More capable form controls

- Event-based API
- Browser compatibility
- Form-associated custom elements
  - Defining a form-associated custom element
  - Setting a value {: #setting-a-value }
  - Input validation
  - Lifecycle callbacks
    - void formAssociatedCallback(form)
    - void formDisabledCallback(disabled)
    - void formResetCallback()
    - void formStateRestoreCallback(state, mode)
  - Restoring form state {: #restoring-form-state }
  - A working example
- Feature detection
- Conclusion

Many developers build custom form controls, either to provide controls that aren't built in to the browser, or to customize the look and feel beyond what's possible with the built-in form controls.

However, it can be difficult to replicate the features of built-in HTML form controls. Consider some of the features an `<input>` element gets automatically when you add it to a form:

- The input is automatically added to the form's list of controls.
- The input's value is automatically submitted with the form.
- The input participates in form validation. You can style the input using the `:valid` and `:invalid` pseudoclasses.
- The input is notified when the form is reset, when the form is reloaded, or when the browser tries to autofill form entries.

Custom form controls typically have few of these features. Developers can work around some of the limitations in JavaScript, like adding a hidden `<input>` to a form to participate in form submission. But other features just can't be replicated in JavaScript alone.

Two new web features make it easier to build custom form controls, and remove the limitations of current custom controls:

- The `formdata` event lets an arbitrary JavaScript object participate in form submission, so you can add form data without using a hidden `<input>`.
- The Form-associated custom elements API lets custom elements act more like built-in form controls.

These two features can be used to create new kinds of controls that work better.

Building custom form controls is an advanced topic. This article assumes a certain knowledge of forms and form controls. When building a custom form control, there are many factors to consider, especially making sure that your controls are accessible to all users. To learn more about forms, go to the MDN guide on forms.

Event-based API

The `formdata` event is a low-level API that lets any JavaScript code participate in form submission. The mechanism works like this:

1. You add a `formdata` event listener to the form you want to interact with.
2. When a user clicks the submit button, the form fires a `formdata` event, which includes a `FormData` object that holds all of the data being submitted.
3. Each `formdata` listener gets a chance to add to or modify the data before the form is submitted.

Here's an example of sending a single value in a `formdata` event listener:

```
const form = document.querySelector('form');
// FormData event is sent on <form> submission, before transmission.
// The event has a formData property
form.addEventListener('formdata', ({formData}) => {
  // https://developer.mozilla.org/docs/Web/API/FormData
  formData.append('my-input', myInputValue);
});
```

Try this out using our example on Glitch. Be sure to run it on Chrome 77 or later to see the API in action.

https://glitch.com/edit/#!/formdata-event

Browser compatibility

Form-associated custom elements

You can use the event-based API with any kind of component, but it only allows you to interact with the submission process.

Standardized form controls participate in many parts of the form lifecycle besides submission. Form-associated custom elements aim to bridge the gap between custom widgets and built-in controls. Form-associated custom elements match many of the features of standardized form elements:

- When you place a form-associated custom element inside a `<form>`, it's automatically associated with the form, like a browser-provided control.
- The element can be labeled using a `<label>` element.
- The element can set a value that's automatically submitted with the form.
- The element can set a flag indicating whether or not it has valid input. If one of the form controls has invalid input, the form can't be submitted.
- The element can provide callbacks for various parts of the form lifecycle—such as when the form is disabled or reset to its default state.
- The element supports the standard CSS pseudoclasses for form controls, like `:disabled` and `:invalid`.

That's a lot of features! This article won't touch on all of them, but will describe the basics needed to integrate your custom element with a form.

This section assumes a basic familiarity with custom elements. For an introduction to custom elements, see Custom Elements v1: Reusable Web Components on Web Fundamentals.

Defining a form-associated custom element

To turn a custom element into a form-associated custom element requires a few extra steps:

- Add a static `formAssociated` property to your custom element class. This tells the browser to treat the element like a form control.
- Call the `attachInternals()` method on the element to get access to extra methods and properties for form controls, like `setFormValue()` and `setValidity()`.

- Add the common properties and methods supported by form controls, like `name`, `value`, and `validity`.

Here's how those items fit into a basic custom element definition:

```
// Form-associated custom elements must be autonomous custom elements--
// meaning they must extend HTMLElement, not one of its subclasses.
class MyCounter extends HTMLElement {

  // Identify the element as a form-associated custom element
  static formAssociated = true;

  constructor() {
    super();
    // Get access to the internal form control APIs
    this.internals_ = this.attachInternals();
    // internal value for this control
    this.value_ = 0;
  }

  // Form controls usually expose a "value" property
  get value() { return this.value_; }
  set value(v) { this.value_ = v; }

  // The following properties and methods aren't strictly required,
  // but browser-level form controls provide them. Providing them helps
  // ensure consistency with browser-provided controls.
  get form() { return this.internals_.form; }
  get name() { return this.getAttribute('name'); }
  get type() { return this.localName; }
  get validity() {return this.internals_.validity; }
  get validationMessage() {return this.internals_.validationMessage; }
  get willValidate() {return this.internals_.willValidate; }

  checkValidity() { return this.internals_.checkValidity(); }
  reportValidity() {return this.internals_.reportValidity(); }

  …
}
customElements.define('my-counter', MyCounter);
```

Once registered, you can use this element wherever you'd use a browser-provided form control:

```
<form>
  <label>Number of bunnies: <my-counter></my-counter></label>
  <button type="submit">Submit</button>
</form>
```

Setting a value {: #setting-a-value }

The `attachInternals()` method returns an `ElementInternals` object that provides access to form control APIs. The most basic of these is the `setFormValue()` method, which sets the current value of the control.

The `setFormValue()` method can take one of three types of values:

- A string value.
- A `File` object.
- A `FormData` object. You can use a `FormData` object to pass multiple values (for example, a credit card input control might pass a card number, expiration date, and verification code).

To set a simple value:

```
this.internals_.setFormValue(this.value_);
```

To set multiple values, you can do something like this:

```
// Use the control's name as the base name for submitted data
const n = this.getAttribute('name');
const entries = new FormData();
entries.append(n + '-first-name', this.firstName_);
entries.append(n + '-last-name', this.lastName_);
this.internals_.setFormValue(entries);
```

The `setFormValue()` method takes a second, optional `state` parameter, used to store the internal state of the control. For more information, see Restoring form state.

Input validation

Your control can also participate in form validation by calling the `setValidity()` method on the internals object.

```
// Assume this is called whenever the internal value is updated
onUpdateValue() {
  if (!this.matches(':disabled') && this.hasAttribute('required') &&
      this.value_ < 0) {
    this.internals_.setValidity({customError: true}, 'Value cannot be negative.');
  }
  else {
    this.internals_.setValidity({});
  }
  this.internals.setFormValue(this.value_);
}
```

You can style a form-associated custom element with the `:valid` and `:invalid` pseudoclasses, just like a built-in form control.

Although you can set a validation message, Chrome currently fails to display the validation message for form-associated custom elements.

Lifecycle callbacks

A form-associated custom element API includes a set of extra lifecycle callbacks to tie in to the form lifecycle. The callbacks are optional: only implement a callback if your element needs to do something at that point in the lifecycle.

`void formAssociatedCallback(form)`

Called when the browser associates the element with a form element, or disassociates the element from a form element.

`void formDisabledCallback(disabled)`

Called after the `disabled` state of the element changes, either because the `disabled` attribute of this element was added or removed; or because the `disabled` state changed on a `<fieldset>` that's an ancestor of this element. The `disabled` parameter represents the new disabled state of the element. The element may, for example, disable elements in its shadow DOM when it is disabled.

`void formResetCallback()`

Called after the form is reset. The element should reset itself to some kind of default state. For `<input>` elements, this usually involves setting the `value` property to match the `value` attribute set in markup (or in the case of a checkbox, setting the `checked` property to match the `checked` attribute.

`void formStateRestoreCallback(state, mode)`

Called in one of two circumstances:

- When the browser restores the state of the element (for example, after a navigation, or when the browser restarts). The `mode` argument is `"restore"` in this case.
- When the browser's input-assist features such as form autofilling sets a value. The `mode` argument is `"autocomplete"` in this case.

The type of the first argument depends on how the `setFormValue()` method was called. For more details, see Restoring form state.

Restoring form state {: #restoring-form-state }

Under some circumstances—like when navigating back to a page, or restarting the browser, the browser may try to restore the form to the state the user left it in.

For a form-associated custom element, the restored state comes from the value(s) you pass to the `setFormValue()` method. You can call the method with a single value parameter, as shown in the earlier examples, or with two parameters:

```
this.internals_.setFormValue(value, state);
```

The `value` represents the submittable value of the control. The optional `state` parameter is an *internal* representation of the state of the control, which can include data that doesn't get sent to the server. The `state` parameter takes the same types as the `value` parameter—it can be a string, `File`, or `FormData` object.

The `state` parameter is useful when you can't restore a control's state based on the value alone. For example, suppose you create a color picker with multiple modes: a palette or an RGB color wheel. The submittable *value* would be the selected color in a canonical form, like `"#7fff00"`. But to restore the control to a specific state, you'd also need to know which mode it was in, so the *state* might look like `"palette/#7fff00"`.

```
this.internals_.setFormValue(this.value_,
    this.mode_ + '/' + this.value_);
```

Your code would need to restore its state based on the stored state value.

```
formStateRestoreCallback(state, mode) {
  if (mode == 'restore') {
```

```
    // expects a state parameter in the form 'controlMode/value'
    [controlMode, value] = state.split('/');
    this.mode_ = controlMode;
    this.value_ = value;
  }
  // Chrome currently doesn't handle autofill for form-associated
  // custom elements. In the autofill case, you might need to handle
  // a raw value.
}
```

In the case of a simpler control (for example a number input), the value is probably sufficient to restore the control to its previous state. If you omit `state` when calling `setFormValue()`, then the value is passed to `formStateRestoreCallback()`.

```
formStateRestoreCallback(state, mode) {
  // Simple case, restore the saved value
  this.value_ = state;
}
```

A working example

The following example puts together many of the features of form-associated custom elements. Be sure to run it on Chrome 77 or later to see the API in action.

https://glitch.com/edit/#!/form-associated-ce

Feature detection

You can use feature detection to determine whether the `formdata` event and form-associated custom elements are available. There are currently no polyfills released for either feature. In both cases, you can fall back to adding a hidden form element to propagate the control's value to the form. Many of the more advanced features of form-associated custom elements will likely be difficult or impossible to polyfill.

```
if ('FormDataEvent' in window) {
  // formdata event is supported
}
```

```
if ('ElementInternals' in window &&
    'setFormValue' in window.ElementInternals.prototype) {
  // Form-associated custom elements are supported
}
```

Conclusion

The `formdata` event and form-associated custom elements provide new tools for creating custom form controls.

The `formdata` event doesn't give you any new capabilities, but it gives you an interface for adding your form data to the submit process, without having to create a hidden `<input>` element.

The form-associated custom elements API provides a new set of capabilities for making custom form controls that work like built-in form controls.

Hero image by Oudom Pravat on Unsplash.

□□□ #3
□ □□□ □□□ 2026-10-31 22:04:03 UTC
□ cocobook-sync □□□ 2026-11-04 17:37:35 UTC