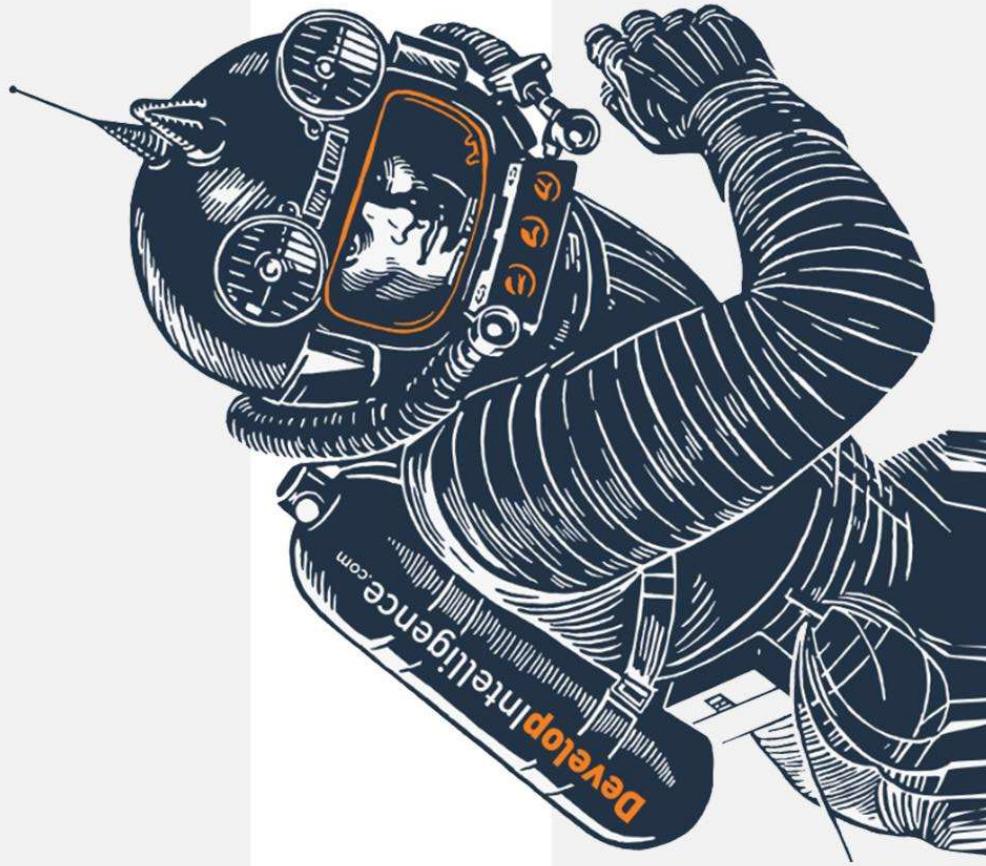


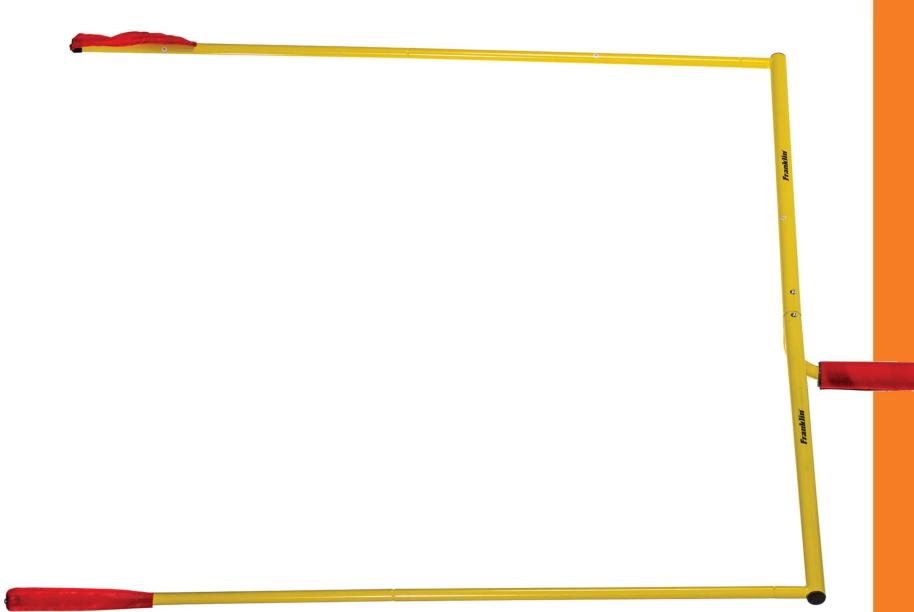
State

Develop Intelligence



Goals

1. Describe useState
2. Explain userReducer
3. List the **rules of hooks**





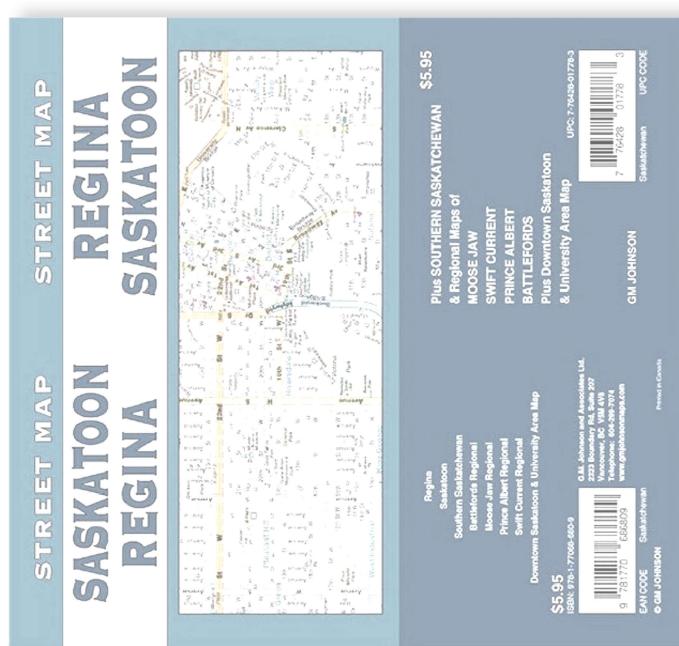
Roadmap

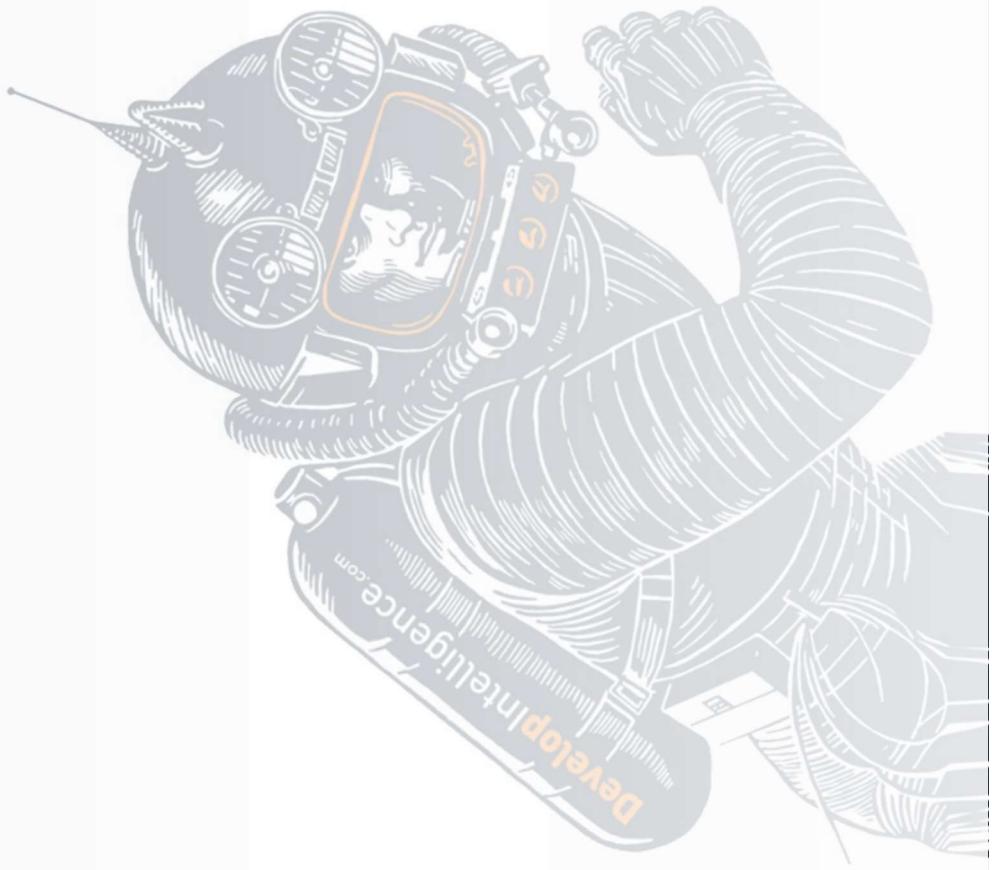
1. Overview

2. Hooks

3. State Molecules

4. Reducers

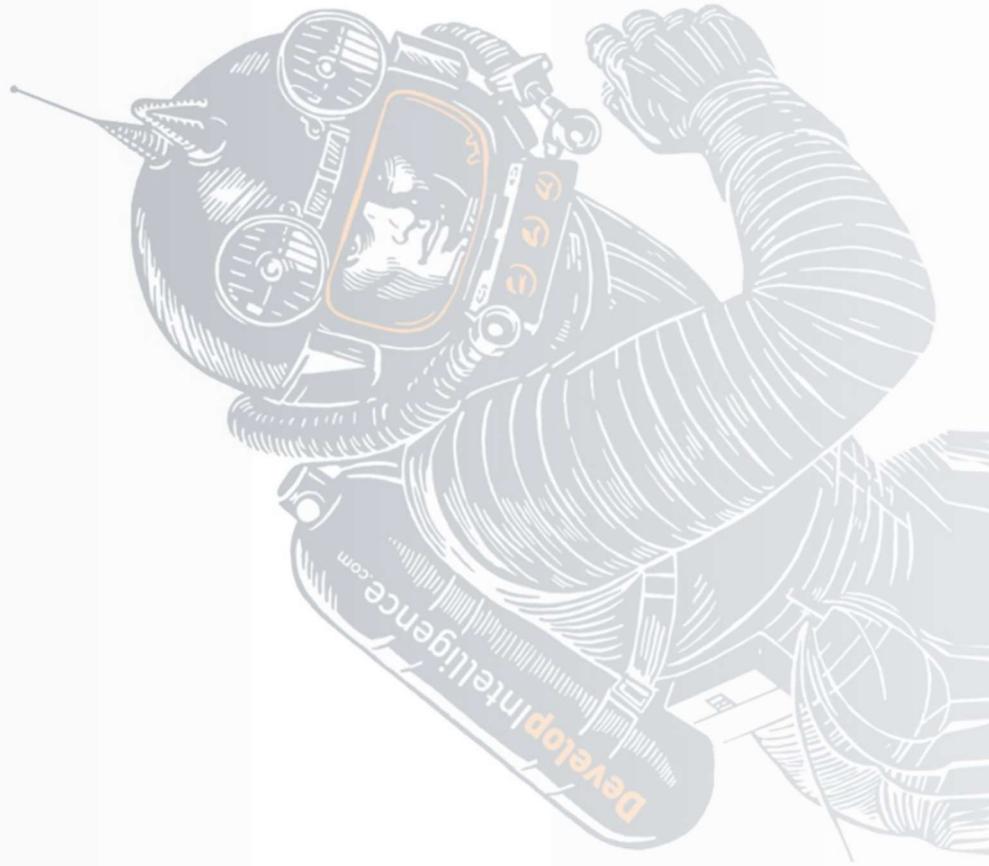




Develop
Intelligence



Overview





What's State?

- **State** is any data that directly effects what's on the screen.
- Components are rendered under 2 conditions:
 - Initial render
 - Updated state
- Functional components are called for each render



Motivating Example

```
1 export const App = () => {
2   let i = 1;
3   return <>
4     <h1>Your number is: {i}</h1>
5     <button onClick={() => (i = i + 1)}>Increment</button>
6   </>;
7 };
```

- It's just a function:
 - Invocation initializes local variables
 - Exiting recovers memory from the stack
- Nothing tells React that state has changed

What we Need

- Functional components are just functions
- So we need
 - Something to keep track of state across multiple invocations of a functional component
 - Kind of like data members of a class



useState to the Rescue

- A 'hook' function lets you 'hook' into React behavior
- useState is a 'hook' providing
 - A **state variable** to retain the data between renders.
 - A **state setter** function to update the variable and trigger React to render the component again.
- ***Unbreakable rule:*** Only update state through its setter function

```
1 | const [i, setI] = useState(1); //Array destructuring
```

Fixed



- With useState even a functional component can remember

```
1 import { useState } from 'react';
2
3 export const App = () => {
4   const [i, setI] = useState(1);
5   return <>
6     <h1>Your number is: {i}</h1>
7     <button
8       onClick={() => setI(i+1)}
9     >
10       Increment
11     </button>
12   </>;
13};
```



useState Bloopers

Why isn't this working?

```
1 export const App = () => {
2   let [i, setI] = useState(1);
3   return (
4     <>
5       <h1>Your number is: {i}</h1>
6       <button onClick={() => i=i+1}>Increment</button>
7     </>
8   );
9 }
```



State == Snapshot

State variables might look like regular JavaScript variables that you can read and write to. However, state behaves more like a snapshot. Setting it does not change the state variable you already have, but instead triggers a re-render.

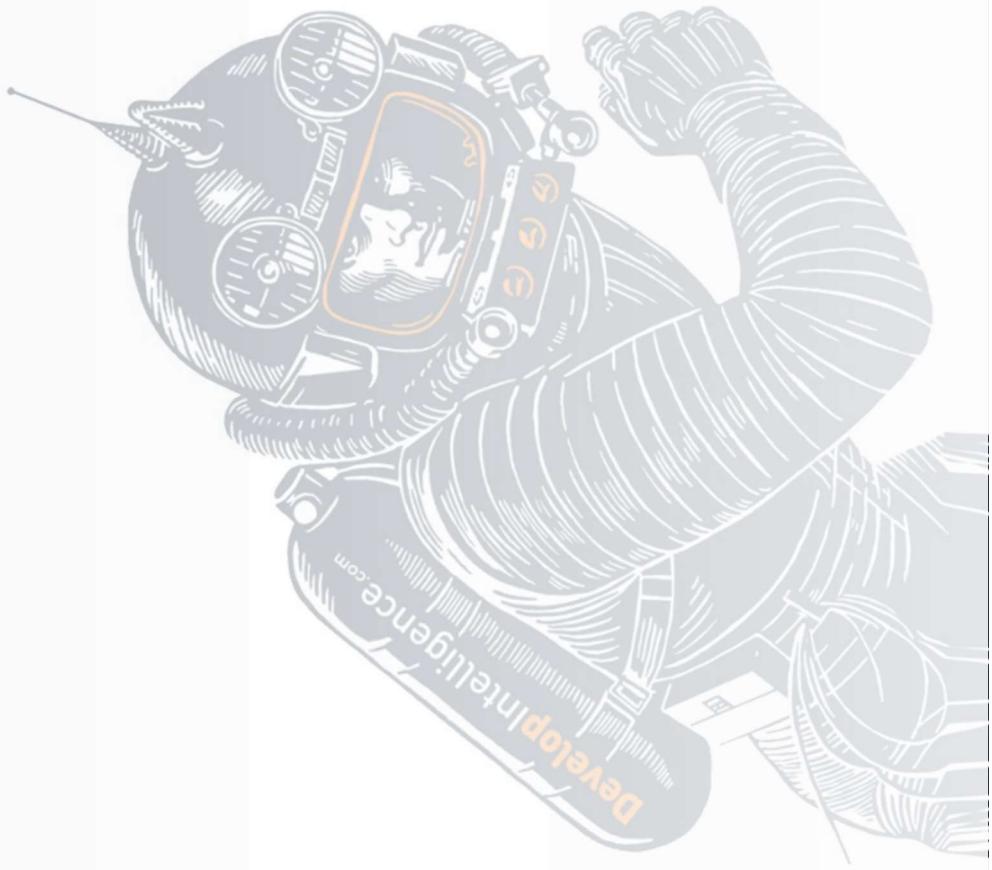
-- [React.dev](#)

Proof



Why isn't this working?

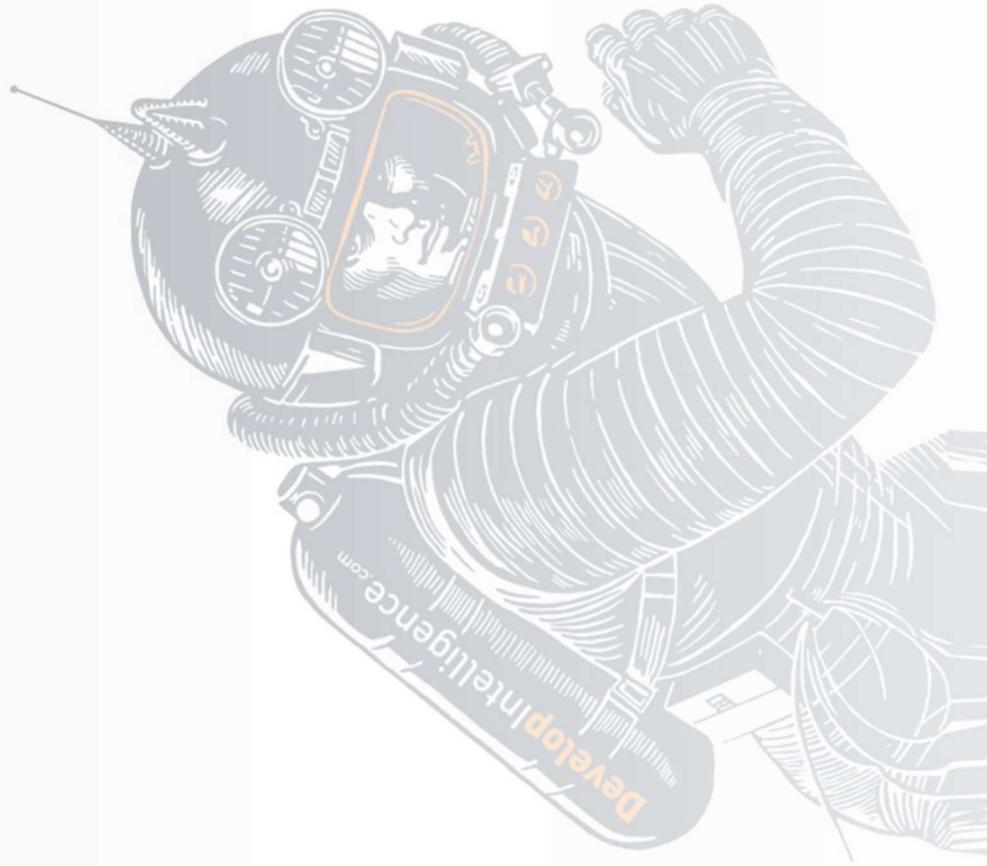
```
1  export const App = () => {
2    const [i, setI] = useState(1);
3    const handleClick = () => {
4      setI(i + 1);
5      setI(i + 1);
6      setI(i + 1);
7    };
8    return <>
9      <h1>Your number is: {i}</h1>
10     <button onClick={handleClick}>Increment</button>
11   </>;
12 },
```



Develop
Intelligence



Hooks





About Hooks

- A *hook* is a special function
 - Only available during rendering
 - Has a use prefix
- Lets you 'hook into' react features
 - The most fundamental hook is `useState` (maybe)



Rules of Hooks

- **Always** call a hook at the top level of a component or custom hook
- **Don't** call a hook
 - Inside conditions or loops
 - After a conditional return statement
 - From class components
- **Good news:** The linter is good about warnings

Bad Ideas

```
1  export const BadIdea = ({ hasTheme }) => {
2    if (hasTheme) {
3      const theme = useContext(ThemeContext);
4    }
5    // ...
6  }
7
8  export const AnotherBadIdea = ({ hidden }) => {
9    if (hidden) {
10      return;
11    }
12    const [clicks, setClicks] = useState(0);
13    // ...
14  }
```



Not Normal Functions

More good news: Error messages are actually helpful!

Console

- ✖ Hooks can only be called inside the body of a function component.



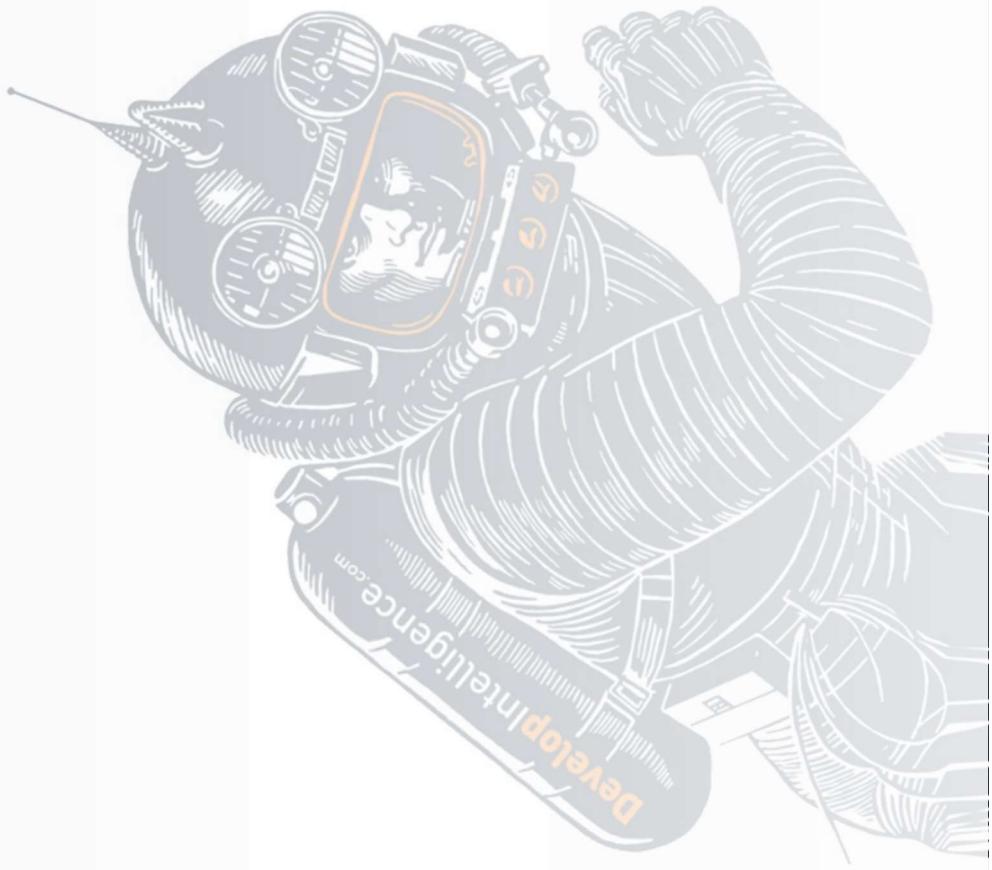
Complaints About Hooks

- Feels weird
 - Sort of magical
 - Different from class-based state management
- Violates the explicit dependency principle
- React lock-in
- Too much magic



Reasons to Love Hooks

- Composable
- Functional
- Everyone else uses them

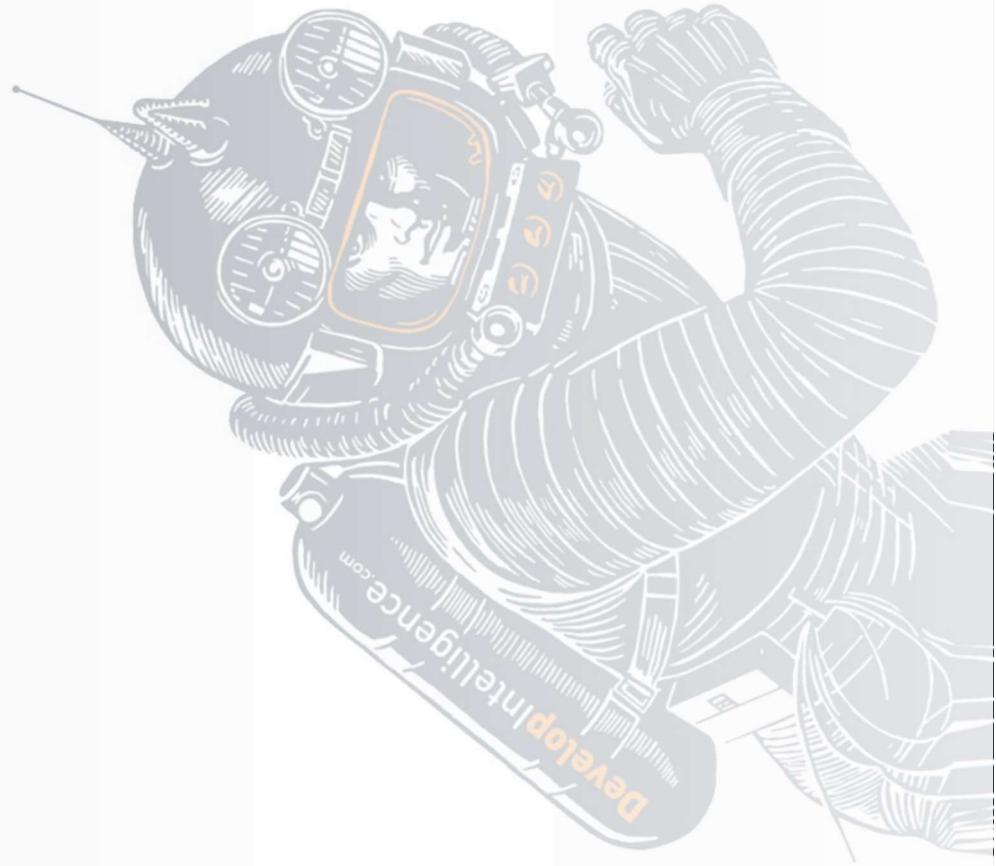


Develop
Intelligence



State Molecules

Develop Intelligence





Overview

- Most real-world state is non-atomic
 - Lists
 - Object graphs
- React tracks state changes based on identity



Multiple useState

```
1  export const UserEditor = () => {
2    const [givenName, setGivenName] = useState('');
3    const [surname, setSurname] = useState('');
4    return <form>
5      <h1>Hello, {givenName} {surname}</h1>
6      <input type="text"
7        onChange={event => setGivenName(event.target.value)}
8        value={givenName}
9      />
10     <input type="text"
11       value={surname}
12       onChange={event => setSurname(event.target.value)}
13     />
14   </form>;
15 }
```



Objects as State

- There's nothing wrong with more than one useState
- **But for related data, it's often better to use an object**

```
1 interface User{  
2   givenName:string;  
3   surname:string;  
4 }
```

Failed User Object

```
1  export const UserEditor = () => {
2    const [user, setUser] = useState<User>({givenName: '', surname: ''});
3    return <form> <h1>Hello, {user.givenName} {user.surname}</h1>
4    <input type="text"
5      onChange={event => {
6        user.givenName = event.target.value;
7        setUser(user);
8      }
9      value={user.givenName}
10     />
11     <input type="text"
12       onChange={event => {
13       user.surname = event.target.value;
14       setUser(user);
15     }
16     value={user.surname}
```



Why So Much Fail?

- Two important points
 - React only re-renders based on changes to state
 - React uses Object.is to determine whether state's changed
- So
 - State changes can't be mutations



Problematic Mutation

```
1 onChange={event => {
2   user.surname = event.target.value;
3   setUser(user);
4 }}
```

Chad Copy

```
1 onChange={event => {
2   setUser({ ...user, surname:event.target.value });
3 }}
```

Fixed



```
1  export const UserEditor = () => {
2    const [user, setUser] = useState<User>({givenName: '', surname: ''});
3    return <form> <h1> Hello, {user.givenName} {user.surname}</h1>
4      <input type="text"
5        onChange={event => {
6          setUser({ ...user, givenName: event.target.value });
7        }
8        value={user.givenName}
9      />
10     <input type="text"
11       onChange={event => {
12         setUser({ ...user, surname: event.target.value });
13       }
14       value={user.surname}
15     />
16   </form>;
17 }
```



Arrays too

- Mutating an array in-place doesn't change its identity
 - If the array changes (or something in it changes), make a copy of the array
 - e.g.
 - Array.map
 - Array.filter
 - Array.slice
 - Spread syntax e.g. `[...originalArray]`



Quiz

What's the output?

```
1 const ws = [1,2,3];
2 const xs = ws;
3 xs[0] = 500;
4 console.log(` Is xs ws? ${Object.is(xs,ws)})`)

5
6 const ys = ws.map(w=>w);
7 console.log(` Is ys ws? ${Object.is(ys, ws)})`);

8
9 const zs = ws.filter(w=>true);
10 console.log(` Is zs ws? ${Object.is(zs, ws)})`);
```

Fail #1

```
1 export const Planets = () => {
2   const [planets, setPlanets] = useState(['Mercury', 'Venus', 'Earth', 'Mars']);
3   return (
4     <>
5       {planets.map((planet, i)=>
6         <button onClick={
7           ()=>planets.splice(i,1)
8         }
9       > Nuke {planet}
10      </button>
11    )})
12   </>
13   );
14 );
15 );
```

Fail #2



```
1 export const Planets = () => {
2   const [planets, setPlanets] = useState(['Mercury', 'Venus', 'Earth', 'Mars']
3   return (
4     <>
5       {planets.map((planet, i) => (
6         <button onClick={() => {
7           planets.splice(i, 1);
8           setPlanets(planets);
9         }}
10        >Nuke {planet}</button>
11      ))}
12    </>
13  );
14};
```

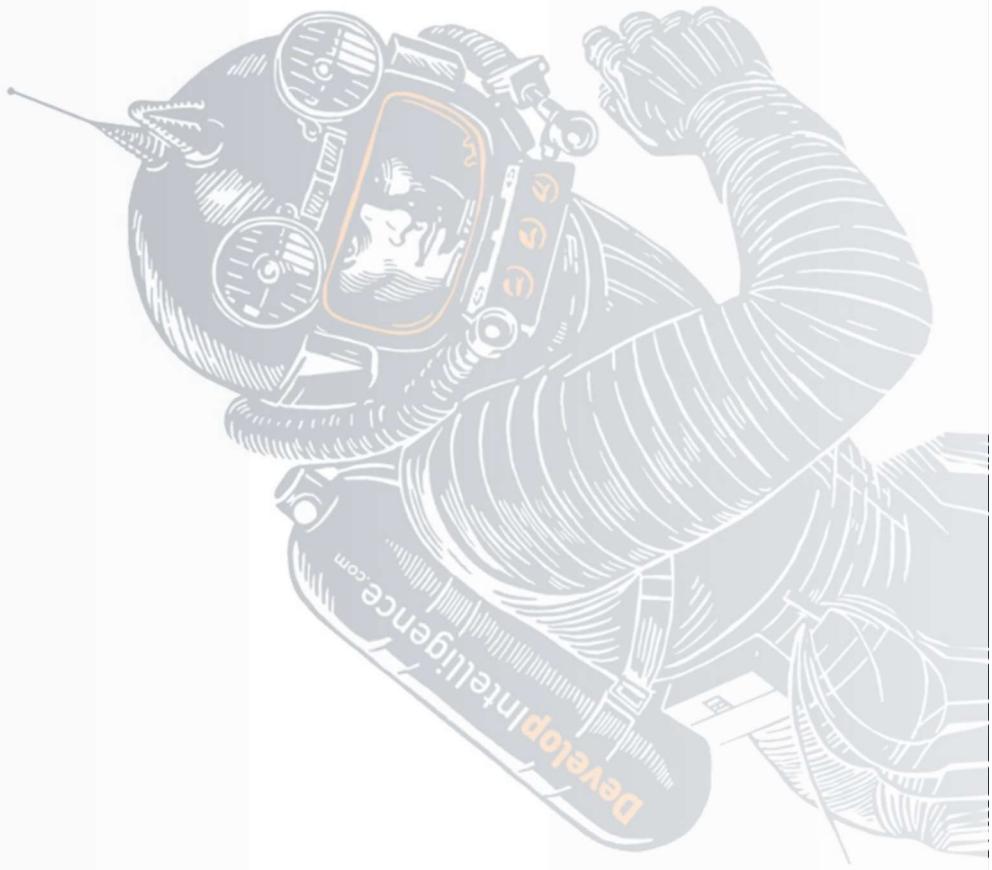


Success #1

```
1 export const Planets = () => {
2   const [planets, setPlanets] = useState(['Mercury', 'Venus', 'Earth', 'Mars']
3   return (
4     <>
5       {planets.map((planet, i) => (
6         <button onClick={() => {
7           planets.splice(i, 1);
8           setPlanets([...planets]); //Update with shallow copy!
9         }}
10        >Nuke {planet}</button>
11      ))}
12    </>
13  );
14};
```

Success #2

```
1  export const Planets = () => {
2    const [planets, setPlanets] = useState(['Mercury', 'Venus', 'Earth', 'Mars']
3    return (
4      <>
5        {planets.map(planet => (
6          <button
7            onClick={
8              event => setPlanets(planets.filter(p => p !== planet))
9            } //Filter creates a new array.
10         >
11           Nuke {planet}
12           </button>
13         ))}
14       </>
15     );
16   );
```

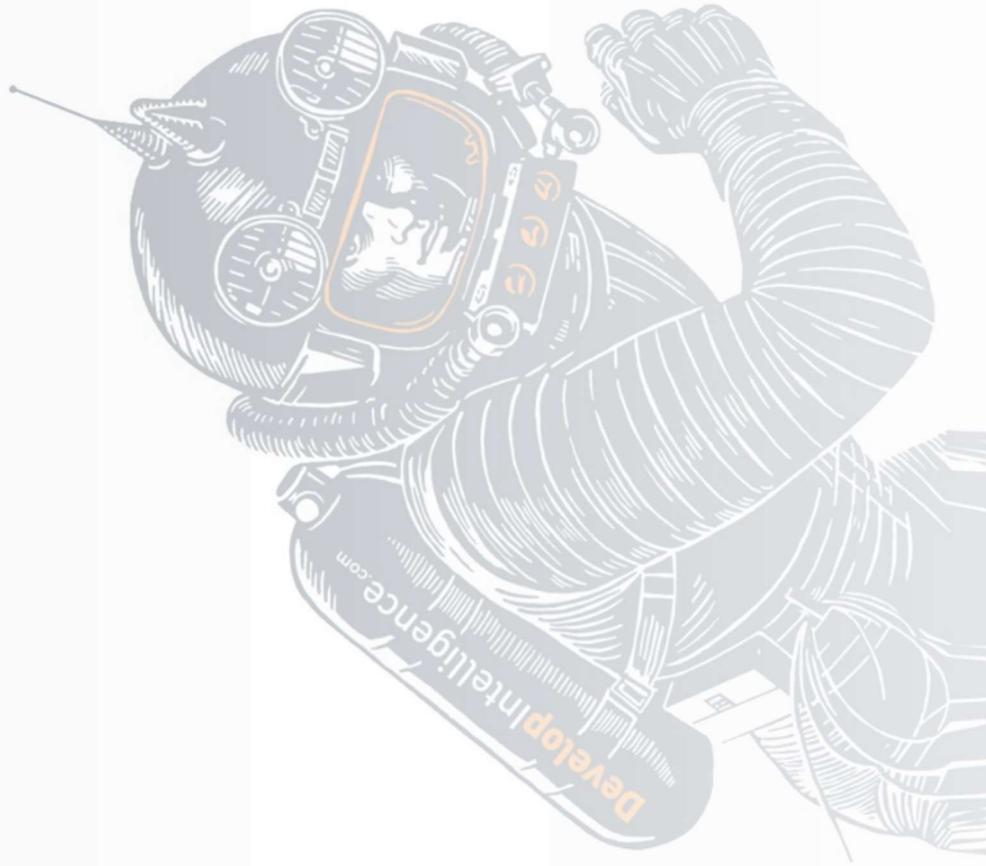


Develop
Intelligence



Reducers

Develop Intelligence





Overview



Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called a reducer.

-- [React.dev](#)



What's a 'reducer'?

- Function that takes (a) the current state and (b) a piece of data to update the state, returning a new state.
- A.k.a **Fold**
- e.g. ECMAScript's Array.reduce

```
1 const sum = (xs:Array<number>)
2 =>xs.reduce(
3   //This is the reducer function
4   (state,x)=>state+x, // Create a new state with additional item
5   0 // This is the initial state
6 );
7
8
9 console.log(sum([1,2,3,4,5]))
```



Command Pattern

Category	Behavioral
Problem	<ul style="list-style-type: none">• Initiating a request tightly coupled it its fulfillment.• You want a history of actions• Other components want to know
Solution	<ul style="list-style-type: none">• Create an abstraction to represent the action





React's `useReducer`

Reducers are a different way to handle state. You can migrate from `useState` to `useReducer` in three steps:

- 1. Move from setting state to dispatching actions.*
- 2. Write a reducer function.*
- 3. Use the reducer from your component.*



Dumb Example

```
1 const counterReducer=(state:number,action?:unknown)=>{
2   return state+1;
3 }
4
5 export const Counter = () => {
6   const [i, increment] = useReducer(counterReducer,0);
7   return (
8     <>
9       <h1>Your number is: {i}</h1>
10      <button onClick={increment}>Increment</button>
11      </>
12    );
13  };
```



Better Example: Planets

- Look under `src/demos/planets/`
- Two actions:
 - `UpdatePlanetDescriptionAction`
 - `DestroySatelliteAction`
- Reducer: `planetsReducer`
- Typescript helps a lot





Advantages of `useReducer`

1. Testability
2. Fewer bugs (?)
3. Separation of concerns



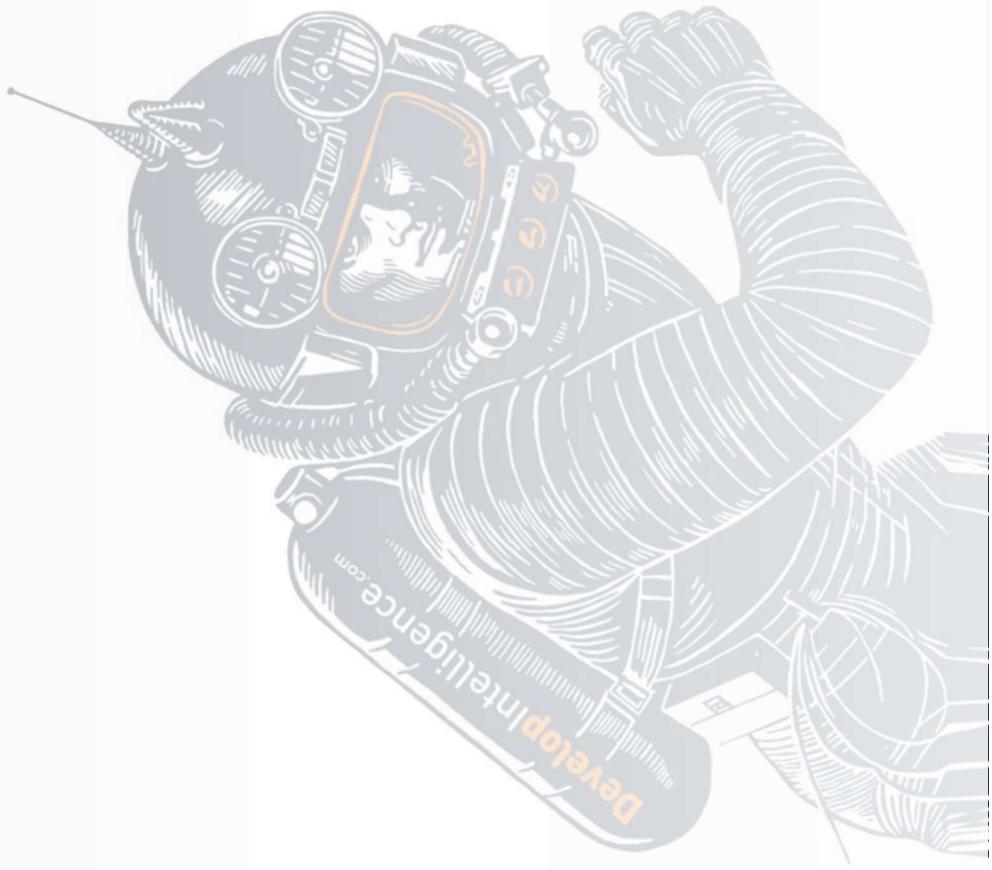
Lab: Word Processor

Instructions: src/practice/word-processor

Summary:

- Simple word processor
- Display word count
- Enable simple editing
 - Color





Develop Intelligence



Review



1. Describe useState
2. Explain userReducer
3. List the **rules of hooks**

