



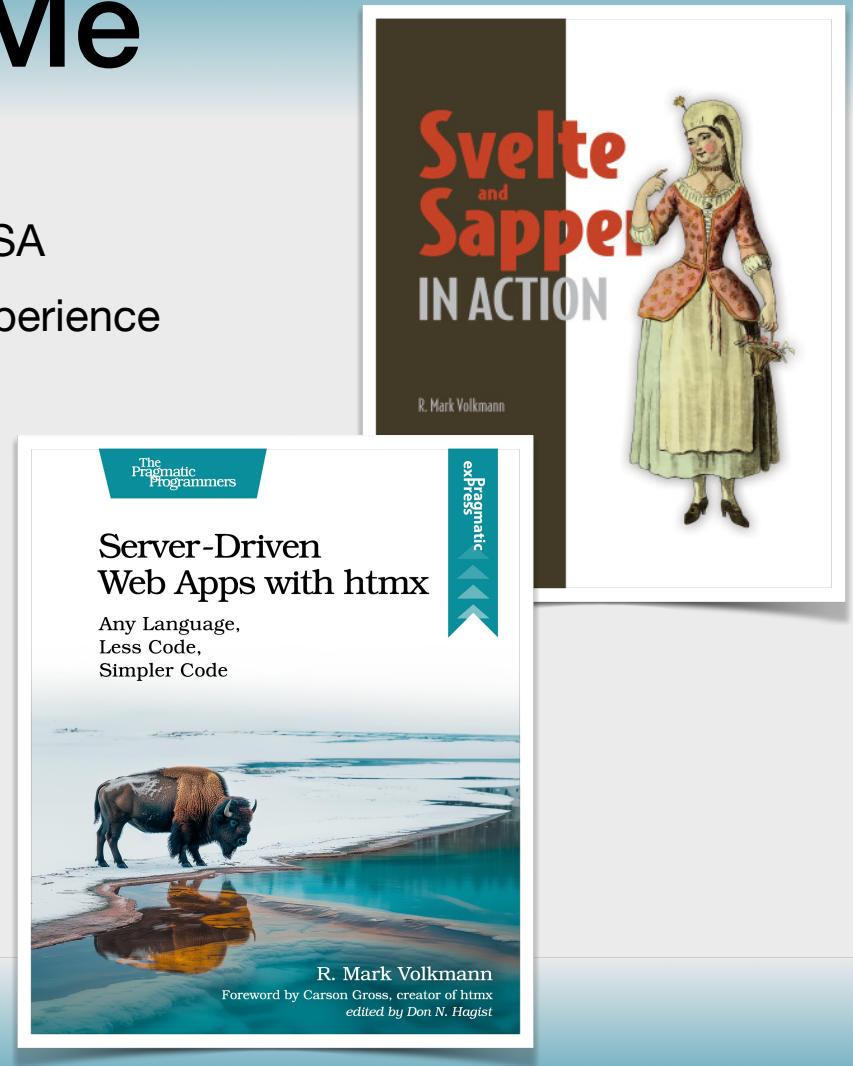
Web REactive Components (wrec)

R. Mark Volkmann
Object Computing, Inc.
<https://objectcomputing.com>
r.mark.volkmann@gmail.com

Slides at <https://github.com/mvolkmann/talks/>
Code at <https://github.com/mvolkmann/wrec-talk-code>

About Me

- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 44 years of professional software development experience
- Writer and speaker
- **Blog** at <https://mvolkmann.github.io/blog/>
- Author of Manning book “**Svelte ... in Action**”
- Author of Pragmatic Bookshelf book “**Server-Driven Web Apps with htmx**”
- Currently writing a book on web components



What is wrec

- Library that simplifies development of web components
 - <https://www.npmjs.com/package/wrec>
- Strong focus on simplicity and reactivity
- To define a web component
 - define a class that extends **wrec**
 - define properties with static property named **property**
 - specify scoped CSS with static property named **css**
 - specify HTML to render with static property named **html**



Getting Started ...

- Create project directory and `cd` to it
- Create `package.json` by entering `npm init -y`
- Install Vite by entering `npm i -D vite`
- Add `dev` script to `package.json` →
- Install wrec by entering `npm i wrec`
- Create `hello-world.js`

The `html` tag function enables syntax highlighting and code completion of HTML inside tagged template literals.

```
package.json
{
  "name": "hello-world",
  "scripts": {
    "dev": "vite"
  },
  "devDependencies": {
    "vite": "^7.3.1"
  },
  "dependencies": {
    "wrec": "^0.18.5"
  }
}
```

```
hello-world.js
import { html, Wrec } from "wrec";

class HelloWorld extends Wrec {
  static html = html`<h1>Hello, World!</h1>`;
}

HelloWorld.define("hello-world");
```



... Getting Started



- Create `index.html`
- Start local server by entering `npm run dev`
- Browse `localhost:5173`

Hello, World!

```
<!doctype html>
<html lang="en">
  <head>
    <script type="module" src="hello-world.js"></script>
  </head>
  <body>
    <hello-world></hello-world>
  </body>
</html>
```

index.html

Better hello-world

- Can specify **name** attribute

```
import { css, html, Wrec } from "wrec";                                hello-world.js

class HelloWorld extends Wrec {
  static properties = {
    name: { type: String, value: "World" },
  };

  static css = css`  
  h1 {  
    color: blue;  
  }  
`;  
  
  static html = html`<h1>Hello, <span>this.name</span>!</h1>`;  
}

HelloWorld.define("hello-world"); <hello-world name="Earth"></hello-world>
```

Hello, Earth!

Best hello-world

- Can specify `color` and `name` attributes

```
import { css, html, Wrec } from "wrec";                                hello-world.js

class HelloWorld extends Wrec {
  static properties = {
    color: { type: String, value: "blue" },
    name: { type: String, value: "World" },
  };

  static css = css`

# color: this.color; `; static html = html`<h1>Hello, <span>this.name</span>!</h1>`; } HelloWorld.define("hello-world"); <hello-world color="red" name="Earth"></hello-world>


```

Hello, Earth!

Vanilla Version

requires much more code

```
class HelloWorld extends HTMLElement {          hello-world.js
  static get observedAttributes() {
    return ["color", "name"];
  }

  #color = "black";
  #name = "World";
  #p = document.createElement("p");

  attributeChangedCallback(name, _oldValue, newValue) {
    if (name === "color") {
      if (newValue !== this.#color) this.color = newValue;
    } else if (name === "name") {
      if (newValue !== this.#name) this.name = newValue;
    }
  }

  connectedCallback() {
    this.attachShadow({ mode: "open" });
    this.shadowRoot.replaceChildren(this.#p);
  }
}
```

```
get color() {
  return this.#color;
}

get name() {
  return this.#name;
}

set color(value) {
  this.#color = value;
  this.setAttribute("color", value);
  this.#p.style.color = value;
}

set name(value) {
  this.#name = value;
  this.setAttribute("name", value);
  this.#p.textContent = `Hello, ${value}!`;
}

customElements.define("hello-world", HelloWorld);
```



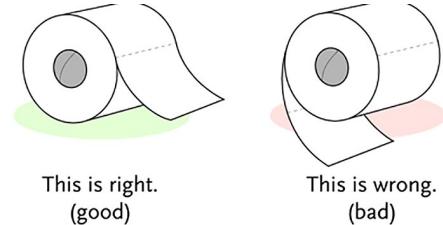
JavaScript Expressions



- wrec supports JavaScript expressions that refer to component properties with `this . someName`
- Can appear in element text content, attribute values, and CSS property values

Lit doesn't support JavaScript expressions in CSS
- Automatically updated when referenced property values change

this . Controversy



- Should those JavaScript expressions be surrounded by some kind of delimiter like curly braces?
- Is searching for `this.someName` in plain text sufficient?
- When would plain text ever contain `this.` followed by a letter that is not part of a JavaScript expression?
- Answer: never
- But there is an escape mechanism
 - `this` followed by two periods is not treated as part of a JavaScript expression and will be replaced by `this` followed by a single period

Property/Attribute Reflection

- It's automatic!
- Open DevTools and click Elements tab
- Change an attribute of a custom element
 - like `color` or `name` attribute in a `hello-world` element
 - updates what is rendered AND corresponding property
- Select a custom element and click Console tab
- Change properties of `$0`
 - for example, `$0.name = 'Moon'`
 - updates what is rendered AND corresponding attribute

demonstrate with
hello-world project



Secret Sauce

- The first time an instance of a given web component class is used, `wrec` searches all attribute values, element text content, and CSS properties for expressions matching `this.{property-name}` using a regular expression.
- Expressions in CSS properties are replaced by a local variable reference and a CSS variable declaration whose value is the expression is added before it.
- A static map named `propToExprsMap` is created (one per `Wrec` subclass) whose keys are property names and whose values arrays of the expressions where they are found.
- A map named `exprToRefsMap` is created for each `Wrec` subclass instance whose keys are the expressions and whose values are arrays of references to the elements, attributes, and CSS variable declarations where they appear.
- For each property, `Object.defineProperty` is called to add getter and setter methods.
- When an instance property change is detected by its setter method, a list of the expressions that use the property is retrieved from `propToExprsMap` and new values are computed.
- Then for each expression whose value was computed, a list of all references to the expression is retrieved from `exprToRefsMap` and they are updated with the new value.

```
h1 {  
  --color: this.color;  
  color: var(--color);  
}
```



Form Control 2-way Bindings

Lit doesn't do this

- **input & select** element **value** attribute gets 2-way binding to a property ↪
- Expressions that use property automatically update when value changes

```
import { css, html, Wrec } from "wrec";

class CircleCalculator extends Wrec {
  static properties = {
    radius: { type: Number, value: 0 },
  };

  static css = css`  
  :host {  
    font-family: sans-serif;  
  }  
  label {  
    font-weight: bold;  
  }  
`;
```

Radius:

Circumference: 6.283185307179586
Area: 3.141592653589793

```
static html = html`  
  <div>  
    <label for="radius">Radius:</label>  
    <input id="radius" type="number" value="this.radius" />  
  </div>  
  <div>  
    <label for="circumference">Circumference:</label>  
    <span id="circumference">2 * Math.PI * this.radius</span>  
  </div>  
  <div>  
    <label for="area">Area:</label>  
    <span id="area">Math.PI * this.radius</span>  
  </div>  
`;  
  
CircleCalculator.define("circle-calculator");
```

Disabling



- Adding **disabled** attribute to a wrec component instance or removing it, automatically does the same to all applicable descendant elements, even those in nested shadow DOMs
- Applicable elements include **button**, **fieldset**, **input**, **select**, and **textarea** elements, and also nested wrec component instances

Parent-Child 2-way Bindings ...

- Parent components can pass their properties to child components
- Automatically creates a 2-way binding
- When child property changes,
corresponding parent property is directly updated
 - doesn't require dispatching events from child and listening for them in parent
 - uses `propToParentPropMap`



... Parent-Child 2-Way Bindings

```
import { css, html, Wrec } from "wrec";

class ToggleButtons extends Wrec {
  static properties = {
    labels: { type: String },
    value: { type: String },
  };
  static css = css`...`;
  static html = "this.makeButtons(this.labels)";

  handleClick(event) {
    this.value = event.target.textContent.trim();
  }

  makeButtons(labels) {
    const labelArray = labels.split(",");
    return labelArray.map((label) => {
      const classExpr = `this.value === '${label}' ? 'selected' : ''`;
      return html`
        <button class="${classExpr}" onClick="handleClick">${label}</button>
      `;
    });
  }
}

ToggleButtons.define("toggle-buttons");
```

```
import { html, Wrec } from "wrec";
import "./hello-world";
import "./toggle-buttons";

class MyApp extends Wrec {
  static properties = {
    color: { type: String, value: "red" },
  };
  static html = html`
    <hello-world color="this.color"></hello-world>
    <toggle-buttons
      labels="red,green,blue"
      value="this.color"
    ></toggle-buttons>
  `;
}

MyApp.define("my-app");
```

Hello, World!



number-slider

used later in color-picker

```
import { css, html, Wrec } from "wrec";

class NumberSlider extends Wrec {
  static properties = {
    label: { type: String },
    labelWidth: { type: String },
    max: { type: Number, value: 100 },
    min: { type: Number, value: 0 },
    value: { type: Number },
  };

  static css = css`...`;
}
```

details omitted

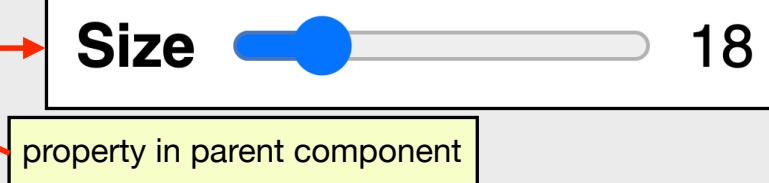
```
static html = html`this.label
<input
  type="range"
  min="this.min"
  max="this.max"
  value:input="this.value"
/>
this.value`;
}

NumberSlider.define("number-slider");
```



By default, updates on change events. Adding :input causes update on every mouse drag, not just when focus moves away.

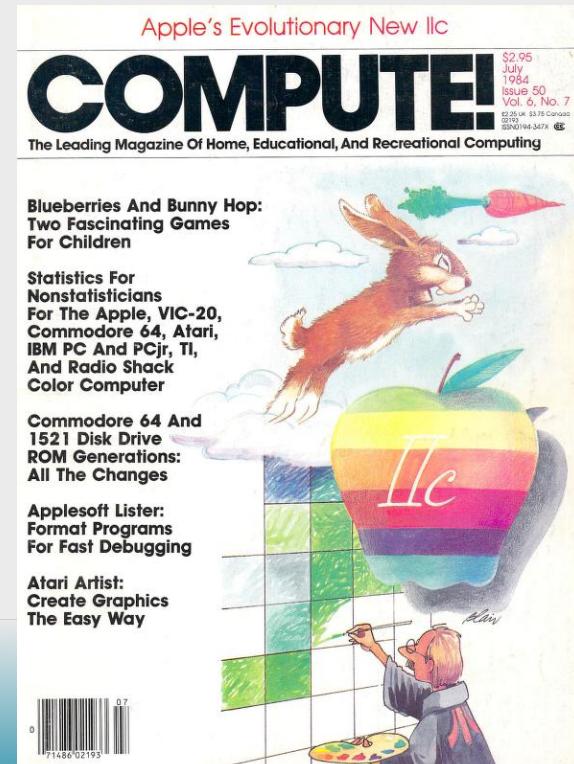
```
<number-slider
  label="Size"
  max="48"
  min="12"
  value="this.size"
></number-slider>
```



property in parent component

Computed Properties

- Defined by adding `computed` property in property options
 - see `color` property on next slide
- Uses static `propToComputedMap`
 - tracks which computed properties to recompute after a change to a property used to compute it



color-picker



```
import { css, html, Wrec } from "wrec";
import "./number-slider";

class ColorPicker extends Wrec {
  static properties = {
    labelWidth: { type: String, value: "3rem" },
    red: { type: Number },
    green: { type: Number },
    blue: { type: Number },
    color: {
      type: String,
      computed: `rgb(${this.red},${this.green},${this.blue})`,
    },
  };
  static css = css`

Red



Green



Blue

:host {
  display: flex;
  gap: 0.5rem;
}
#sliders {
  display: flex;
  flex-direction: column;
  justify-content: space-between;
}
#swatch {
  background-color: this.color;
  height: 5rem;
  width: 5rem;
}`;
```

```
static html = html`

<${this.makeSlider("Red")}  
${this.makeSlider("Green")}  
${this.makeSlider("Blue")}

`;
static makeSlider(label) {
  return html`<number-slider
  label=${label}
  label-width="this.labelWidth"
  max="255"
  value="this.${label.toLowerCase()}">
```

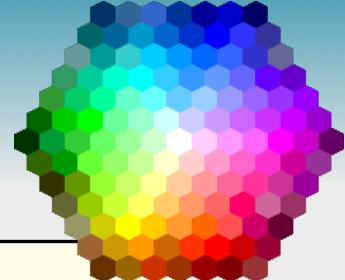
Computed with “uses”

- Computed properties support optional `uses` property
 - needed when expression doesn't reference every property needed to compute value
 - for example, the `color` property on previous slide can be defined like this

```
static properties = {
  ...
  color: {
    type: String,
    computed: "this.getColor()",
    uses: "red,green,blue"
  },
};

getColor() {
  return `rgb(${this.red},${this.green},${this.blue})`;
}
```

color-demo ...



```
import { css, html, Wrec } from "wrec";
import "./color-picker";
import "./number-slider";

class ColorDemo extends Wrec {
  static properties = {
    color: { type: String },
    size: { type: Number, value: 18 },
  };

  static css = css`  

    :host {  

      display: flex;  

      flex-direction: column;  

      gap: 0.5rem;  

      font-family: sans-serif;  

    }  

    p {  

      color: this.color;  

      font-size: this.size + "px";  

    }  

`;
```

```
static html = html`  

<color-picker color="this.color"></color-picker>  

<number-slider  

  label="Size"  

  max="48"  

  min="12"  

  name="size"  

  value="this.size"  

></number-slider>  

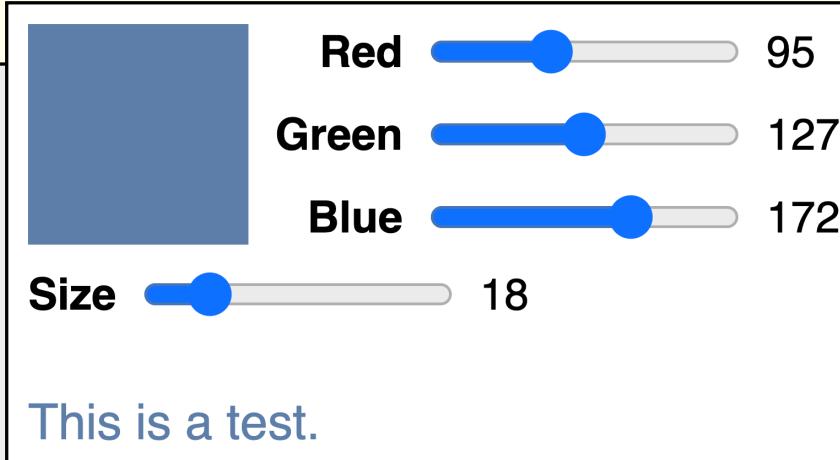
<p>This is a test.</p>
`;  

}  
  

ColorDemo.define("color-demo");
```

... color-demo

```
<!doctype html>
<html lang="en">
  <head>
    <script src="color-demo.js" type="module"></script>
  </head>
  <body>
    <color-demo></color-demo>
  </body>
</html>
```



Event Dispatching ...

- Can opt into having change events dispatched when a property changes
- Just add **dispatch: true** to property description

```
static properties = {  
  value: {type: String, dispatch: true}  
};
```

- Events have **detail** property
whose value is an object with properties
tagName, property, oldValue, and value

custom element name

Event Dispatching

```
class ToggleButtons extends Wrec {
```

```
  static properties = {
```

```
    labels: { type: String },
```

```
    value: { type: String, dispatch: true },
```

```
  };
```

```
  ...
```

```
}
```

```
<!doctype html>
```

```
<html lang="en">
```

```
  <head>
```

```
    <script type="module" src="toggle-buttons.js"></script>
```

```
    <script type="module">
```

```
      const tb = document.querySelector('toggle-buttons');
```

```
      tb.addEventListener('change', e => {
```

```
        const {tagName, property, oldValue, value} = e.detail;
```

```
        alert(`${tagName} ${property} changed from ${oldValue} to ${value}.`);
```

```
      });
```

```
    </script>
```

```
  </head>
```

```
  <body>
```

```
    <toggle-buttons labels="red,green,blue" value="red"></toggle-buttons>
```

```
  </body>
```

```
</html>
```

change



localhost:5173 says

toggle-buttons value changed from red to green.

OK



Non-primitive Properties ...

```
<!doctype html>
<html lang="en">
  <head>
    <style>
      body { font-family: sans-serif; }
    </style>
    <script src="table-wired.js" type="module"></script>
    <script type="module">
      const propNames = ['name', 'age', 'occupation'];
      const headings = ['Name', 'Age', 'Occupation'];
      const data = [
        {name: 'Alice', age: 30, occupation: 'Engineer'},
        {name: 'Bob', age: 25, occupation: 'Designer'},
        {name: 'Charlie', age: 35, occupation: 'Teacher'}
      ];
      const tableWired = document.querySelector('table-wired');
      tableWired.propNames = propNames;
      tableWired.headings = headings;
      tableWired.data = data;
    </script>
  </head>
  <body>
    <table-wired></table-wired>
  </body>
</html>
```

index.html

non-primitive properties

can't specify non-primitive properties with attributes

Name	Age	Occupation
Alice	30	Engineer
Bob	25	Designer
Charlie	35	Teacher



... Non-primitive Properties ...

```
import {css, html, Wrec} from '../wrec';           table-wired.ts

class TableWired extends Wrec {
  static properties = {
    headings: {type: Array<string>},
    propNames: {type: Array<string>},
    data: {type: Array<object>}
  };
  static css = css`...`;
  static html = html`



`;
  sortAscending = true;
  sortHeader: HTMLTableCellElement | null = null;
```

see code at <https://github.com/mvolkmann/wrec-talk-code/blob/main/table-demo/table-wired.ts>

bind is needed here
because makeTh uses this

continued on next slide



... Non-primitive Properties

```
makeTd(dataIndex: number, prop: string) {  
  const value = this.data[dataIndex][prop];  
  return html`<td>${value}</td>`;  
}  
  
makeTh(heading: string, index: number) {  
  return html`  
    <th  
      aria-label="sort by ${heading}"  
      data-property="${this.propNames[index]}"  
      onclick="sort"  
      role="button"  
      tabindex="0"  
    >  
      <span>${heading}</span>  
      <span class="sort-indicator"></span>  
    </th>  
  `;  
}
```

```
makeTr(dataIndex: number) {  
  return html`  
    <tr>  
      this.propNames.map(propName =>  
        this.makeTd(${dataIndex}, propName))  
    </tr>  
  `;  
}  
  
sort(event: Event) {  
  ...  
}  
}  
  
TableWired.define("table-wired");
```

see code at <https://github.com/mvolkmann/wrec-talk-code/blob/main/table-demo/table-wired.ts>

Can update from DevTools Console after selecting a `table-wired` element.

```
data = [...$0.data]  
data[1] = {name: "Leonardo", age: 49, occupation: "Artist"}  
$0.data = data
```

Forms ...

- When form control elements are nested in a form element, their values are automatically included in `form` submissions
 - includes `input`, `select`, and `textarea` elements
- Instances of web components can also contribute to form submissions
 - add `static formAssociated = true;`
 - to contribute single value in property named `value`, specify name to submit with instance `name` attribute
 - just like in form control elements
 - to contribute multiple values, specify mappings from property names to names to submit with `form-assoc` attribute

Contact

First name

Last name

Phone

Date and time

Day Month Year Time

Seating preference

Indoor
 Outdoor

Additional requests

Yes, subscribe me to your newsletter.

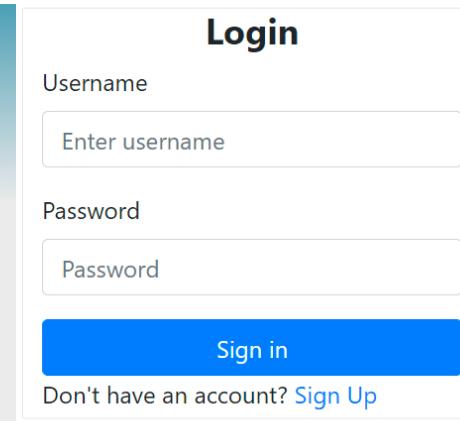
Request booking

... Forms

- Example
 - in `number-slider.js` and `color-picker.js`, add `static formAssociated = true;`
 - in parent component, add following where
 - `number-slider value` property is used for form data property `score`
 - `color-picker` properties `red`, `green`, and `blue` are used for form data properties `r`, `g`, and `b`

```
<form method="post" action="/save-game">
  <number-slider label="Score" name="score"></number-slider>
  <color-picker form-assoc="red: r, green: g, blue: b"></color-picker>
  <button type="reset">Reset</button>
  <button type="submit">Submit</button>
</form>
```

Duplicating this in vanilla web components is far more complicated.
It requires implementing the methods `formAssociatedCallback` and `formResetCallback`.
It also requires creating a `FormData` object, calling `this.attachInternals`, and calling `setFormValue`.



State ...

- **WrecState** class maps state properties to wrec component properties
 - changing either updates the other to keep them in sync
- Optionally persists to **localStorage**
 - supports page refreshes without losing data
- Removes need for prop drilling
- Example uses
 - **hello-world** component (shown earlier)
 - **labeled-input** component 

```
import { css, html, Wrec } from "wrec";

class LabeledInput extends Wrec {
  static properties = {
    id: { type: String, required: true },
    label: { type: String, required: true },
    name: { type: String },
    value: { type: String },
  };
  static css = css`

display: flex;
    align-items: center;
    gap: 0.5rem;
  `;
  static html = html`<div>
    <label for="this.id">this.label</label>
    <input
      id="this.id"
      name="this.name"
      type="text"
      value="this.value"
    />
  </div>
`;
}

LabeledInput.define("labeled-input");


```

... State ...

```
<!doctype html>
<html lang="en">
  <head>
    <style>
      body {
        font-family: sans-serif;
        padding: 1rem;
      }
    </style>
  </head> ←
  <body>
    <labeled-input
      id="name"
      label="Name"
    ></labeled-input>
    <hello-world></hello-world>
    <button>Reset</button>
  </body>
</html>
```

```
<script type="module">
  import { WrecState } from "wrec";
  import "./hello-world.js";
  import "./labeled-input.js";

  const state = new WrecState("demo", false, { name: "World" });

  const li =
    document.querySelector("labeled-input");
  li.useState(state, { name: "value" });

  const hw =
    document.querySelector("hello-world");
  hw.useState(state);

  const button = document.querySelector("button");
  button.addEventListener("click", () => {
    state.name = "World";
  });
</script>
```

specifies whether state should be persisted to localStorage

maps state `name` property to `labeled-input value` property

maps state `name` property to `hello-world name` property

Name

Hello, World!

Reset

... State

- State can be accessed, inspected, and modified from any code and from DevTools console
- To get reference to state object,
enter `state = WrecState.get("demo")`
- To see all properties in state object, enter `JSON.stringify(state)`
- To see value of `name` property, enter `state.name`
 - outputs "World"
- To change value of `name` property, enter `state.name = "Earth"`
 - updates `hello-world` and `labeled-input` components



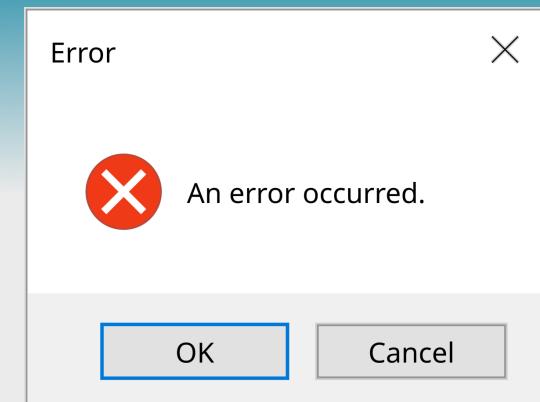
The screenshot shows the DevTools Console tab with the following interactions:

```
> state = WrecState.get("demo")
< st2 {}
> JSON.stringify(state)
< '{"name":"World"}'
> state.name
< 'World'
> state.name = "Earth"
< 'Earth'
```

A yellow callout box with a black border and a red arrow points from the text "name assigned to our state object" to the variable "state" in the first line of the console output.

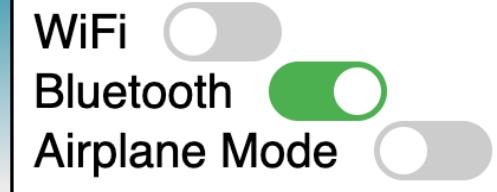
Error Checking

- wrec checks for many kinds of errors and throws an **Error** when they are found
- Look for messages in DevTools console
- Kinds of errors detected include
 - custom elements that are not defined
 - attribute names with no matching property declaration
 - attribute values with type that differs from property type
 - expressions in attribute values or element text content that reference undeclared properties
 - expressions in element text content that do not evaluate to a string or number
 - event handling function names that don't match a method name in web component



```
static html = html`  
  <bad-1></bad-1>  
  <component-a  
    bad-2="no matching property"  
    bad-3="should be a number"  
    bad-4="this.noSuchProperty  
>  
    this.bad4Undeclared  
  </component-a>  
  <component-b>  
    this.bad5NonPrimitive  
  </component-b>  
  <button onclick="bad6">  
    Click Me  
  </button>  
`;
```

Generating with AI



- An LLM prompt like the following can generate a web component that uses wrec
 - Implement a web component for a switch that has a label using the wrec library which can be found at <https://github.com/mvolkmann/wrec> and is described in the blog page at <https://mvolkmann.github.io/blog/wrec/>.*
 - Claude** gets this mostly correct
 - imports from `./wrec.min.js` instead of just `wrec`
 - used an invalid class attribute value
 - Google Gemini** gets this mostly correct
 - imports from `https://cdn.jsdelivr.net/npm/wrec/dist/wrec.min.js` instead of just `wrec`
 - thinks Boolean attributes in CSS have value `true` instead of attribute name
 - ChatGPT, Microsoft Copilot, and Perplexity** all fail badly

```
changed "switch this.checked ? 'checked' : ''"  
to "this.checked ? 'switch checked' : 'switch'"
```

```
changed .track[data-on="true"]  
to .track[data-on="data-on"]
```

Tests

- wrec has an extensive set of Playwright tests
- To run the tests
 - clone **wrec** repository
 - cd to **wrec** directory
 - Enter **npm install**
 - To run headless
 - enter **npm run test**
 - To run interactive
 - enter **npm run testui**
 - click right pointing triangle



wrec Resources

- **Blog page**
 - <https://mvolkmann.github.io/blog/wrec>
- **npm package**
 - <https://www.npmjs.com/package/wrec>
- **GitHub repository**
 - <https://github.com/mvolkmann/wrec>
 - view and create issues here
- **Support email**
 - r.mark.volkmann@gmail.com



wrec Benefits

- Simplifies creating web components
- More reactivity than Lit components
 - DOM updates when properties change
 - automatic 2-way bindings between attributes and properties
 - automatic 2-way bindings between properties and form controls
 - reactive CSS through JS expressions in CSS property values
 - syncing changes between parent and child components without dispatching events and listening for them
 - but can automatically dispatch `change` events when properties change
 - computed properties
- Simplifies contributing data to form submissions
- Simplifies sharing state between components

