



Mentally Friendly [Follow](#)

A Sydney based agency of super passionate strategists, designers and developers.  
Oct 4, 2016 · 8 min read

## Creating a Web Component with Polymer

by Liam Fiddler—Senior Full-Stack Developer



Creating a Web Component with Polymer - Part 1

This is the first in a three part guide which will demonstrate an approach for writing, documenting, and testing a Web Component with Polymer.

- Part 1 will explain what a Web Component is and build a component that shows/hides list items based on user input.
- Part 2 will extend the component by adding support for variable list item content and case-insensitive filtering. It will also provide an introduction to component documentation.
- Part 3 will cover the writing and execution of tests for the component using the Web Component Tester library.

. . .

### What is a Web Component?

Web Components are a collection of standardised features currently being added by the W3C to the HTML and DOM specifications. They allow you to encapsulate HTML, CSS, and Javascript into reusable widgets or components in web documents and web applications. Part of

this encapsulation is handled by the Shadow DOM that scopes your component's style and DOM.

Web Components have native support in over 55% of web browsers globally at the time of writing, with greater support being added in every new release. For the remaining browsers we'll be using Polymer as a kind of 'polyfill'.

## Assumed Knowledge & Prerequisites

A working understanding of HTML, CSS, and Javascript will be required to follow this guide.

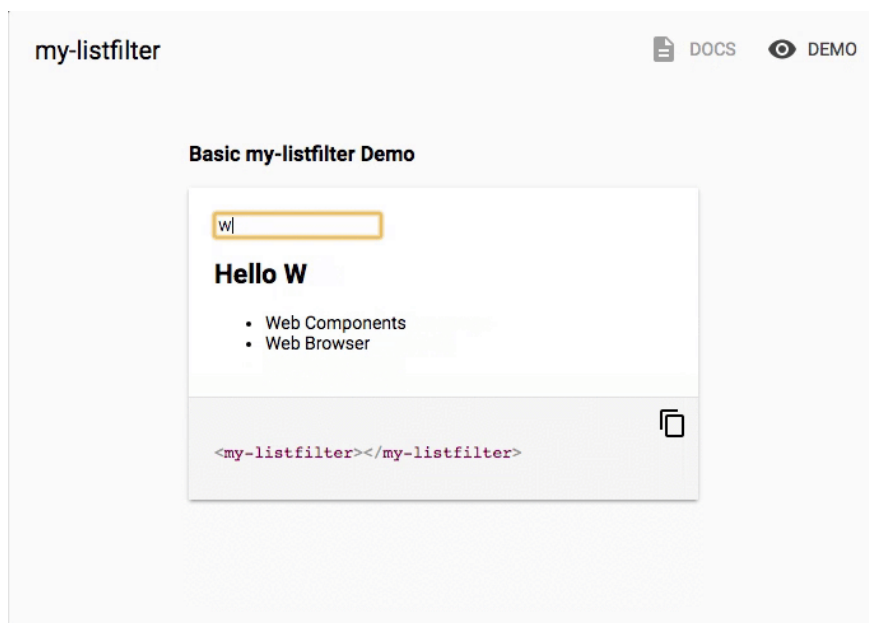
Additionally, this guide assumes you have the Polymer CLI installed on your system. Polymer CLI is a command-line interface for Polymer projects. It includes a build pipeline, a boilerplate generator for creating elements and apps, a linter, a development server, and a test runner.

Finally, although it's not a requirement, Chrome was the first browser to ship with support for all of the new Web Component standards so it's the recommended browser for this guide.

## Our Goal

For this guide we'll be building a component that allows a user to filter list items based on a search string.

It will look something like this:



# Getting Started

We're going to call our new component **my-listfilter** (the custom elements specification requires the component name contain a dash).

Open your preferred commandline shell and paste the following:

```
mkdir my-listfilter
cd my-listfilter
```

This will create a new directory for the project and change to that directory. Next, we want to initialise the project. We'll use the Polymer CLI to set it up:

```
polymer init
```

You'll be asked a series of questions about the project you're setting up. Select element, name the project **my-listfilter** and enter a description for the project (something like 'A filterable list component'). Polymer CLI will then generate a number of files and directories for your element and install the required dependencies.

## The Component Skeleton

Open *my-listfilter/my-listfilter.html* in your preferred editor. It should look something like this:

```

1 <link rel="import" href="../polymer/polymer.html">
2 <!--
3 `my-listfilter`
4 A filterable list component
5
6 @demo demo/index.html
7 -->
8 <dom-module id="my-listfilter">
9   <template>
10     <style>
11       :host {
12         display: block;
13       }
14     </style>
15     <h2>Hello [[prop1]]</h2>
16   </template>
17   <script>
18     Polymer({
19       is: 'my-listfilter',

```

First we see a link tag, this is importing the Polymer dependencies into our component.

Next is a HTML comment containing the name of our component and the description of the component we provided to Polymer CLI. This is used to generate the documentation for the component.

The comment ends on a line that starts with *@demo*, this line specifies that the component comes with a demo page that can be found at *demo/index.html* (this will come into play in Part 2 of this article series).

Further down the file we find the *DOM-MODULE* tag. The *DOM-MODULE* is broken into two sections; a *TEMPLATE* which houses the component styles & markup, and a *SCRIPT* which contains the logic & functionality.

Inside the *TEMPLATE* tag you will see *[[prop1]]*. This is known as a one-way binding. When the component is put on a page this text will be replaced with the value of a variable called “prop1”.

The “prop1” variable and its value are set in the *SCRIPT* section within the *properties* object. By default it is set to the string “my-listfilter”.

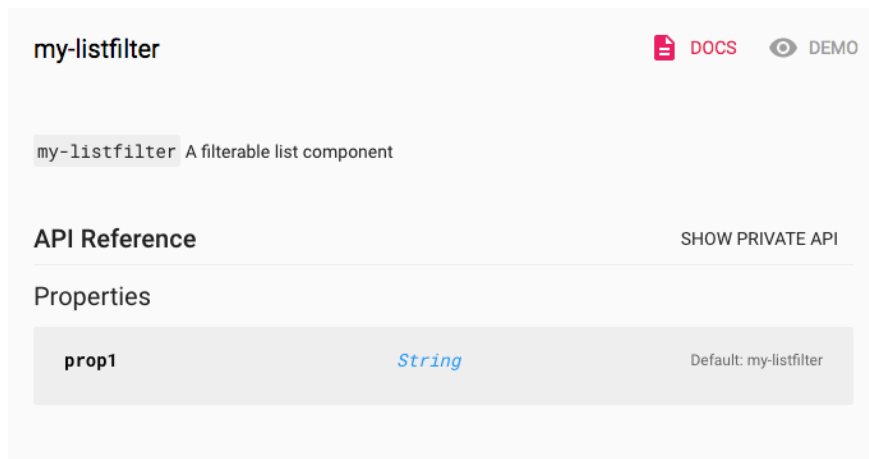
## Viewing the Component in the Browser

At this stage you should be able to view the component in your web browser.

Back in your commandline shell type the following:

```
polymer serve
```

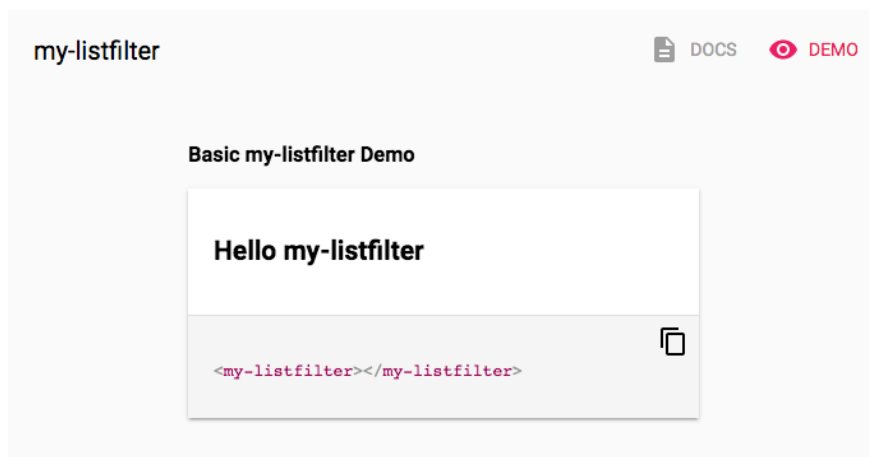
A tool called “polyserve” will start and a couple of URLs will be displayed. Open the “reusable components” URL in your web browser, it will be something like *http://localhost:8080/components/my-listfilter/*



The screenshot shows the documentation page for the 'my-listfilter' component. At the top left is the component name 'my-listfilter'. To the right are links for 'DOCS' and 'DEMO'. Below the name is a description: 'my-listfilter A filterable list component'. There is an 'API Reference' section with a 'SHOW PRIVATE API' link. Underneath is a 'Properties' section with a table:

| Property Name | Type   | Default Value          |
|---------------|--------|------------------------|
| prop1         | String | Default: my-listfilter |

This is the documentation that Polymer has generated for your component. Clicking the “demo” link in the top corner should take you to another page that looks like this:



The screenshot shows the demo page for the 'my-listfilter' component. At the top left is the component name 'my-listfilter'. To the right are links for 'DOCS' and 'DEMO'. Below the name is the title 'Basic my-listfilter Demo'. The main content area shows a white box with the text 'Hello my-listfilter'. Below this box is a code block containing the HTML template: `<my-listfilter></my-listfilter>`. A copy icon is visible next to the code block.

On this page we can test the component and make sure it's working correctly.

## Adding The Filter Input

One of the key pieces in our listfilter component will be the text input field used for filtering. To accomplish this we'll be using a standard HTML input tag.

Jump back to your code editor and insert the following on a new line after the closing *STYLE* tag and before the opening *H2* tag:

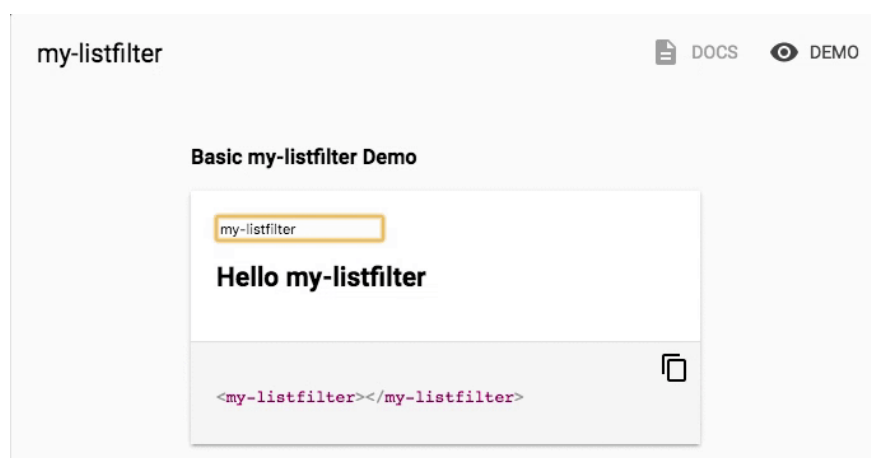
```
<input value="{{prop1::input}}" />
```

This creates a text input on the page.

You'll note we've also added a value attribute to the input. The content of the attribute is what makes the tag special. By specifying prop1 in curly braces we've told Polymer to create a two-way binding between the "prop1" variable and the input's value property. Additionally, by specifying *::input* after the variable name we've told Polymer to automatically update the variable's value whenever the value property changes.

Your *my-listfilter.html* file should now look like this.

Save the file and refresh the demo page in your web browser. Now when you type into the input you should see the content of the *H2* tag change!



## Adding the List Markup and Styles

The other key piece in our listfilter component will be the list itself. For this guide we'll be using a *UL* tag to represent a list.

Paste the following in your code editor after the closing *H2* tag:

```
<ul id="list">
  <li>Hypertext Markup Language</li>
  <li>Cascading Style Sheets</li>
  <li>Javascript</li>
  <li>Web Components</li>
  <li>Polymer</li>
  <li>Web Browser</li>
  <li>Mentally Friendly</li>
</ul>
```

Note that we've added an id attribute to the *UL* tag. This will allow us to easily target the element when we start writing the filtering functionality.

Then, between the style tags in the template we'll add some CSS to hide list items:

```
.hidden {
  display: none;
}
```

Your *my-listfilter.html* file should now look like this.

If you refresh the demo page in your browser you will now see the list items below the *INPUT* tag, but the list is not being filtered yet. On to the next step!

## Marking List Items as Hidden

At this stage we've produced the markup for an input (whose value is attached to a variable) and a list. The next step will be to write a function that can show or hide the list items if they contain the input value.

Add the following to the script section, before the final `});`

```
filterList: function() {
  var items = this.$.list.children;

  for (var i = 0; i < items.length; i++) {
    var item = Polymer.dom(items[i]);

    if (item.textContent.includes(this.prop1)) {
      item.classList.remove('hidden');
    } else {
      item.classList.add('hidden');
    }
  }
}
```

There's a lot going on in the above snippet so let's go through it step-by-step.

The first line defines a name for our function, “filterList”.

The second line uses a neat trick in Polymer to retrieve the list items and store them in an array. When you create a node in the template section with an *id* attribute it will be automatically added to the *this.\$* hash with the id as its key. This means we can access the list we created earlier as *this.\$list* without having to query the DOM manually!

Over the next two lines we loop through the list items and store them in the variable *item*. Here we use the *Polymer.dom* method to get a reference to the node in the Shadow DOM, ensuring any changes we make to the item are properly maintained when the component is rendered.

We then check to see if the text content of the item includes the filter text. If the filter text is found we remove the “hidden” class from the node, otherwise the “hidden” class is added.

Your *my-listfilter.html* file should now [look like this](#).

## Running a Function when the Input Changes

We've got a function that can show/hide list items based on the search text, but how do we get it to run when the user types new text in the input field?

In the script section there is a block of code that looks like this:



```
prop1: {
  type: String,
  value: 'my-listfilter',
},
```

This defines the *prop1* variable we're using. The type of the variable is set to *String* and the default value is *my-listfilter*.

Let's change the value to an empty string so the input field is cleared by default, and add an observer property:

```
prop1: {
  type: String,
  value: '',
  observer: 'filterList'
},
```

Observers are methods invoked when observable changes occur to the element's data, this includes when the data is first defined and any changes that occur thereafter (even if it becomes undefined again). In this case we've told Polymer to call the the function we created earlier whenever the value is updated.

. . .

## Review

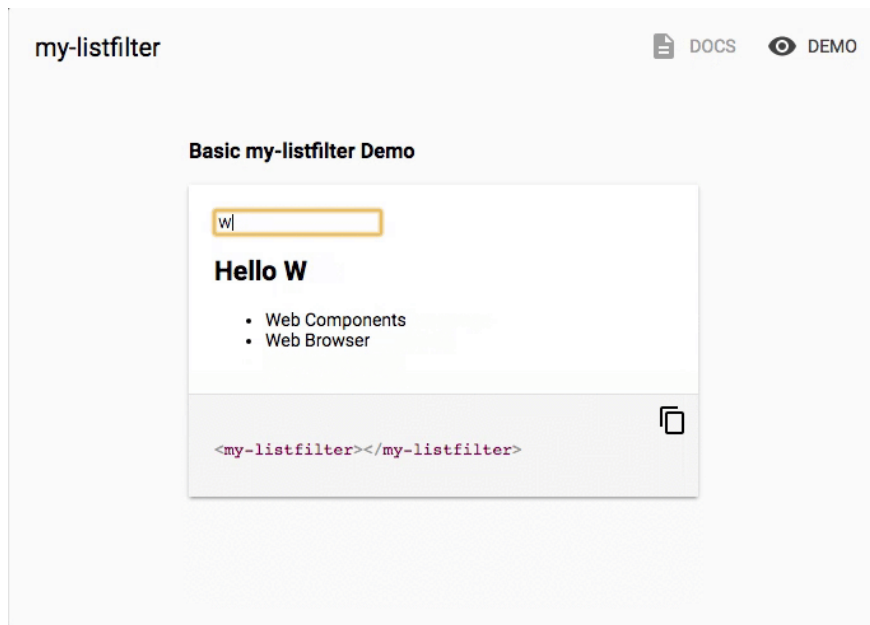
If you've been following along your *my-listfilter.html* file will now look like this:

```

1 <link rel="import" href="../polymer/polymer.html">
2 <!--
3 `my-listfilter`
4 A filterable list component
5
6 @demo demo/index.html
7 -->
8 <dom-module id="my-listfilter">
9   <template>
10     <style>
11       :host {
12         display: block;
13       }
14       .hidden {
15         display: none;
16       }
17     </style>
18     <input value="{{prop1::input}}" />
19     <h2>Hello [[prop1]]</h2>
20     <ul id="list">
21       <li>Hypertext Markup Language</li>
22       <li>Cascading Style Sheets</li>
23       <li>Javascript</li>
24       <li>Web Components</li>
25       <li>Polymer</li>
26       <li>Web Browser</li>
27       <li>Mentally Friendly</li>
28     </ul>
29   </template>
30   <script>
31     Polymer({
32       is: 'my-listfilter',
33       properties: {
34         prop1: {
35           type: String,

```

Save the file and refresh the demo page in your web browser. If everything's gone to plan you should be able to type into the text field and see the list items disappear if they don't match!



## Next Steps

In the next article we'll extend the component by adding support for variable list item content and case-insensitive filtering. We'll also look at methods for improving the readability of the code, as well as documenting the component.