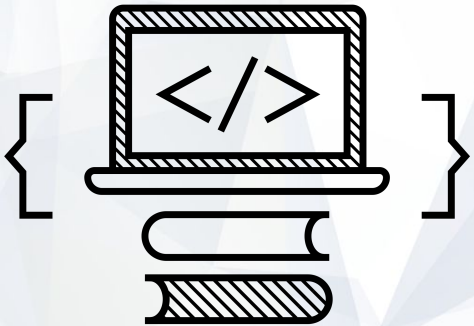
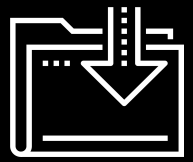


# State Management in React

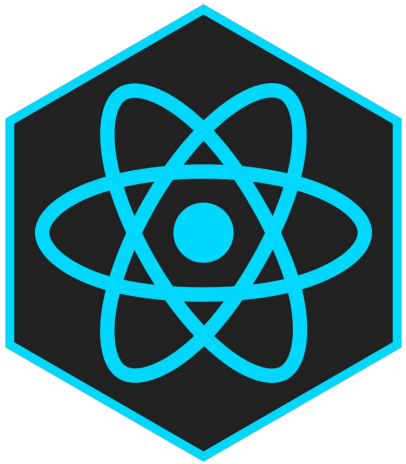
Coding Boot Camp

Module 22



# State: Basics

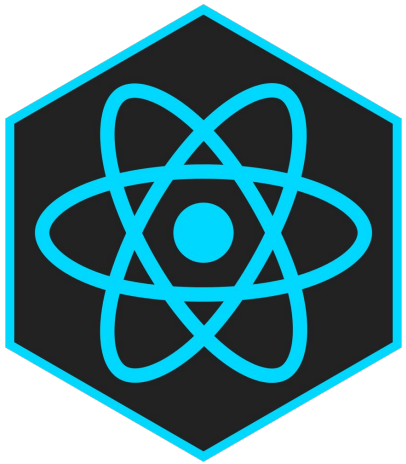
In a React application, state refers to the data that represents the current state of the application. It could include things like user input, fetched data, or any other relevant information.



**In larger applications, managing state can become complex. That's where the idea of a global state comes in.**

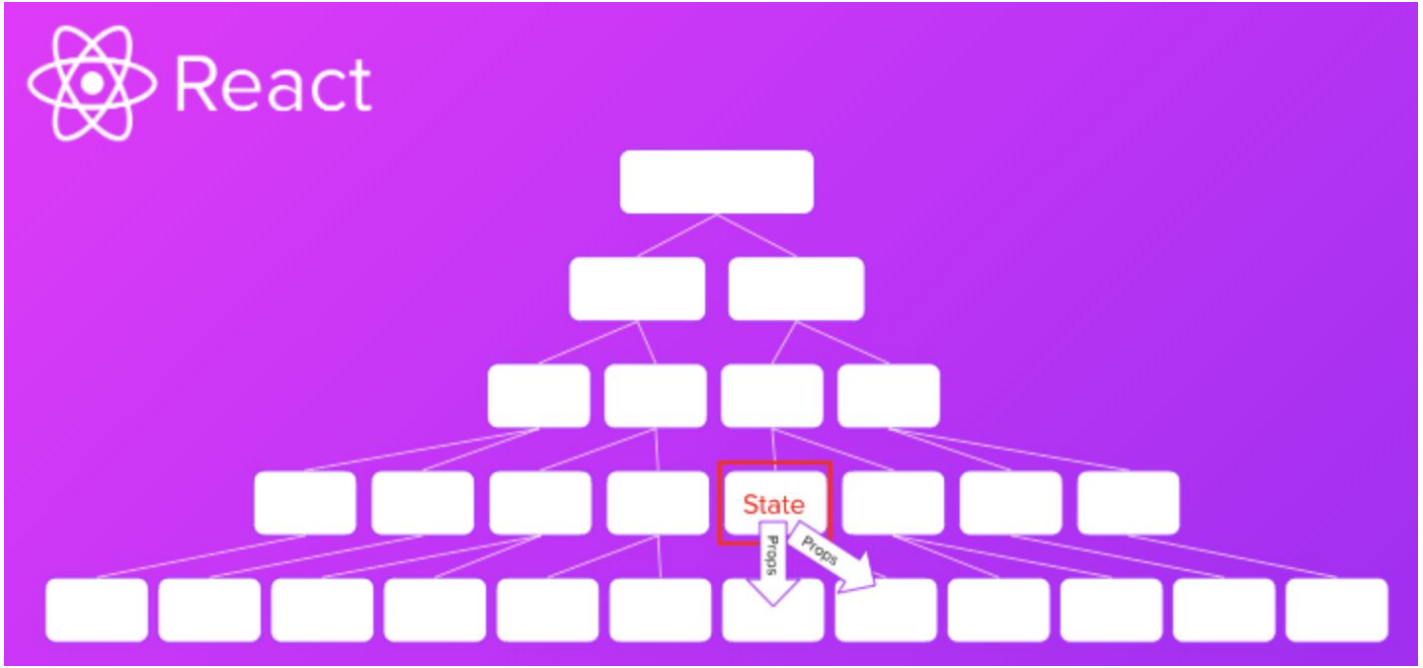
# React Hooks and Context API

In more complex applications, 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.



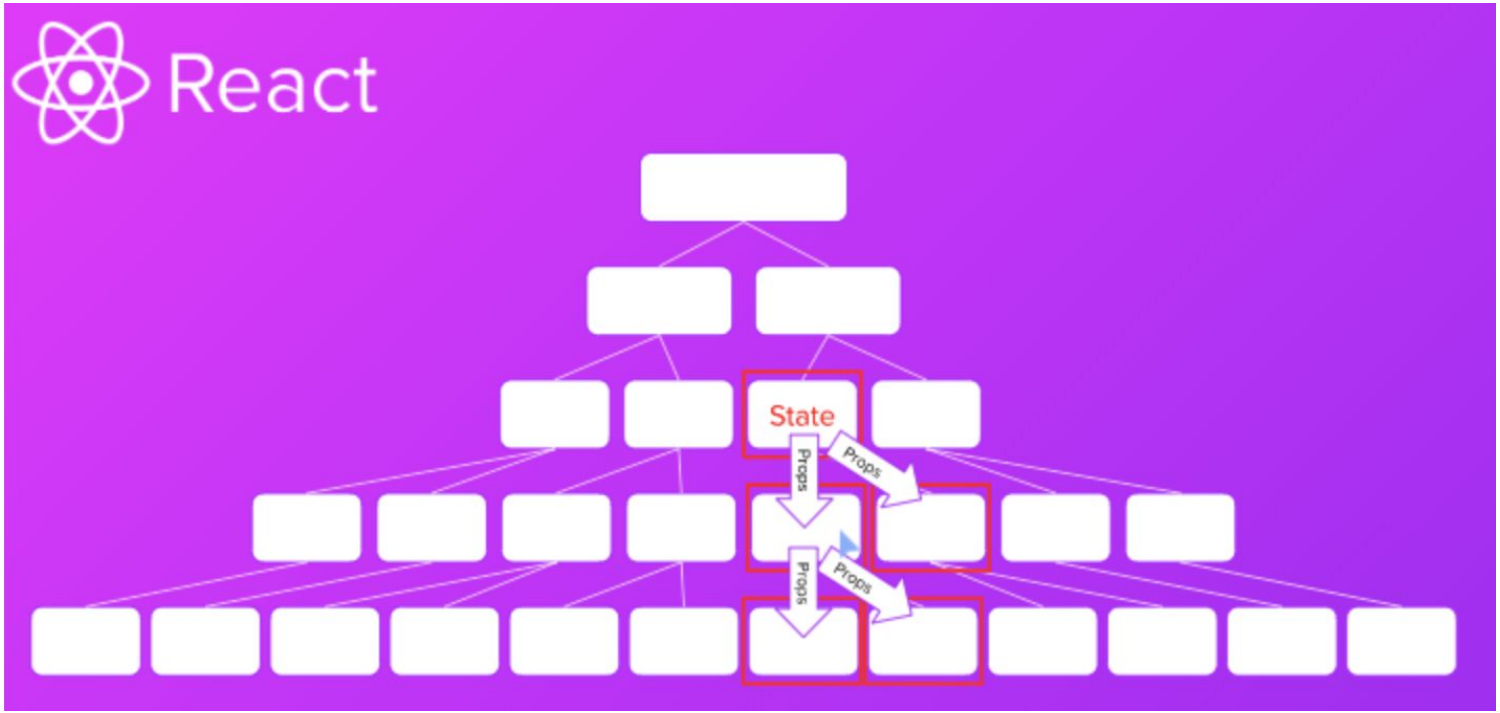
**A good use for the Context API would be making a user's account information available to all subscribing components.**

# Global State



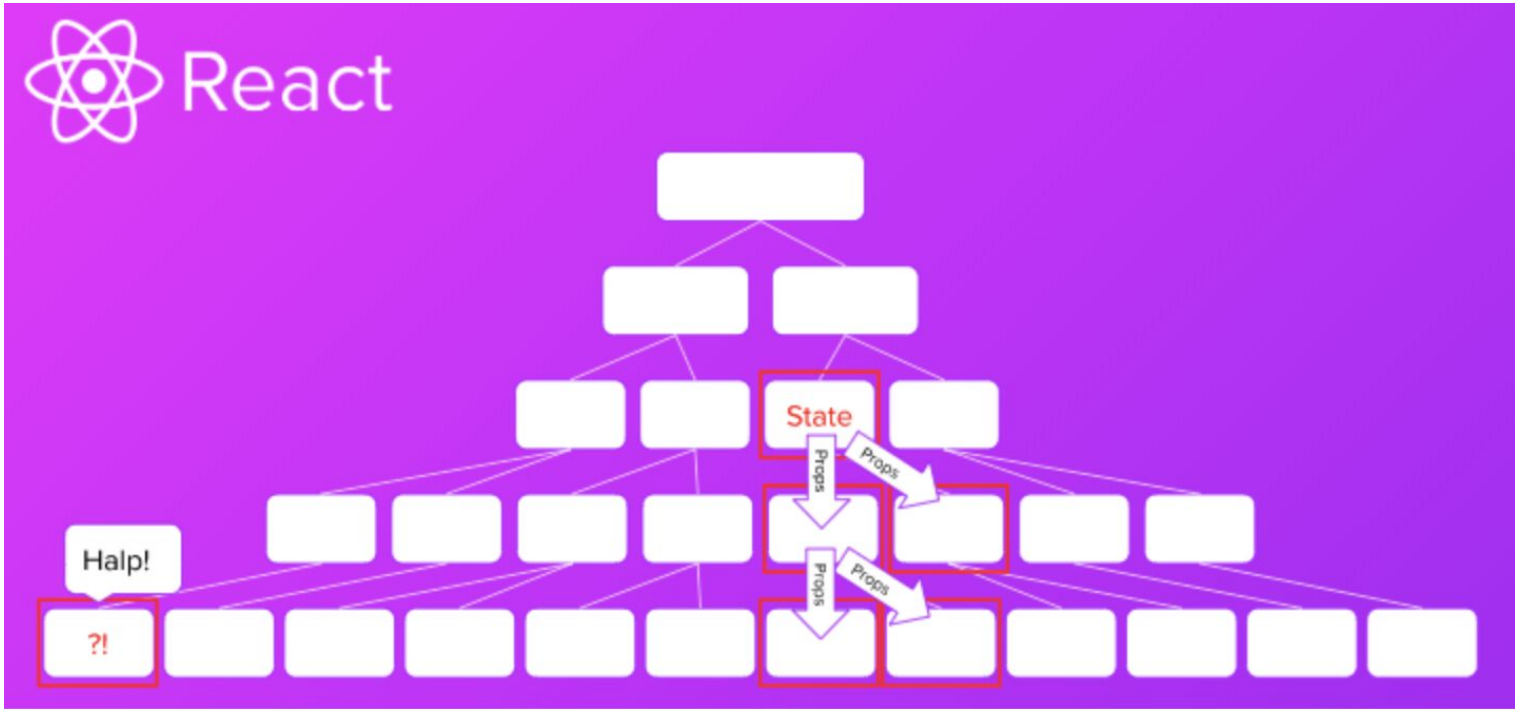
# Global State

To apply state to more components, you need to raise the state up the tree



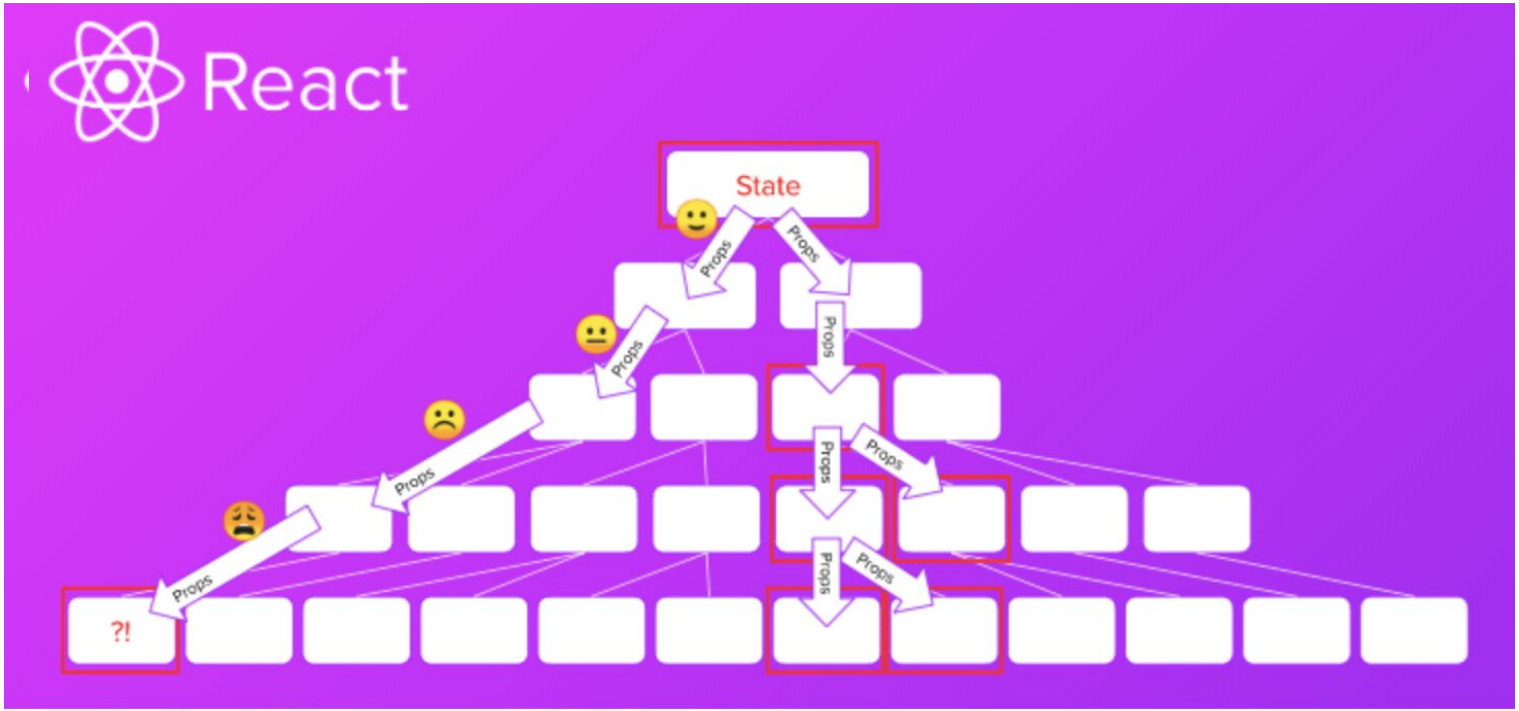
# Global State

To apply state to more components, you need to raise the state up the tree



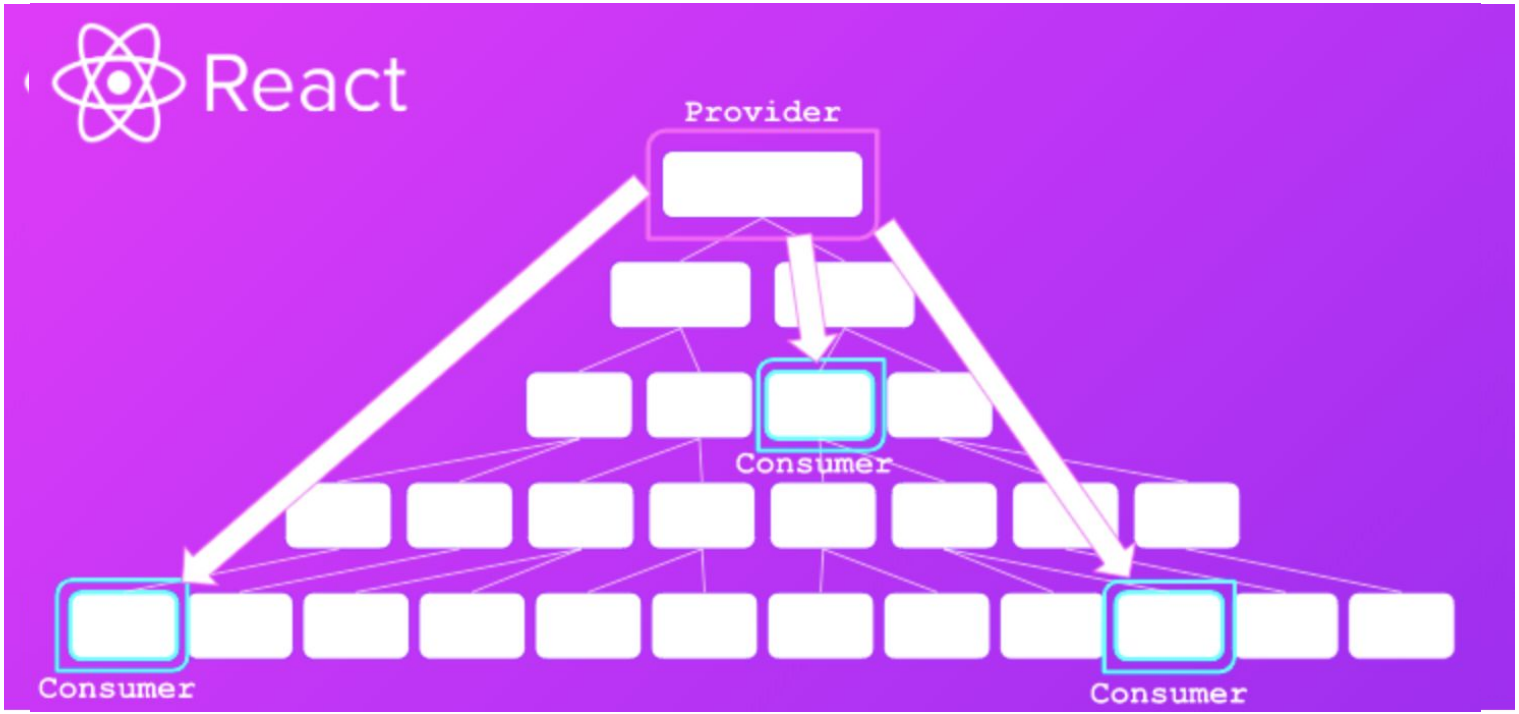
# Global State

This can lead to *prop drilling*



# Global State

Solution: **`Context API`**



# 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 and Consumers

By using the Provider and Consumer components together, you avoid prop drilling and easily share data across components without explicitly passing props at each level.

## Providers

Responsible for creating a context and making the data or state available to its child components. It wraps the component tree and all the child components can access this value.

## Consumers

Acts as a subscriber to the shared context provided by the nearest ancestor Provider component. Enables components to effortlessly consume shared data without the need for explicit prop passing.

# The Redux Parts: Actions, Reducers, and Store

We will also cover three core Redux concepts:

## Actions

Plain JavaScript objects that describe the changes you want to make to the state.

## Reducers

Functions responsible for applying the changes described in the actions to the state tree.

## Store

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

# The Principles of Redux

# Principle 1: State Tree

The global state is a single source that holds all the data needed by different components of the application. We access this data as an 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 instead of directly modifying the state, you declare what changes you want to happen by dispatching actions. The actual modification of the state is handled by reducers.

```
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. Two examples of pure functions:

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

```
function reducer(state, action) {
  switch(action.type) {
    case 'COMPLETE_TODO':
      return state.map((todo, index) => {
        if (index === action.index) {
          return {
            ...todo,
            completed: true
          };
        }
        return todo;
      });
    default: return state
  }
}
```

# 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

- Only depend on the input arguments.
- Don’t modify the input arguments.
- Always produce a predictable output.
- And don’t have side effects.

## 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.

# Instructor Demonstration

## Mini-Project