DEV.BYTES

Web Components Custom Elements

Eiji Kitamura
@agektmr

Eiji is a Developer Advocate at Google. Eiji works closely with web application developers, helping them understand HTML5 and open web technologies.

Introduction to Custom Elements

BY EIJI KITAMURA November 23rd, 2014

HTML is the most important factor for the web platform. It provides various low level features to structure sites and apps. But it also is easy to end up with div soup once you start implementing a complex component using native HTML tags. What if the web platform could allow you to create your original component? What if you can give it an arbitrary tag name? What if you can extend features of an existing HTML tag? Custom Elements allow you to do those things.

TAGS

- Introduction
- Custom Elements
- Shadow DOM
- Template

What are Custom Elements?

Custom Elements enable developers to create their own custom HTML tags, let them use those tags in their sites and apps, and enable easier component reuse.

How to build a custom element

Defining a custom element is simple. Just call `document.registerElement()` with its tag name as the first argument.

```
var XComponent = document.registerElement('x-component');
```

Now you can use `<x-component>` wherever you want in the document.

```
<x-component></x-component>
```

Note: `<x-component>` can appear in the document before the definition of the custom element execution. See HTML5Rocks article for details.

To detect the availability of Custom Elements, check if `document.registerElement` is available. Otherwise, you can simply load `webcomponents.js` to polyfill it.

```
<script src="bower_components/webcomponentsjs/webcomponents.min.js"></script>
```

Naming rules

You need to have at least one '`-`' inside the name of a custom element. Any tag names without '`-`' will result in an error.

Good

- x-component
- x-web-component

Bad

- web_component

- xelement
- XElement

Imperative usage

A defined custom tag can be used declaratively by inserting `<x-component>` tag inside HTML, but you can also take an imperative approach.

```
var XComponent = document.registerElement('x-component');
var dom = new XComponent();
document.body.appendChild(dom);
```

The above example is using `new` to instantiate a custom element.

```
document.registerElement('x-component');
var dom = document.createElement('x-component');
document.body.appendChild(dom);
```

This example uses `document.createElement()` to instantiate a custom element.

Adding features to a custom element

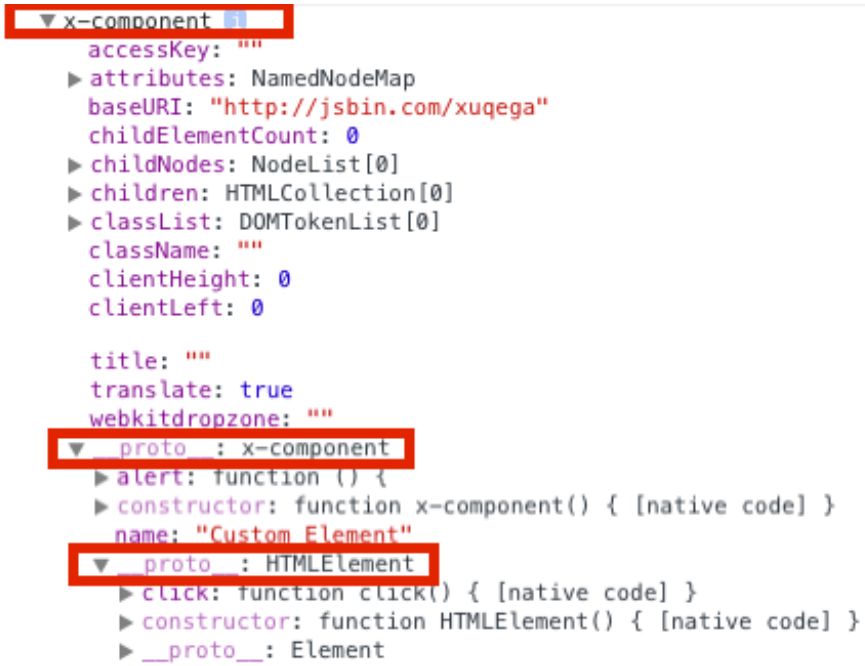
Being able to use a custom tag name itself is fine, but it doesn't do much. Let's add some features to the element.

In order to add features to a custom element, you first need to create a basic prototype object by calling `Object.create()` with `HTMLElement.prototype` as an argument. This gives you an empty prototype object with the basic HTML element feature set in its prototype chain. Add any functions and properties you want to the prototype object, then pass your prototype to document.registerElement as shown below:

```
var proto = Object.create(HTMLElement.prototype);
proto.name = 'Custom Element';
proto.alert = function() {
  alert('This is ' + this.name);
};
document.registerElement('x-component', {
  prototype: proto
});
```

Custom Element Structure

Let's see what's going on in a custom element using Chrome DevTools. Use the "Elements" panel to inspect the `x-component` tag we just created. You can see the `x-component` is an instance of a `x-component` prototype which is an instance of the `HTMLElement` prototype.



Type Extension Custom Element

You can create a custom element that extends a native HTML element's features. This is called a Type Extension Custom Element. To use the element, use the original tag and specify the custom tag name using the `is` attribute.

```
<div is="x-component"></div>
```

To define a type extension:

- Create the base prototype object using the prototype of the extended element, instead of HTMLElement.

- Add an `extends` key in the second argument to `document.registerElement()`, specifying the *tag name* of the extended element.

Following is an example code when extending the `input` element:

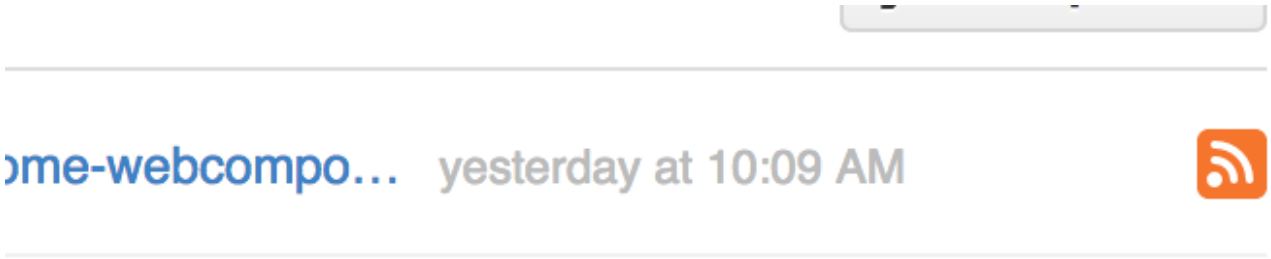
```
var XComponent = document.registerElement('x-component', {
  extends: 'input',
  prototype: Object.create(HTMLInputElement.prototype)
});
```

Notice that it `extends: 'input'` and its prototype is based on `HTMLInputElement` instead of `HTMLElement`. Now you can use `<input is="x-component">` inside your document. By doing so, you can have extended APIs on top of basic `input` element's features.

Note: You may wonder what happens if you set different elements for '`extends`' and '`prototype`'. Yes, it is possible and may cause unexpected results. But as far as I have experimented, you won't get any valuable outcome.

Use case at GitHub

So what's the point of Type Extension Custom Element? Let's look at a great existing example at the GitHub website.



GitHub has a many components that displays date and time. Notice they are not absolute dates/times but relative to the browser's current time. You should be able to imagine how to calculate that but GitHub is doing that using Type Extension Custom Element with `time-elements`.

Let's look into how it works.

```
▼<div class="time">
   <time datetime="2014-06-03T01:09:21Z" is="relative-time" title=
   "Jun 3, 2014 10:09 AM GMT+09:00">yesterday at 10:09 AM</time>
 </div>
```

There are four things you should notice:

- `time` tag is used as a base element
- `datetime` attribute indicates an absolute date/time
- `relative-time` is specified as a type extension
- `TextContent` indicates a relative date/time

This is done by calculating a relative date/time out from an absolute date/time (`datetime`) attribute on the fly as a type extension.

The benefit of using Type Extension Custom Element is that even if JavaScript is turned off or the browser doesn't support Custom Elements (including polyfill), the `time` element will still show the date/time information as a fallback keeping its semantics. Try using DevTools and turning off JavaScript; you'll notice it shows absolute dates and times.

Read webcomponents.org's How GitHub is using Web Components in production for more details about `time-elements`.

Lifecycle callbacks

I mentioned the `relative-time` custom element inserts a relative date/time into `TextContent` on the fly. But when does that happen? You can define functions to be called when certain events happened on Custom Elements, which are called "lifecycle callbacks".

Here's the list of lifecycle callbacks:

.createdCallback() Called after the element is created.

.attachedCallback() Called when the element is attached to the document

.detachedCallback() Called when the element is detached from the document.

.attributeChangedCallback() Called when one of attributes of the element is changed.

In case of `relative-time`, `.createdCallback()` and `.attributeChangedCallback()` are hooked up to insert a relative date/time to `TextContent`.

Example

To use lifecycle callbacks, just define the functions as properties of a prototype object when registering a custom element.

```
var proto = Object.create(HTMLElement.prototype);
proto.createdCallback = function() {
  var div = document.createElement('div');
  div.textContent = 'This is Custom Element';
  this.appendChild(div);
};
var XComponent = document.registerElement('x-component', {
  prototype: proto
});
```

Combining Custom Elements with Templates and Shadow DOM

By using Templates and Shadow DOM in a custom element, you can

make the element easier to handle and resusable. With templates, defining the content of your custom element can be declarative. With Shadow DOM, styles, ids and classes of the content can be scoped to itself.

You can utilize them when the custom element is created using `.createdCallback()`. Let's have a look at a sample code. To learn about Templates and Shadow DOM, read the respective articles (Template, Shadow DOM) written previously.

HTML

```
<!-- Template Definition -->
<template id="template">
  <style>
    ...
  </style>
  <div id="container">
    <img src="http://webcomponents.org/img/logo.svg">
    <content select="h1"></content>
  </div>
</template>
```

```
<!-- Custom Element usage -->
<x-component>
  <h1>This is Custom Element</h1>
</x-component>
```

JavaScript

```
var proto = Object.create(HTMLElement.prototype);
proto.createdCallback = function() {
  // Adding a Shadow DOM
  var root = this.createShadowRoot();
  // Adding a template
  var template = document.querySelector('#template');
  var clone = document.importNode(template.content, true);
  root.appendChild(clone);
}
var XComponent = document.registerElement('x-component', {
  prototype: proto
});
```

Here's a live example.

Supported browsers

Custom Elements are supported by Chrome and Opera. Firefox supports them behind a flag as of November 2014. To check availability, go to chromestatus.com or caniuse.com. For polyfilling other browsers, you can use webcomponents.js (renamed from platform.js).

Resources

So that's the Custom Elements. As you may have noticed, Custom Elements are used in the production of GitHub supporting IE9 by using polyfill. Now is your time to try this feature.

If you are interested in learning more about the Custom Elements, head over to:

- Custom Elements: defining new elements in HTML - HTML5Rocks
- Custom Elements spec