React Lifecycle & State

Prior Knowledge

- Understand how React works: a tree of linked components that are all called via an initial render from ReactDOM.
- Understand the concept of **state** as a place to hold changeable data that represents what's going on in your application.
- Understand the concept of **props** as the way of passing functions and data between your components.

Learning Objectives

- Understand the factors to consider when deciding on state structure.
- Be able to update nested state without mutation.
- Be able to derive information from state.

React State

When making decisions about how to organise data, it's worth spending a little time planning ahead. This applies to React state as much as it does any database.

There are several factors you should consider, and it's possible that some of them may lead to conflicting conclusions, so you will have to make judgment calls. These are some of the criteria decisions should be based on:

- ***Avoiding duplicate data***: this is advice applicable across the wider programming world, and should generally be a deciding factor. Duplicate data is tricky to update and leads to multiple sources of truth, and therefore potential bugs.
- ***Ability to derive necessary information***: store all the information needed (but with as little excess as possible) to get the information that you want.
- ***Keeping data flat***: as state should ***never*** be mutated, having nested data that can be overwritten can prove awkward.
- ***Use cases in your application***: in many to many relationships, consider which dataset should be responsible for holding the other. An app that held information about ***recipes*** and ***ingredients*** might store that data differently if the primary use of the application is to organise your recipe cards or to organise your store cupboard, for example.
- ***Extensibility***: keep in mind how your data may change in the future - allow for extra information fields.
- ***Performance***: should not be your number one concern, but there are easy wins and for a website that is re-rendered regularly, this should be on your mind. Storing collected data in objects is often more efficient to query than finding the correct object in an array, for example.

Derivation from state

The state is our ***source of truth*** for our application. As such we should not repeat data. The state must be a **complete representation of the UI** but this does not mean we need to keep each piece of information in state explicitly. Consider the component below

```
const App = () => {
  const [todos, setTodos] = useState([
    { id: 1, text: 'eat' },
    { id: 2, text: 'sleep' },
    { id: 3, text: 'react' },
  ]);

  return (
    <div>
      <h1>My Todo List</h1>
      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>{todo.text}</li>
        ))}
      </ul>
    </div>
  );
};
```

Say we want to work out how much coding we have to do and how many times 'react' appears in our todoList. We could add some more state to our app to keep a count variable and update this whenever `'react'` is added or removed from our list, but there is no need. We already have enough information in our current state to work out how many times it appears in the list. This information is said to be `derivable`.

```
const Todos = () => {
  const [todos, setTodos] = useState([
    { id: 1, text: 'eat' },
    { id: 2, text: 'sleep' },
    { id: 3, text: 'react' },
    { id: 4, text: 'eat' },
    { id: 5, text: 'sleep' },
    { id: 6, text: 'react' },
  ]);

  // deriving the reactCount from state
  let reactCount = 0;
  todos.forEach((todo) => {
    if (todo.text === 'react') reactCount++;
  });

  return (
    <div>
      <h1>My Todo List</h1>
      <p>How much react I have to do: {reactCount}</p>
      <ul>
        {todos.map(({ id, text }) => (
          <li key={id}>{text}</li>
        ))}
      </ul>
    </div>
  );
};
```

Depending on the complexity of the state and derivation required then the functionality can be extracted to a function that can be built with TDD. This way we can be sure that our functions work before using them to update our UI.

```
// src/utils/todos.js
export const getReactCount = (todos) => {
  let reactCount = 0;
  todos.forEach((todo) => {
    if (todo.text === 'react') reactCount++;
  });
  return reactCount;
};

// src/components/Todos.jsx
import { getReactCount } from '../utils/todos.js';

const Todos = () => {
  const [todos, setTodos] = useState([
    { id: 1, text: 'eat' },
    { id: 2, text: 'sleep' },
    { id: 3, text: 'react' },
    { id: 4, text: 'eat' },
    { id: 5, text: 'sleep' },
    { id: 6, text: 'react' },
  ]);

  const reactCount = getReactCount(todos);

  return (
    <div>
      <h1>My Todo List</h1>
      <p>How much react I have to do: {reactCount}</p>
      <ul>
        {todos.map(({ id, text }) => (
          <li key={id}>{text}</li>
        ))}
      </ul>
    </div>
  );
};
```

nb You can install jest and run unit tests in the usual manner as part of a React project. However you may get some warnings from ESlint that was setup by Vite as it won't recognise jest functions like `describe`. You can add this plugin to add linting support for jest.