React Lecture 13

State Structure and Lifting states

Group Related States

Example you have two x and y co-ordinates and these two will always change together. In this case we can group these two states in one.

```
const [x, setX] = useState(0);
const [y, setY] = useState(0);
```

We can create a single state object with x and y properties, grouping the two related states together.

```
const [position, setPosition] = useState({ x: 0, y: 0 });
```

Avoid redundant states

If you can calculate some information from the component's props or its existing state variables during rendering, you should not put that information into that component's state.

Avoid redundant states - Redundant State

This form has three state variables: firstName, lastName, and fullName. However, fullName is redundant. You can always calculate fullName from firstName and lastName during render, so remove it from state.

```
import { useState } from 'react';

export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');
  const [fullName, setFullName] = useState('');

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
    setFullName(e.target.value + ' ' + lastName);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
    setFullName(firstName + ' ' + e.target.value);
  }

  return (
    <>
      <h2>Let's check you in</h2>
      <label>
        First name:{' '}
        <input
          value={firstName}
          onChange={handleFirstNameChange}
        />
      </label>
      <label>
        Last name:{' '}
        <input
          value={lastName}
          onChange={handleLastNameChange}
        />
      </label>
      <p>
        Your ticket will be issued to: <b>{fullName}</b>
      </p>
    </>
  );
}
```

Avoid redundant states - Avoiding Redundant State

Here, fullName is not a state variable. Instead, it's calculated during render:

As a result, the change handlers don't need to do anything special to update it. When you call setFirstName or setLastName, you trigger a re-render, and then the next fullName will be calculated from the fresh data.

```
export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');

  const fullName = firstName + ' ' + lastName;

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
  }

  return (
    <>
      <h2>Let's check you in</h2>
      <label>
        First name:{' '}
        <input
          value={firstName}
          onChange={handleFirstNameChange}
        />
      </label>
      <label>
        Last name:{' '}
        <input
          value={lastName}
          onChange={handleLastNameChange}
        />
      </label>
      <p>
        Your ticket will be issued to: <b>{fullName}</b>
      </p>
    </>
  );
}
```

Avoid duplication in state

Currently, it stores the selected item as an object in the selectedItem state variable. However, this is not great: the contents of the selectedItem is the same object as one of the items inside the items list. This means that the information about the item itself is duplicated in two places.

```jsx
import { useState } from 'react';

const initialItems = [
  { title: 'pretzels', id: 0 },
  { title: 'crispy seaweed', id: 1 },
  { title: 'granola bar', id: 2 },
];

export default function Menu() {
  const [items, setItems] = useState(initialItems);
  const [selectedItem, setSelectedItem] = useState(
    items[0]
  );

  function handleItemChange(id, e) {
    setItems(items.map(item => {
      if (item.id === id) {
        return {
          ...item,
          title: e.target.value,
        };
      } else {
        return item;
      }
    }));
  }

  return (
    <>
      <h2>What's your travel snack?</h2>
      <ul>
        {items.map((item, index) => (
          <li key={item.id}>
            <input
              value={item.title}
              onChange={e => {
                handleItemChange(item.id, e)
              }}
            />
            {' '}
            <button onClick={() => {
              setSelectedItem(item);
            }}>Choose</button>
          </li>
        ))}
      </ul>
      <p>You picked {selectedItem.title}.</p>
    </>
  );
}
```

Avoid duplication in state

The state used to be duplicated like this:

1. items = [{ id: 0, title: 'pretzels'}, ...]
2. selectedItem = {id: 0, title: 'pretzels'}

But after the change it's like this:

1. items = [{ id: 0, title: 'pretzels'}, ...]
2. selectedId = 0

The duplication is gone, and you only keep the essential state!

Avoid deeply nested state

Deeply nested states should be avoided because it creates complexity in our application and make them difficult to manage and work it.

Try to keep states as flat as possible.

Sharing States Between Components

Introduction

Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as lifting state up, and it's one of the most common things you will do writing React code.

Lifting State Up

1. Remove states from child components
2. Pass hardcoded data from the parent component
3. Add state to the common parent component