

# Rendering on the client

Many ways "work"

- This methods SCALES over time

Similar process to server rendering!

- Have state
- Render output based on state
- Update state in response to user actions

Many variations

- Same base pattern

# Example Client App

For now, let's look solely at browser JS

Imagine a list of "todo" items

Each item in the list

- Is done or not done
- Has text for the task

# Consider State without Presentation

```
const todos = [  
  { task: 'Nap', done: false },  
  { task: 'Knock stuff down', done: true },  
  { task: 'Eat', done: false },  
];
```

- This is a useful coding technique in general
- Describe your data model ("shape")
  - Before writing code
- This is an array. Later we'll see an object is better

# What actions can a user take?

- Toggle a task done/not done
- Delete a task
- Add a task

Also "view all tasks", but we'll do that continuously

- Not really an "action"
- Doesn't change state

# Render State to View (HTML)

First pass (additions to come)

```
<ul class="todos"></ul>
```

```
const html = todos.map( todo => `- ${todo.task}</li>`  
).join('');  
  
document.querySelector('.todos').innerHTML = html;

```

# Need to indicate if complete!

```
const html = todos.map( todo => {  
  const doneClass = todo.done ? "complete" : "";  
  return `  
    <li class="todo ${doneClass}">${todo.task}</li>  
  `;  
}).join('');  
  
document.querySelector('.todos').innerHTML = html;
```

```
// CSS  
.complete {  
  line-decoration: line-through;  
}
```

We write class BASED on state

- Not CHANGING class on existing HTML

# Now put it together

- static `index.html`
  - Loads `styles.css`
  - Loads `todos.js` with one of
    - `<script>` with `defer` attribute in `<head>`
    - `<script>` as last child of `<body>`
  - Has the empty `<ul class="todos"></ul>`
- `todos.js`
  - Defines initial state
  - Has a `render()` function
    - Nothing magic about `render()` name
  - Calls `render()` when script loads

# Changing state through events

- Action to toggle "done" on an item
  - Change the state
  - Call `render()`

What in presentation will trigger this action?

- How about clicking on the todo?



# Small steps: First add click event

Worry about updating state, etc later

- Just capture the click
- `console.log()` to see it worked

One immediate problem

- `<li>` don't exist when the script first runs
- `<li>` are replaced each time `render()` is called
- Many `<li>` would mean many event listeners

# Solution: Event Propagation

All `click` events on `<li>`

- Will **propagate** to ancestor `<ul>`

We can add just one listener on the `<ul>`

- Inspect `e.target` to see which `<li>` was clicked
- We don't to react to clicks on the `<ul>` itself
  - Just ones on the `<li>`s

# Click Event listener

```
const listEl = document.querySelector('.todos');
listEl.addEventListener('click', (e) => {
  if( e.target.classList.contains('todo') ) {
    console.log(`click on a <li>`);
  }
});
```

Now we detect clicks

- But WHICH `<li>` was clicked?
- Which todo item does it match to?

We can use `data-` attributes and `.dataset`

# Rendering the data attribute

```
function render() {  
  const html = todos.map( (todo, index) => {  
    const doneClass = todo.done ? "complete" : "";  
    return `  
      <li  
        class="todo ${doneClass}"  
        data-index="${index}"  
      >  
        ${todo.task}  
      </li>  
    `;  
  }).join('');  
  
  document.querySelector('.todos').innerHTML = html;  
}
```

# Reading the .dataset property

- The `data-` name we choose is up to us
- We use `data-index` since our state is an array
  - For objects we'd have the key
- We could call it `data-bob`
  - Terrible name, but allowed

```
const listEl = document.querySelector('.todos');
listEl.addEventListener('click', (e) => {
  if( e.target.classList.contains('todo') ) {
    const index = e.target.dataset.index;
    console.log(`click on <li> index ${index}`);
  }
});
```

# Updating state and re-render

Update state

- then render()

```
const listEl = document.querySelector('.todos');
listEl.addEventListener('click', (e) => {
  if( e.target.classList.contains('todo') ) {
    const index = e.target.dataset.index;
    todos[index].done = !todos[index].done;
    render();
  }
});
```

No need to say what we are changing in output

- Follows automatically from state

# Deleting an item

- Same principle
  - Detect which item to delete
  - Update state
  - Call render()
- But how are we indicating a delete?

Let's add an "X" in the list

- Many options
- Let's add a button in the HTML
  - Can change appearance with CSS

# Rendering the delete button

```
const html = todos.map( (todo, index) => {  
  const doneClass = todo.done ? "complete" : "";  
  return `  
    <li>  
      <span class="todo ${doneClass}" data-index="${index}">  
        ${todo.task}  
      </span>  
      <button  
        data-index="${index}" class="delete" type="button">  
        X  
      </button>  
    </li>  
  `;  
}).join('');
```

- Extra `<span>` not needed, but cleaner
- No overlapping functionality



# Handling the click to delete

```
const listEl = document.querySelector('.todos');
listEl.addEventListener('click', (e) => {
  if( e.target.classList.contains('todo') ) {
    // ...
    return; // ADDED
  }
  if( e.target.classList.contains('delete') ) {
    const index = e.target.dataset.index;
    todos.splice(index, 1);
    render();
    return;
  }
});
```

Why the `return`?

Here using same click event listener

- Could be separate

# Change didn't add much complexity

Because our render() uses the state

- Not connected to the action we are taking
- Actions just change state
- Our concerns (rendering vs handling action)
  - Separation of Concerns
- Our render doesn't KNOW what happened
- Our actions don't know how it renders
  - Principle of Least Knowledge
    - (in general, not the OOP version)

# Getting the Data to add

- We need a form (or at least an input)
- Todo Task
- We will assume "done" as false
- Form/input does not need to be rerendered
  - Always there regardless of state

# HTML for input

```
<label>
  New Task
  <input class="new-task">
</label>
<button type="button">Add</button>
```

- No `<form>` is allowed
  - No `name` attribute
  - No `submit` event
  - No auto "submit on enter"
    - Can use `click` and `keyup/input`
- We are REQUIRING Javascript for this
  - Always consider if that is right choice

# HTML with form

```
<form action="" class="add-task">
  <label>
    New Task
    <input class="new-task">
  </label>
  <button type="submit">Add</button>
</form>
```

- Need to preventDefault on `submit` event
  - Auto "submit on enter" behavior
  - Don't use click/input/keyup events
- Still requires Javascript
  - Unless we have an action on backend
- `<form>` never submits, but semantically better

# Adding event for adding tasks

```
const addFormEl = document.querySelector('.add-task');
addFormEl.addEventListener('submit', (e) => {
  e.preventDefault();

  const task = document.querySelector('.new-task').value;
  todos.push({ task, done: false });
  render();
});
```

## **Almost feels too easy now**

- User takes action
- Update state
- render()

Only complication is not re-rendering elements that have listeners

# What if it was more complex?

What if...

- Bulk of HTML could be "replaced"?
- Including the list and the add form?
- Let's add a toggle
  - Option 1: See todo list and add task form
  - Option 2: See a picture of a cat



# Changing page contents

Two major approaches:

- All content on page, some set to `display: none;`
  - Best to do this with classes
  - **Do not use `style` attribute**
    - Lots of tutorials and examples will!
    - It works, but makes code hard to change
- Render only the current HTML content
  - But keep the state to recreate as needed

# Option: Hiding Content

## Pros

- Don't need to juggle event listeners
- Showing/Hiding is fairly easy
  - As long as you define the classes

## Cons

- Render gets messier since it has ALL the HTML
- Styling can have impacts from all the HTML
- Debugging can be annoying
  - Lots of hidden elements but still in HTML

# Option: Conditional Render

## Pros

- HTML and CSS stay "clean"
- Once changes made
  - Can easily scale for more

## Cons

- Have to move event listeners to ancestor
- Have to expand and break up render

# Choosing which Option

Personally

- "Hiding" only for the most trivial of content
- "Conditional Rendering" most of the time
  - Always will be another added feature

# Implementing Conditional Rendering

- Expand State
  - Track what to show
- Shift event listeners
  - Common ancestor element
  - Check target of events
- Refactor `render()`
  - Check state for what to show
  - Specific `render()`-like for parts

# Expanded State

```
const state = {  
  todos: [  
    // ...  
  ],  
  page: 'todos',  
};
```

- Doesn't NEED to be one object
- ...but can help organize
- ...can pass all state or sub-portion as/if needed
- `page` as a string?
  - Allows for expansion
  - Can define enum-like

# State with Enum-like

```
const PAGES = {  
  TODOS: 'todos',  
  CAT: 'cat',  
};  
  
const state = {  
  todos: [  
    // ...  
  ],  
  page: PAGES.TODOS,  
};
```

- Makes typos easier to find
- IDE can help fill in

# Shifting the Listeners

Wrap page in an element that will not be replaced

```
<div id="app"></div>
```

```
const appEl = document.querySelector('#app');
appEl.addEventListener('submit', (e) => {
  if ( e.target.classList.contains('add-task') ) {
    // code for adding task
  }
  render();
});
appEl.addEventListener('click', (e) => {
  if ( e.target.classList.contains('todo') ) {
    // code for task done toggle
  }
  if ( e.target.classList.contains('delete') ) {
    // code for deleting task
  }
  render();
});
```



# Breaking up render()

```
function render() {  
  if(state.page === PAGES.TODO) {  
    renderTodos();  
  }  
  if(state.page === PAGES.CAT) {  
    renderCat();  
  }  
}
```

# Another approach

```
const renderFor = {  
  [PAGES.TODO]: renderTodos,  
  [PAGES.CAT]: renderCat,  
};  
  
function render() {  
  renderFor[state.page]();  
}
```

- Pro: Less `if()` "noise"
- Con: Can't easily pass values

Use whatever style works best

# renderTodos ( )

```
function renderTodos() {  
  const listHtml = state.todos.map( (todo, index) => {  
    // Same as before  
  }).join('');  
  
  appEl.innerHTML = `  
    <ul class="todos">  
      ${listHtml}  
    </ul>  
    <form action="" class="add-task">  
      <label>  
        New Task  
        <input class="new-task">  
      </label>  
      <button type="submit">Add</button>  
    </form>  
  `;  
}
```

# renderCat()

```
function renderCat() {  
  appEl.innerHTML = `  
      
  `;  
}
```

# Changing the page

- Specific button classes OR just one with data

Add to `renderTodos()`

```
<button type="button" class="page" data-target="cat">  
  Go to Todos  
</button>
```

Add to `renderCat()`

```
<button type="button" class="page" data-target="todos">  
  Go to Todos  
</button>
```

Add to listeners

```
if (e.target.classList.contains('page')) {  
  state.page = e.target.dataset.target;  
}
```

# Further cleanup?

- split out some functions
  - those functions not shown

```
function renderTodos() {  
  const listHtml = generateTodoHtml(state.todos);  
  const addTaskHtml = generateAddTaskHtml();  
  const navigateHtml = generateNavigateHtml(state);  
  
  appEl.innerHTML = `  
    <ul class="todos">  
      ${listHtml}  
    </ul>  
  
    ${addTaskHtml}  
    ${navigateHtml}  
  `;  
}
```

# These are just one way of organizing details

- Key lesson
  - **separate state changes and rendering**
  - Render html based on state
  - Allows you to generate HTML for any state
    - Without knowing what triggered render
  - Minimizes complexity; Allows easy change
  - Listeners unimpacted by HTML changes

You may notice js file is getting large and cumbersome

- We will address that soon!



# Let's test the ability to change

We will add option to change text of task

- Click will not toggle "done"
  - Instead add checkbox for that
- Click will "select" that todo (new state!)
  - Display that todo text as input
  - Include "Update" button
- On update
  - Change task text
  - De-select the todo

# Summary: Render Loop

- Have state
- Render based on state
- User actions update state
  - Rerender using new state
- That's all of it

# Rendering based on state

- Generate HTML based on state
  - Including what classes are on elements
- Replace original HTML with new HTML
- NOT: Changing a class on an element
- NOT: Changing style attributes on an element

# Pros/Cons of State-Render Loop

- PRO: State always update to date
  - PRO: Can just re-render
    - Reduces cognitive requirements
- I
- CON: Lots of redundant replacement
    - Fixed when we get to React
  - CON: Re-render can lose typing in form fields
    - Fixed when we get to React
    - Could fix with more work
      - Putting in-progress typing into state