COURSEWARE

Lifting State

Contents

Overview

Often, several components need to reflect the same changing data.
It is recommended to lift the shared state up to their closest common ancestor.
Lifting the state prevents sharing too much or too little state in your component tree.
Basically, it is a refactoring that you have to do once in a while to keep your components maintainable and focused on only consuming the sate that they need to consume.

How to lift state up

Let's break this down with an example.
The "Search-a-List" example has three components. Two sibling components - a Search component, and a List component, that are used in an overarching `SearchableList` Component.
All of which are functional components.

```
// Search Component
const Search = ({children}) => {
    const [query, setQuery] = React.useState('');
    const handleQuery = e => {
        setQuery(e.target.value);
    }
    return(
        <>
            <div>
                {children}
                <input type="text" value={query} onChange={handleQuery}/>
            </div>
        </>
    )
}
```

```
//List Component
const List = ({list}) => (
    <ul>
    {list.map(item => (
        <li key={item.id}>{item.name}</li>
    ))}
    </ul>
);
```

The `SearchableList` component uses both components - Search and List component. Therefore, both components become siblings in the component tree:

```
const SearchableList = ({list}) => {
    <div>
        <Search>Search List:</Search>
        <List list={list}/>
    </div>
}
```

However, the above example will not work.
The `search` component knows about the `query` that could be used to filter the list, but the `List` component doesn't know about it.
The state from the `Search` component can only be passed down the component tree, by using `props`, but not up to its parent component.
Therefore, you have to lift the state of the `Search` component up to the `SearchableList` component, to make the `query` state accessible for the List component, in order to filter the list of items eventually.

In other words, we want to share the `query` state in both `List` component and `Search` component.

So, the adjusted `Search` component will look like this:

```
const Search = ({query, handleQuery, children}) => {
    <div>
        {children}
        <input type="text" value={query} onChange={handleQuery}/>
    </div>
}
```

The adjusted `SearchableList` component will look like this:

```
const SearchableList = ({list}) => {
    const [query, setQuery] = React.useState('');
    const handleQuery = e =>{
        setQuery(e.target.value);
    }
    return(
        <div>
            <Search query={query} handleQuery={handleQuery}>Search List:
</Search>
            <List list={list}/>
        </div>
    );
}
```

After the state has been lifted up, the parent component takes care of the local state management.
The state has been lifted to share the local state across the child components.
Finally, we can use `query` to filter the list for the `List` component:

```
...
const filteredList = list.filter(byQuery(query));

return(
    ...
    <List list={filteredList}/>
    ...
);
const byQuery = query => item => !query ||
item.name.toLowerCase().includes(query.toLowerCase());
```

The list gets filtered by the search query before it reaches the `List` component.
An alternative would be passing the `query` state as prop to the `List` component and the `List` component would apply the filter to itself.

Tutorial

In this tutorial, we will create a simple application which lets us add players to our game room.

1. Create a component `Game`:

```
const Game = () => { }

export default Game;
```

2. Create two states, one which stores an array of players, and another which stores a player's name:

```
const Game = () => {
    const [playerName, setPlayerName] = useState("");
    const [players, setPlayers] = useState([]);
}
```

Don't forget to add the following line to the top of your file `import {useState} from 'react';`

3. Create another component `AddGamers` which contains a form, with a text field and button.

```
const AddGamers = () => {
return(
    <form>
        <input type="text" placeholder="Enter name:"/>
        <button type="button"> Add Player </button>
    </form>
)
}
export default AddGamers;
```

4. In `Game.jsx` create a function called `newPlayer`, which has a destructured 'target' object as parameter and in the body of the function, call the `setPlayerName` and pass in `target.value` as parameter:

```
const newPlayer = ({target}) =>{
    setPlayerName(target.value);
}
```

5. Next, we need to associate this function with an input field. Pass the `newPlayer` function as a prop to the `AddGamers` component and add an event listener to the input field which invokes this method.

```
// Game.jsx
const newPlayer = ({target}) => {
    setPlayerName(target.value);
}

return(
    <AddGamers newPlayer={newPlayer}/>
);
```

```
// AddPlayer.jsx
const AddGamers = ({newPlayer}) => {
    return(
        <input type="text" onChange={newPlayer}/>
    );
}
```

6. Perfect, now we need to save the player name into the array on submission of the form.
7. In `Game.jsx` create two functions, one which prevents the default behaviour of the form, and another which saves the player's name into the state array:

```
const submitForm = (event) => {
    // prevent form submission on initial click
    event.preventDefault();
}

const handleAdd = () => {
    // save playername into the array
    setPlayers( players => [...players, playerName]);
}
```

8. Again, we need to associate these functions to the appropriate handler in the `AddGamers` component:

```
// Game.jsx
return(
    <AddGamers newPlayer={newPlayer} submitHandler={submitForm} handleAdd=
{handleAdd}/>
)
```

```
// AddGamers.jsx
const AddGamers = ({newPlayer, submitHandler, handleAdd}) => {
    return(
        <form onSubmit={submitHandler}>
            <input type="text" placeholder="Enter name:" onChange={newPlayer}/>
            <button type="button" onClick={handleAdd}> Add Player </button>
        </form>
    )
}
```

9. Now, we need to print the player names to the screen.
10. Create another component called `GameRoom.jsx`:

```
const GameRoom = () => {
    return()
}
export default GameRoom;
```

11. Call this component in `Game` as a child, after `AddGamers`

```
// Game.jsx

return(
    <>
        <AddGamers newPlayer={newPlayer} submitHandler={submitForm} handleAdd=
{handleAdd}/>
        <GameRoom/>
    </>
);
```

12. We need to loop over the array in order to print it to the screen, so first pass `Players` as a prop to the `<GameRoom/>` component:

```
// Game.jsx
return(
    <>
        // ...
        <GameRoom players={players}/>
    </>
);
```

13. In the function declaration of GameRoom, destructure the object passed in as a property:

```
const GameRoom = ({players}) => {
    //...
}
```

14. Now we have access to the array, we must loop through the entries. Use a map to loop through the array and print the value to the screen:

```
const GameRoom = ({players}) => {

    const Display = ({playerName}) => <li> {playerName} </li>

    return(
        <ul>
            {players.map((playerName,i) => (
                <Display key={i} playerName={playerName}/>
            ))}
        </ul>
    )
}
```

And that's it! You've learnt how to pass information from one state up to a common ancestor and use the value in another component!

▶ Code

Exercises

Create a simple product manager application which meets the following requirements:

- A user should be able to add an item to their basket
- At a later stage, the user should be able to amend the items in their basket, including removing or updating the quantity of an item.