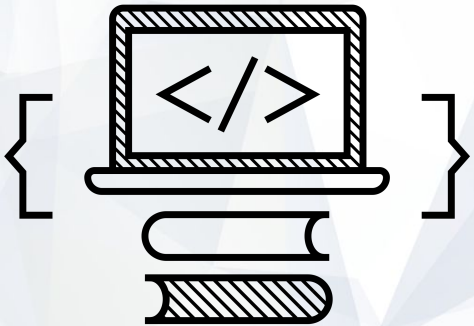
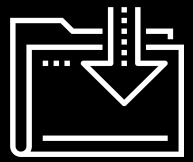


State Management in React

Coding Boot Camp

Module 22



Pure vs. Impure Functions

Let’s Talk About Pure Functions

Pure functions keep data passed to the function untouched by the function itself.

Pure Functions

- Are straightforward with singular intention.
- Yield the same result when passed the same arguments—every time.
- Store modified data in new variables.

Impure Functions

- Include side effects like database and network calls.
- Often modify the values that are passed in.

Impure and Pure Examples

Pure functions keep data passed to the function untouched by the function itself.

Pure Functions

- Regardless of what number we pass in, we will get a predictable result every time.

```
// Pure function
const doubleIt = (int) ⇒ int * 2;

doubleIt(5); // returns 10

```

Impure Functions

- The data that gets passed to impure functions can be mutated.
- Might include calls to a database or API, possibly returning something unexpected.

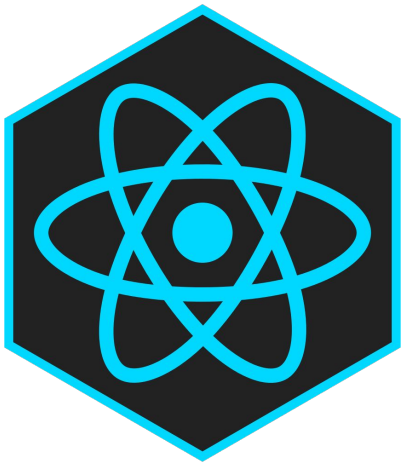
```
const exampleFetch = async () ⇒ {
  const response = await fetch('https://reqres.in/api/users/random');
  const json = await response.json();
  return json;
};

exampleFetch(); // returns a random user
```

Creating pure functions is an effective technique to apply to all of your JavaScript, but it is especially useful when creating React components.

React Hooks and Context API

React manages state through a combination of Hooks and the Context API. React Context enables us to share data globally and follows a very similar design pattern to Redux, a similar but separate library. Learning one will help you learn the other.



A good use for the Context API would be making a user's account information available to all subscribing components.

Redux

Redux is another library that helps us manage the state of complex applications. Redux requires a lot of boilerplate code, but its utility increases exponentially with an application's complexity.

Instead of learning about the Redux library, we will focus on the concepts and design patterns that Redux uses. After learning how to create each function from scratch, we will implement a Redux-like state management system in a small CMS app.

The React Parts: Providers, Consumers, and Reducers

Today we will cover three core React state components.
Some of these will overlap with Redux:

Providers

Allow child components to subscribe to context changes.

Consumers

Allow for consumption of the data made available by the provider.

Reducers

Take the current state and a desired action and return a new copy of state.

The Redux Parts: Actions, Reducers, and Store

We will also cover three core Redux concepts:

Actions

Declarative objects that define what the store should do to the state.

Reducers

Handle all actions by taking in the previous state and returning a new state.

Store

Grants the entire application access to the reducer function and the global state.

The Principles of Redux

Principle 1: State Tree

Every stateful aspect of your application can be represented by a single JavaScript object, known as the **state tree**.

```
console.log(store.getState())

/* Prints
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
*/
```

Principle 2: Read-Only

The state tree is **read-only**—meaning that to change the state tree, you need to dispatch an action. The action describes the change that the state will undergo, in a declarative manner.

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
})
```

Principle 3: Pure Functions

Try to make your reducers pure functions. It is considered bad practice to use side effects inside reducer functions.

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}
```

```
case 'COMPLETE_TODO':
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
    default:
      return state
  }
}
```

Instructor Demonstration

Mini-Project