POLYMER SYD

Liam Fiddler

@liamfiddler
liam@mf.agency

[INTRO]

A few weeks ago Google held the 2017 Polymer Summit.

It was around this time last year Polymer 2 was announced and a technical preview made available, which brought with it some pretty significant changes.

JavaScript objects were gone, classes were in, Shadow DOM support was growing, and the V1 web component spec was not only in development, but it was actively landing in modern browsers..

How were they going to top that this year?

What changes could be on the horizon, and how will they impact our workflow?

Polymer 3!?

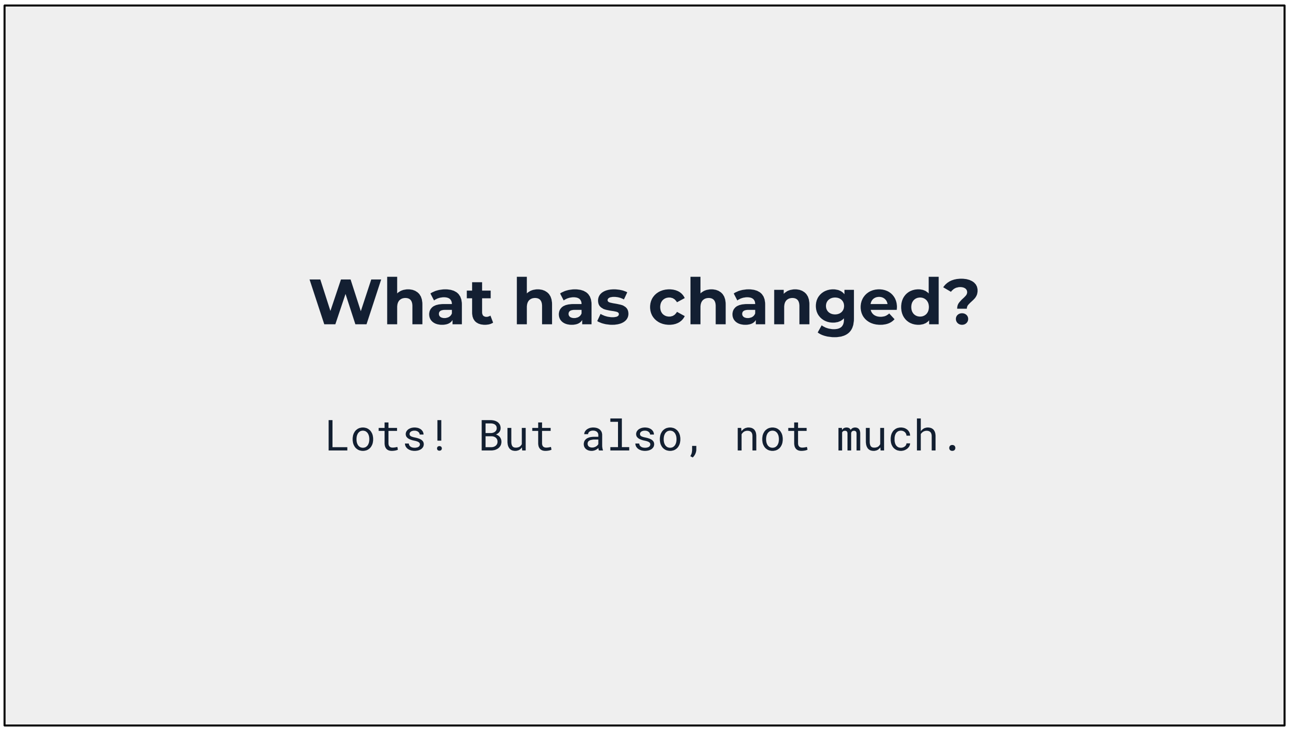
So you can imagine my surprise when Polymer 3 was announced.

I mean, have people even been using Polymer 2 long enough to warrant a new version?

What kind of changes would warrant a new major point increase?

Well the development team is quick to point out that there’s still lots of work do before Polymer 3 is ready.

They’re releasing an early access preview for you to start using and they’re asking for feedback around how it works. Particularly if you’re using Polymer to develop shareable components.



What has changed?

Lots! But also, not much.

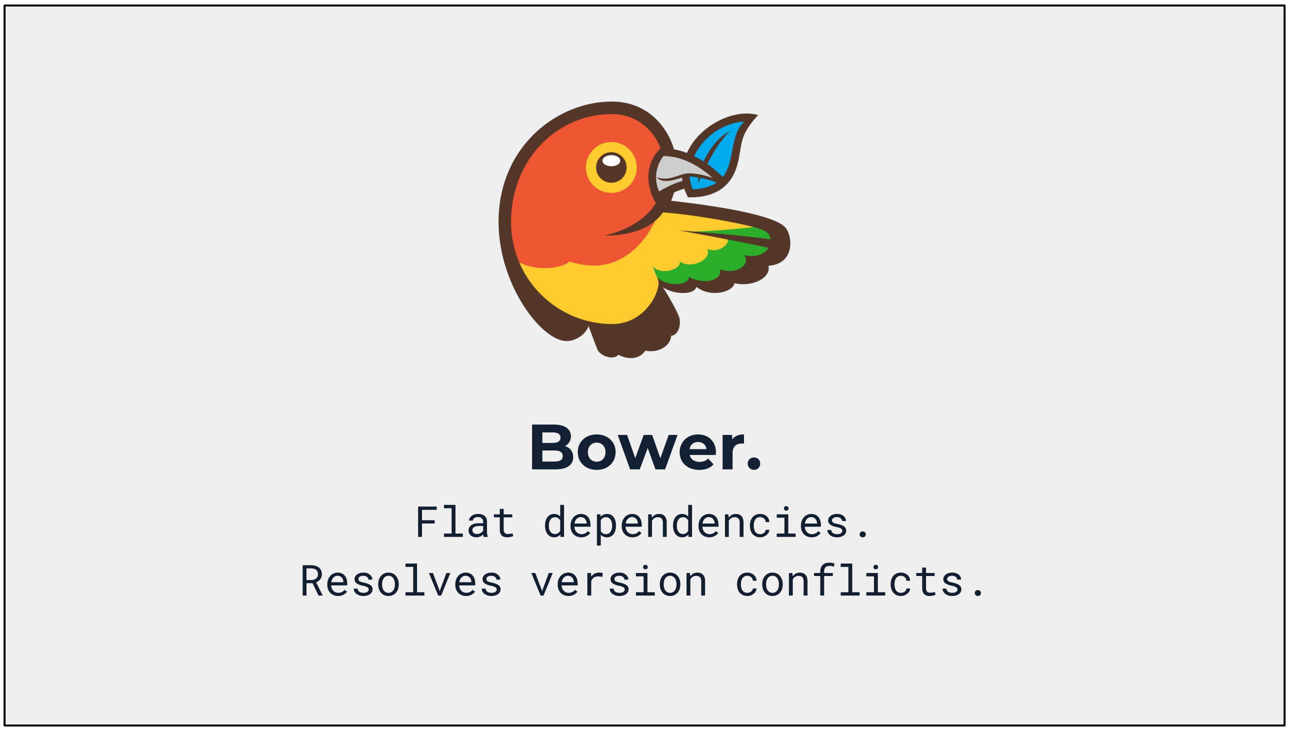
There's two really big things that you'll probably want to be aware of:

#1 the package manager is being swapped out.

#2 the method of importing files is changing.

The core API will remain the same and your component code will basically port over from Polymer 2.

In fact, the Polymer team has developed a tool that will automatically convert your code to use the new syntax (I'll talk about this soon).



Bower.

Flat dependencies.
Resolves version conflicts.

Bower had some great things going for it.

It was one of the first on the scene, that I can remember, that could handle a flat dependency tree, deduplicate multiple dependencies, and handle resolution of version conflicts.

There are a bunch of tools that can do this for you as a build step after you’ve written whatever code you need, but this is something that is really best handled by a package manager before you write any code.

But the community slowly faded away. It has been languishing in maintenance purgatory since it was deprecated.

NPM on the other hand. THAT has a really active community.

But NPM doesn’t manage flat dependencies or resolve version conflicts.

Yarn.

yarnpkg.com

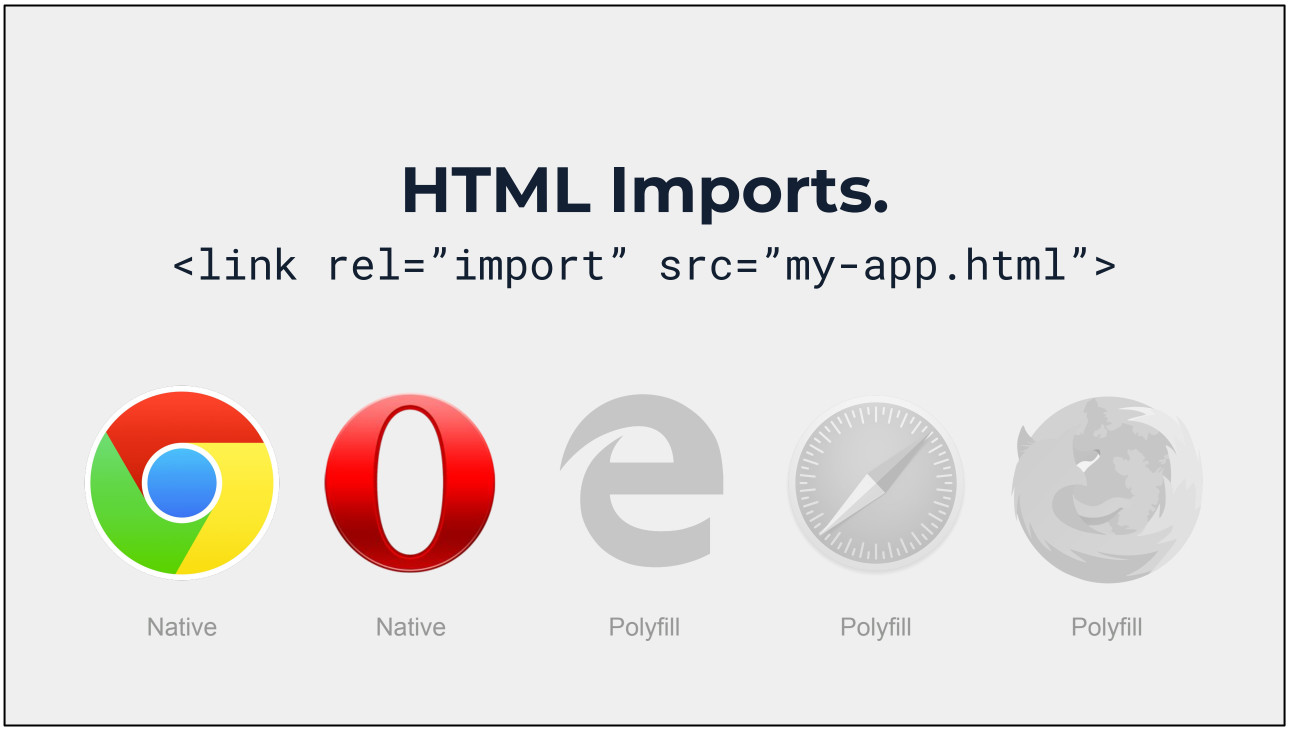
Enter Yarn.

Yarn is an alternative client for NPM that adds a few missing features.

But the two keys features it adds are setting up the dependencies in a flat tree and handling version conflicts!

So now we have access to the hundreds of thousands of packages and community support of NPM, without losing the benefits afforded by Bower.

Also it has all these sweet illustrations of cats on it’s homepage. So there’s that.



HTML Imports.

```
<link rel="import" src="my-app.html">
```

Native Native Polyfill Polyfill Polyfill

HTML imports is a spec that allows you to natively include other HTML files.

Those files can define and load their own dependencies. Which can load their dependencies, and so on.

And the great thing about HTML is it just knows how to deal with all sorts of files.

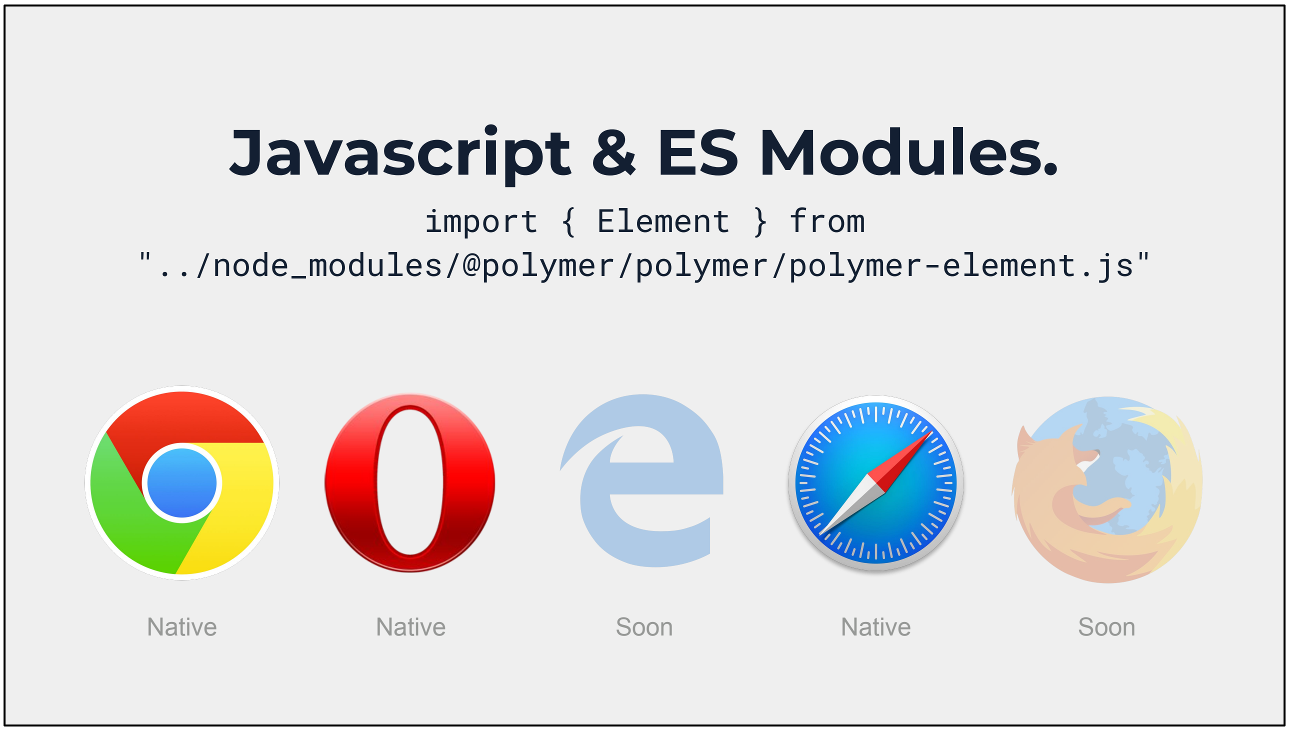
You can embed CSS or JavaScript. Load images. Render a canvas. Whatever.

On top of that it could include things at a file-level, so no bundling or additional tooling is required.

BUT unfortunately it never quite managed to get full browser support.

Edge, Safari, & Firefox all haven't moved forward with development of the spec for a number of different reasons.

And that means that polyfills are still required in most browsers, and probably will be for a very long time. Maybe forever.



Javascript & ES Modules.

```
import { Element } from
"../node_modules/@polymer/polymer/polymer-element.js"
```

Native Native Soon Native Soon

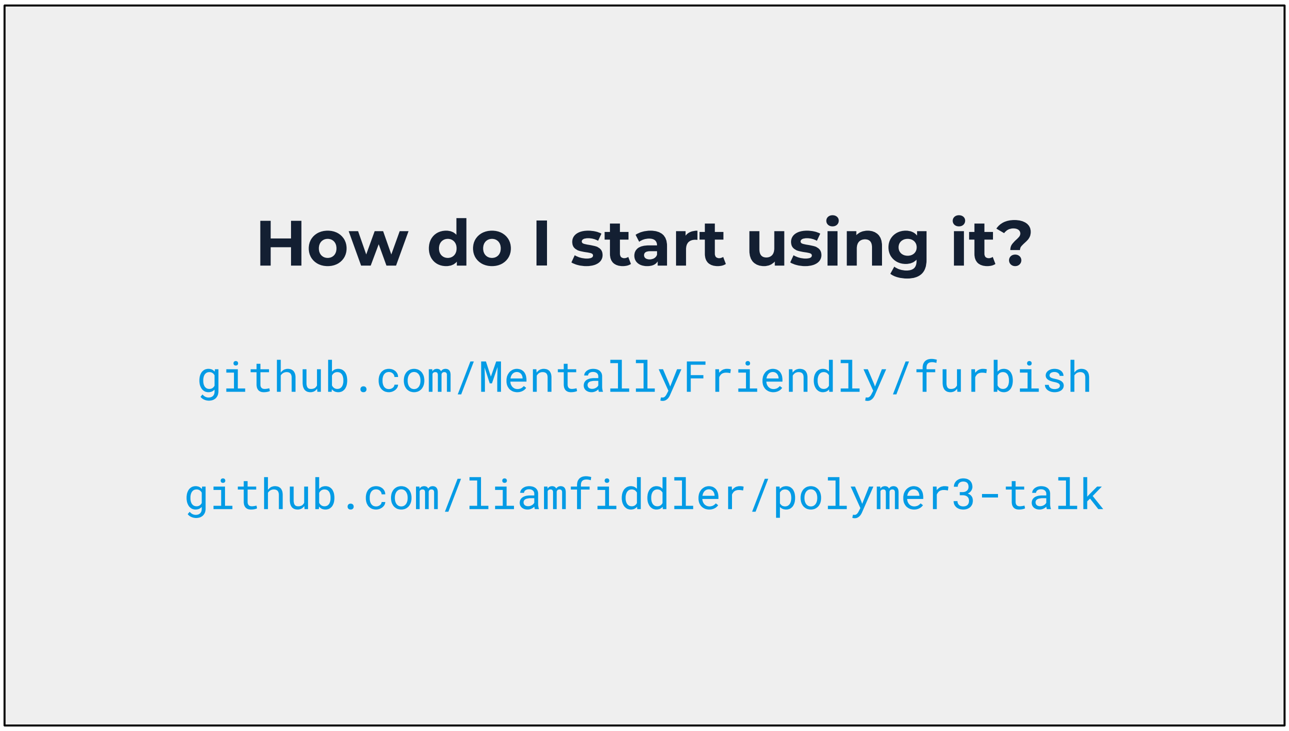
There’s another spec out there that can accomplish many of the same things HTML Imports set out to do; ES Modules.

ES Modules are a native spec that can load files with a deep dependency tree, and importantly it has fairly good support.

Chrome, Safari, & Opera have already shipped browsers with full native support. While Edge and Firefox are actively developing with an aim to release soon.

Coincidentally, Chrome, Safari, & Opera all have native support for template tags, Shadow DOM, and Custom Elements out of the box. So once the switch to ES Modules is made Polymer (& web components in general) will have full native support without any polyfills in 3 of the 5 major browsers!

JavaScript’s support of other languages is a little weak the moment, styles and markup are a particular kind of pain, but there are lots of libraries and approaches that can be taken to make it work.


How do I start using it?

github.com/MentallyFriendly/furbish

github.com/liamfiddler/polymer3-talk

So at this point you’re probably wondering “how do I get started?”

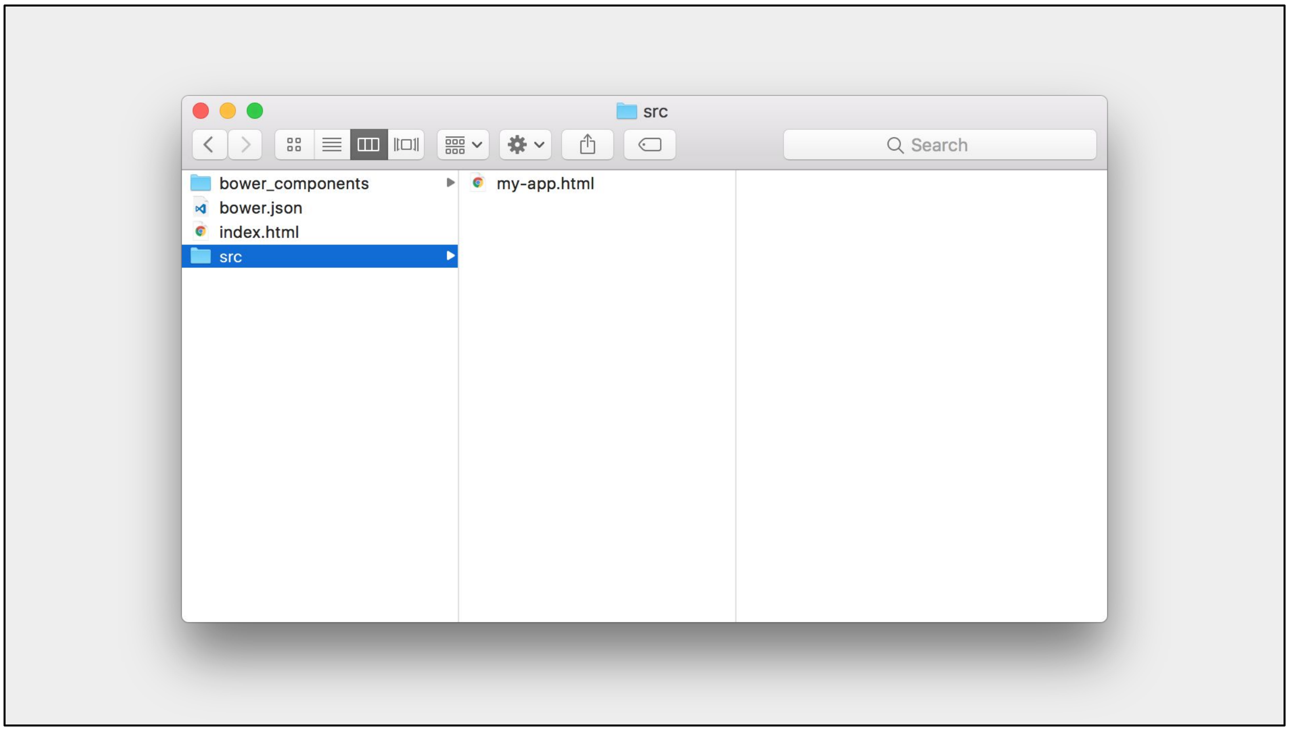
I definitely was!

I’m happy to say it’s not difficult, but at this stage there are a few steps involved.

We built a script named Furbish that bootstraps out a basic app structure using Polymer 3 and installs all the dependencies for you. You can find it at this URL. It’s really easy, you don’t need to clone the repo and pull and files down, it’s designed to run directly from GitHub.

I’ve also put together some code examples at the URL below which demonstrate the process I went through while testing Polymer 3.

So, now, let’s take a look at a pretty basic Polymer 2 component and see what’s involved in getting it ready for Polymer 3.



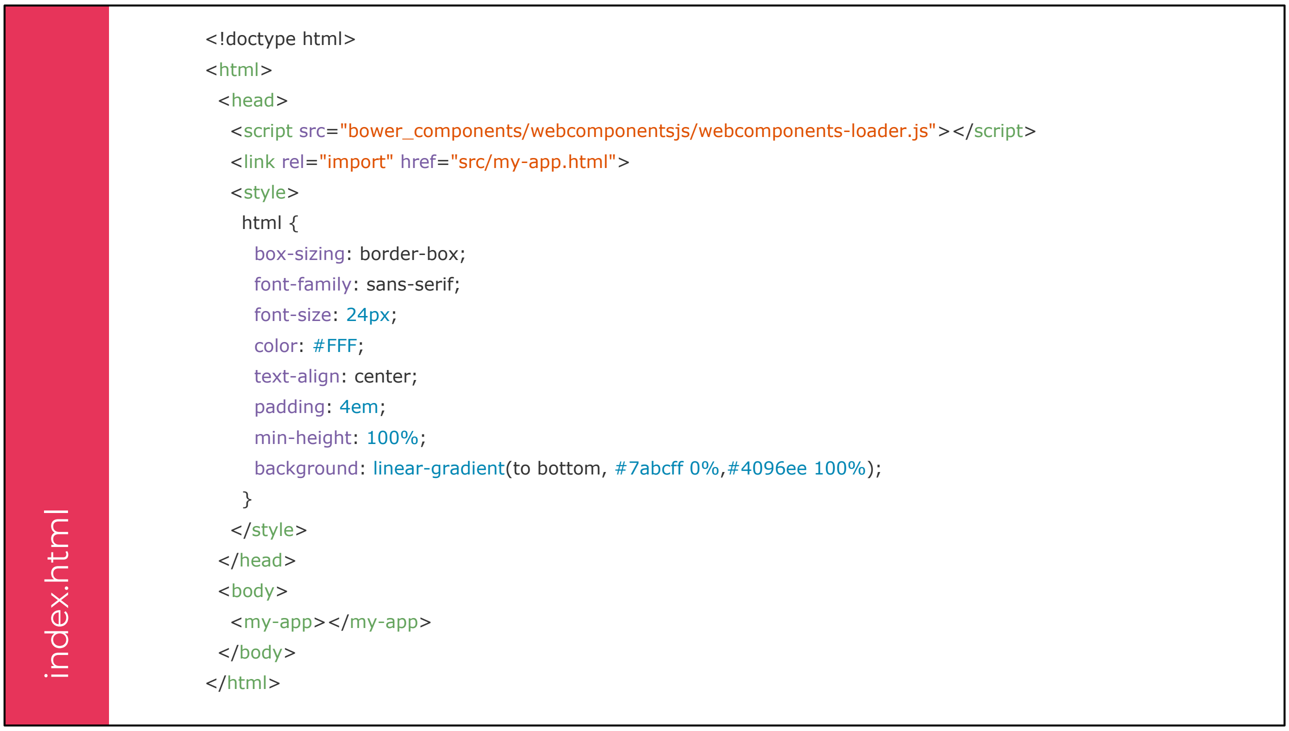
We start off with an app structure that has two main files, an index.html file and a my-app.html file.

And of course, our bower.json file & bower_components directory.

This project has only two dependencies installed via bower.

Bower install polymer
Bower install webcomponentsjs

This should look pretty familiar to any Polymer 2 devs in the room.

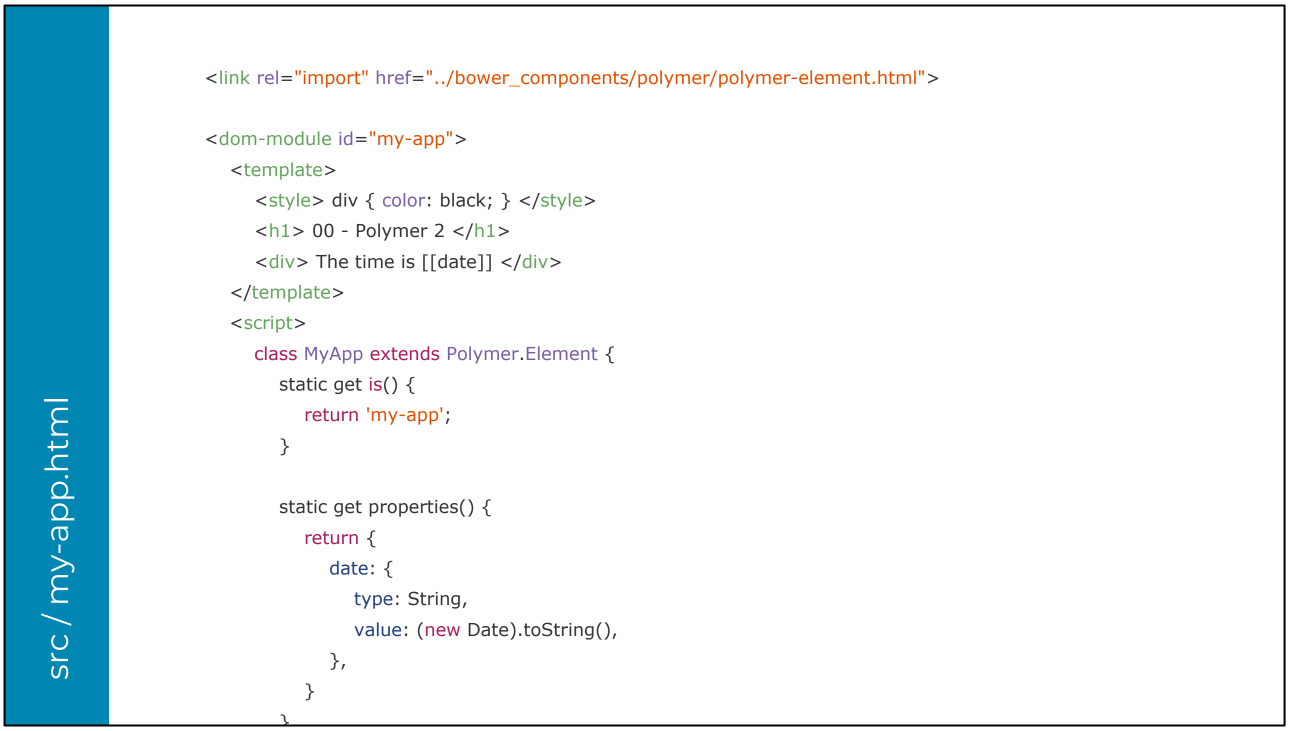


index.html

```html
<!doctype html>
<html>
 <head>
  <script src="bower_components/webcomponentsjs/webcomponents-loader.js"></script>
  <link rel="import" href="src/my-app.html">
  <style>
   html {
    box-sizing: border-box;
    font-family: sans-serif;
    font-size: 24px;
    color: #FFF;
    text-align: center;
    padding: 4em;
    min-height: 100%;
    background: linear-gradient(to bottom, #7abcff 0%,#4096ee 100%);
   }
  </style>
 </head>
 <body>
  <my-app></my-app>
 </body>
</html>
```

The index.html file will look something like this…

We've got the web components loader in there, followed by a HTML Import with our my-app.html file, some styles, and finally we're using our my-app tag in the body.



src / my-app.html

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

<dom-module id="my-app">
  <template>
    <style> div { color: black; } </style>
    <h1> 00 - Polymer 2 </h1>
    <div> The time is [[date]] </div>
  </template>
  <script>
    class MyApp extends Polymer.Element {
      static get is() {
        return 'my-app';
      }

      static get properties() {
        return {
          date: {
            type: String,
            value: (new Date).toString(),
          },
        }
```

And this is our my-app.html file.

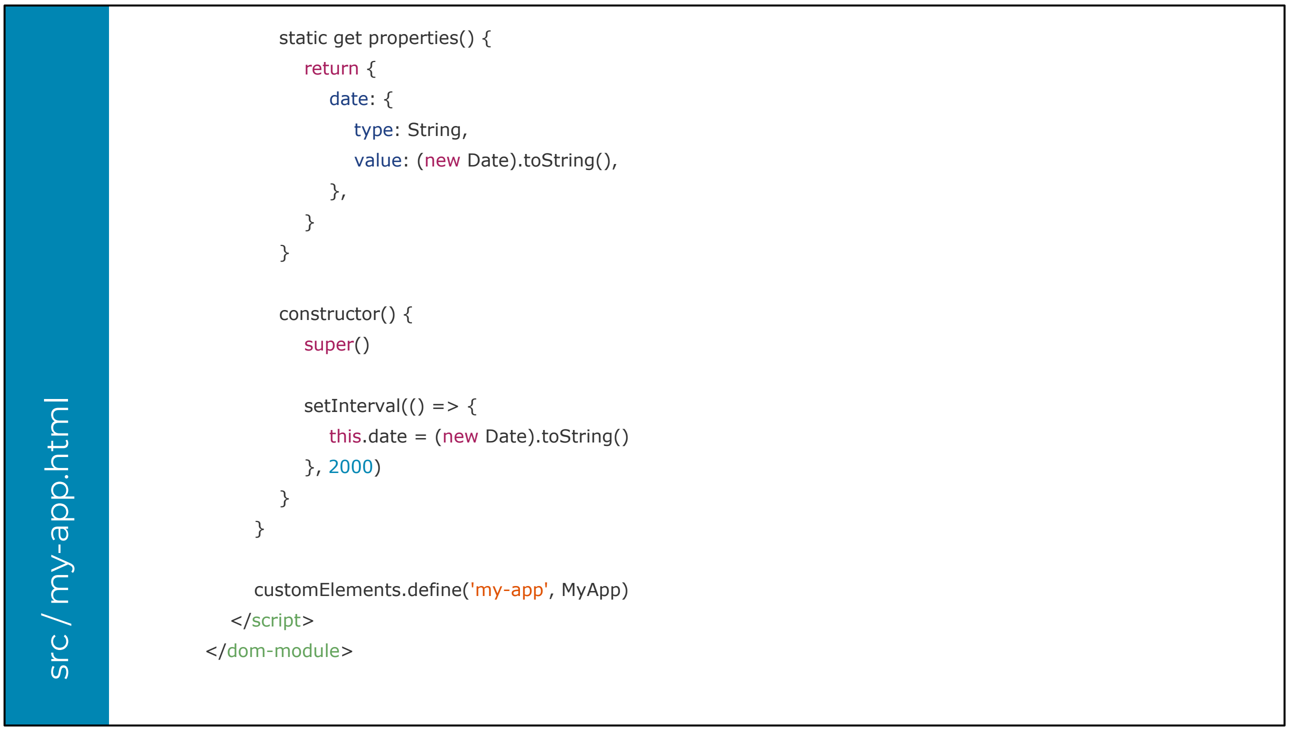
We start by importing our bower dependencies, Polymer Element.

Open a dom-module with a unique ID.

Create a template with some scoped styles, a heading, and a one-way binding to a date variable that we define later in the file.

After that comes the script tag where we extend Polymer Element, set that unique ID again so when/if you elect to use a build step Polymer knows which template to use.

And configure some properties. We'll set that date variable to be a string.


src / my-app.html

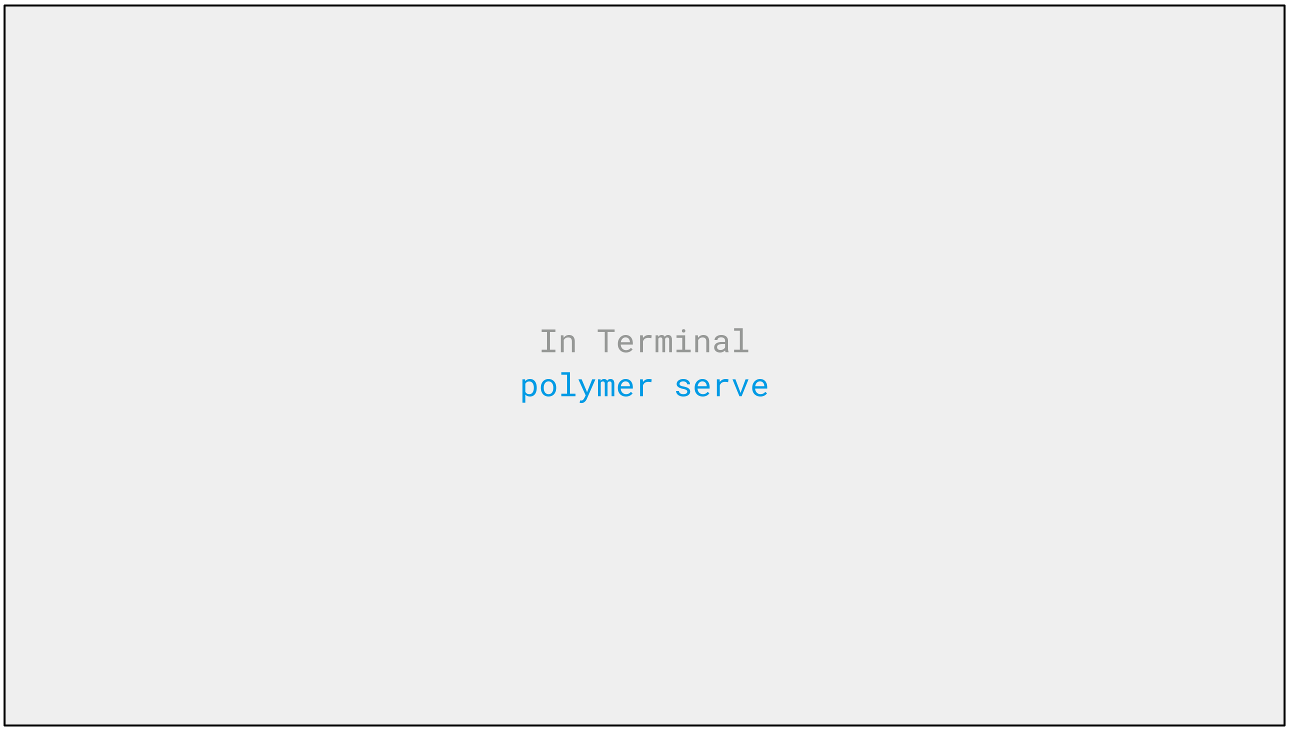
```
        static get properties() {
          return {
            date: {
              type: String,
              value: (new Date).toString(),
            },
          }
        }

        constructor() {
          super()

          setInterval(() => {
            this.date = (new Date).toString()
          }, 2000)
        }
      }

      customElements.define('my-app', MyApp)
    </script>
  </dom-module>
```

I've just scrolled the code a little here, but you can see our properties at the top of the screen now.

Because this is a just a dumb example component we're going to set an interval in the constructor that updates the date string every two seconds.

In the real world we'd probably using something other than setInterval, but this is just a test.

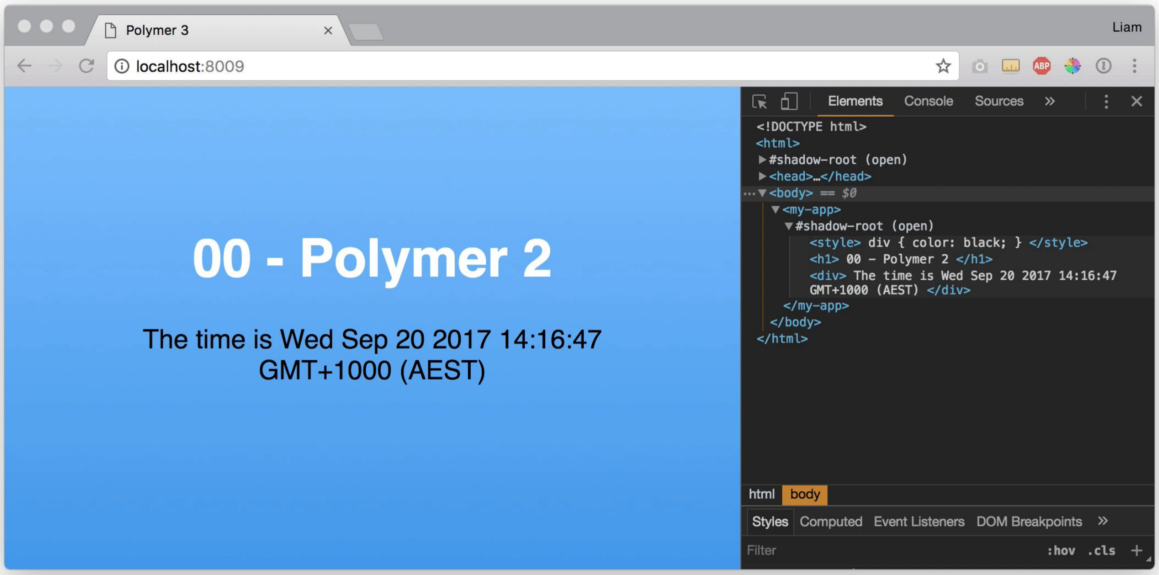
Finally, we call customElements.define and our component is ready to use.



```
In Terminal
polymer serve
```

Then we serve the project using the Polymer CLI.

And we open our web browser to find….



```
<!DOCTYPE html>
<html>
  ▶#shadow-root (open)
  ▶<head>…</head>
…▼<body> == $0
    ▼<my-app>
      ▼#shadow-root (open)
          <style> div { color: black; } </style>
          <h1> 00 - Polymer 2 </h1>
          <div> The time is Wed Sep 20 2017 14:16:47
          GMT+1000 (AEST) </div>
      </my-app>
    </body>
</html>
```

This!

This is what it should look like.

OK this is pretty good, but let’s get started converting this to Polymer 3!

```
In Terminal
rm -r ./bower*
yarn init
mv ./src/my-app.html ./src/my-app.js

In package.json
“flat: true”

In Terminal
yarn add @polymer/polymer@next
yarn add @webcomponents/webcomponentsjs@^1.0.11
```

First we remove bower and the bower components directory, while leaving the rest of our source code in place.

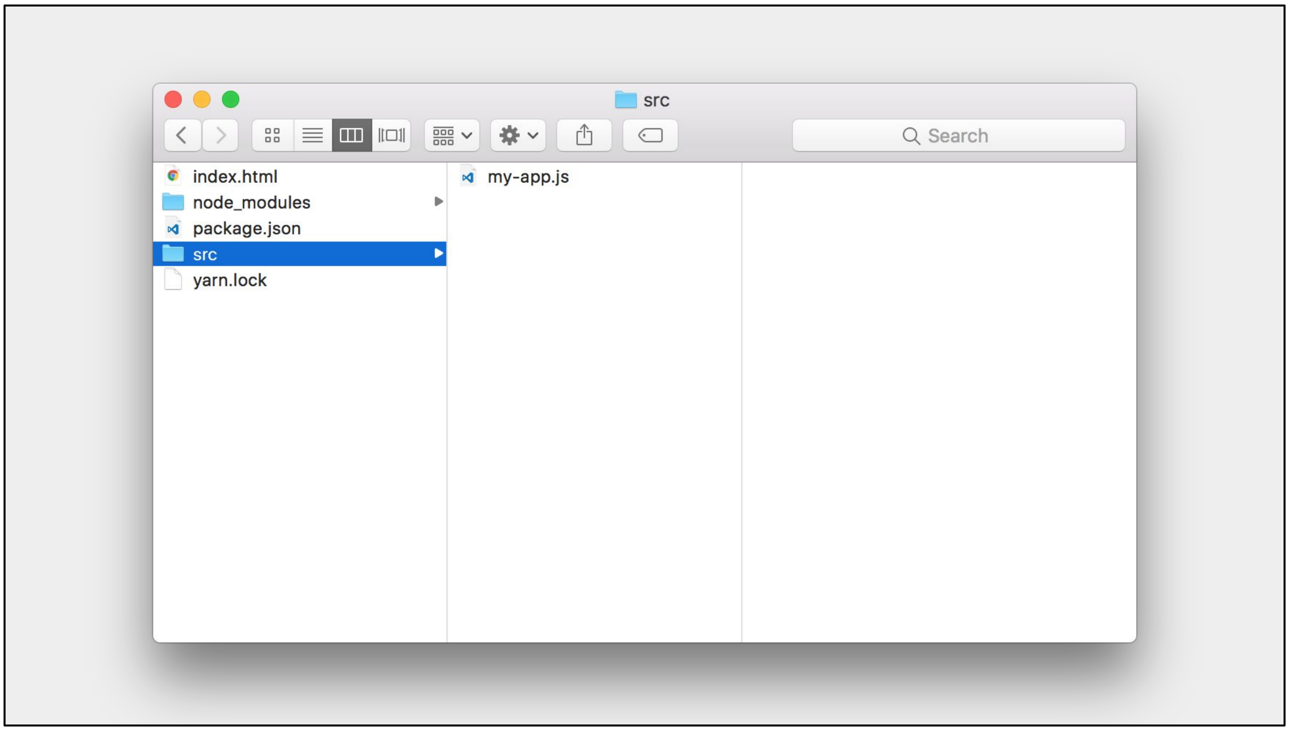
Then we initialise the project with Yarn.

Yarn will ask you a few questions, like what’s your project name, is it private, what’s the license, all the normal NPM stuff.

Then we rename our HTML Import file to have a .js file extension

Then we open the resulting package.json file and add a new key, “flat: true”. This tells Yarn to install our dependencies in a flat tree.

Back in our terminal we add and install our dependencies.

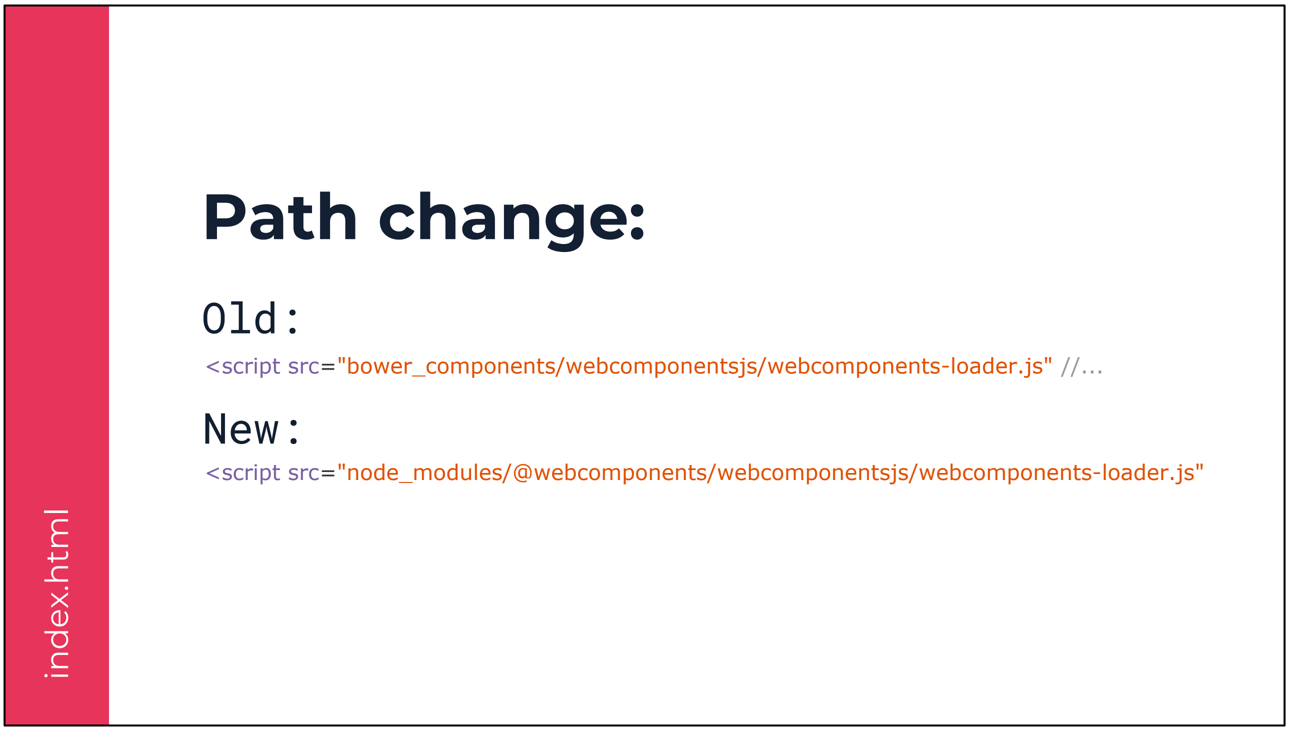


Our directory will look something like this.

index.html

```
<!doctype html>
<html>
  <head>
    <script src="bower_components/webcomponentsjs/webcomponents-loader.js"></script>
    <link rel="import" href="src/my-app.html">
    <style>
      html {
        box-sizing: border-box;
        font-family: sans-serif;
        font-size: 24px;
        color: #FFF;
        text-align: center;
        padding: 4em;
        min-height: 100%;
        background: linear-gradient(to bottom, #7abcff 0%,#4096ee 100%);
      }
    </style>
  </head>
  <body>
    <my-app></my-app>
  </body>
</html>
```

So this is just a reminder of what index.html looked like



index.html

Path change:

Old:

```
<script src="bower_components/webcomponentsjs/webcomponents-loader.js" //...
```

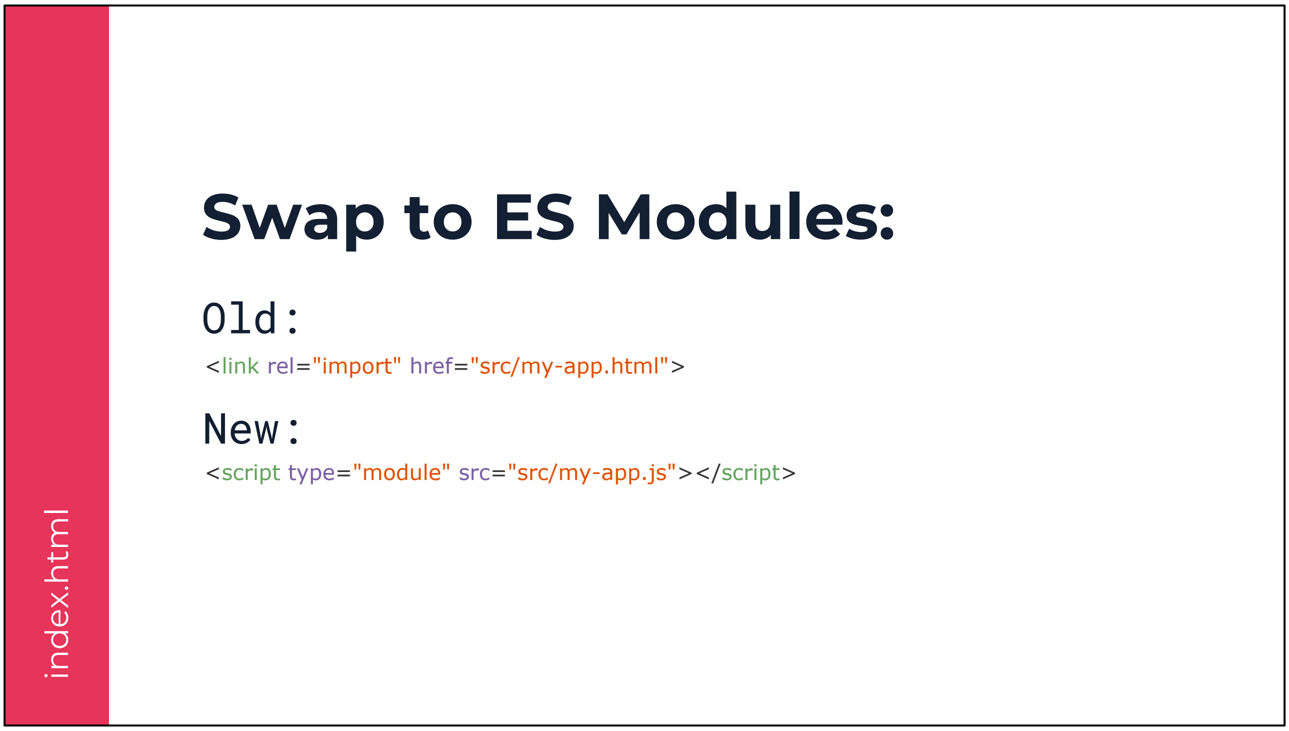
New:

```
<script src="node_modules/@webcomponents/webcomponentsjs/webcomponents-loader.js"
```

We can then jump into our index.html file and make a few small changes...

First we need to update the old bower_components path to this new node_modules path.

Note the namespacing of @webcomponents



index.html

Swap to ES Modules:

Old:

```html
<link rel="import" href="src/my-app.html">
```

New:

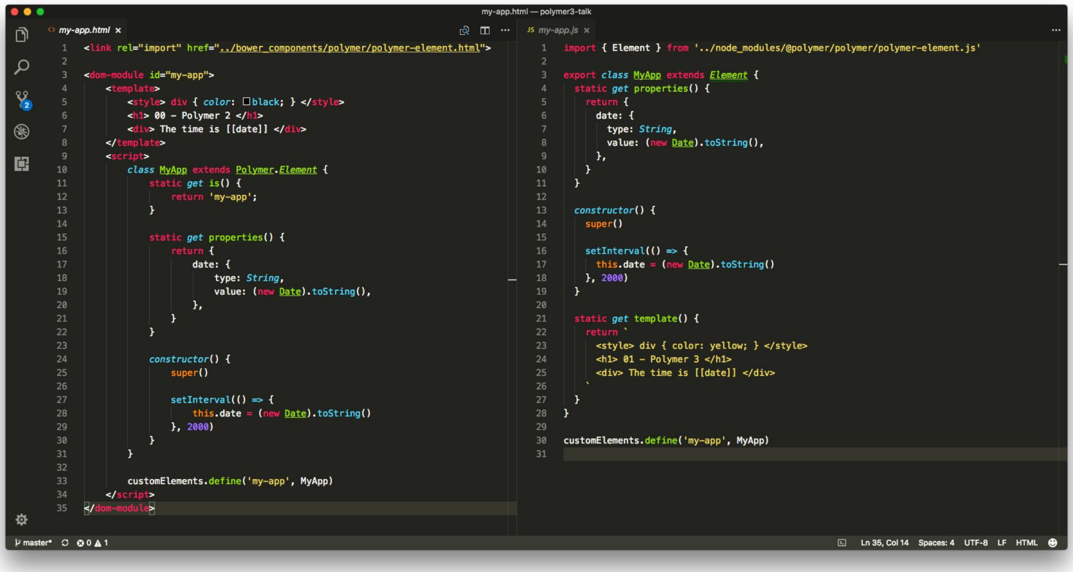
```html
<script type="module" src="src/my-app.js"></script>
```

Then we need to update the method by which we’re importing our custom element.

HTML Imports are out, and module imports are in.

Note we’ve also updated the file extension to JS here.

src / my-app.js



my-app.html — polymer3-talk

my-app.html

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

<dom-module id="my-app">
    <template>
        <style> div { color: black; } </style>
        <h1> 00 - Polymer 2 </h1>
        <div> The time is [[date]] </div>
    </template>
    <script>
        class MyApp extends Polymer.Element {
            static get is() {
                return 'my-app';
            }

            static get properties() {
                return {
                    date: {
                        type: String,
                        value: (new Date).toString(),
                    },
                }
            }

            constructor() {
                super()

                setInterval(() => {
                    this.date = (new Date).toString()
                }, 2000)
            }
        }

        customElements.define('my-app', MyApp)
    </script>
</dom-module>
```

my-app.js

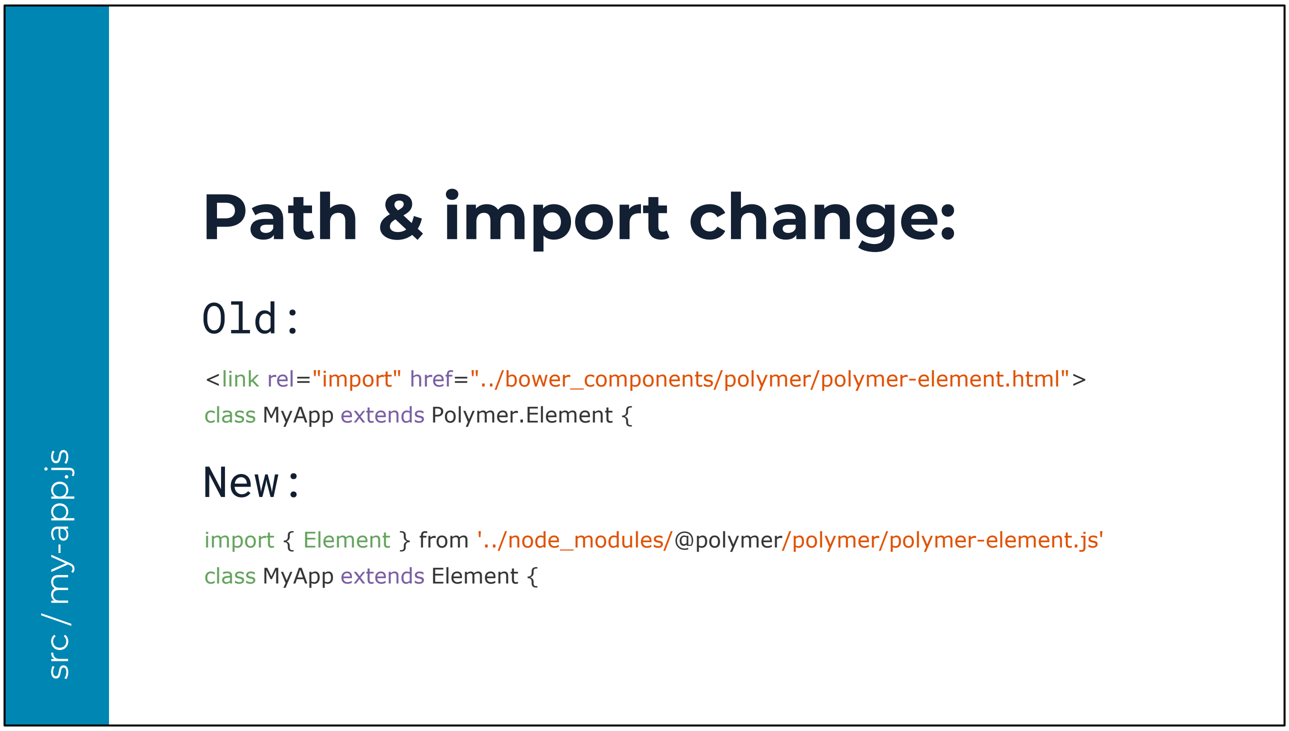
```js
import { Element } from '../node_modules/@polymer/polymer/polymer-element.js'

export class MyApp extends Element {
  static get properties() {
    return {
      date: {
        type: String,
        value: (new Date).toString(),
      },
    }
  }

  constructor() {
    super()

    setInterval(() => {
      this.date = (new Date).toString()
    }, 2000)
  }

  static get template() {
    return `
      <style> div { color: yellow; } </style>
      <h1> 01 - Polymer 3 </h1>
      <div> The time is [[date]] </div>
    `
  }
}

customElements.define('my-app', MyApp)
```

```
master*  ⊗ 0 ⚠ 1                Ln 35, Col 14  Spaces: 4  UTF-8  LF  HTML
```

Over to the my-app.js file we have some more significant changes to deal with.

Overall the code complexity is about the same. The new version is a couple of lines shorter if you’re golfing.

The biggest changes are related to the import at the top of the file and the way templates are handled.



src / my-app.js

Path & import change:

Old:

```
<link rel="import" href="../bower_components/polymer/polymer-element.html">
class MyApp extends Polymer.Element {
```

New:

```
import { Element } from '../node_modules/@polymer/polymer/polymer-element.js'
class MyApp extends Element {
```

We're now using Javascript imports which allows us to pull in named bits and pieces from modules into our scope.

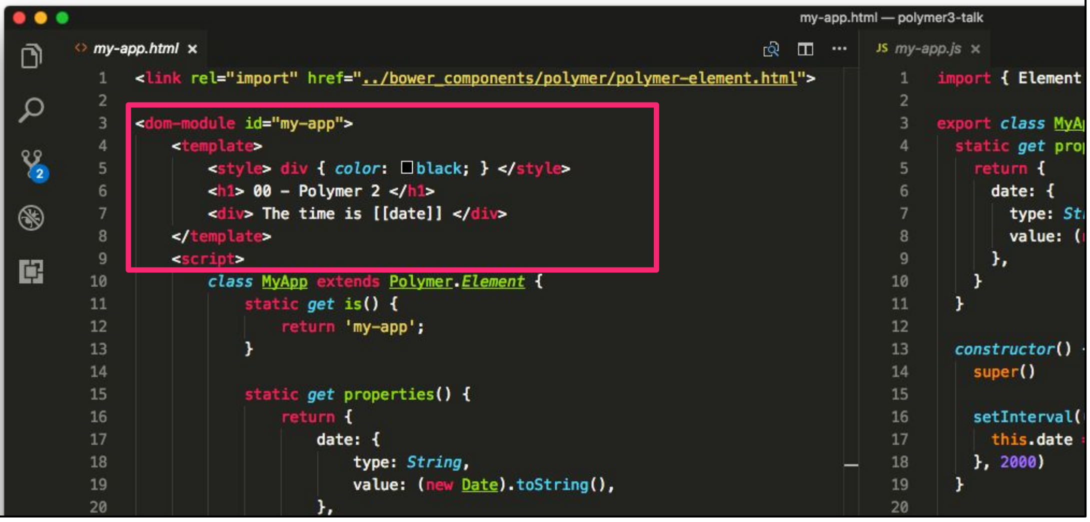
You might already be using this type of import in your code if you work with other libraries, but please note that this uses the native implementation of modules.

That means we need to make the path explicit. The browser doesn't know which directories are in your NODE_PATH, for examples, and it doesn't automatically append your file extension.

We also update the name of the class we're extending to just “Element” to match our import statement.

src / my-app.js

Remove all the HTML:



my-app.html — polymer3-talk

my-app.html

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

<dom-module id="my-app">
    <template>
        <style> div { color: ■black; } </style>
        <h1> 00 - Polymer 2 </h1>
        <div> The time is [[date]] </div>
    </template>
    <script>
        class MyApp extends Polymer.Element {
            static get is() {
                return 'my-app';
            }

            static get properties() {
                return {
                    date: {
                        type: String,
                        value: (new Date).toString(),
                    },
```

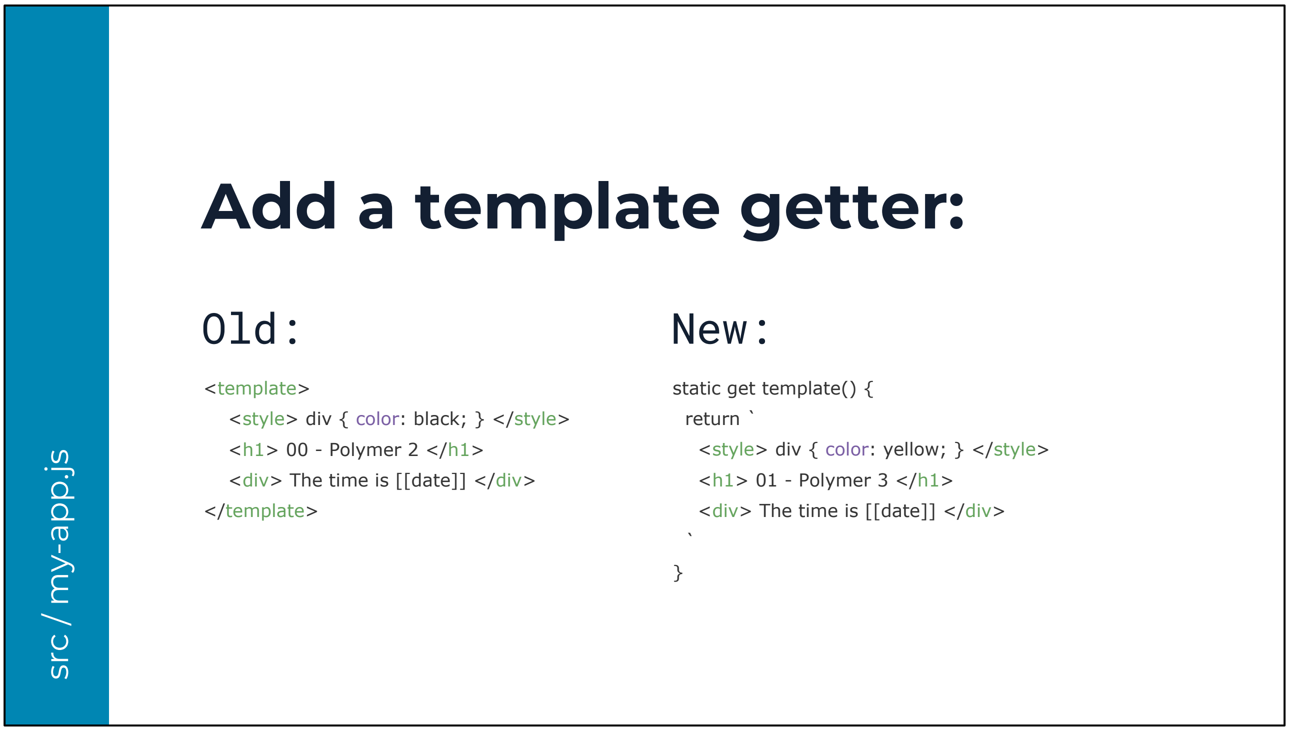
my-app.js

```js
import { Element

export class MyA
  static get prop
    return {
      date: {
        type: St
        value: (
      },
    }
  }

  constructor()
    super()

    setInterval(
      this.date
    }, 2000)
  }

```

Now we remove all the HTML stuff in here.

This means we get rid of the dom-module and script tags that surround our Javascript class, and the markup in our template.

I recommend you copy the contents of the template into your clipboard because you'll need it in a moment.



src / my-app.js

Add a template getter:

Old:

```
<template>
   <style> div { color: black; } </style>
   <h1> 00 - Polymer 2 </h1>
   <div> The time is [[date]] </div>
</template>
```

New:

```
static get template() {
  return `
    <style> div { color: yellow; } </style>
    <h1> 01 - Polymer 3 </h1>
    <div> The time is [[date]] </div>
  `
}
```

The next thing we need to do is add a template getter method to our class.

The template getter is a new method in Polymer 3 that simply allows you to define a Javascript string which returns your template.

We can literally copy and paste the content from between the template tags in our Polymer 2 component and drop it into our Polymer 3 component.

I've actually changed the heading text and div colour here instead of doing a copy and paste. This was just a visual aide for me as I was developing so I know I was looking at the new version and not something in my cache.

Also I know what you're thinking, because I was thinking it too.

HTML in JS template literals is yucky.

Quick aside:

github.com/PolymerLabs/lit-html

youtube.com/watch?v=ruql541T7gc

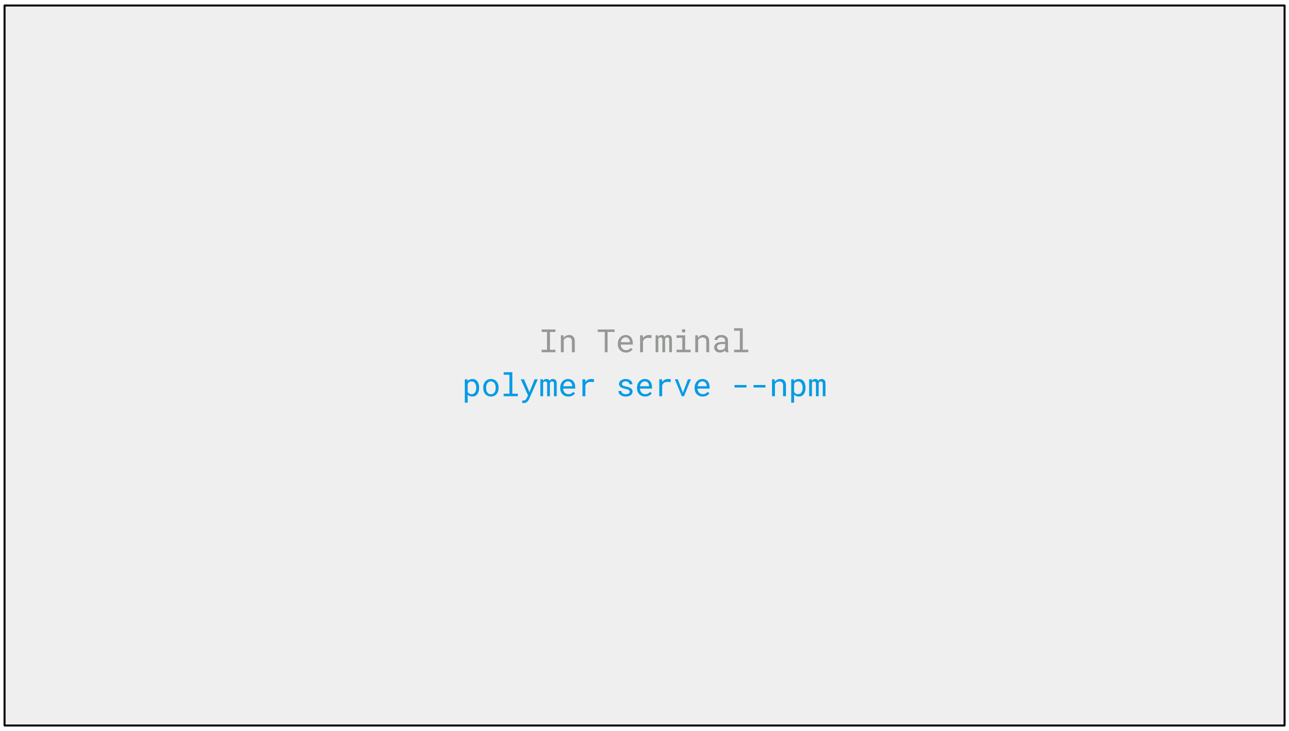
The Polymer team have mentioned that they're exploring some options for this.

I think it's safe to assume it won't stay like this, there's likely to be a third-party library involved.

Not JSX.

One of the lead devs, Justin Fangiani, has been working something called lit-html which takes template literals and efficiently re-renders them to the DOM. If I had to place a bet I'd say that's where Polymer is heading.

He did a great talk at the Summit. The link is on screen.

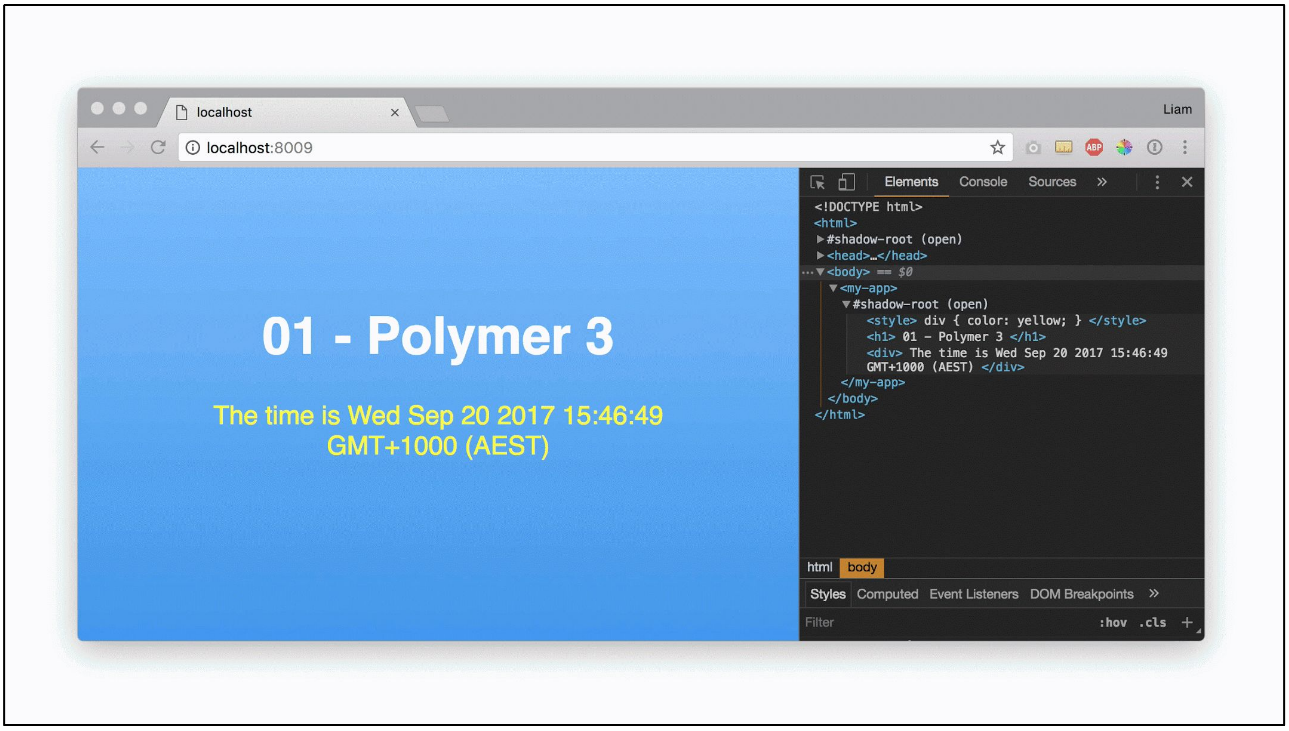


In Terminal

```
polymer serve --npm
```

Then we serve the project using the Polymer CLI again, except this time we add the --npm flag.

And we open our web browser to find….



```
<!DOCTYPE html>
<html>
 ▶#shadow-root (open)
 ▶<head>…</head>
…▼<body> == $0
   ▼<my-app>
     ▼#shadow-root (open)
        <style> div { color: yellow; } </style>
        <h1> 01 - Polymer 3 </h1>
        <div> The time is Wed Sep 20 2017 15:46:49
        GMT+1000 (AEST) </div>
    </my-app>
  </body>
</html>
```

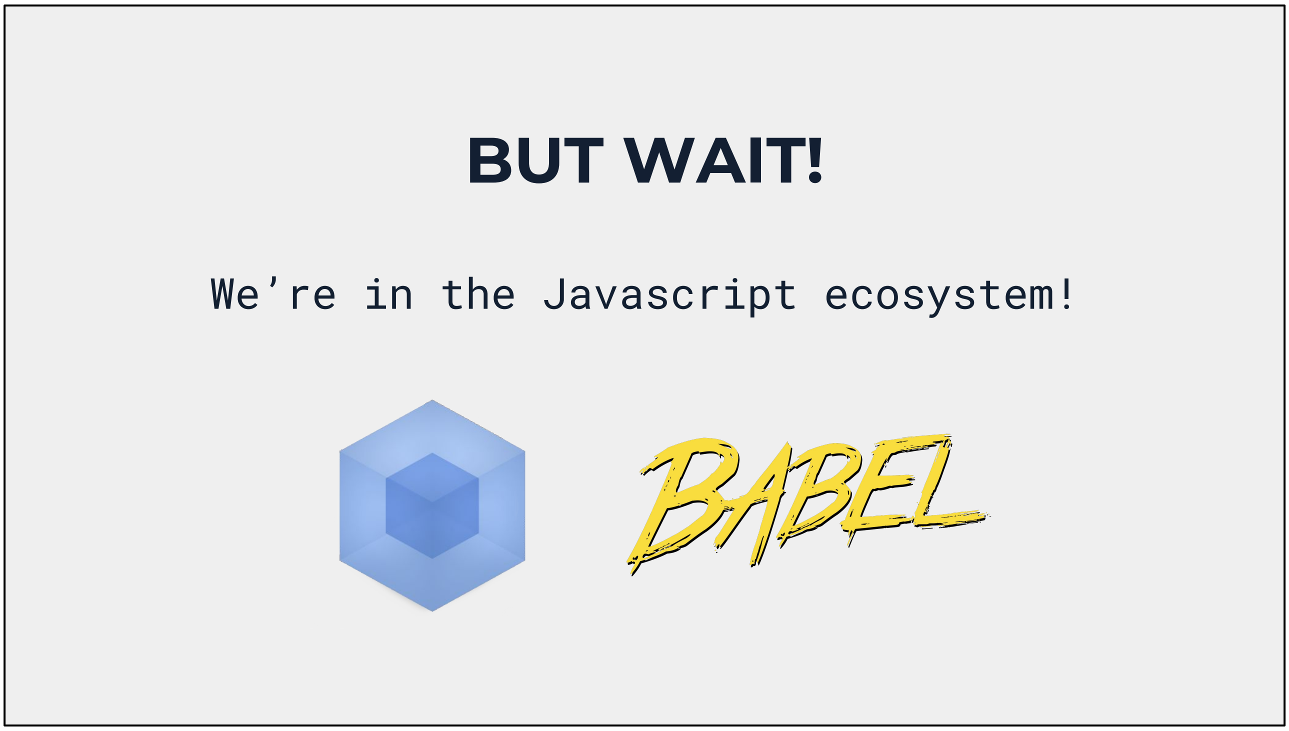
Excellent! It works!

Sooooo what's the benefit here?

Well until the Polymer CLI and appropriate tooling are updated, it's a little difficult to recommend this.

The browser support is broad, but not very deep (only the latest versions of each evergreen browser) and don't even consider running this in Internet Explorer.

The Polymer team said they'll be tackling tooling over the next few months so maybe we'll see what they had in mind soon.



BUT WAIT!

We're in the Javascript ecosystem!

BABEL

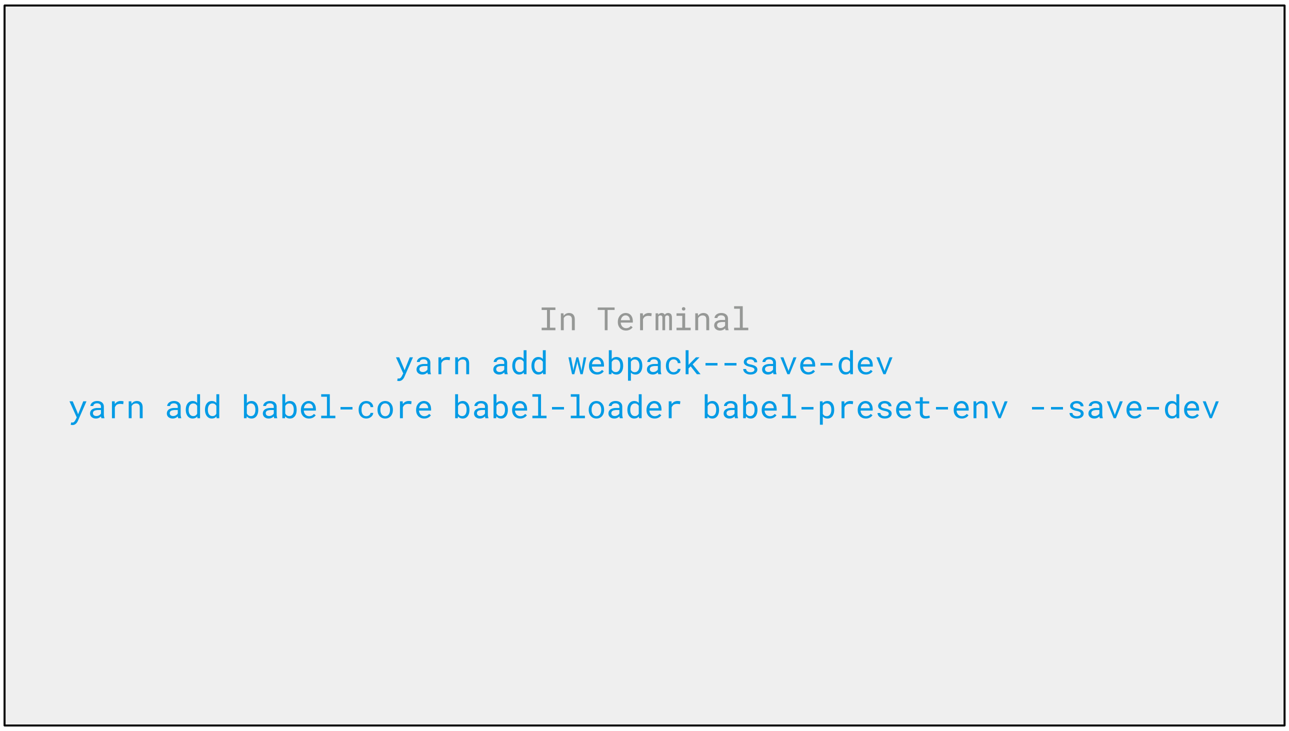
BUT WAIT!

We're writing standard JS now.

There are countless tools out there that can help us.

Maybe Webpack & Babel?

I've used them in a few other projects now, and though I'm not a huge fan of adding a build step to web components I was curious to see what's involved in making it work and how deep we could get support.

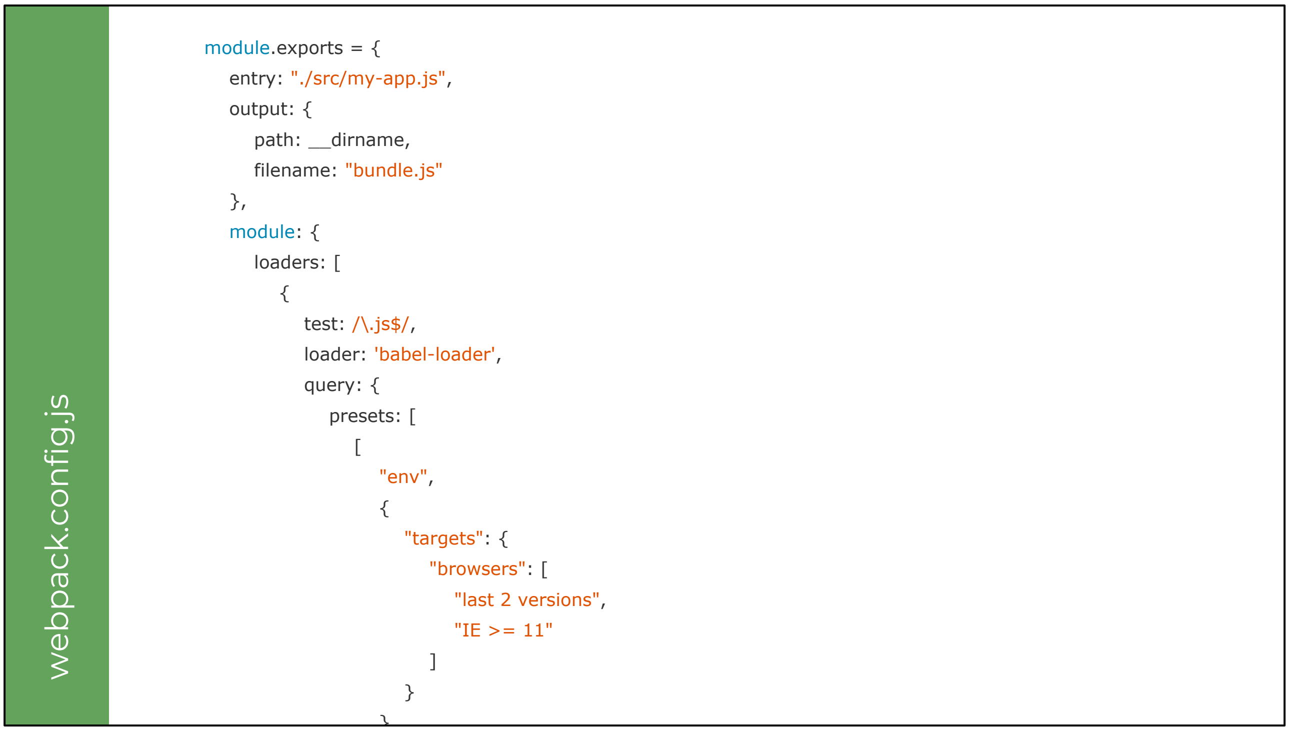


In Terminal

```
yarn add webpack--save-dev
yarn add babel-core babel-loader babel-preset-env --save-dev
```

Let's hurry back to the terminal and install these packages.

Once again we use Yarn to add dependencies to the project. In this case we're running the command with the --save-dev flag so they get saved as dev dependencies.


webpack.config.js

```
module.exports = {
    entry: "./src/my-app.js",
    output: {
        path: __dirname,
        filename: "bundle.js"
    },
    module: {
        loaders: [
            {
                test: /\.js$/,
                loader: 'babel-loader',
                query: {
                    presets: [
                        [
                            "env",
                            {
                                "targets": {
                                    "browsers": [
                                        "last 2 versions",
                                        "IE >= 11"
                                    ]
                                }
```

We created a Webpack config file that processes our component into a bundle.js file.

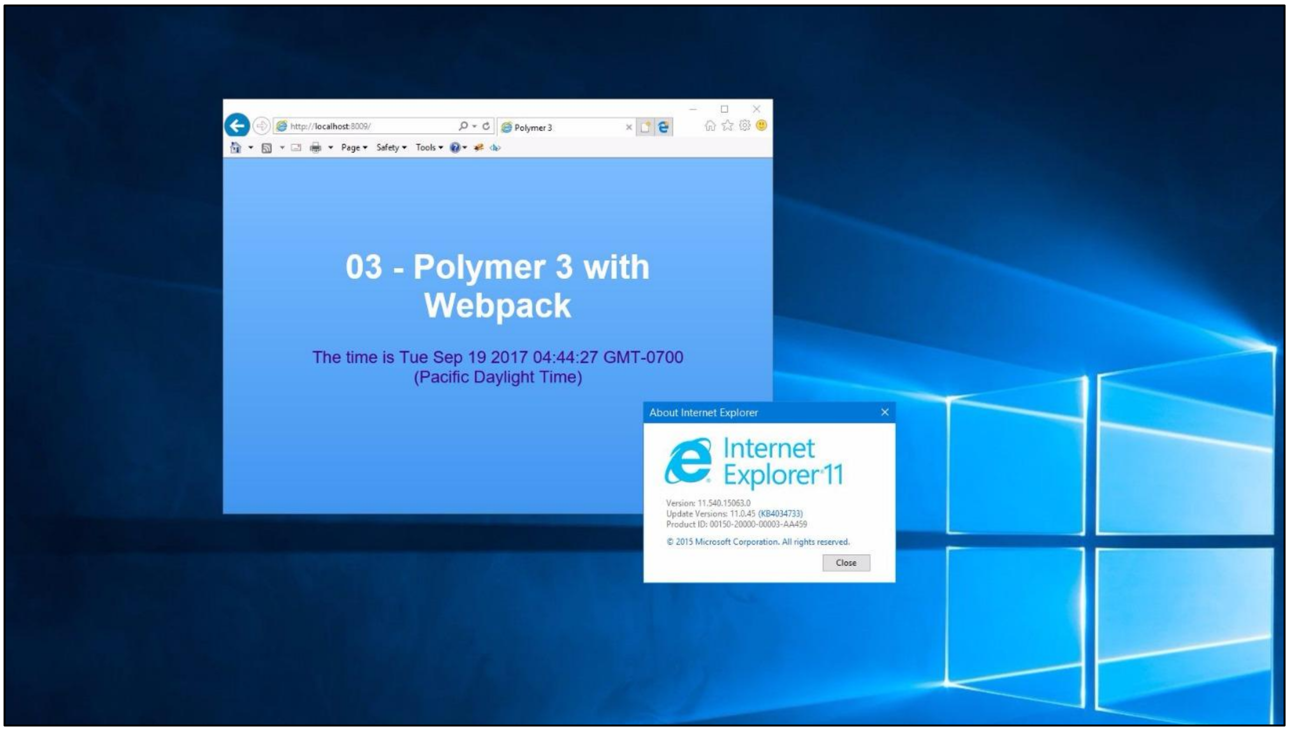
Added babel-loader which will process our .js files and targets support for IE11 or newer.

Then updated the path to our script in index.html to reference bundle.js, ran webpack in the terminal, and opened IE11...

Turns out many older browsers are a little picky about the order in which scripts run.

Fortunately I'm not the first person to encounter this and I was able to borrow a few lines of code from the Polymer Webpack Loader library that defer loading of bundle.js until the webcomponents polyfill has fired it's ready event and an ES5 shim has run.

Our index.html is looking a little more bloated, but I ran Webpack again, crossed my fingers, and refreshed IE11.



03 - Polymer 3 with Webpack

The time is Tue Sep 19 2017 04:44:27 GMT-0700 (Pacific Daylight Time)

About Internet Explorer

Internet Explorer 11

Version: 11.540.15063.0
Update Versions: 11.0.45 (KB4034733)
Product ID: 00150-20000-00003-AA459

© 2015 Microsoft Corporation. All rights reserved.

IT WORKED!

Riding that high I decided to tackle the next big thing that was bothering me. HTML in Javascript.

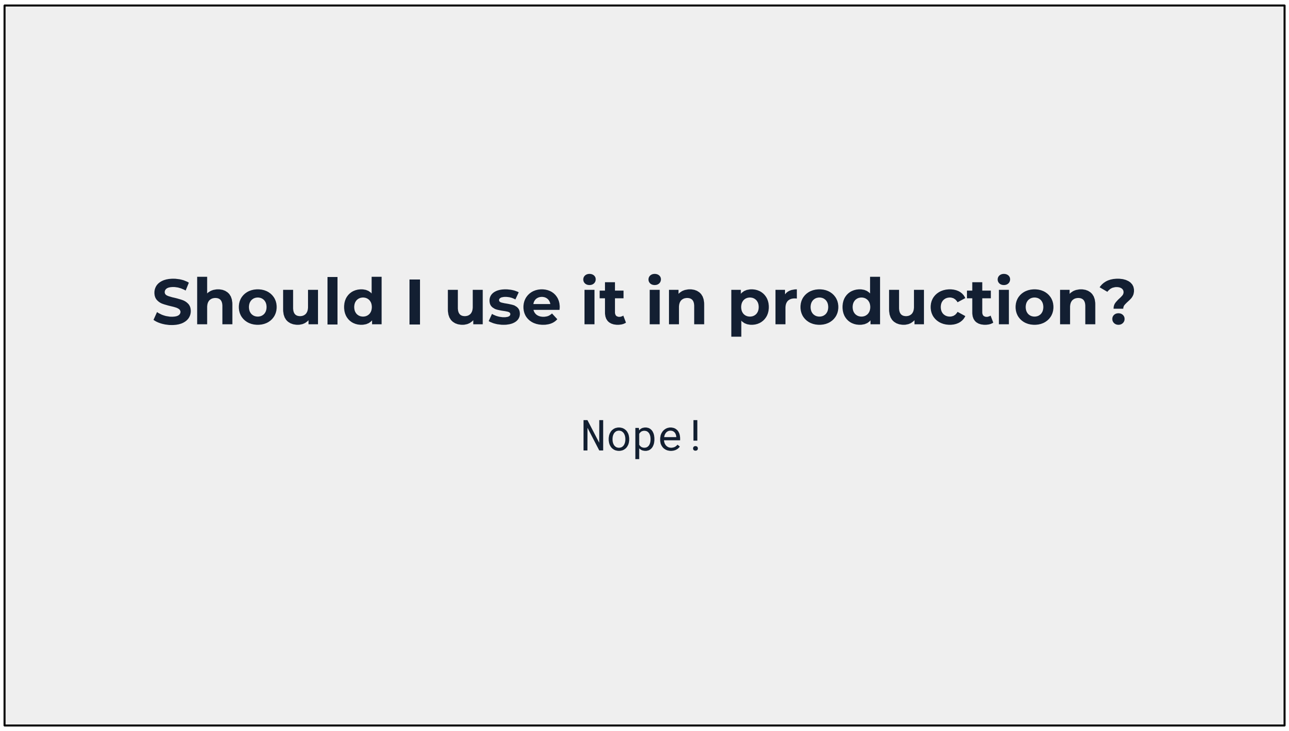
I split the HTML and CSS out into their own files and imported them back into the component file. Then added a Webpack loader to handle this process.

It was… Fine. I guess.

I tend to flip flop between wanting my styles and markup right there in the component, and wanting them separated.

I did like that I could author my CSS and HTML in editors and receive full syntax highlighting and code completion support.

But I also feel like those things are only an editor plugin out of reach.



Should I use it in production?

Nope!

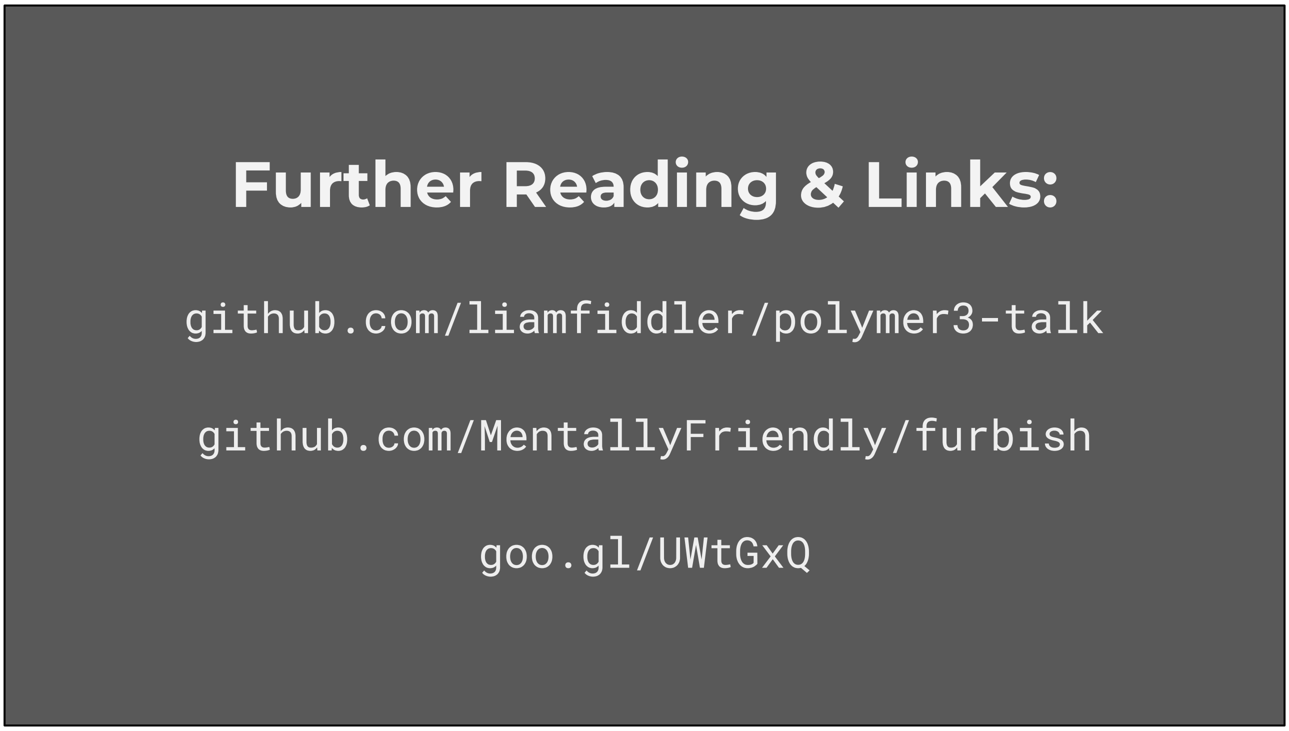
Don’t do it. It’s not ready yet.

Maybe you can make some prototypes using it. Or some projects you’re working on just for fun.

But the best approach is probably going to be to just keep using Polymer 2, then when Polymer 3 is officially released you can use the automatic conversion tool the Polymer team is working on to migrate.

Tooling just isn’t there yet and they’re still working out what to do about markup and styles in Javascript.

I am really excited about the possibilities here. And think this is a good move for the platform.



Further Reading & Links:

github.com/liamfiddler/polymer3-talk

github.com/MentallyFriendly/furbish

goo.gl/UWtGxQ

I’ve put all my source code into a repo on github. It contains everything you’ve seen here tonight, plus an experiment using lit-html.

The Furbish script here will help you start a new project from scratch using Polymer 3 if you want to try your own hand at some of this stuff.

Finally I’d really recommend you take a look at the YouTube channel for the Polymer Summit 2017, it’s the link at the bottom of the slide.

Questions?

Thanks for your time!