How to create State

React Component functions run each time we render

- How do we get variables with persistent values?
- **hooks**!
    - Outside functions to read/write state changes
- JSX **renders with current state**
- Event listeners (using onEVENT) **update state**
- JSX **automatically rerenders** when state changes

State Example

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

function App() {
  const [count, setCount] = useState(0);
    return (
    <>
      <div className="card">
        <button
            onClick={() => setCount((count) => count + 1)}
        >
          count is {count}
        </button>
      </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 [count, setCount] = useState(0);`

`useState()` returns an array

Above code is the same as:

```
const returnedArray = useState(0);
const count = returnedArray[0]; // value from state
const setCount = returnedArray[1]; // setter function
```

`useState()` *always* returns array of two values

- We **destructure** to declare and assign 2 variables

SO MUCH - useState returns

`useState()` always returns array of two values:

- **value from state**
- **setter function**

Value is:

- The last value set with setter function
- If NEVER set, uses value passed to `useState()`
- Value passed to `useState()` ignored once setter called

SO MUCH - onClick

```
<button
    onClick={() => setCount((count) => count + 1)}
>
    count is {count}
</button>
```

Let's simplify to better understand:

```
<button onClick={() => setCount(count + 1)} >
    count is {count}
</button>
```

Simplified onClick

```
<button onClick={() => setCount(count + 1)} >
    count is {count}
</button>
```

- `count is {count}` will show "current" `count`
- `onClick()` is passed a callback handler function
    - Just like a `click` event listener
- Handler function calls `setCount()`
- `setCount()` changes stored state value
- Triggers re-render

Notice the difference here

```
<button onClick={() => setCount(count + 1)} >
    count is {count}
</button>
```

- `onClick` is passed a function callback to call
- `setCount()` called **when** that callback is called

```
<button onClick={setCount(count + 1)} > /* BAD */
    count is {count}
</button>
```

- `onClick()` is passed result of calling `setCount()`
- `setCount()` is called IMMEDIATELY
- `setCount()` triggers rerender, calls `setCount()`
- Web app crashes (infinite loop)

Passing Function Wrapper

- **Event handlers are passed a function to run**
  - Just like `.addEventListener`
- NOT result of calling a function immediately

```
/* Correct Version: */
<button onClick={() => setCount(count + 1)} >
    count is {count}
</button>

/* Bad Version: */
<button onClick={setCount(count + 1)} >
    count is {count}
</button>
```

SO MUCH - automatic rerender

`setCount()` changes the value of `count` in state

- Page shows changed HTML/text!

When a state setter function is called

- Output **automatically** re-renders

When does state variable change?

```
function App() {
    const [count, setState] = useState(0);

    console.log("on render", count);

    return (
        <button onClick={ () => {
            console.log("before setter", count);
            setCount(count + 1);
            console.log("after setter", count);
        }} >
            count is {count}
        </button>
    );
}
```

```
on render 0
before setter 0
after setter 0
on render 1
```

Important State Update Confusion!

`setCount()` does NOT change `count`

```
<button
  onClick={() => {
      setCount(count + 1);
      console.log(count);
  }}
>
  count is {count}
</button>
```

- `console.log()` shows that `count` didn't change!
- But page shows that `count` DID change?!

State isn't actually IN component

- Component function called after state changes
- Component **gets a copy of state** from `useState()`
- Setter updates state **outside of component**
    - Queues up new call to component function
        - To render HTML
    - Doesn't happen until current code finishes
    - Copies of state values are STALE until then
- **https://react.dev/reference/react/useState#ive-updated-the-state-but-logging-gives-me-the-old-value**

Passing a function to a setter?

What does this mean?

- `setCount((count) => count + 1)`

Consider:

```
<button onClick={() => {
    setCount(count + 1);
    setCount(count + 1);
}} >
  count is {count}
</button>
```

- Page shows count only going + 1
- Because `count` is a stale copy of state

Why pass a function to a state setter

You can pass a value to a state setter

- `setState(count + 1)`
- Value will be new value for state

You can also pass a function to a state setter

- `setState( (count) => count + 1 )`
- Passed function is itself passed current state value
    - ACTUAL current value of state, not copy
- Passed function should return new value for state

Results of passing function to setter

```
<button onClick={ () => {
    setCount( count => count + 1);
    setCount( cat => cat + 1);
}} >
  count is {count}
</button>
```

- Now increases by 2
- Functions were passed ACTUAL value of state
    - Not the possibly stale copy that is `count`
- param name in passed function just a name
    - In its own scope
    - That's why `cat` still changed `count` state
    - But please use meaningful variables

Another example

State values can be any value, not just numbers!

Let's consider an example with text

Input Example

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

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

SO MUCH - onInput

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

- `name` will always be latest value
- `onInput()` runs whenever there is typing
    - `input` event
    - Including backspace/delete
- `e.target` is the input field here
- Notice the self-closing input tag!
    - JSX requires a close

Putting the Parts together

- When App() is called (when `<App/>` renders
    - `name` is set to `''`
    - HTML renders to the screen
    - `<input>` has value `''`
- User types 'J'
    - `onInput` callback fires
        - calls `setName` with 'J'
- Change in state triggers rerender (App() is called)
    - `name` is set to 'J'
    - HTML renders `<input>` with value = 'J'

Why State?

Remember the concept we are using

- State is variable(s) of values that can change
- **Rendering** is setting HTML based on state
- Events will change state
- After state changes, **render**

True both in React and in advanced plain JS SPAs

Every component defines part of HTML

- Based on state and props

Revisit Example

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

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

Component is output HTML

- Based on current state/props
- Defines event handlers
- Event Handlers can change state
    - Which would cause new **render**
    - Which would reflect updated state

More Example

```
function App() {
  const [inProgress, setInProgress] = useState('');
  const [saved, setSaved] = useState('');
  return (
    <>
      <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>
    </>
  );
}
```

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"

See the State-Render cycle at work

- We have State variables and props
- The output HTML is based on the variables
- User events change the state
- Output HTML is automatically updated
    - Based on new state

Trigger for render was the change in state

- Not the user event
- User event was the trigger for the change in state

Components can call other components

```
import ShowThing from './ShowThing';

function App() {
  const [isOn, setIsOn] = useState(false);
  return (
    <>
      <ShowThing isImpressive={isOn}/>
      <button onClick={ () => setIsOn(!isOn) } >Flip</button>
    </>
  );
}
```

```
function ShowThing({ isImpressive }) {
  const className = isImpressive ? "thing--on" : "";
  return (
    <div className="thing__container">
      <div className={`thing ${className}`}/>
    </div>
  );
}
```

Component calls other component

```
import ShowThing from './ShowThing';

function App() {
  const [isOn, setIsOn] = useState(false);
  return (
    <>
      <ShowThing isImpressive={isOn}/>
      <button onClick={ () => setIsOn(!isOn) } >Flip</button>
    </>
  );
}
```

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

- Components can have Components amidst HTML
- Can have multiple instances of any Component

State became a prop

```
const [isOn, setIsOn] = useState(false);
return (
  <>
    <ShowThing isImpressive={isOn}/>
```

- `isOn` state passed to `<ShowThing>` as a prop
- name of prop changed! (`isImpressive`)
    - Does not need to change/stay the same
    - Passing a parameter to a function
    - New variable, can be same or different name
    - Does MATTER! A lot!
        - Some names are better changed
        - Some names are better staying the same

Component ignorant of source of prop

```
function ShowThing({ isImpressive }) {
  const className = isImpressive ? "thing--on" : "";
  return (
    <div className="thing__container">
      <div className={`thing ${className}`}/>
    </div>
  );
}
```

- Doesn't know `isImpressive` was set by state
    - That's good. **decoupled**
- Rerendered when prop changes
- Notice template literal `` ` ` `` with `className`
- Used to embed in string

Showing a list

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

```
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: Each child in a list should have a unique "key" prop.`
- Warnings don't prevent things from working
    - Do indicate a problem
- Errors indicate something definitely wrong
- Fix BOTH Warnings and Errors immediately
    - Don't ignore because things are "working"
    - Regret will come if you let them accumulate

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 5,6,7,8,9 changed; 10 deleted
- `key` props allow to see which really changed
    - Must be unique within list
    - 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!
- Embedded list in JSX

JSX removes leading/trailing spaces

```
function App() {
  return (
    <div>
       This is on a new line
       This has     many     spaces
    </div>
  );
}
```

Results

```
<div>This is on a new line This has many spaces</div>
```

- Unlike HTML, no leading/trailing space/newline
- 99% of the time this is GREAT!

When React is too eager to remove whitespace

Sometimes React removes expected whitespace:

```
function App() {
    return (
        <>
            <span>Name:</span>
            <span>Jorts</span>
        </>
    );
}
```

- You might expect: `Name: Jorts` in the browser
- Instead you see: `Name:Jorts`

Forcing Spaces

```
function App() {
    return (
        <>
            <span>Name:</span>
            {' '}
            <span>Jorts</span>
        </>
    );
}
```

Sophisticated Output

React does not render `false`, `null`, or `undefined`

```
function App() {
  return (
    <>
        Testing output
        { <p>Test</p> }
        { false }
        { null }
        { undefined }
        { 0 }
    </>
  );
}
```

Both `<p>Test</p>` and `0` will render

- `false`, `null`, `undefined` do NOT

Using Short-Circuiting

Remember when we said `&&` and `||` "short-circuit"?

- Return the left-side or right-side value

React does not render `false`, `null`, or `undefined`

- Combine with `&&` or `||` inside `{}`!

Alternatively, use **conditional operator** (`? :`)

- Cannot use `if (condition)` inside `{}`
- Can use `{ condition ? <p>Was Truthy</p> : <p>Was Falsy</p> }`

Conditional Rendering

Conditional Rendering = Deciding what to show

```
function App() {
  const [count, setCount] = useState(0);
  return (
    <>
      <p>Count is {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increase
      </button>
      { count === 8 && <p className="wild">8!!!</p>}
      { count ? <p>Count is truthy</p> : null}
      { count && <p>Dang that 0!</p> }
    </>
  );
}
```

0/NaN WILL render!

Conditional Rendering is great

- But remember **some falsy values WILL render**
- Notably `0` and `NaN`
- Option: Use conditional operator
- Option: Convert to boolean

```
// Bad!
{ messages.length && <p>You've got mail!</p> }
// Good!
{ messages.length !== 0 && <p>You've got mail!</p>}
{ !!messages.length && <p>You've got mail!</p>}
{ messages.length ? <p>You've got mail!</p> : null }
```

Composing Content

How to organize when you have options for content?

- Example:
    - If user is NOT logged in:
        - Show Login Form to login
    - If user IS logged in:
        - Show "content"
        - Show Logout button

A Conditional Example

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

That was messy

- Worked
- Hard to read
- Annoying to decipher

Solution: Move parts to different components

Conditional Rendering of "Pages"

- SPA is a "single page"
- We can change content
- Sometimes a little content
- Sometimes a lot of content
- Sometimes EVERYTHING

Our app can show different "pages" based on state

- Completely different "pages"
- Or just different parts
- "Screens", "views", "pages"
    - No actual terminology

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" to variables in parent
- Parent must pass some function to change
    - Direct setter (Ex: `setName`, etc)
    - OR wrapper of direct setter

Example of passing wrapper of setter function

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

  const catify = () => { // "wrapper" around setName
    setName('Jorts');
  }

  return (
    <>
        <p>Name is {name}</p>
        <CatifyButton catify={catify} />
    </>
  );
}

function Catify({ catify }) {
    return (
        <button onClick={catify}>Catify!</button>
    );
}
```

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={onLogout}>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>);
}
```

Where should you `useState()`?

- Generally, declare that the "nearest common ancestor" of all Components that need that state

```
stateA is used by ComponentB and ComponentC

ComponentC is a "child" of ComponentD
ComponentB is a "child" of ComponentE
ComponentE is a "child" of ComponentD
```

ComponentD is the "nearest common ancestor"

- Have `useState()` for stateA in ComponentD
- Pass state and any setters/wrappers from ComponentD to child elements

Often a LOT of state ends up at "top"

- Most state lives in App.jsx
    - Most state matters to most Components
- Temp state like "as you are typing" username
    - Kept out of App.jsx
    - Declared in their specialized components
    - Any "final" version passed to handlers received from ancestor
        - Ex: Login sends FINAL username to App
        - Using the `onLogin` prop it was passed

Summary - State

- `import { useState } from 'react';`
- `useState()` is a React **hook**
- Pass `useState()` initial value for a state variable
- Returns array of two parts
    - We **destucture** array into two variables
    - State value (a COPY)
    - Setter function
- State value will be:
    - Last value passed to setter function
    - useState() argument if setter never called

Summary - Changing State

- Component returns HTML based on state
    - **conditional rendering**
- Can have multiple `useState()` calls
    - Each a different state variable
- When state changes, component **rerenders**
- set `onEVENT` (onClick, onSubmit, etc) props
    - If set on "native" HTML element
        - Callback called when event on element
    - Callback can call setter to change state

Conditional Rendering

- React renders value from `{}`
    - Will not render `false`, `null`, `undefined`
- Short Circuiting in `{}` can conditionally render JSX
    - Watch out for `0` or `NaN`
    - Convert to boolean OR
    - Use conditional operator
- Conditional operator in `{}` conditionally renders
- Can't use `if ()` in `{}`

Summary - Passing State

A Component

- Can pass state as props to other components
- CANNOT call setter functions they don't have
- CAN be passed functions as props
- CAN pass setter functions to other components
- CAN pass wrapper functions to other components