React Lecture - 7

Pure Components

Introduction

Some JavaScript functions are pure. Pure functions only perform a calculation and nothing more. By strictly only writing your components as pure functions, you can avoid an entire class of baffling bugs and unpredictable behavior as your codebase grows. To get these benefits, though, there are a few rules you must follow

What is a Pure Function ?

In computer science (and especially the world of functional programming), a pure function is a function with the following characteristics:

- **It minds its own business.** It does not change any objects or variables that existed before it was called.
- **Same inputs, same output.** Given the same inputs, a pure function should always return the same result.

Example

The given function will always give same output for a given input.

Ex: If Input is 2 output will always be 4

If input is 3 output will always be 6

This is pure function because if does not modify any pre existing variables and always return same output for same input

```
function double(number) {
  return 2 * number;
}
```

Pure Component Example

Here you can see given input will always give a certain output. It will never change in between re renders

```
1  function Recipe({ drinkers }) {
2    return (
3      <ol>
4        <li>Boil {drinkers} cups of water.</li>
5        <li>Add {drinkers} spoons of tea and {0.5 * drin
6        <li>Add {0.5 * drinkers} cups of milk to boil ar
7      </ol>
8    );
9  }
10
11 export default function App() {
12   return (
13     <section>
14       <h1>Spiced Chai Recipe</h1>
15       <h2>For two</h2>
16       <Recipe drinkers={2} />
17       <h2>For a gathering</Sh2>
18       <Recipe drinkers={4} />
19     </section>
20   );
21 }
22
```

Impure Component Example

React's rendering process must always be pure. Components should only return their JSX, and not change any objects or variables that existed before rendering—that would make them impure!

Here is a component that breaks this rule:

```
let guest = 0;

function Cup() {
  // Bad: changing a preexisting variable!
  guest = guest + 1;
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
  return (
    <>
      <Cup />
      <Cup />
      <Cup />
    </>
  );
}
```

Impure Component Example

This component is reading and writing a guest variable declared outside of it. This means that calling this component multiple times will produce different JSX!

You can fix this component by passing guest as a prop instead:

```
let guest = 0;

function Cup() {
  // Bad: changing a preexisting variable!
  guest = guest + 1;
  return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
  return (
    <>
      <Cup />
      <Cup />
      <Cup />
    </>
  );
}
```

Strict Mode

React offers a “Strict Mode” in which it calls each component’s function twice during development. By calling the component functions twice, Strict Mode helps find components that break these rules.

Notice how the original example displayed “Guest #2”, “Guest #4”, and “Guest #6” instead of “Guest #1”, “Guest #2”, and “Guest #3”. The original function was impure, so calling it twice broke it. But the fixed pure version works even if the function is called twice every time. Pure functions only calculate, so calling them twice won’t change anything—just like calling double(2) twice doesn’t change what’s returned

Strict Mode has no effect in production, so it won’t slow down the app for your users. To opt into Strict Mode, you can wrap your root component into <React.StrictMode>. Some frameworks do this by default.

Local Mutation

In the above example, the problem was that the component changed a preexisting variable while rendering. This is often called a “mutation” to make it sound a bit scarier. Pure functions don’t mutate variables outside of the function’s scope or objects that were created before the call—that makes them impure!

However, it’s completely fine to change variables and objects that you’ve just created while rendering.

Local Mutation

It's completely fine to change variables and objects that you've just created while rendering.

In this example, you create an [] array, assign it to a cups variable, and then push a dozen cups into it:

If the cups variable or the [] array were created outside the TeaGathering function, this would be a huge problem! You would be changing a pre existing object by pushing items into that array.

```
1  function Cup({ guest }) {
2    return <h2>Tea cup for guest #{guest}</h2>;
3  }
4
5  export default function TeaGathering() {
6    let cups = [];
7    for (let i = 1; i <= 12; i++) {
8      cups.push(<Cup key={i} guest={i} />);
9    }
10   return cups;
11 }
12
```