React State Patterns

Springboard

Download Demo Code

Goals

- Pass event handlers down as props to child components
- Understand the ***key*** prop that React asks for when mapping over data
- Use the callback pattern for ***useState*** to ensure state changes happen as expected
- Learn about storing mutable items in state & updating them

Passing functions to child components

How data flows

A common scenario in React:

- A parent component defines a function
- The function is passed as a prop to a child component
- The child component invokes the prop
- The parent function is called, usually setting new state
- The parent component is re-rendered along with its children

What it looks like

demo/numbers-app/src/NumberList.js

```
function NumberList() {
  const [nums, setNums] = useState(
    [1, 2, 3, 4]
  );

  // Remove num. Passed to/called by child
  function removeNum(num) {
    setNums(nums.filter(n => n !== num));
  }

  const numsList = nums.map(n => (
    <NumberItem
      value={n}
      remove={removeNum}
    />)
  );

  return <ul>{numsList}</ul>;
}
```

demo/numbers-app/src/NumberItem.js

```
function NumberItem(props) {

  /** Delete num via parent fn */
  function handleRemove(evt) {
    props.remove(props.value);
  }

  return (
    <li>
      {props.value}
      <button onClick={handleRemove}>
        X
      </button>
    </li>
  );
}
```

Lists and Keys

demo/numbers-app/src/NumberList.js

```
function NumberList() {
  const [nums, setNums] = useState(
    [1, 2, 3, 4]
  );

  // Remove num. Passed to/called by child
  function removeNum(num) {
    setNums(nums.filter(n => n !== num));
  }

  const numsList = nums.map(n => (
    <NumberItem
      value={n}
      remove={removeNum}
    />)
  );

  return <ul>{numsList}</ul>;
}
```

- When code runs, warning that a key should be provided for list items.
- ***key*** is a special string attr to include when creating lists of elements.

Adding keys

Let's assign a key to our list items inside ***numbers.map()***

demo/numbers-app/src/KeyedNumberList.js

```
function KeyedNumberList() {
  const [nums, setNums] = useState([1, 2, 3, 4]);

  function removeNum(num) {
    setNums(nums.filter(n => n !== num));
  }

  const numsList = nums.map(n => (
    <NumberItem
      value={n}
      key={n}
      remove={removeNum}
    />)
  );

  return <ul>{numsList}</ul>;
}
```

Keys

- Keys help React identify which items are changed/added/removed.
- Keys should be given to repeated elems to provide a stable identity.

Picking a key

- Best way: use string that *uniquely* identifies item among siblings.
- Most often you would use IDs from your data as keys:

```
let todoItems = todos.map(todo =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

Last resort

When you don't have stable IDs for rendered items, you may use the iteration index as a key as a last resort:

```
// Only do this if items have no stable IDs

const todoItems = fortodos.map((todo, index) =>
  <li key={index}>
    {todo.text}
  </li>
);
```

- Don't use indexes for keys if item order may change or items can be deleted.
  - This can cause performance problems or bugs with component state.

Note: More details on Keys

Here is some further reading on keys, if you're interested:

Key props and rendering in React

Index as a key is an anti-pattern

Passing Arguments to Event Handlers

- Inside a loop, you'll often want an pass arguments functions you pass down.
- But event handlers always receive the event object as their argument!
- To fix, you can wrap your state-changing function inside of an anonymous function

NumberList Revisited

We could have written our ***NumberList*** like this:

demo/numbers-app/src/NumberListAlt.js

```
function NumberListAlt() {
  const [nums, setNums] = useState(
    [1, 2, 3, 4]
  );

  /** Remove num.
   * Wrapped ver passed to/called by child */
  function removeNum(num) {
    setNums(nums.filter(n => n !== num));
  }

  const numsList = nums.map(n => (
    <NumberItemAlt
      value={n}
      key={n}
      remove={evt => removeNum(n)}
    />)
  );

  return <ul>{numsList}</ul>;
}
```

demo/numbers-app/src/NumberItemAlt.js

```
function NumberItemAlt(props) {
  return (
    <li>
      {props.value}
      <button onClick={props.remove}>X</button>
    </li>
  );
}
```

More detail

- Using arrow functions in the parent simplifies the child
- However, there are performance considerations
- We will favor the pattern in ***NumberList*** over the pattern in ***NumberListAlt***
- If you need to pass the event object to the child, send it along as another argument: `evt => remove(n, evt)`

Changing State

Changing State Review

Always change the state using the second destructured value to ***.useState()***.

```
const [data, setData] = useState(initialState);
```

- During the initial render, the returned state (data) is the same as the value passed as the first argument (initialState).
- The setData function is used to update the state. It accepts a new state value and enqueues a re-render of the component.
- The convention is to always name the second value **setX** where X is the name of the first value.

Normally, variables "disappear" when the function exits but state variables are preserved by React.

Setting State Using State

Sometimes your new state depends on the value of your previous state.

demo/simple-counter/src/SimpleCounter.js

```
function SimpleCounter() {
  const [num, setNum] = useState(0);
  function clickUp() {
    setNum(num + 1);
  }

  function clickUpBy2() {
    setNum(num + 1);
    setNum(num + 1);
  }

  return (
    <div>
      <h3>Count: {num}</h3>
      <button onClick={clickUp}>Up</button>
      <button onClick={clickUpBy2}>Up By 2</button>
    </div>
  );
}
```

What's the problem here?

- Click on "Up" button: requests changing state from 0 → 1 **(ok!)**
- Click on "Up By 2" button:
  - First state change isn't complete, just does 1 → 2 twice **(grrr!)**

If your new state depends on the previous state, you should use the *callback pattern* for ***useState***

The function returned by ***useState*** can accept a callback function.

The callback is called when all *already requested* state changes have finished.

It is passed the state as an argument & should return new state.

A better approach:

demo/simple-counter/src/BetterSimpleCounter.js

```
function BestSimpleCounter() {
  const [num, setNum] = useState(0);
  function clickUp() {
    setNum(n => n + 1);
  }

  function clickUpBy2() {
    setNum(n => n + 1);
    setNum(n => n + 1);
  }

  return (
    <div>
      <h3>Count: {num}</h3>
      <button onClick={clickUp}>Up</button>
      <button onClick={clickUpBy2}>Up By 2</button>
    </div>
  );
}
```

Even better:

demo/simple-counter/src/BestSimpleCounter.js

```
function BestSimpleCounter() {
  const [num, setNum] = useState(0);
  function clickUp() {
    setNum(n => n + 1);
  }

  function clickUpBy2() {
    setNum(n => n + 2);
  }

  return (
    <div>
      <h3>Count: {num}</h3>
      <button onClick={clickUp}>Up</button>
      <button onClick={clickUpBy2}>Up By 2</button>
    </div>
  );
}
```

Mutable Data Structures

So, we've just had primitive values (strings & numbers) in our state.

But state also stores things like objects & arrays.

```
import React, { useState } from "react";
import NumberItem from "./NumberItem";

/** Renders & manages list of numbers.
 *
 * State:
 * - nums: array of numbers: [1, 2, 3, 4]
 */

function NumberList() {
  const [nums, setNums] = useState(
    [1, 2, 3, 4]
  );

  // Remove num. Passed to/called by child
  function removeNum(num) {
    setNums(nums.filter(n => n !== num));
  }

  const numsList = nums.map(n => (
    <NumberItem
      value={n}
      remove={removeNum}
    />)
  );

  return <ul>{numsList}</ul>;
}
// end

export default NumberList;
```

Changing Mutable State

Just mutating a value in the state won't work:

demo/colorful-circles/src/BrokenColorfulCircles.js

```
function BrokenColorfulCircles() {
  const [circles, setCircles] = useState([]);

  function addCircle(newColor) {
    // FIXME: this doesn't work: without using setCircles,
    // component doesn't know that it needs to re-render
    circles.push(newColor);
  }

  return (
    <div>
      <ColorButtons addCircle={addCircle} />
      {circles.map((color, i) => (
        <Circle color={color} key={i} idx={i} />
      ))}
    </div>
  );
}
```

Pushing inside of the state setter also doesn't work: React sees a reference to the same array in memory!

demo/colorful-circles/src/StillBrokenColorfulCircles.js

```
function StillBrokenColorfulCircles() {
  const [circles, setCircles] = useState([]);

  function addCircle(newColor) {
    // FIXME still doesn't work: array reference is unchanged
    setCircles(circles => {
      circles.push(newColor);
      return circles;
    });
  }

  return (
    <div>
      <ColorButtons addCircle={addCircle} />
      {circles.map((color, i) => (
        <Circle color={color} key={i} idx={i} />
      ))}
    </div>
  );
}
```

You may find that sometimes your app works even though you are mutating the current state.

- However, We strongly recommend you don't do this!
  - It makes it much harder to optimize your React app later
  - It can make debugging any state bugs harder

A better way is to make a new copy of the data structure:

demo/colorful-circles/src/ColorfulCircles.js

```
function ColorfulCircles() {
  const [circles, setCircles] = useState([]);

  function addCircle(newColor) {
    // FIXED make a *new* array so reference changes
    setCircles(circles => [...circles, newColor]);
  }

  return (
    <div>
      <ColorButtons addCircle={addCircle} />
      {circles.map((circle, i) => (
        <Circle color={circle} key={i} idx={i} />
      ))}
    </div>
  );
}
```

This will let us later optimize our React apps/use advanced features.

Note: Affect Runtime!

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 side effects.

Nested Mutable States

State can also be things like arrays-of-objects.

Imagine we want our circles to be positioned randomly on the page:

```
[
  { color: "red",      x: 10.2, y: 50.1},
  { color: "honeydew", x: 30.7, y: 99.9 },
  // etc
]
```

demo/colorful-circles/src/PositionedColorfulCircles.js

```
/** Get random int min..max (not incl max) */
function randRange(min = 0, max = 100) {
  return Math.random() * (max - min) + min;
}

/** Manage positioned & re-positionable circles.
 *
 * State:
 * - circles: array of circles: [ {x, y, color }, ... ]
 **/

function PositionedColorfulCircles() {
  const [circles, setCircles] = useState([]);
      // then change copy
      circlesCopy[i] = {
        ...circles[i],
        x: randRange(),
        y: randRange(),
      };
      // return circlesCopy;
      return circlesCopy;
    });
  // end

  return (
    <div>
      <ColorButtons addCircle={addCircle} />
```

What if we want to click on a circle to change its position?

To do this, we need to update state by modifying an object in an array.

One approach:

```
const changePosition = i => {
  setColors(colors => {
    // create a copy of state array
    const colorsCopy = [...colors];

    // update the object at index i
    colorsCopy[i].x = getRandom();
    colorsCopy[i].y = getRandom();

    // return colorsCopy;
    return colorsCopy;
  });
};

New array, but mutated `colorsCopy[i]`
```

A better way:

```
  /** Add a circle w/newColor */
  function addCircle(newColor) {
    setCircles(circles => [
      ...circles,
      { color: newColor, x: randRange(), y: randRange() },
    ]);
  }

  /** Change position of circle at index i */
  function changePosition(i) {
    setCircles(circles => {
      // create copy of state array
      const circlesCopy = [...circles];
      // create copy of object at idx i,
```

New array *and* object at ***i*** is a new obj! Now we're not mutating *anything*.

Do I *Have* To Do This?

No, not for normal React

But some add-on features & debugging tools will require this.

It's a very good idea to do this — never mutate *any part of state*

A Common Pattern

Often, you can do this with JS "pure functions", like ***map*** or ***filter***:

remove specific colors w/id

```
setColors(colors => colors.filter(colorObj => colorObj.color !== color))
```

another way to change position

```
setColors(colors => (
  colors.map((colorObj, idx) => (
    idx === i
      ? { ...colorObj, x: getRandom(), y: getRandom() }
      : colorObj
  ))
))
```

These are good intermediate idioms to practice