bind(this)`

This is not React-specific behavior; it is a part of [how functions work in JavaScript](#). Generally, if you refer to a method without () after it, such as `onClick={this.handleClick}`, you should bind that method.

If you forget to bind `this.handleClick` and pass it to `onClick`, `this` will be `undefined` when the function is actually called.

In React the event is also passed to functions that are event handlers. This is handled differently based on the syntax you use.

- Arrow function
  - `<button onClick={(e) => this.handleClick(id, e)}>Delete Row</button>`
  - must explicitly pass the event(e)
  - No need to bind
- Function binding
  - `<button onClick={this.handleClick}>Delete Row</button>`
  - automatically passes the event
  - In the constructor you need to bind this method
  - `this.handleClick = this.handleClick.bind(this);`

React events follow the W3C spec so you don't have to worry about cross-browser compatibility, but they don't work exactly the same as native events.

Another difference is that you cannot return `false` to prevent default behavior in React. You must call `preventDefault` explicitly.

- `e.preventDefault()` rather than `return false`

More information on handling for elements can be found in the React docs

<https://reactjs.org/docs/forms.html>

```
constructor(props) {  
  super(props);  
  this.state = { mood: 'good' };  
  this.handleChange = this.handleChange.bind(this);  
}  
  
handleChange(e) {  
  const newMood = e.target.value;  
  this.setState({ mood: newMood });  
}  
  
render() {  
  return  
  <div>  
    <h1>I'm feeling {this.state.mood}</h1>  
    <select onChange={this.handleChange} />  
  </div>  
}
```

```
</div>
}
```

Here is how the component's state would be set:

1. A user triggers an `onChange` event by selecting a value in the select list.
2. The `onChange` event from is being listened for on the select element.
3. When the event fires it calls the *event handler* function assigned to it -- `this.handleChange`
4. `handleChange(e)` takes an *event object* as an argument. Using the event object you can access the target's value. The target is the element that the event was fired on.
5. That value is saved in the constant `newMood`
6. `this.setState()` is sent the object `{mood: newMood}` to change the value of `mood` to the value in `newMood` and the component's state is changed.

Any time that you call `this.setState()`, `this.setState()` AUTOMATICALLY calls `.render()` as soon as the state has changed.

Think of `this.setState()` as actually being two things: `this.setState()`, immediately followed by `.render()`.

*That* is why you can't call `this.setState()` from inside of the `.render()` method. `this.setState()` automatically calls `.render()`. If `.render()` calls `this.setState()`, then an infinite loop is created.

Example:

<https://repl.it/@aileenjp/react-favorites-state>

- `render()` now returns `votes(state)`, `name(props)`, and a button
- The button uses the `onClick` event to call the `vote` function to increase the number of votes
  - To follow naming conventions we'll change the method name from `vote` to `handleClick`
  - `{this.handleClick}` is an event handler and must be bound in the constructor
  - Because `handleClick()` is bound in the constructor the event (`e`) is automatically passed into the function
- Added styling to the Favorite component
  - Note that the styling objects must be in `render()` but before `return()`
  - Apply `itemStyle` to the main `div`
  - Apply `nameStyle` to the `name`
  - Apply `buttonStyle` to the button
- Added styling in `App.css` and updated the `App` component