Controlled vs Uncontrolled

A Tale of Two Forms

A brownbag deep-dive at

MX®

by Seth House

@whiteinge
seth@eseth.com

Why?

“We had everything before us, we had nothing before us.”

A (quick) introduction

Controlled

```
<input type="text"
    value={this.state.foo}
    onChange={(ev) => this.setState({foo: ev.target.value})} />
```

Controlled

```
<input type="text"
    value={this.state.foo}
    onChange={(ev) => this.setState({foo: ev.target.value})} />
```

- Data stored explicitly (component state, hook, Redux, etc).
- Change to data triggers a re-render.
- Retrieve current value by referencing external state.

Controlled

```
<input type="text"
    value={this.state.foo}
    onChange={(ev) => this.setState({foo: ev.target.value})} />
```

- Data stored explicitly (component state, hook, Redux, etc).
- Change to data triggers a re-render.
- Retrieve current value by referencing external state.

React controls the current value.

Uncontrolled

```
<input type="text"
    defaultValue={this.state.foo}
    name="foo" />
```

Uncontrolled

```
<input type="text"
    defaultValue={this.state.foo}
    name="foo" />
```

- Data stored implicitly (in the DOM tree).
- Change to data does not trigger a render.
- Retrieve current value by reacting to DOM events, or by finding the DOM element.

Uncontrolled

```
<input type="text"
    defaultValue={this.state.foo}
    name="foo" />
```

- Data stored implicitly (in the DOM tree).
- Change to data does not trigger a render.
- Retrieve current value by reacting to DOM events, or by finding the DOM element.

The browser controls the current value.

A side-rant about refs

A side-rant about refs

Refs are an API "escape hatch":

A side-rant about refs

Refs are an API "escape hatch":

- Useful when necessary.
- Easy to create a mess when overused.
- Always imperative.

A side-rant about refs

Refs are an API "escape hatch":

- Useful when necessary.
- Easy to create a mess when overused.
- Always imperative.

"Common wisdom" is uncontrolled inputs means using refs. That's only true when you need imperative access to values. DOM events, `FormData` instances, and `HTMLFormElement` instances are very flexible.

A quick HTML form primer

A quick HTML form primer

```
<form id="myform">
  <fieldset>
    <legend>Inputs</legend>
    <label><input type="text" name="mytext" placeholder="Some Text">
    <br>
    <label><input type="number" name="mynumber" placeholder="Some Nu
  </fieldset>

  <fieldset>
    <legend>Radio and Checkbox</legend>
    <label><input type="radio" name="myradio" value="foo">Foo</label
    <label><input type="radio" name="myradio" value="bar">Bar</label
    <br>
    <label><input type="checkbox" name="mycheckbox"> Check?</label>
  </fieldset>

  <fieldset>
    <legend>Buttons</legend>
    <button type="submit">Submit</button>
    <button type="reset">Reset</button>
  </fieldset>
</form>
```

FormData

```
const urlEncodedData = new FormData(myform)
```

FormData

```
const urlEncodedData = new FormData(myform)
```

```
// Automatically sets Content-Type to "multipart/form-data"
fetch('/some/path', {method: 'POST', body: urlEncodedData})
```

FormData

```
const urlEncodedData = new FormData(myform)
```

```
// Automatically sets Content-Type to "multipart/form-data"
fetch('/some/path', {method: 'POST', body: urlEncodedData})
```

```
// As a JavaScript object:
const data = Object.fromEntries(new FormData(myform))
```

`HTMLFormElement`

- Form attributes.
- Form elements: `myform.elements`.
    - Careful not to name any form fields that will shadow existing DOM attributes like `length` or `submit`.
- Available via events: `ev => ev.target.form`.

`HTMLFormElement`

- Form attributes.
- Form elements: `myform.elements`.
    - Careful not to name any form fields that will shadow existing DOM attributes like `length` or `submit`.
- Available via events: `ev => ev.target.form`.

Note checkboxes and radio collections don't provide defaults to `FormData` but those are accessible via `HTMLFormElement`:

```
const data = Array.from(myform.elements)
        .reduce((acc, cur, idx, collection) => {
  if (cur.type === 'checkbox') { acc[cur.name] =
    collection[cur.name].checked }
  if (cur.type === 'radio') { acc[cur.name] =
    collection[cur.name].value }

  return acc
}, Object.fromEntries(new FormData(myform)))
```

Update another part of the page (controlled)

Update another part of the page (controlled)

```
/* ...snip State logic up here... */
<form>
    <h1>{this.state.campaignName || 'Campaign'}</h1>

    <input
        value={this.state.campaignName}
        placeholder="Campaign Name"
        onChange={(ev) =>
            this.setState({campaignName: ev.target.value})} />
</form>
```

Update another part of the page (uncontrolled)

Update another part of the page (uncontrolled)

```
<form>
    <h1>
        <output htmlFor="campaignName" name="nameDisplay">
            Campaign
        </output>
    </h1>

    <input
        defaultValue=""
        placeholder="Campaign Name"
        name="campaignName"
        onChange={(ev) =>
            ev.target.form.elements.nameDisplay.value =
                ev.target.value
                || ev.target.form.nameDisplay.defaultValue} />
</form>
```

Disable submit until the form is valid (controlled)

Disable submit until the form is valid (controlled)

```
<form>
  <input type="text" value={this.state.name} placeholder="name" />

  <br/>

  <button type="submit" disabled={this.isFormValid()}>
    Submit
  </button>
</form>
```

Disable submit until the form is valid (uncontrolled)

Disable submit until the form is valid (uncontrolled)

```
<form onChange={(ev) =>
    ev.target.form.submit.disabled =
        !ev.target.form.checkValidity()}>

  <input type="text" name="name" placeholder="name" required />

  <br/>

  <button type="submit" name="submitbtn" disabled={true}>
    Submit
  </button>
</form>
```

Form errors (controlled)

Form errors (controlled)

- State management for values.

Form errors (controlled)

- State management for values.
- State management for errors.

Form errors (controlled)

- State management for values.
- State management for errors.
    - Set error message.
    - Highlight input.

Form errors (controlled)

- State management for values.
- State management for errors.
    - Set error message.
    - Highlight input.

```
<input type="text" value={this.state.values.name} />
{this.state.errors.name && (
    <span className="error-msg">{this.state.errors.name}</span>
)}
```

Form errors (uncontrolled)

Form errors (uncontrolled)

```
<input type="text" name="name" onChange={(ev) => {
    if (ev.target.value === 'forbidden') {
        ev.target.setCustomValidity('Doh!')
    } else {
        ev.target.setCustomValidity('')
    }
    ev.target.reportValidity()
}} />
```

Form errors (uncontrolled)

```
<input type="text" name="name" onChange={(ev) => {
    if (ev.target.value === 'forbidden') {
        ev.target.setCustomValidity('Doh!')
    } else {
        ev.target.setCustomValidity('')
    }
    ev.target.reportValidity()
}} />
```

```
<style>
{`
input:invalid {border: 1px solid red;}
input:valid {border: 1px solid green;}
`}
</style>
```

Update component state from uncontrolled inputs

Update component state from uncontrolled inputs

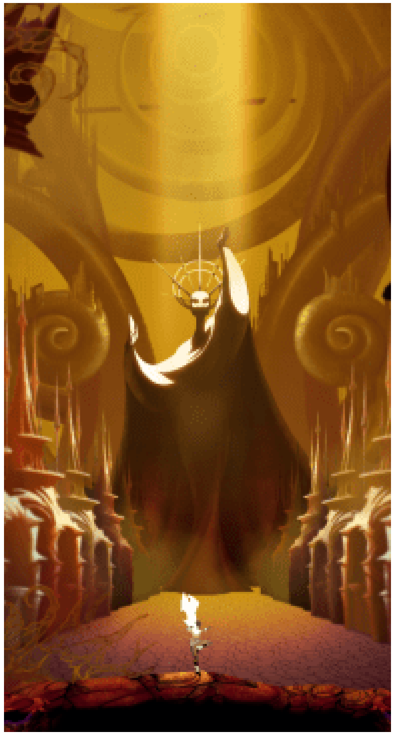
```
<form onChange={(ev) => setState(oldState => ({
    ...oldState,
    [ev.target.name]: ev.target.value,
  }))}>

  <input type="text"
    name="name"
    defaultValue={state.name} />

  <br />

  <input type="number"
    name="zipcode"
    defaultValue={state.zipcode} />
</form>
```

Summary?

What can controlled inputs learn from from uncontrolled inputs?

What can controlled inputs learn from from uncontrolled inputs?

Embrace or resist (the platform).



Addendum

`dialog`

`dialog`

```
<dialog id="mydialog">
    <form method="dialog" >
        <p>Modal content here!</p>
        <button type="submit">Close</button>
    </form>
</dialog>

<button type="button" onclick="mydialog.showModal()">Show</button>

<style>
dialog[open] { max-width: 80%; max-height: 80% }
dialog::backdrop { background: 'black'; opacity: 90% }
</style>
```

`dialog` plus async

```
<dialog id="mydialog">
    <form
        method="dialog"
        onsubmit="((ev) => {
            ev.preventDefault(); // Stop the modal from closing.
            ev.target.elements.submitbtn.disabled = true
            ev.target.elements.spinner.value = '🌀'
            setTimeout(() => {
                ev.target.submit(); // Close the modal later.
                ev.target.elements.submitbtn.disabled = false
                ev.target.elements.spinner.value = ''
            }, 2000);
        })(event)"
    >
        <button type="submit" name="submitbtn">
            Wait 2s then close.
            <output name="spinner" for="mydialog"></output>
        </button>
    </form>
</dialog>

<button type="button" onclick="mydialog.showModal()">Show</button>
```