The State-Render Cycle

- We react to events
- We read form fields (as needed)
- Use `.classList` to change classes on elements
- Use `.innerText`/`.innerHTML` to change text/HTML

This works, but:

- HTML/text/classes scattered among HTML & JS
- Grows more complex as app grows
- Too complex at large sizes

Addressed by **state-render cycle**

What is "State"

- Everything about your app that can change
- Think a vending machine
    - Tracks how much money you've put in (state)
    - Has a limited number of ways it can change
- Web apps have state
    - Values entered into form fields
    - Changing concepts based on "clicks"
        - Ex: "open", "hidden", "selected"

Core State-Render loop

- App has variables for current state
- Events can change state values
    - State changes trigger **render**
- "render" **replaces** HTML chunk
    - New HTML is based on current state

Simple state-render example

```
const contentEl = document.querySelector('.content');
const catNames = ['Jorts', 'Jean', 'Nyancat', ];
let catNum = 0;
render();

contentEl.addEventListener('click', (e) => {
  if(!e.target.classList.contains('change-cat')) {
    return;
  }
  catNum = catNum < (catNames.length - 1) ? catNum + 1 : 0;
  render();
});

function render() {
  contentEl.innerHTML = `
    <div>Your cat is ${catNames[catNum]}</div>
    <button class="change-cat" type="button">Next</button>
  `;
}
```

Explaining the simple example

- Our state is `catNum`
- User actions can change state
    - +1 to catNum, wrap around to 0
- When state changes, we call `render()`
- `render()` replaces innerHTML of `.content`

Differences when using state-render

- We replace a chunk of HTML
  - In extreme cases, entire `<body>` contents
- Write HTML with or without classes
  - Instead of `.classList.add/remove/toggle`
- Attach event listener to ancestor
  - Possibly listen for multiple events
  - Make sure event is on correct element!

An example of removing/adding elements

```
const contentEl = document.querySelector('.content');
const showExample = false;
render();

contentEl.addEventListener('click', (e) => {
  if(!e.target.classList.contains('toggle')) {
    return;
  }
  showExample = !showExample;
  render();
});

function render() {
  contentEl.innerHTML = `
    ${ showExample ? <div>Some Text</div> : "" }
    <button
      class="toggle"
      type="button"
    >${showExample ? "Hide" : "Show"}</button>
  `;
}
```

Important element things to notice

- The `<div>` is present when `showExample === true`
    - And not otherwise
    - Not hidden with CSS
- The `<button>` text changes
- All elements are replaced on each `render()`
    - Even if nothing changed
    - Can't listen for clicks on the button!
        - Different `<button>` each render!

An example with class names

```
const contentEl = document.querySelector('.content');
const isActive = false;
render();

contentEl.addEventListener('click', (e) => {
  if(!e.target.classList.contains('activator')) {
    return;
  }
  isActive = !isActive;
  render();
});

function render() {
  contentEl.innerHTML = `
    <div class="example ${isActive ? 'active' : ''}" >
      Example
    </div>
    <button class="activator" type="button" >Toggle</button>
  `;
}
```

Important class name things to notice

- The class attribute changes!
  - Always `example`
  - Sometimes `active`
  - Not using `.classList`

Why use the state-render cycle?

Pros:

- HTML in one place
    - Class names changes in HTML (one place)
- Scales better as app gets more complex

Cons:

- Many elements are replaced
    - Often without change
    - Can cause problems with form fields
- Must use event delegation to listen for events

A Particular Note

React is a fancy, easier, more efficient version

- We will define **components**
  - These output chunks of HTML
  - That are replaced/changed
  - Output is based on state changes
  - State changes when **events** happen
- We aren't using React YET
  - But we CAN use state-render without React

What are you required to do?

This section of course does NOT require state-render

- Can select elements
    - Update their classes/text
- OR you can select ancestors
    - Replace chunks of HTML
        - Including class names
- Both solutions used on jobs

Our unit 3 will use React which does this for you

- Easier to understand with early practice

Complex decisions MUST use state!

- If you are changing any values
    - AND need to make decision
    - Have variables that track current situation
        - This is **state**
- NEVER read text/classes of HTML to decide
    - This is a proven source of pain
    - `.classList.add/remove/toggle` allowed

Anything more than `.classList.add/remove/toggle`?

- Should track current situation in variable

Summary - State

- Using variables to track current decisions
    - Can be different keys in a state object
    - Allows new decisions based on previous
- Prevents code from growing needlessly complex

Summary - Render

- `render()` is just a name
    - Could be any name
    - Can have many functions
- Creates/replaces HTML based on state
- Keeps HTML/class names in one place
- Often requires event delegation

Summary - Using State-Render Cycle

- Have state variables
- Listen on ancestor for events
    - Filter out irrelevant events
    - Change state as user indicates
    - Call a render-style function
- Render-style functions replace block of HTML
    - Write HTML contents based on style
    - Write HTML class names based on style
    - Write, not change (replacing elements)
- DO NOT use unsanitized user data in output