Chapter 4: State in React

Section 4.1: Basic State

State in React components is essential to manage and communicate data in your application. It is represented as a JavaScript object and has *component level* scope, it can be thought of as the private data of your component.

In the example below we are defining some initial state in the `constructor` function of our component and make use of it in the `render` function.

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

    // Set-up our initial state
    this.state = {
      greeting: 'Hiya Buddy!'
    };
  }

  render() {
    // We can access the greeting property through this.state
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

Section 4.2: Common Antipattern

You should not save `props` into `state`. It is considered an anti-pattern. For example:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            url: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            url: this.props.url + '/days=?' + e.target.value
        });
    }

    componentWillMount() {
        this.setState({url: this.props.url});
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />
```

```
                URL: {this.state.url}
            </div>
        )
    }
}
```

The prop `url` is saved on `state` and then modified. Instead, choose to save the changes to a state, and then build the full path using both `state` and `props`:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            days: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            days: e.target.value
        });
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.props.url + '/days?=' + this.state.days}
            </div>
        )
    }
}
```

This is because in a React application we want to have a single source of truth - i.e. all data is the responsibility of one single component, and only one component. It is the responsibility of this component to store the data within its state, and distribute the data to other components via props.

In the first example, both the MyComponent class and its parent are maintaining 'url' within their state. If we update state.url in MyComponent, these changes are not reflected in the parent. We have lost our single source of truth, and it becomes increasingly difficult to track the flow of data through our application. Contrast this with the second example - url is only maintained in the state of the parent component, and utilised as a prop in MyComponent - we therefore maintain a single source of truth.

Section 4.3: setState()

The primary way that you make UI updates to your React applications is through a call to the `setState()` function. This function will perform a *shallow merge* between the new state that you provide and the previous state, and will trigger a re-render of your component and all decedents.

Parameters

1. `updater`: It can be an object with a number of key-value pairs that should be merged into the state or a function that returns such an object.

2. `callback (optional)`: a function which will be executed after `setState()` has been executed successfully. Due to the fact that calls to `setState()` are not guaranteed by React to be atomic, this can sometimes be useful if you want to perform some action after you are positive that `setState()` has been executed successfully.

Usage:

The `setState` method accepts an `updater` argument that can either be an object with a number of key-value-pairs that should be merged into the state, or a function that returns such an object computed from `prevState` and `props`.

Using `setState()` with an Object as `updater`

```
//
// An example ES6 style component, updating the state on a simple button click.
// Also demonstrates where the state can be set directly and where setState should be used.
//
class Greeting extends React.Component {
    constructor(props) {
        super(props);
        this.click = this.click.bind(this);
        // Set initial state (ONLY ALLOWED IN CONSTRUCTOR)
        this.state = {
            greeting: 'Hello!'
        };
    }
    click(e) {
        this.setState({
              greeting: 'Hello World!'
        });
    }
    render() {
        return(
            <div>
                <p>{this.state.greeting}</p>
                <button onClick={this.click}>Click me</button>
            </div>
        );
    }

}
```

Using `setState()` with a Function as `updater`

```
//
// This is most often used when you want to check or make use
// of previous state before updating any values.
//

this.setState(function(previousState, currentProps) {
  return {
    counter: previousState.counter + 1
  };
});
```

This can be safer than using an object argument where multiple calls to `setState()` are used, as multiple calls may be batched together by React and executed at once, and is the preferred approach when using current props to set state.

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

```
this.setState({ counter: this.state.counter + 1 });
```

These calls may be batched together by React using `Object.assign()`, resulting in the counter being incremented by 1 rather than 3.

The functional approach can also be used to move state setting logic outside of components. This allows for isolation and re-use of state logic.

```
// Outside of component class, potentially in another file/module

function incrementCounter(previousState, currentProps) {
    return {
        counter: previousState.counter + 1
    };
}

// Within component

this.setState(incrementCounter);
```

Calling `setState()` with an Object and a callback function

```
//
// 'Hi There' will be logged to the console after setState completes
//

this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

Section 4.4: State, Events And Managed Controls

Here's an example of a React component with a "managed" input field. Whenever the value of the input field changes, an event handler is called which updates the state of the component with the new value of the input field. The call to `setState` in the event handler will trigger a call to `render` updating the component in the dom.

```
import React from 'react';
import {render} from 'react-dom';


class ManagedControlDemo extends React.Component {

  constructor(props){
    super(props);
    this.state = {message: ""};
  }

  handleChange(e){
    this.setState({message: e.target.value});
  }

  render() {
    return (
      <div>
        <legend>Type something here</legend>
          <input
            onChange={this.handleChange.bind(this)}
            value={this.state.message}
            autoFocus />
        <h1>{this.state.message}</h1>
      </div>
   );
```

```
  }
}


render(<ManagedControlDemo/>, document.querySelector('#app'));
```

Its very important to note the runtime behavior. Every time a user changes the value in the input field

- `handleChange` will be called and so
- `setState` will be called and so
- `render` will be called

Pop quiz, after you type a character in the input field, which DOM elements change

1. all of these - the top level div, legend, input, h1
2. only the input and h1
3. nothing
4. whats a DOM?

You can experiment with this more here to find the answer