React State I

Goals

- Understand the concept of state in web applications
- Learn how to model state in React
- Use events to trigger state changes

What is State?

Thinking About State

In any sufficiently advanced web application, the user interface has to be stateful.

- logged-in users see a different screen than logged-out users
- clicking “edit profile” opens up a modal (pop-up) window
- sections of a website can expand or collapse, for instance clicking “read more”

The state of the client interface (frontend) is not always directly tied to state on the server.

Why would the server need to know if a modal is open?

State Changes

State is designed to constantly change in response to events.

A great way to think about state is to think of games, for instance chess. At any point in time, the board is in a complex state.

 <_images/chess.gif>

Every new move represents a single discrete state change.

What Does State Track?

There are two types of things state on the client/frontend keeps track of:

- ***UI logic*** - the changing state of the interface e.g., there is a modal open right now because I'm editing my profile
- ***business logic*** - the changing state of data e.g., in my inbox, messages are either read or unread, and this in turn affects how they display.

Vanilla / jQuery State

The way we kept track of state with jQuery was by selecting DOM elements and seeing if they were displayed/hidden, or if they had certain styles or attributes.

```
// getting a text input value
var firstName = $('#firstNameInput').val();

// setting a text input value
$('#firstNameInput').val('Michael');
```

In other words, we were inferring the state of the application from the DOM itself.

React is going to do the opposite!

React State

Core React Concept Review

- component
  - building block of React
  - combines logic (JS) and presentation (JSX)
- prop
  - data passed to a component *(or found via defaults)*
  - immutable; component cannot change its own props
- state
  - internal data specific to a component
  - data that changes over time!

What is React State?

In React, state is an instance attribute on a component.

It's always an object (POJO), since you'll want to keep track of several keys/values.

```
// what is the current state of my component?
console.log(this.state);
```

```
{
  playerName: "Whiskey",
  score: 100
}
```

Initial State

State should be initialized as soon as the component is created.

So we set it in the constructor function:

```
class ClickCount extends Component {
  constructor(props) {
    super(props);
    this.state = {
      numClicks: 0  // start at zero clicks
    };
  }
}
```

React Constructor Function

If your component is stateless, you can omit the constructor function.

If you are building a component with state, you need a standard React constructor

```
constructor(props) {
  super(props);
  this.state = {
    /* values we want to track */
  };
}
```

- ***constructor*** takes one argument, ***props***
- You must call `super(props)` at start of constructor, which "registers" your class as a React ***Component***
- Inside the instance methods, you can refer to `this.state` just like you did `this.props`

> **Note: (Advanced) ESNext Class Fields Syntax**
>
> State can be defined using the new public fields syntax <https://github.com/tc39/proposal-class-fields>, which is going to be added to a future version of JavaScript (ES2019 or ES2020):
>
> ```
> class ClickCount extends Component {
>   state = { numClicks: 0 };
>
>   // ...
> }
> ```
>
> Notice that it is ***state***, not ***this.state***, when done like this.
>
> You may need to update your linter if you choose to use this syntax:

```
npm install --global babel-eslint  # global add-on for eslint
```

~/.eslintrc.json

```
// add "parser": "babel-eslint" to your linter config:
{
  // ...other settings
  "parser": "babel-eslint",
  // ...other settings
}
```

Example

demo/basicExample.js

```
class Game extends Component {
  constructor(props) {
    super(props);
    this.state = {
      player: 'Whiskey',
      score: 0
    };
  }

  render() {
    return (
      <div>
        <h1>Battleship</h1>
        <p>Current Player: {this.state.player}</p>
        <p>Score: {this.state.score}</p>
      </div>
    );
  }
} // end
```

Changing State

this.setState() is the built-in React method of changing a component's state.

```
this.setState({ playerName: "Matt", score: 0 })
```

- Can call in any instance method except the constructor
- Takes an object describing the state changes
- Patches state object — keys that you didn't specify don't change
- Asynchronous!
  - The component state will *eventually* update.
  - React controls when the state will actually change, for performance reasons.
- Components re-render when their state changes

demo/click-me/src/Rando.js

```
class Rando extends Component {
  constructor(props) {
    super(props);
    this.state = { num: 0 };
    this.makeTimer();
  }

  makeTimer() {
    setInterval(() => {
      this.setState({
        num: Math.floor(Math.random() * this.props.maxNum)
      });
    }, 1000);
  }

  render() {
    return <button>Rando: {this.state.num}</button>;
  }
} // end
```

React Events

State most commonly changes in direct response to some event.

In React, every JSX element has built-in attributes representing every kind of browser event.

They are camel-cased, like ***onClick***, and take callback functions as event listeners.

```
<button onClick={function(e) { alert('You clicked me!'); } }>
  Click Me!
</button>
```

> **Note: (Advanced) React Events**
>
> React Events are a bit of an abstraction on top of regular browser events.
>
> They're called synthetic events <https://reactjs.org/docs/events.html#overview>, but in practice they behave the same and you don't have to worry about the abstraction.
>
> Check out the React documentation for all types of supported events <https://reactjs.org/docs/events.html#supported-events>.

Broken Click

If we're updating state in response to an event, we'll have to call a method with `this.setState()`:

demo/click-me/src/BrokenClick.js

```
class BrokenClick extends Component {
  constructor(props) {
    super(props);
    this.state = { clicked: false };
  }

  handleClick() {
```

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

  render() {
    return (
      <div>
        <h1>The Button is
          {this.state.clicked ? 'clicked' : 'not clicked'}
        </h1>
        <button onClick={this.handleClick}>Broken</button>
      </div>
    );
  }
} // end
```

`this` is back

But ***this*** is undefined!

- *Who* is calling ***handleClick*** for us?
  - React is, on click
- *What* is it calling it on?
  - 🤷 it doesn't remember to call it on our instance
  - The method was called "out of context"
- *What* do we do?
  - `.bind()` it!

Fixed Click

We'll fix the situation by binding our instance methods in the constructor.

demo/click-me/src/FixedClick.js

```
class FixedClick extends Component {
  constructor(props) {
    super(props);
    this.state = { clicked: false };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({ clicked: true });
  }

  render() {
    return (
      <div>
        <h1>The Button is
          {this.state.clicked ? 'clicked' : 'not clicked'}
        </h1>
        <button onClick={this.handleClick}>Fixed</button>
      </div>
    );
  }
} // end
```

Full Example: Click Rando

demo/click-me/src/ClickRando.js

```
class ClickRando extends Component {
  constructor(props) {
    super(props);
    this.state = { num: 0 };
    this.handleClick = this.handleClick.bind(this);
  }

  setRandom() {
    this.setState({
      num: Math.floor(Math.random() * this.props.maxNum)
    });
  }

  handleClick(evt) {
    this.setRandom();
  }

  render() {
    return (
        <button onClick={this.handleClick}>
          Click Rando: {this.state.num}
        </button>
    );
  }
} // end
```

State vs. Props

State *vs* Props

State and ***Props*** are the most important concepts in React (after knowing what a “component” is).

term	structure	mutable	purpose
state	POJO `{}`	yes	stores changing component data
props	POJO `{}`	no	stores component configuration

State *as* Props

A common pattern we will see over and over again is a stateful (“smart”) parent component passing down its state values as props to stateless (“dumb”) child components.

```
class CounterParent extends Component {
  constructor(props) {
    super(props);
    this.state = {count: 5};
  }
  render() {
    // passing down parent state as a prop to the child
    return (
```

```
      <div>
        <CounterChild count={this.state.count} />
      </div>
    );
  }
}
```

This idea is generalized in React as ***"downward data flow"***. It means that components get simpler as you go down the component hierarchy, and parents tend to be more stateful than their children.

> **Note: Further Reading**
>
> Check out the React docs <https://reactjs.org/docs/faq-state.html>.
>
> They link to two great resources to help further your understanding:
>
> - Props vs State <https://github.com/uberVU/react-guide/blob/master/props-vs-state.md>.
> - ReactJS: Props vs. State <http://lucybain.com/blog/2016/react-state-vs-pros/>.