How to State

If React is **declarative**, how do we manage state?

- hooks!
    - outside functions to read/write state changes
- render JSX with current state
- event listeners (using onXXX) update state

Input Example

```
import { useState } from 'react';

function App() {
  const [name, setName] = useState('');
  return (
    <div className="app">
      <p>Last name seen was {name}</p>
      <label>
        <span>Name: </span>
        <input
          value={name}
          onInput={ (e) => setName(e.target.value) }
        />
      </label>
    </div>
  );
}
export default App;
```

SO MUCH - import

```
import { useState } from 'react';
```

This is one of those "other" ways to import

- A file can have one "default" export
    - import and give a name of your choice
- A file can have many "named" exports
    - that you import inside `{}` using their name
    - you can change it with `as`:

```
import { useState as someOtherVar } from 'react';
```

- importing from a library (react) involves no path

SO MUCH - array destructure

`const [name, setName] = useState('');`

`useState()` returns an array

Above code is the same as:

```
const returnedArray = useState('');
const name = returnedArray[0];
const setName = returnedArray[1];
```

`useState()` *always* returns two values

SO MUCH - useState returns

`useState()` always returns two values:

- a value
- a setter function

The value is the last value set with setter function

- defaults to value passed to `useState()`
- value passed to `useState()` ignored once setter called

SO MUCH - automatic rerender

When a state setter function is called

- output re-renders
- no need to call render()
- Component IS a render() function

SO MUCH - onInput

```
<input
  value={name}
  onInput={ (e) => setName(e.target.value) }
/>
```

- `name` will always be latest value
- `onInput()` runs whenever there is typing
    - including backspace/delete
- `e.target` is the input field here
- notice the self-closing input tag!
    - React translates to actual HTML

More Example

```
function App() {
  const [inProgress, setInProgress] = useState('');
  const [saved, setSaved] = useState('');
  return (
    <div className="app">
      <p>Name in progress is {inProgress}</p>
      <p>Last Saved name was {saved}</p>
      <label>
        <span>Name: </span>
        <input
          value={inProgress}
          onInput={ (e) => setInProgress(e.target.value) }
        />
        <button
          type="button"
          onClick={ () => setSaved(inProgress) }
        >Save</button>
      </label>
    </div>
  );
}
```

Two useState()s

```
const [inProgress, setInProgress] = useState('');
const [saved, setSaved] = useState('');
```

Each `useState()` will track a separate value

- order in file in meaningful
- you can't put useState() inside an `if(){}`

Different State Updates

```
<input
  value={inProgress}
  onInput={ (e) => setInProgress(e.target.value) }
/>

<button
  type="button"
  onClick={ () => setSaved(inProgress) }
>Save</button>
```

- One "as you type"
- One "after you click"

Components can call other components

```
import Switch from './Switch';

function App() {
  const [isOn, setIsOn] = useState(false);
  return (
    <div className="app">
      <Switch isFlipped={isOn}/>
      <button onClick={ () => setIsOn(!isOn) } >Flip</button>
    </div>
  );
}
```

```
function Switch({ isFlipped }) {
  const switchState = isFlipped ? "switch--on" : "";
  return (
    <div className="switch__container">
      <div className={`switch ${switchState}`}/>
    </div>
  );
}
```

Component calls other component

```
import Switch from './Switch';

function App() {
  const [isOn, setIsOn] = useState(false);
  return (
    <div className="app">
      <Switch isFlipped={isOn}/>
      <button onClick={ () => setIsOn(!isOn) } >Flip</button>
    </div>
  );
}
```

Both `App.jsx` and `Switch.jsx` are components

- No limits to putting them together

State became a prop

```
const [isOn, setIsOn] = useState(false);
return (
  <div className="app">
    <Switch isFlipped={isOn}/>
```

- `isOn` state passed to `<Switch` as a prop
- name of prop changed! (`isFlipped`)
  - doesn't have to
  - passing a parameter to a function
  - new variable, can be same or different name

Component ignorant of source of prop

```
function Switch({ isFlipped }) {
  const switchState = isFlipped ? "switch--on" : "";
  return (
    <div className="switch__container">
      <div className={`switch ${switchState}`}/>
    </div>
  );
}
```

- Doesn't know `isFlipped` was set by state
- Rerendered when parent rerendered
- Notice template literal `` ` ` `` with `switchState`
- Used to embed in string

Showing a list

```
function App() {
  const [todos, setTodos] = useState([
    'Pounce',
    'Chase Laser Pointer',
    'Nap',
  ]);
  return (
    <div className="app">
      <TodoList list={todos}/>
    </div>
  );
}
```

```
function TodoList({ list }) {
  const items = list.map(
    item => ( <li>{item}</li> )
  );
  return (
    <ul className="todos">
      {items}
    </ul>
  );
}
```

Check the console for errors and warnings!

- Warning: `'setTodos' is assigned a value but never used no-unused-vars`
- Error: `Warning: Each child in a list should have a unique "key" prop.`

Why does the Error say "Warning"? Grr.

- Warnings don't prevent things from working, but may indicate a problem
    - This is coming from the linting tool, which has a rule about unused variables
- Errors indicate something definitely wrong

Rendered lists and "key" prop

Rendered lists in React need a "key" prop

- React does comparison logic to decide what to actually change in DOM
    - Delete item 5 out of 10: looks like changed 5 items and deleted last
- key props allow to see what really changed
    - must be unique
    - must stay the same between renders
        - generally bad to use index

Fixing our key prop

```
function TodoList({ list }) {
  const items = list.map(
    item => ( <li key={item}>{item}</li> )
  );
  return (
    <ul className="todos">
      {items}
    </ul>
  );
}
```

- Unique `key` prop added

Understanding the List

```
function TodoList({ list }) {
  const items = list.map(
    item => ( <li key={item}>{item}</li> )
  );
  return (
    <ul className="todos">
      {items}
    </ul>
  );
}
```

- map list of items to list of JSX elements
- NO JOIN
- NOT A STRING
- embed list in JSX

Arrays and Objects as State

`useState()` can manage any type of JS value

- including arrays and objects

React assumes the state only changes when you call the setter function.

- This means arrays and objects can be a problem
- You can **mutate** these by changing an element/property
    - without calling the setter function
- This would confuse React

Solution: Don't do that

Treat arrays and objects in state as **immutable**

- No React confusion

But how do you change the state?

- pass a NEW array/object to the setter function

Updating array in state example

```
function SomeComponent() {
  const [todos, setTodos] = useState(['nap','sleep','rest']);
  const [newTodo, setNewTodo] = useState('');
  return (
    <div>
      <TodoList list={todos}/>
      <input
        value={newTodo}
        onInput={ (e) => setNewTodo(e.target.value) }
      />
      <button
        onClick={ () => setTodos( [...todos, newTodo]);
      />
    </div>
  );
}
```

Setting a new array

- Setting the state to a new array using `[]`
- Using the **spread** operator (`...`)
  - fills new array with contents of existing array
  - copies array

https://beta.reactjs.org/learn/updating-arrays-in-state

Replacing array mutations for state update

Changing an element:

- DO NOT set the element to a new value
- DO copy the array, change copy, set state to copy

Adding an element:

- DO NOT use `.push()` or `.unshift()`
- DO use spread (`...`) or `.slice()` (to copy array)

Removing an element:

- DO NOT use `.pop()` or `.shift()`
- DO use `.slice()` or alter a copy

Updating object in state example

```
function SomeOtherComponent() {
  const [student, setStudent] = useState({
    name: 'Jorts', grade: '87'
  });
  const [grade, setGrade] = useState(student.grade);
  return (
    <div>
      <div>Name: {student.name}</div>
      <div>
        Grade:
        <input
          value={grade}
          onInput={ (e) => setGrade(e.target.value) }
        />
      </div>
      <button
        onClick={ () => setStudent( {...student, grade }); 
      />
    </div>
  );
}
```

Setting a new object

- Setting the state to a new object using `{}`
- Using the **spread** operator (`...`)
  - fills new object with existing object contents
  - copies object

https://beta.reactjs.org/learn/updating-objects-in-state

Any key/value pairs after spread op override key/values in copied object

More about Object copying

```
onClick={ () => setStudent( {...student, grade });
```

Remember this is the same as saying:

```
onClick={ () => setStudent({
  ...student,
  grade: grade,
});
```

`grade` property gets the value of the `grade` variable

- and here, overrides any `grade` key/value pair in `student`

Replacing object mutations for state update

Changing an element:

- DO NOT set the element to a new value
- DO copy the object, change copy, set state to copy

Adding an element:

- DO NOT define the new property value
- DO use spread (`...`) (to copy object)

Removing an element:

- DO NOT use `delete` on object property
- DO alter a copy, set new state as copy

This can feel daunting

But the rules itself is straight-forward

- Do not change an array/object that is in state
- set state to a new array/object
    - that was set from the existing array/object
    - and has the changes

How to show different content sometimes

What if you want to have different options for content

- Example: Login form vs content + Logout?

A Conditional Example

```
const [isLoggedIn, setIsLoggedIn] = useState(false);
const [username, setUsername] = useState('');
return (
  <div className="app">
  { isLoggedIn
    ? <div>
        Hello {username}
        <button onClick={() => setIsLoggedIn(false)}>Logout</button>
      </div>
    : <form>
        <label>
          <span>Username: </span>
          <input value={username} onInput={(e) => setUsername(e.target.value)}/>
        </label>
        <button type="button" onClick={() => setIsLoggedIn(true)}>Login</button>
      </form>
   }
  </div>
);
```

A Different Conditional Example

```
const [isLoggedIn, setIsLoggedIn] = useState(false);
const [username, setUsername] = useState('');
const content =
( <div>
  Hello {username}
  <button onClick={() => setIsLoggedIn(false)}>Logout</button>
</div>);
const login =
(<form>
  <label>
    <span>Username: </span>
    <input value={username} onInput={(e) => setUsername(e.target.value)}/>
  </label>
  <button type="button" onClick={() => setIsLoggedIn(true)}>Login</button>
</form>);

return (
  <div className="app">
  { isLoggedIn ? content : login }
  </div>
);
```

Yet Another Conditional Example

```
const [isLoggedIn, setIsLoggedIn] = useState(false);
const [username, setUsername] = useState('');
let content;
if (isLoggedIn) {
  content = ( <div>
    Hello {username}
    <button onClick={() => setIsLoggedIn(false)}>Logout</button>
  </div>);
} else {
  content = (<form>
    <label>
      <span>Username: </span>
      <input value={username} onInput={(e) => setUsername(e.target.value)}/>
    </label>
    <button type="button" onClick={() => setIsLoggedIn(true)}>Login</button>
  </form>);
}

return (
  <div className="app"> { content } </div>
);
```

State goes "down"

```
function App() {
  const [todos, setTodos] = useState([
    'Pounce',
    'Chase Laser Pointer',
    'Nap',
  ]);
  return (
    <div className="app">
      <TodoList list={todos}/>
    </div>
  );
}
```

- State is passed "down"
    - to children

What if a child wants to change state?

Child component has no access to setter!

- cannot reach "up"
- Parent must pass some function to change
    - direct setter
    - OR wrapper of direct setter

A Better Conditional Example

```
import Content from './Content';
import Login from './Login';

function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  const [username, setUsername] = useState('');
  return (
    <div className="app">
      { isLoggedIn
        ? <Content
            username={username}
            setLoggedIn={setLoggedIn}
          />
        : <Login
            username={username}
            setUsername={setUsername}
            setLoggedIn={setLoggedIn}
          />
      }
    </div>
  );
}
```

The other components

```
function Content({ username, setLoggedIn }) {
  return ( <div>
    Hello {username}
    <button onClick={() =>
      setIsLoggedIn(false)}>Logout</button>
  </div>);
}
```

```
function Login({ username, setUsername, setIsLoggedIn }) {
  return ( <form>
    <label>
      <span>Username: </span>
      <input value={username}
        onInput={(e) => setUsername(e.target.value)}/>
    </label>
    <button type="button"
      onClick={() => setIsLoggedIn(true)}>Login</button>
  </form>);
}
```

You can be more generic

```
    const onLogin = (username) => {
      setUsername(username);
      setIsLoggedIn(true);
    };
    const onLogout = () => setIsLoggedIn(false);

    return (
      <div className="app">
        { isLoggedIn
          ? <Content
              username={username}
              onLogout={onLogout}
            />
          : <Login
              onLogin={onLogin}
            />
        }
      </div>
    );
  }
```

The more generic parts

```
function Content({ username, onLogout }) {
  return ( <div>
    Hello {username}
    <button onClick={onLogin}>Logout</button>
  </div>);
}
```

```
function Login({ onLogin }) {
  const [username, setUsername] = useState('');
  return ( <form>
    <label>
      <span>Username: </span>
      <input value={username}
        onInput={(e) => setUsername(e.target.value)}/>
    </label>
    <button type="button"
      onClick={() => onLogin(username)}>Login</button>
  </form>);
}
```

Each component can have state

See the `useState()` here!

- distinct from the username of `App`
- allows for custom behavior

```
function Login({ onLogin }) {
  const [username, setUsername] = useState('');
  return ( <form>
    <label>
      <span>Username: </span>
      <input value={username}
        onInput={(e) => setUsername(e.target.value)}/>
    </label>
    <button type="button"
      onClick={() => onLogin(username)}>Login</button>
  </form>);
}
```

Extra useState notes

Some details aren't needed for all useState uses

- but they are good to know about

Expensive Initialization

Common question:

What is the value passed to useState?

- As mentioned, the very first, initial, value
- ignored afterwards
- still evaluated every render

```
const [value, setValue] = useState( calcValues() );
```

- Can avoid repeat calculations by passing a function:

```
const [value, setValue] = useState( () => calcValues() );
```

Current state doesn't change!

What is wrong here?

```
const [count, setCount] = useState(0);

return (
  <div onClick={
    () => {
      setCount(count + 1);
      setCount(count + 1);
    }
  }>
    {count}
  </div>
);
```

Passing function to setter

A callback passed to the setter

- will be called with the current value

```
const [count, setCount] = useState(0);

return (
  <div onClick={
    () => {
      setCount( prevCount => prevCount + 1);
      setCount( prevCount => prevCount + 1);
    }
  }>
    {count}
  </div>
);
```

Often helpful with objects/arrays

- if doing multiple changes not all at once

Summary - Hooks

- Hooks are functions for React that
    - manage state
    - and/or interact with render cycle

Summary - useState Basics

- `import { useState } from 'react';`
- each `useState()` creates a distinct value
- `useState()` returns an array
    - containing the current value
    - and setter function to change stored vale
    - always destructured into two variables
- `useState()` passed an initial value
    - only used the first render
    - can be a function
        - when getting initial value is expensive

Summary - Calling useState()

- multiple useState() = multiple distinct state values
    - cannot be inside an `if`
- Assign meaningful variable names
    - React doesn't "know" meaning, only order
- Current value only "changes" when useState() called!
- Each component can have their own state
    - You should scope to who "owns" a value
- Parents can pass state values as props to children

Summary - Calling state setter

Calling the setter returned by `useState()`

- Sets the new value for the NEXT render
- Queues a new render
    - AFTER current code finishes
    - If all renders change state, infinite loop
- Can be passed a callback
    - will be called with current/pending state value

Summary - Setting an object/array

Current state should be **immutable**

- numbers, strings, booleans already are
- objects and arrays can mutate
    - so you should make sure not to do that
- updates are setting to new object/array
    - populated with original object/array
        - common to use "spread" operator (`...`)
    - except for changed values

Summary - Conditional Rendering

Can hide/show section with CSS

- by deciding current classes
- React can add/remove classes
    - but you will redeclare output HTML

Simply include/omit Components/HTML

- Common: we redeclare output HTML anyway
- Can't have if/then inside `{}` in JSX
    - if/then doesn't return a value
- Conditional operator (`? :`) does
- Or set variables and substitute those

Summary - Events causing state changes

- form fields (`<input>`, `<select>`, etc)
    - you set the `value` prop to current state value
    - onInput/onChange read `e.target.value`
    - feels "heavy", but browser does anyway
- other interactions (click, etc)
    - Call setter function in event handler callback

Summary - Component state vs Application state

Each component can have state

If state is only meaningful to component and children

- manage that state in that component If state is used in many components
- manage at a common ancestor component

Usually

- top level component has "application state"
- lower components manage temporary limited state
    - values that are being typed
    - UI-related state for a section
        - Ex: Is a section open vs collapsed