React State Patterns

Goals

- Learn how to update state based off of existing state
- Properly manage state updates for mutable data structures
- Discuss best practices for modeling state and designing components

Setting State Using State

We've established that ***setState()*** is asynchronous…

So: it's risky to assume previous call has finished when you call it. Also, React will sometimes batch (squash together) calls to ***setState*** together into one for performance reasons.

If a call to ***setState()*** depends on current state, the safest thing is to use the alternate "callback form".

setState Callback Form

`this.setState(callback)`

Instead of passing an object, pass it a callback with the current state as a parameter.

The callback should return an object representing the new state.

```
this.setState(curState => ({ count: curState.count + 1 }));
```

Abstracting State Updates

The fact that you can pass a function to `this.setState` lends itself nicely to a more advanced pattern called ***functional setState***.

Basically you can describe your state updates abstractly as separate functions. But why would you do this?

```
// elsewhere in the code
function incrementCounter(prevState) {
  return { count: prevState.count + 1 };
}
// somewhere in the component
this.setState(incrementCounter);
```

Because testing your state changes is as simple as testing a plain function:

```
expect(incrementCounter({ count: 0 })).toEqual({ count: 1 });
```

This pattern also comes up all the time in Redux!

> **Note: (Advanced) Functional setState**
>
> Here is a nice opinionated article on the subject of using functional setState:
> https://medium.freecodecamp.org/functional-setstate-is-the-future-of-react-374f30401b6b
> <https://medium.freecodecamp.org/functional-setstate-is-the-future-of-react-374f30401b6b>

Mutable Data Structures in State

Mutable Data Structures

Until now, we've been setting state to primitives: mainly numbers and strings.

But component state also commonly includes objects, arrays, and arrays of objects.

```
this.state = {
  // store an array of todo objects
  todos: [
    { task: 'do the dishes', done: false, id: 1 },
    { task: 'vacuum the floor', done: true, id: 2 }
  ]
};
```

You have to be extra careful modifying your array of objects!

```
completeTodo(id) {
  const theTodo = this.state.todos.find(t => t.id === id);
  theTodo.done = true; // NOOOOO

  this.setState({
    todos: this.state.todos // bad
  });
}
```

Why? It's a long story…

Mutating nested data structures in your state can cause problems w/ React. (A lot of the time it'll be fine, but that doesn't matter. Just don't do it!)

Immutable State Updates

A much better way is to make a new copy of the data structure in question. We can use any ***pure function*** to do this…

```
completeTodo(id) {

  // Array.prototype.map returns a new array
  const newTodos = this.state.todos.map(todo => {
```

```
    if (todo.id === id) {
      // make a copy of the todo object with done -> true
      return { ...todo, done: true };
    }
    return todo;  // old todos can pass through
  });

  this.setState({
    todos: newTodos // setState to the new array
  });
}
```

Pure functions such as ***.map***, ***.filter***, and ***.reduce*** are your friends. So is the ***...spread operator***.

There is a slight efficiency cost due to the O(N) space/time required to make a copy, but it's almost always worth it to ensure that your app doesn't have extremely difficult to detect bugs due to mischevious side effects.

Immutable State Summary

- While it sounds like an oxymoron, immutable state just means that there is an old state object and a new state object that are both snapshots in time.
- The safest way to update state is to make a copy of it, and then call ***this.setState*** with the new copy.
- This pattern is a *good habit* to get into for React apps and *required* for using Redux.

Designing State

Designing the state of a React application (or any modern web app) is a challenging skill! It takes practice and time!

However, there are some easy best-practices that we can talk about in this section to give you a jump-start.

Minimize Your State

In React, you want to try to put as little data in state as possible.

Litmus test

- does *x* change? If not, *x* should not be part of state. It should be a prop.
- is *x* already captured by some other value *y* in state or props? Derive it from there instead.

Bad Example of State Design

Let's pretend we're modelling a Person…

```
this.state = {
  firstName: 'Matt',
  lastName: 'Lane',
  birthday: '1955-01-08T07:37:59.711Z',
  age: 64,
```

```
  mood: 'irate'
};
```

- Does Matt's first name or last name ever change? Not often I hope…
- Does Matt's birthday ever change? How is that even possible!
- Matt's ***age*** *does change*, however if we had `this.props.birthday` we could easily derive it from that.
- Therefore, the only property here that is truly stateful is arguably ***mood*** (although Matt might dispute this 😉).

Fixed Example of State Design

```
console.log(this.props);
{
  firstName: 'Matt',
  lastName: 'Lane',
  birthday: '1955-01-08T07:37:59.711Z',
  age: 64
}

console.log(this.state);
{
  mood: 'insane'
}
```

State Should Live On the Parent

As previously mentioned, we want to support the "downward data flow" philosophy of React.

In general, it makes more sense for a parent component to manage state and have a bunch of "dumb" stateless child display components.

This makes debugging easier, because the state is centralized. It's easier to predict where to find state:

Is the current component stateless? Find out what is rendering it. There's the state.

Todo Example:

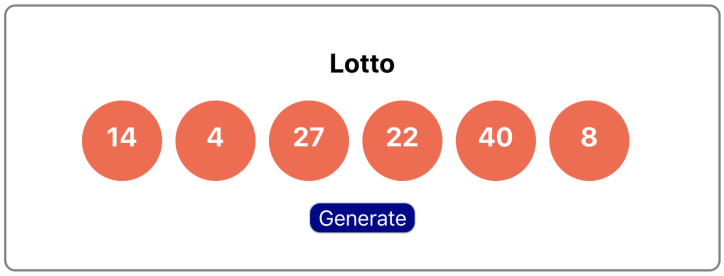
```
class TodoList extends Component {
  constructor(props) {
    super(props);
    this.state = {
      todos: [
        { task: 'do the dishes', done: false, id: 1 },
        { task: 'vacuum the floor', done: true, id: 2 }
      ]
    };
  }
  /* ... lots of other methods ... */
  render() {
    return (
      <ul>
        {this.state.todos.map(t => <Todo {...t} />)}
      </ul>
    );
```

```
  }
}
```

TodoList is a smart parent with lots of methods, while the individual ***Todo*** items are just `<li>` tags with some text and styling.

Example Design: Lottery

Let's Design an App!



<_images/lottery.png>

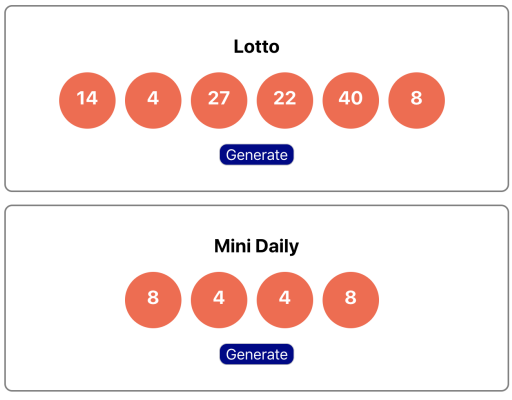
in App.js

```
<Lottery />
```

Should show 6 balls

Value 1-40 generated when button clicked

Should Be Reusable, Flexible



<_images/lottery-many.png>

in App.js

```
<div>
  <Lottery />
  <Lottery title="Mini Daily" numBalls={4} maxNum={10} />
</div>
```

Should be able to control title, num balls to show, and max value

Design

- What components will we need?
- What props will they need?
- What state will we need?

Lottery Component

- Props
  - ***title***: title of the lottery
  - ***numBalls***: num of balls to display
  - ***maxNum***: max value of ball
- State
  - ***nums***: array of `[num, num, num, ...]` for balls
- Events
  - ***onClick***: regenerate nums in state

LotteryBall Component

- Props
  - ***num***: value on this ball
- State
  - none!
- Events
  - none!