DEV PREVIEW

Polymer in 10 minutes

Creating elements

Edit on GitHub

▶ **Table of contents**

Polymer makes it simple to create web components, declaratively. Custom elements are defined using our custom element, `<polymer-element>`, and can leverage Polymer's special features. These features reduce boilerplate and make it even easier to build complex, web component-based applications:

- Two-way data binding
- Declarative event handling
- Declarative inheritance
- Property observation

Setup

1. Install Polymer

Install the latest version of Polymer as described in Getting the code.

If you want to play with Polymer without installing anything, skip to Using Polymer's features. You can run and edit the samples online using Plunker.

2. Build a Polymer element

Polymer provides extra goodies for creating declarative, souped-up custom elements. We call these "Polymer elements". From the outside they look just like any other DOM element, but inside they're filled with handy features like two-way data binding and other bits of Polymer magic. These features make it easy to build complex components with much less code.

To create a new element:

1. Load the Polymer core (`polymer.html`).
2. Declare your custom element using `<polymer-element>`.

In the following example, we define a new element named `<my-element>`, save it to a file `elements/my-element.html`, and use an HTML Import to load the `polymer.html` dependency.

my-element.html

```
<link rel="import" href="../bower_components/polymer/p
```

```
<polymer-element name="my-element" noscript>
  <template>
    <span>Hello from <b>my-element</b>. This is my Sha
  </template>
</polymer-element>
```

Two items to notice:

- The `name` attribute is required and **must** contain a "-". It specifies the name of the HTML tag you'll instantiate in markup (in this case `<my-element>`).
- The `noscript` attribute indicates that this is a simple element that doesn't include any script. An element declared with `noscript` is registered automatically.

Reusing other elements

By composing simple elements together we can build richer, more complex components. To reuse other elements in your `<polymer-element>`, install the element in your app:

```
bower install Polymer/core-ajax
```

and include an import that loads the new dependency in

`my-element.html`:

```
<link rel="import" href="../bower_components/polymer/p
<link rel="import" href="../bower_components/core-aja

<polymer-element name="my-element" noscript>
  <template>
    <span>I'm <b>my-element</b>. This is my Shadow DO
    <core-ajax url="http://example.com/json" auto resp
    <textarea value="{{resp}}"></textarea>
  </template>
</polymer-element>
```

3. Create an app

Lastly, create an `index.html` that imports your new element. Remember to include `webcomponents.min.js` to load polyfills for the native APIs.

Here's the full example:

```
<!DOCTYPE html>
<html>
  <head>
    <!-- 1. Load platform support before any code that
    <script src="bower_components/webcomponentsjs/webc
```

```
    <!-- 2. Load the component using an HTML Import -->
    <link rel="import" href="elements/my-element.html">
  </head>
  <body>
    <!-- 3. Declare the element by its tag. -->
    <my-element></my-element>
  </body>
</html>
```

Note: You must run your app from a web server for the HTML Imports to work properly. They cannot be loaded from `file://` due to the browser's security restrictions.

Your final directory structure should look something like this:

```
yourapp/
  bower_components/
    webcomponentsjs/
    polymer/
    core-ajax/
  elements/
    my-element.html
  index.html
```

Now that you've got the basic setup, it's time to start using the features!

Using Polymer's features

Polymer provides a number of sugaring APIs for authoring web components. Below are a few of the concepts. Consult the API reference for detailed information on each of these features.

Add properties and methods

When you're creating a new element, you'll often need to expose a public API so users can configure it. To define a public API, include a `<script>` tag that calls the `Polymer(...)` constructor. The `Polymer(...)` constructor is a convenience wrapper for `document.registerElement`, but also endows the element with special features like data binding and event mapping. The Polymer constructor takes as an argument an object that defines your element's prototype.

proto-element.html index.html EDIT ON PLUNKER

```
<link rel="import"
      href="../bower_components/polymer/polymer.html">

<polymer-element name="proto-element">
  <template>
    <span>I'm <b>proto-element</b>. Check out my proto
  </template>
  <script>
    Polymer({
```

```
      ready: function() {
        //...
      }
    });
  </script>
</polymer-element>
```

Result

I'm **proto-element**. Check out my prototype.

Adding lifecycle methods

Lifecycle callbacks are special methods you can define on your element which fire when the element goes through important transitions.

When a custom element has been registered it calls its `created()` callback (if one has been defined). When Polymer finishes its initialization, the `ready()` method is called. The `ready` callback is a great place to do constructor-like initialization work.

ready-element.html index.html EDIT ON PLUNKER

```
<link rel="import"
      href="../bower_components/polymer/polymer.html">
```

```
<polymer-element name="ready-element">
  <template>
    This element has a ready() method.
    <span id="el">Not ready...</span>
  </template>
  <script>
    Polymer({
      owner: "Daniel",
      ready: function() {
        this.$.el.textContent = this.owner +
                                " is ready!";
      }
    });
  </script>
</polymer-element>
```

Result

This element has a ready() method. Daniel is ready!

Learn more about all of the lifecycle callbacks.

Declarative data binding

Data binding is a great way to quickly propagate changes in your element and reduce boilerplate code. You can bind properties in your component using the “double-mustache” syntax (`{{}}`). The `{{}}` is replaced by the value of the property referenced between the brackets.

name-tag.html index.html EDIT ON PLUNKER

```
<link rel="import"
      href="../bower_components/polymer/polymer.html">

<polymer-element name="name-tag">
  <template>
    This is <b>{{owner}}</b>'s name-tag element.
  </template>
  <script>
    Polymer({
      owner: 'Daniel'
    });
  </script>
</polymer-element>
```

Result

This is **Daniel**'s name-tag element.

Note: Polymer's data-binding is powered under the covers by a sub-library called TemplateBinding, designed for other libraries to build on top of.

Binding to markup

You can use binding expressions in most HTML markup, except for tag

names themselves. In the following example, we create a new property on our component named `color` whose value is bound to the value of the `color` style applied to the custom element. Bindings ensure that any time a property like `color` is changed, the new value will be propagated to all binding points.

fav-color.html index.html EDIT ON PLUNKER

```
<link rel="import"
      href="../bower_components/polymer/polymer.html">

<polymer-element name="fav-color">
  <template>
    This is <b>{{owner}}</b>'s fav-color element.
    {{owner}} likes the color
    <span style="color: {{color}}">{{color}}</span>.
  </template>
  <script>
    Polymer({
      owner: "Daniel",
      color: "red"
    });
  </script>
</polymer-element>
```

Result

This is **Daniel**'s fav-color element. Daniel likes the color red.

Binding between components and built-in elements

You can use bindings with built-in elements just like you would with Polymer elements. This is a great way to leverage existing APIs to build complex components. The following example demonstrates binding component properties to attributes of native input elements.

age-slider.html index.html EDIT ON PLUNKER

```
<link rel="import"
      href="../bower_components/polymer/polymer.html">

<polymer-element name="age-slider">
  <template>
    This is <b>{{owner}}</b>'s age-slider.
    <b>{{name}}</b> lets me borrow it.
    He likes the color <span style="color: {{color}}">
    I am <b>{{age}}</b> years old.
    <p><label for="ageInput">Age:</label>
    <input id="ageInput" type="range"
           value="{{age}}">
    <label for="nameInput">Name:</label>
    <input id="nameInput" value="{{name}}"
           placeholder="Enter name..."></p>
  </template>
  <script>
```

```
    Polymer({
      age: 25,
      name: "Daniel",
      color: "red",
      owner: "Eric",
      nameChanged: function() {
        if (this.name) {
          // Ensure name is capitalized
          this.name = this.name[0].toUpperCase() +
                      this.name.slice(1);
        }
      }
    });
</script>
</polymer-element>
```

Result

This is **Eric**'s age-slider. **Daniel** lets me borrow it. He likes the color red. I am **25** years old.

Age: Name: Daniel

Note: Giving `age` an initial value of `25` gives Polymer a hint that this property is an integer.

In this example, `nameChanged()` defines a property changed watcher.

Polymer will then call this method any time the `name` property is updated. Read more about changed watchers.

Publishing properties

Published properties can be used to define an element's "public API". Polymer establishes two-way data binding for published properties and provides access to the property's value using `{{}}`.

Publish a property by listing it in the `attributes` attribute in your `<polymer-element>`. Properties declared this way are initially `null`. To provide a more appropriate default value, include the same property name directly in your prototype.

The following example defines two data-bound properties on the element, `owner` and `color`, and gives them default values:

color-picker.html index.html EDIT ON PLUNKER

```
<link rel="import"
      href="../bower_components/polymer/polymer.html">

<polymer-element name="color-picker"
         attributes="owner color">
  <template>
    This is <strong>{{owner}}</strong>'s color-picker
    He likes the color <b style="color: {{color}}">{{
  </template>
```

```
  <script>
    Polymer({
      // These default values are overridden
      // by the user's attribute values.
      color: "red",
      owner: "Daniel"
    });
  </script>
</polymer-element>
```

Result

This is **Scott**'s color-picker. He likes the color **blue**.

In this example the user overrides the defaults for `owner` and `color` by configuring the element with initial attribute values (e.g. `<color-picker owner="Scott" color="blue">`).

Note : When binding a property that takes a type other than String, it's important to hint a property's type. Polymer relies on this information to correctly serialize and de-serialize values.

Learn more about published properties.

Automatic node finding

The use of the `id` attribute has traditionally been discouraged as an anti-

pattern because the document requires element IDs to be unique. Shadow DOM, on the other hand, is a self-contained document-like subtree; IDs in that subtree do not interact with IDs in other trees. This means the use of IDs in Shadow DOM is not only permissible, it's actually encouraged. Each Polymer element generates a map of IDs to node references in the element's template. This map is accessible as `$` on the element and can be used to quickly select the node you wish to work with.

editable-color-picker.html index.html EDIT ON PLUNKER

```
<link rel="import"
      href="../bower_components/polymer/polymer.html">

<polymer-element name="editable-color-picker" attribut
  <template>
    <p>
    This is a <strong>{{owner}}</strong>'s editable-co
    He likes the color <b style="color: {{color}}">{{c
    </p>
    <button on-click="{{setFocus}}">Set focus to text
    <input id="nameInput" value="{{owner}}"
           placeholder="Your name here...">
  </template>
  <script>
    Polymer({
      color: "red",
      owner: "Daniel",
      setFocus: function() {
```

```
        this.$.nameInput.focus();
      }
    });
  </script>
</polymer-element>
```

Result

This is a **Daniel**'s editable-color-picker. He likes the color **red**.

Set focus to text input Daniel

Learn more about automatic node finding

Next steps

Now that you know how to create your own elements, follow the tutorial to create your first Polymer app, or dive deeper and read up on Polymer's core API. Continue on to:

→ API DEVELOPER GUIDE

→ YOUR FIRST POLYMER APP

+POLYMER @POLYMER

/POLYMER FILE A BUG