Chris Buckett

Web Components IN ACTION

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Web Components in Action
Version ς

For more information on this and other Manning titles go to
www.manning.com

Welcome

Thank you for purchasing the MEAP for *Web Components in Action*. I'm excited to see the book reach this stage, and look forward to its continued development and eventual release. This is an intermediate level book designed for anyone who's writing client-side web applications and wants to get ahead with the latest in browser technology.

Web Components represents a significant evolution in the web developer's toolbox, providing native browser support for creating your own HMTL tags, right up to application-level complexity. I've tried to show this in the book, by taking you through a simple, but realistic example application, using the various tools that Web Components provide.

With Google's Polymer framework (currently in alpha, and very much still undergoing change), the ability to write web applications that work across multiple browsers is now very real. Thanks to Google's polyfill libraries, which fill in the technology blanks where native implementation doesn't exist, you can start building applications using Web Components today.

We're releasing the first three chapters to start. The first chapter provides a good overview of the various parts of Web Components, and introduces the example application we'll be playing with through the book.

In chapter 2, you'll get to grips with the idea of templates, something that exists in other popular HTML frameworks, but now exists as a first-class entity within the browser. You'll create reusable HTML templates and insert them into your page

Chapter 3 takes this one step further, but showing how you can use the <polymer-element> to mix templates and data to produce reusable HTML elements.

Looking ahead, the book will cover more advanced topics, like using and extending the data binding expressions library, event handling in web components, understanding the web component lifecycle, and understanding the impact the Shadow DOM has on your components. We expect to have updates to the book every few weeks, whether that is a new chapter, or an update to an existing chapter.

As you are reading, I hope you'll take advantage of the Author Online forum. I'll be reading your comments and responding, and your feedback is helpful in the development process.

Chris Buckett

brief contents

1

The World of Web Components

This chapter contains

- Defining the term Web Components
- Discovering Platform.js and Polymer
- Building your first custom element

New web technologies appear every week, most with the promise of making it easier or quicker to build complex web applications, so what makes Web Components special? In this chapter, you'll discover that some browser vendors are already using web component technology to build some of the standard HTML5 tags that you're familiar with. Now, with the help of some experimental additions appearing in browsers, and some JavaScript polyfills to fill in the gaps where native support doesn't exist, you too can use this technology to develop components, widgets, and even entire applications as custom HTML tags.

In this chapter, you'll learn about the how the various specifications come together under the umbrella name of "Web Components" to provide a rich syntax for developing web applications, and how the technologies they represent layer upon each other to provide a compelling way to build web applications. These specifications and technologies cover Templates for adding static HTML, Model Driven Views for data binding, Custom Elements that let you invent your own browser elements, HTML Imports which provide a mechanism to package and reuse your custom elements, and the Shadow DOM, which lets you hide the internal workings of your custom elements.

We'll look at how you'll cope with browser support issues using Platform.js, which brings support for the underlying Web Component technologies to all modern browsers. You'll see how Google's Polymer library and Mozilla's X-Tag libraries sit on top of Platform.js, adding syntactic sugar to the standard Web Components spec, helping you to develop your application in a modular fashion. Finally, you'll start building your own custom element, based upon

Polymer's supercharged element framework, paving the way for a larger example that we'll use throughout the book.

1.1 What is a web component?

The web component technologies let you create your own HTML elements. They are a suite of complementary technologies for encapsulating HTML, script, and styles into reusable packages, with native support available in the browser. To find out what that actually means we need to take a high-level look at how the built-in HTML elements work.

1.1.1 HTML elements primer

Each HTML element, whether it's a `<button>`, `<ul>` or `<input>` element has a defined API and support built into the browser. When you use these tags, the browser takes data supplied from your markup, renders the element, and reacts accordingly to user and script actions. Some of these are quite simple containers, such as the `<div>` or `<span>` elements, but many of these are more complex. Some HTML5 elements, like the date picker shown in figure 1.1, is an example of one of the more complex elements. It is composed of many child components that show and hide in response to user input, according to code within the component itself.

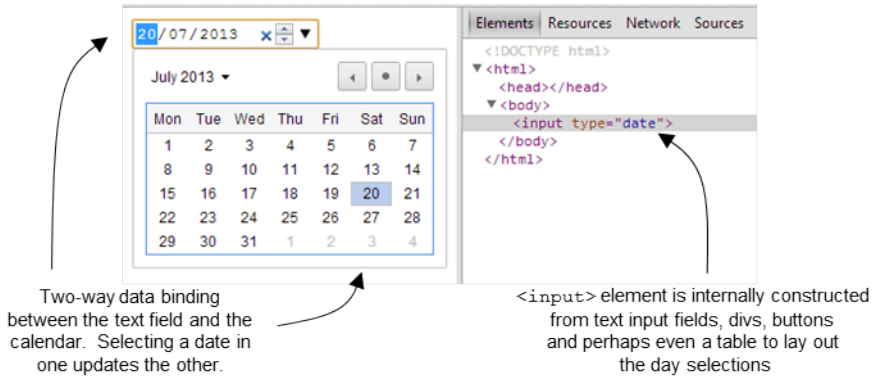

Figure 1.1 Inspecting the `input` element with the chrome developer tools.

The date picker has data bound elements; the date value entered into the text field, and the date value shown in the drop-down calendar are bound together internally. When you update the date in one place, the browser automatically updates it in the other, without any externally visible script. Additionally, there is styling applied to the date picker, such as the days of the week heading. No additional CSS scripts imported, and the styling does not leak out to the rest of the page. The element itself encapsulates the internal HTML, script, and styling.

> **TIP:** Many of the screen shots in this book, like that in figure 1.1, include the Chrome Developer Tools. The Inspect Element command (found under the right click menu) is

especially useful - and you should be familiar with navigating around inside the elements tab.

You use these modular pieces of HTML to construct your application, reading data in and out of the element, providing custom content, and reacting to events raised by the element in the form of event handlers. Figure 1.2 shows these ways you can interact with an element.

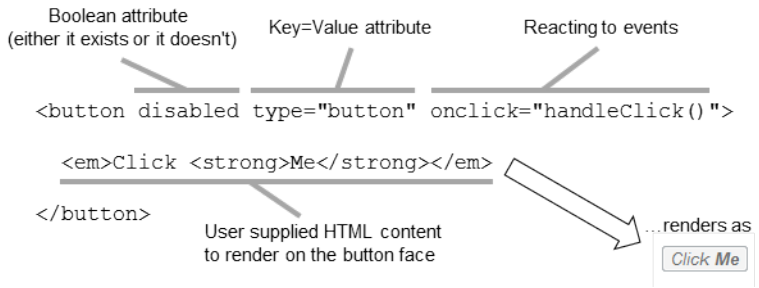

Figure 1.2 Interacting with a button element through its formal API.

The attributes and event handlers represent the element's formal API - you don't need to worry what's going on inside the element, as long as it works as expected.

1.1.2 Using other people's web components

So how does the workings of regular HTML elements impact web components? This brings us back to the original definition: web component technologies let you create your own HTML elements, with encapsulated HTML, script, and styles that you can use to build web applications and expose web APIs. There are web frameworks with similar functionality, that within their own ecosystem do this already, but the difference here is that the elements look and feel like real HTML elements, and because the technology is backed by W3C specification, browser manufacturers are building in native support for web components, thereby giving a performance boost.

> **NOTE**: Frameworks such as handlebars.js and mustache.js provide templating solutions, allowing you to bind data to reusable templates.

As a web developer, you will inevitably become a consumer of other people's web components, but also as a web developer, you will be an author of your own web components. Let's see how.

Think about the last web page you looked at - it probably had some social media buttons - Facebook Like, Twitter Follow, and so on. It might even have had a Google Map embedded on it, and it probably has some analytics. The developer of that web page has embedded third party HTML, Script and CSS into the site. To add a Google Map, you need to add an `iframe`

and link in some script. To add a Twitter Follow button, you add a `div` and some custom script, and to add a Facebook "Like" button, your page needs two `divs`, and even more script. You can see a sample of this in Listing 1.1.

Listing 1.1 Sample view of importing third party HTML

```
<!DOCTYPE html>
<html>
<body>
  <iframe width="425" height="350" frameborder="0" scrolling="no" #A
        marginheight="0" marginwidth="0"                          #A
        src="https://www.google.co.uk/maps?ie=UTF8&z=10           #A
[CA]         &ll=51.511214,-0.119824&output=embed"></iframe>      #A

  <a href="https://twitter.com/chrisdoesdev"                      #B
[CA]      class="twitter-follow-button"                           #B
[CA]      data-show-count="false"                                 #B
[CA]      data-lang="en">Follow @chrisdoesdev</a>                 #B
  <script>                                                        #B
[CA] !function(d,s,id){var js,fjs=d.getElementsByTagName(s)[0];   #B
[CA]    if(!d.getElementById(id)){js=d.createElement(s);          #B
[CA]    js.id=id;js.src="//platform.twitter.com/widgets.js";      #B
[CA]    fjs.parentNode.insertBefore(js,fjs);}}                    #B
[CA]    (document,"script","twitter-wjs");                        #B
  </script>                                                       #B

  <div id="fb-root"></div>                                        #C
  <script>(function(d, s, id) {                                   #C
    var js, fjs = d.getElementsByTagName(s)[0];                   #C
    if (d.getElementById(id)) return;                             #C
    js = d.createElement(s); js.id = id;                          #C
    js.src = "//connect.facebook.net/en_GB/all.js#xfbml=1";       #C
    fjs.parentNode.insertBefore(js, fjs);                         #C
  }(document, 'script', 'facebook-jssdk'));</script>              #C
  <div class="fb-like" data-href="http://example.com"><div>       #C
</body>
</html>
```

#A Google Map's iframe
#B Twitter's Follow link and script
#C Facebook's divs and script

In each case, you've embedded some custom functionality into your page. Search engines and other machines can't infer meaning from the scripts, divs and iframes, and it becomes harder for the third parties to modify their implementations.

When you use web component custom elements, however, embedding third party APIs in your page becomes both simple and standardized. Imagine using `<google-map>`, `<fb-like>` and `<twitter-follow>` elements directly into your page. All the script and style information that would usually end up in your HTML are wrapped-up inside the tag, leaving your page HTML clean and uncluttered. It would look something like listing 1.2, where you'll see some HTML Imports and Custom Elements. We'll talk more about what that means in a few pages, but for now, you can see that as a user of those third party components, it becomes more straightforward to follow the same standard each time.

Listing 1.2 Sample view of third party web components

```
<!DOCTYPE html>
<html>
<head>
  <link rel="import" href="http://google.com/embed-map.html">          #A
  <link rel="import" href="http://twitter.com/follow.html">            #A
  <link rel="import" href="http://facebook.com/fb-like.html">          #A
</head>
<body>
  <google-map lat="51.5171" lng="-0.1062" zoom="10"></google-map>      #B
                                                                       #B
  <twitter-follow lang="en" show-count>                                #B
    @chrisbuckett                                                      #B
  </twitter-follow>                                                    #B
                                                                       #B
  <fb-like url="http://example.com"></fb-like>                         #B
</body>
</html>
```

#A HTML Imports
#B Custom Elements

Now you've seen what other people's web components might look like, let's expand on that to see how web components might look in your own web application.

1.1.3 Using your own web components

In order to demonstrate the power of web components, we're going to look at an example application in this section. We'll be returning to it throughout the book, looking at how you might build different parts of it with the various different technologies. All you need to know is that it's an application that lets you vote on beers (at a beer tasting event), and it looks a bit like the mockup shown in figure 1.3. This isn't a walkthrough guide to build the entire application, instead, we'll return to the example throughout the book as we explore new features of web components.

> **NOTE:** I was once paid (yes, paid), to attend a market research event where I had to taste different beers, and rate them. It was before the time of tablets, so the rating happened on paper. From what I remember, it was great fun!

Figure 1.3 "Rate your beers" app mockup.

THE APPLICATION AS A WEB COMPONENT

Web component technology lets you create arbitrarily complex HTML elements. These may be as straightforward as a Twitter Follow button, or as complex as an entire application. By combining the HTML, CSS and script that makes up an application into an html element, the final page looks something like listing 1.3, where our Beer Rating app is represented by a single tag: `<beer-app>`.

Listing 1.3 Example beer application

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import" href="beerapp.html">      #A
  </head>
  <body>
    <beer-app></beer-app>                        #B
  </body
</html>
```

#A Import the custom element...
#B ... and use it

In the same way that the built-in HTML date picker component has internal complexity, so does the `<beer-app>`. Just like the date picker, all of the HTML, CSS, and script that makes our app work is wrapped up inside the web component. Figure 1.4 shows how these pieces come together to produce a reusable, modular element that looks and feels like it is a real, built-in HTML element.

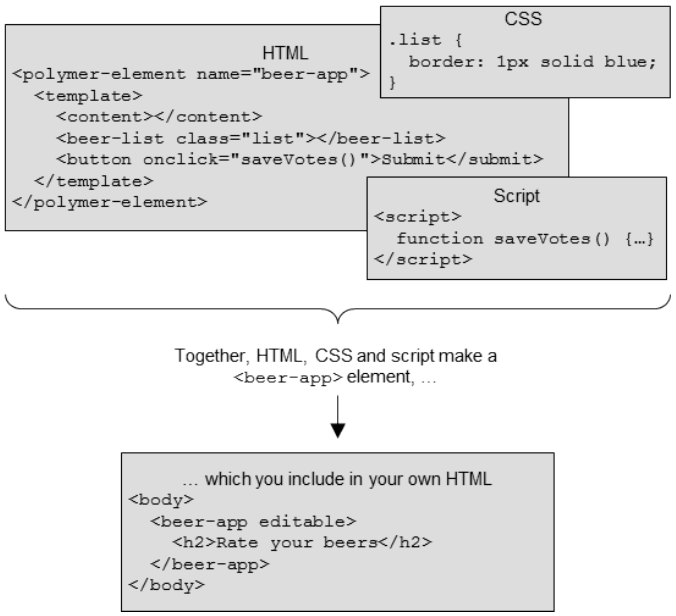

Figure 1.4 Web Components encapsulate HTML, script, and CSS into reusable modules.

If you're not thinking this is awesome, then you're probably thinking that you still have HTML, CSS and Script, and it's still complex, it's just that it's now in *another* set of files that you have to deal with, except that they're one page removed from the main HTML page. Here's the thing - designing an application in this way is different to how you've been doing it before. By designing an app with web components, you break your app down into the smallest reusable pieces of functionality, and bolt them together to make larger reusable pieces of functionality, until you have built your application. This lets your team create UI elements (such as 'like' buttons or login screens) and background, hidden elements (for example, a common user-preference element to persist data in local browser storage). These have discreet, modular functionality with published APIs. You can mix your own components with those from third parties (just like real HTML tags), and other teams can take your components and re-use them in their apps.

> **IMPORTANT**: All this boils down to the ability to create your own building blocks of HTML, customized for your particular use case, of arbitrary complexity, up-to and including an entire application represented as a single HTML tag.

SMALLEST REUSABLE FUNCTIONALITY AS A WEB COMPONENT

Our Beer Rating application needs to show a list of beer, each beer has an image and a name, so the lowest level of reusable complexity could well be a combination of an `<img>` and an `<span>` element (figure 1.5). By packaging the `img` and `span` elements together into single reusable tag, such as `<beer-listitem>`, we can use it in multiple locations in our app:

```
<beer-listitem>India Pale Ale</beer-listitem>
<beer-listitem>Porter</beer-listitem>
<beer-listitem>Wheat Beer</beer-listitem>
```

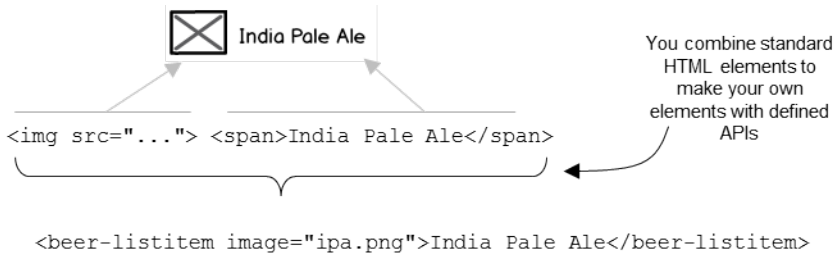

Figure 1.5 Conceptually combining `img` and `span` elements to make a `beer-listitem`

Perhaps then we create a `<beer-ratings>` element, which uses `<beer-listitem>` elements, and enriches them - kind of like an `<ol>` (ordered list) element does with a `<li>` (list item) element:

```
<beer-ratings totalstars=5>                          #A
  <beer-listitem>India Pale Ale</beer-listitem>      #B
  <beer-listitem>Porter</beer-listitem>              #B
  <beer-listitem>Wheat Beer</beer-listitem>          #B
```

```
</beer-ratings>
```

#A Ratings element
#B contains list item elements

Figure 1.6 shows how the web-component layers expand from the top-level HTML page to combine simple HTML elements into more complex application elements.

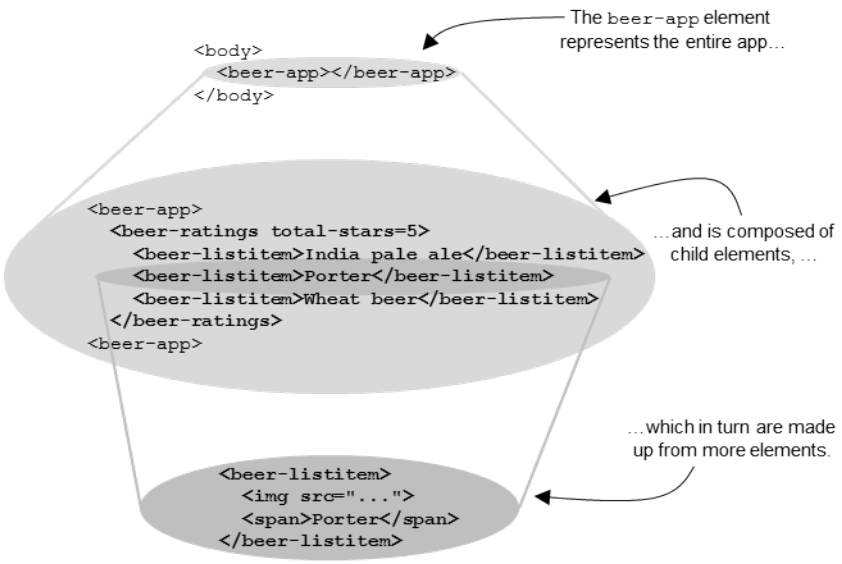

Figure 1.6 How an application is composed of web-component layers

Web component technology lets you create your own client-side components that work the same way as the browser's built-in components, ultimately supported natively in the browser - which leads us on nicely to talk about browser compatibility.

1.2 Browser support for web components

Developers on the Chrome and Mozilla teams are the primary creators of web component technology, so there's no surprise that support is greatest in these browsers. Opera, which also uses Chrome's Blink engine from the Chromium project, is also well supported. At the time of writing, the future is less clear for other browsers - understandable as the various W3C specifications that form web component technologies are still in draft state.

All is not lost, however. At Google I/O 2013, a team of engineers working at Google, but developing in the community, showed off a new set of tools they called Polymer. Polymer is a layered approach providing cross-browser support for web components by using polyfills. Polyfills are scripts that detect whether a browser supports a feature natively; if a browser does, the native implementation is used, otherwise, a JavaScript implementation fills in the gaps.

NOTE: In addition to JavaScript, Polymer also has a Dart implementation, allowing you to write web components with Google's new structured web language as well as JavaScript.

The Polymer developers called these polyfills `platform.js`, and it's designed to work in the latest *evergreen* browsers. In practice though, this means current versions of Chrome, Firefox, Opera, Safari 6+, IE10+, Mobile Safari, Chrome Android. The rationale for providing support for only the latest browser, is that as a group of developers they are primarily interested in driving the web forwards by "simulating the future."

NOTE: The term evergreen browser was coined in 2012 at the Web Performance SF meetup, and is a browser that solves the problem where people don't check for and download browser updates automatically. Evergreen browsers automatically update themselves. Commonly, these are the various mobile browsers (Opera, Safari, Chrome, Firefox), and their desktop equivalents. There is still some debate as to whether Internet Explorer (v10 onwards) is an Evergreen browser because although it updates through Windows Update, unlike the other browsers, this doesn't happen transparently on browser startus.

1.2.1 Looking at the Polymer layers

While platform.js provides polyfills for any missing native support within the browser itself, on top of this is the framework layer. Both Polymer (from Google developers), and X-Tag (from Mozilla developers) have built element frameworks on top of platform.js, and you use the element frameworks to build your own rich elements and interact with web component technologies, in the same way you use jQuery to interact with the DOM rather than deal with the DOM directly. This layering is shown in the diagram in figure 1.7.

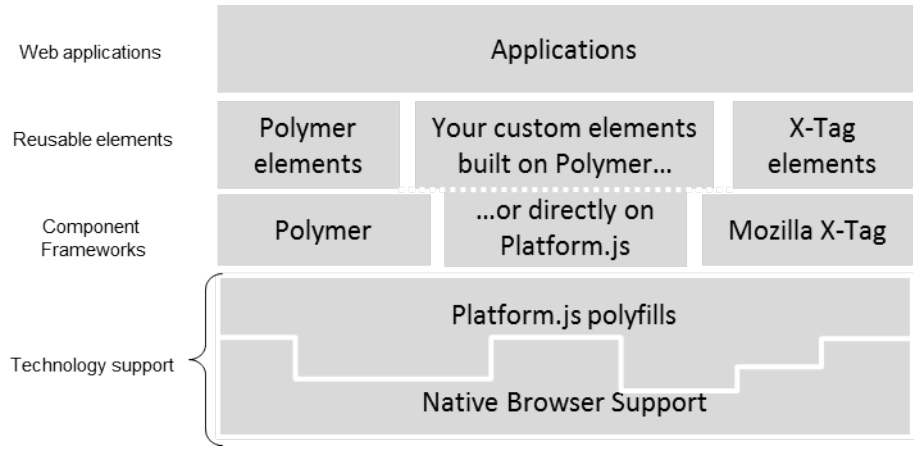

Figure 1.7 How the web component layers fit together.

In this book, we'll be including platform.js where needed to fill in the missing native browser support, and polymer.js when we're using specific features of the polymer library (which itself includes platform.js).

1.2.2 Importing the polymer and platform library layers

The Polymer project's homepage is available at www.polymer-project.org, and also available at GitHub.com/Polymer (the popular source-code hosting tool). From the project's homepage you can download a zip file containing all the files you need. Unzip the file and provide a link to the platform.min.js or polymer.min.js files in your pages:

```
<script src="platform.min.js"></script>    #A
<script src="polymer.min.js"></script>     #B
```

#A Platform polyfills, or
#B Polymer framework

As we go through the examples, I'll be using Google Chrome, which represents the gold standard for web component support. Some of the early examples will require no additional help from platform.js, but as we progress, we'll need to start using platform.js to do the feature detection and provide support for missing implementation elements. As more browsers support the specifications, over time your use of platform.js internals will diminish, with platform forming a thin façade over the native browser.

We'll return to polymer.js itself later in the chapter, but first, let's look at the core technologies that make up term "web components."

1.3 Core technologies

Underpinning Web Components is a number of W3C draft specifications. You've briefly seen two of these in use already, namely Custom Elements and HTML imports. In this section, you'll see how to create custom element, and we'll look at three other technologies, Shadow DOM, Templates and Model Driven Views (MDV). Table 1.1 gives an overview of these areas.

Table 1.1 Key web component technologies

Technology	Purpose
Templates	Define inert pieces of markup that aren't used by the browser until you need them.
Model Driven Views	Declarative data binding between model objects defined in script and the custom element.
Custom Elements	Provide a mechanism to "invent" your own tags
HTML Imports	Encapsulate your custom elements as stand-alone packages that are imported into your HTML
Shadow DOM	A way of hiding implementation and preventing style and script leaking into (and out of) your custom elements.

NOTE: The term Web Components is a wide umbrella term, and there are a few other technologies that are also covered, namely decorators, pointer events, web animations, and mutation observers. These are in various states of maturity and support, even within Chrome and Firefox, and there is still debate as to what form they will take. For that reason, they've been left out of the book.

You'll use the technologies in table 1.1 throughout the book to build reusable client-side APIs and applications. The W3C Web Components introduction document provides a succinct analysis:

> "When used in combination, Web Components enable Web application authors to define widgets with a level of visual richness and interactivity not possible with CSS alone, and ease of composition and reuse not possible with script libraries today."
>
> W3C Web Components Introduction, July 2013

Over the next few pages we're going to take a whistle-stop tour of these technologies, to see how they build on each other. Don't worry if everything doesn't make sense yet; as you go through the book you'll learn how to harness the power of these technologies, but it's important that you get to see all the pieces early to help you visualize them working together to provide a modular, reusable stack development stack.

Let's go back to our beer rating application example to help get a handle on how the various parts work together. It contains a list of beers that you're going to rate, and for the next few pages, we're going to concentrate only on the list of beer names and images. Figure 1.8 highlights the parts we're interested in:

Figure 1.8 The following examples only refer to the beer name and image

1.3.1 Templates

The first piece in the stack is templates. Currently, when you want to insert some HTML tags dynamically, you build them up using JavaScript (or a JavaScript framework), which dynamically creates elements and inserts them into the page's DOM. The following is a contrived snippet of Angular that we might use to list our beer items. Don't worry if you haven't seen Angular before, the point I'm making here is that templates are already in use in the real world, they just need an arbitrary JavaScript library to process them.

```
<ul>
  <li ng-repeat="beer in beerList">                #A
    <img src="bottle.png" onload="imageLoaded()">  #B
    <span>{{beer.name}}</span>                     #B
  </li>
</ul>
```

#A For each item in the list
#B Elements representing a beer list item

Behind the scenes, Angular's JavaScript is analyzing the DOM for special content like `ng-repeat` and `{{beer.name}}`, and creating new elements and inserting them into the DOM dynamically, creating HTML output like the snippet shown below:

```
<ul>
  <li>                                             #A
    <img src="bottle.png" onload="imageLoaded()">  #A
    <span>India Pale Ale</span>                    #A
  </li>                                            #A
</ul>
```

#A Using the template

The problem with this, though, is that the browser understands and processes the elements as the page loads; before the JavaScript library gets to them. The browser loads the `bottle.png` image resource and runs the `imageLoaded()` script even if there are no actual beers in the list to display. In a typical Single Page Application, there are likely to be many templates of HTML that are displayed dynamically as the user navigates around the user interface. Instead of the browser parsing all the HTML, script and CSS as the application starts up, it would be better (for startup times), if only the HTML, script and CSS required for the first screen of the application is processed by the browser. Templates, in the web component context, help achieve this.

INERT HTML, CSS, AND SCRIPT

A template is a block of inert HTML, CSS, and script that the browser does not insert into the DOM, even though it forms part of your HTML page. Let's see this in action with a simple page containing an "Add Beer" button. The page also has a pre-defined `beerTemplate`, which contains an image, a span for the beer name, and some script :

```
<template id="beerItem">
  <img src="beer.png">
  <span>Beer</span>
  <script>alert("Beer!");</script>
</template>
```

When the Add Beer button is clicked, we'll clone the nodes from the template and insert them into the DOM, turning them from inert template elements to real elements within the DOM.

Figure 1.9 shows, using the Chrome Developer Tools, how using a template doesn't cause the `beer.png` image to load until you use the template by clicking the "Add Beer" button. This also runs an `alert("Beer!")` script that forms part of the template.

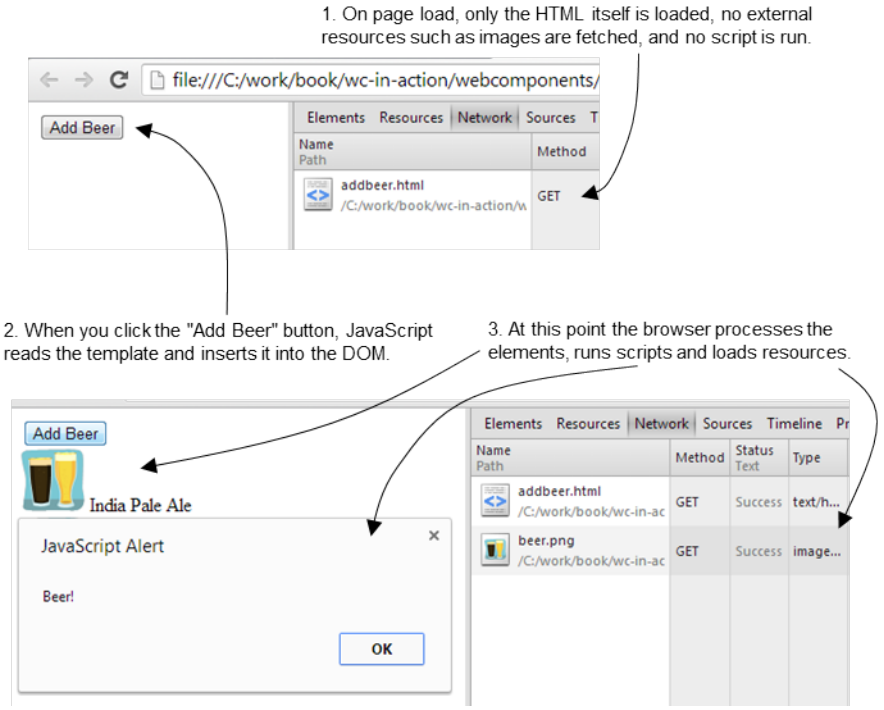

Figure 1.9 No resources are loaded until we use the template by clicking "Add Beer"

The browser doesn't execute any scripts within the template, and no images (or other external resources) are loaded until it you insert an instance of the template into the DOM, perhaps in response to some data being received or a user clicking a page element.

When you define your template, declaratively, using HTML, the browser needs to be told when to insert an instance of that template. You'll use script to clone the template's HTML content, set any properties, and then insert it into the page's DOM where the browser acts upon it.

While it is perfectly possible to achieve this with existing JavaScript frameworks, or by building up the elements with plain JavaScript and inserting them into the DOM, the difference

here is that support for templates is natively available in the browser, by using the new `<template>` tag, which lets you define your template markup declaratively in HTML.

CLONING THE TEMPLATE

Listing 1.4 shows the "Add Beer" `<template>` in a complete HTML page (note the absence of any JavaScript frameworks). The template is ignored when the page is loaded, but when you click the "Add Beer" button, the template's content is cloned and inserted into the DOM. The template is reusable; we clone the template each time you click the "Add Beer" button, we never modify the actual template itself.

Listing 1.4 addbeer.html: Using a template

```
<!DOCTYPE html>
<html>
<body>
  <template id="beerTemplate">                  #A
    <img src="beer.png">
    <span>Beer</span>
    <script>alert("Beer!");</script>
  </template>

  <button onClick="addBeer()">Add Beer</button> #B
  <script>
    function addBeer() {
      var template = document.querySelector("#beerTemplate");
      var element =  template.content.cloneNode(true);             #C
      element.querySelector("span").textContent = "India Pale Ale";
      document.body.appendChild(element);                          #D
    }
  </script>
</body>
</html>
```

#A The beer template
#B The Add Beer button
#C Cloning the template's content
#D Inserting the clone into the DOM

If you inspect the HTML elements created using the Chrome developer tools, shown in figure 1.10, below, you'll see, for each click of the "Add Beer" button, you'll get a copy of the elements that form the template. The template's nodes are contained within a `#document-fragment`, which is an efficient container for DOM nodes.

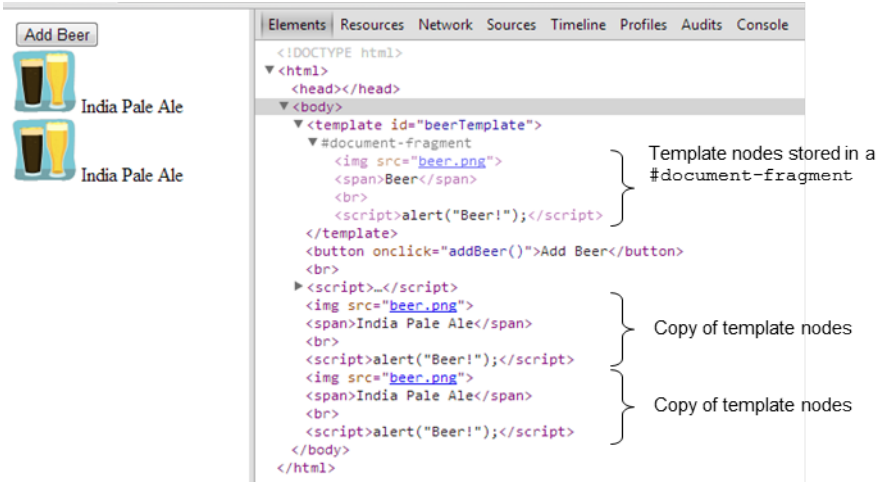

Figure 1.10 Inspecting the DOM after two template insertions.

> **Web components with existing frameworks**
>
> It's no accident that I picked the Angular framework as an example; the Angular framework is just one of many that have announced that they are going to support web components inside their frameworks. You'll still need frameworks like Angular, Ember and others, because their scope is wider than that provided by web components, but you will be able to build web components using your favorite framework, and make use of other people's web components in your favorite framework.

We'll look in more detail at templates in chapter 2. Next, you'll see how to bind a template to a data model, giving you two-way data binding, which shows how you can bind internal data to visible UI elements.

1.3.2 Model Driven Views

Model Driven Views give you a way to separate your internal data model (defined as an object in script), and your user interface markup. As the values in the data model are updated, that update is reflected back to the UI elements, and if the data is updated through user interaction (such as entering text in a textbox), the data is updated in the underlying object.

Figure 1.11 shows an example of our beer item, now with a text input box added. When you enter text into the input box, character by character, the read-only value in the span is updated.

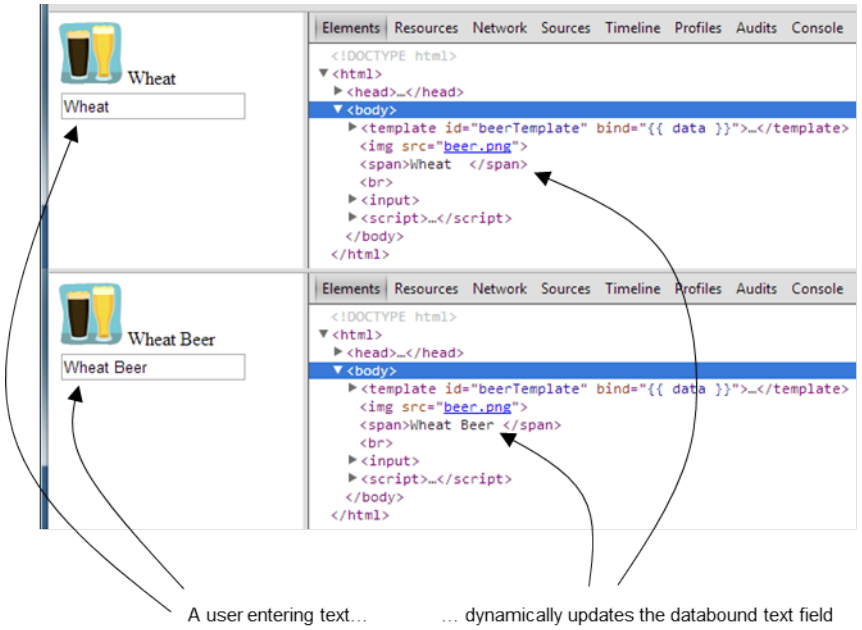

Figure 1.11 Text updates are synchronized between the two elements using data binding.

You achieve this by defining a bind value on the template, and attaching a JavaScript data model to the template. Data values are bound using a `{{ }}` syntax, which provides a binding between a name in the template markup, and the name of the object's field. Listing 1.5, below, shows us binding a `{{beerInfo}}` object to the template, with the `{{beerName}}` field inserted into the template's content, both as a label in the `<span>` element, and as the value for the text input field. Typing in the input field updates the underlying `beerInfo.beerName` field, which in turn updates the static text inside the `<span>` element. Our script gets a handle to the template once the DOMContentLoaded event is raised, and attaches a data model to the template, which creates an instance of the template.

Listing 1.5 Data binding with model driven views

```
<!DOCTYPE html>
<html>
<head>
  <script src="platform.min.js"></script>
</head>
<body>
  <template id="beerTemplate" bind="{{ beerInfo }}">        #A
    <img src="beer.png">
    <span>{{ beerName }} </span><br />                      #B
    <input value="{{ beerName }}"></input>                  #B
  </template>
```

```
    <script>
      document.addEventListener('DOMContentLoaded', function() {
        var template = document.querySelector("#beerTemplate");
        template.model = {                          #C
          beerInfo: { beerName: "Wheat" }           #C
        };

        //Platform.performMicrotaskCheckpoint();          #D
      });
    </script>
  </body>
  </html>
```

#A Binding to the beerInfo data model
#B Binding to the data model's beerName
#C Adding the beerInfo data model to the template
#D Platform polyfill - not required for chrome

Model driven views use a mechanism called `Object.observe` to determine if the data model changes. This exists in Chrome, but not in some other browsers, so the platform.js polyfill adds in support for it. To enable this, you need to ensure that if you modify the data model in your own code, as we have done here when we insert it into the model, you need to manually trigger the polyfills using `Platform.performMicrotaskCheckpoint();`.

Model Driven Views are a flexible technology, and this is just breaking the surface. In chapter 4, we'll look at using model driven views to drive data binding against lists and more complex objects, and see how you can modify the binding syntax `{{ beerName }}` with your own, to provide additional custom functionality.

In the next section, we'll take templates and use them with custom elements to create your own HTML tags.

1.3.3 Custom Elements

Custom elements are one of the key pieces of web components, and this section is going to be an overview that we'll cover in more depth later in the book. Custom elements let you define your own HTML elements; elements that you can invoke using HTML markup (declaratively), or through script (imperatively). The snippet below shows how you create a `beer-item` element - we'll look at how you actually define the beer-item element next.

```
  <beer-item>Porter</beer-item>    #A

  <script>
    var beer = new BeerItem();              #B
    beer.textContent = "Wheat Beer";        #B
    document.body.appendChild(beer);        #B
  </script>
```

#A Creating a BeerItem element declaratively as an HTML tag
#B Creating a BeerItem element imperatively with script

When run, this is going to use our `<beer-item>` custom element that we'll define to create HTML that renders a screen like that shown in figure 1.12.

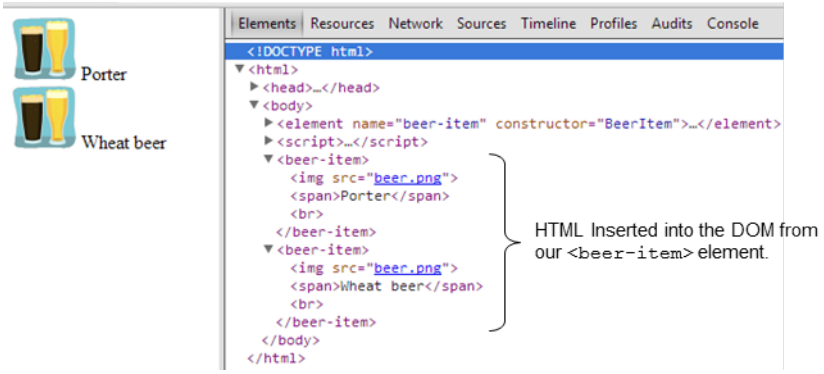

Figure 1.12 Creating <beer-item> elements from a custom element

Just like the two ways you can use a pre-made custom element (imperatively with script or declaratively with HTML), you also define a custom element either by script alone, or with a new HTML element called (confusingly) `HTMLElement`.

DEFINING ELEMENTS WITH THE HTMLELEMENT ELEMENT

In chapter 3, we'll look at how to create your own custom elements using script, but for now, we'll just take a peek at how you might define a `<beer-item>` using the new `HTMLElement`, shown in Listing 1.6. The key parts are firstly the `BeerItemPrototype`, which provides instructions for the browser to create the element when you clone the `<template>`, (you'll see this in listing 1.6), and secondly, inserting the elements into the `<beer-item>` using the code you saw earlier:

```
var template = document.querySelector("#beerTemplate");  #A
var beerItem =  template.content.cloneNode(true);        #A

this.appendChild(beerItem);   #B
```

#A Get the template and clone it
#B Insert the template elements into this <beer-item>

We also need to take the `<beer-item>`'s `textContent` property and use it to populate the from the template. Because you can create an element using script, and set the textContent after the item is created, we need to override the `<beer-item>`'s default textContent setter to forward the value to the span's textContent using code like this:

```
var contentSpan = beerItem.querySelector("span"); #A

Object.defineProperty(this, 'textContent', {
  set: function(inValue) {
    contentSpan.textContent = inValue;            #B
  }
});
```

#A Get the span
#B Set span's content when textContent is updated

Finally, once the prototype is defined, you pass the prototype into `document.register()`, which registers the custom element with the browser, returning a `BeerItem` constructor you can use in script. Listing 1.6 shows the complete custom element.

Listing 1.6 beer-item.html: Defining an element

```
<template id="beerTemplate">                    #A
  <img src="beer.png">                          #A
  <span><!-- content goes here --></span><br /> #A
</template>                                     #A

<script>
  var BeerItemPrototype = Object.create(HTMLElement.prototype); #B
  BeerItemPrototype.createdCallback = function() {           #B

    var template = document.querySelector("#beerTemplate");  #C
    var beerItem =  template.content.cloneNode(true);        #C

    var contentSpan = beerItem.querySelector("span");   #D
    contentSpan.textContent = this.textContent;         #D
    this.textContent = '';                              #D

    this.appendChild(beerItem);                         #E

    Object.defineProperty(this, 'textContent', {    #F
      set: function(inValue) {                      #F
        contentSpan.textContent = inValue;          #F
      }                                             #F
    });                                             #F
  };

  BeerItem = document.register("beer-item", {       #G
    prototype: BeerItemPrototype                    #G
  });                                               #G
</script>
```

#A Template
#B Prototype Element
#C Construct the element when created
#D Set the content of the span
#E Clone the template
#F Allow setting text content via script
#G Register the prototype with the browser

When you call `document.register()`, it registers the string `"beer-item"` with the browser. This maps the `BeerItemPrototype` to the html `<beer-item>` tag.

> **IMPORTANT:** Custom Elements names must contain a hyphen. This distinguishes them from the built-in HTML elements, and lets element creators provide a prefix so that common elements share the same prefix. When the browser encounters a tag with a hyphen, it treats it as an HTMLElement, rather than an HTMLUnknownElement, as it would do for unknown tags without a hyphen.

To actually create an instance of the beer-item element that appears on the page, you only need to include a `<beer-item>` tag, like this:

```
<beer-item>Porter</beer-item>
```

When the browser sees this markup, it creates an `HTMLElement` in the DOM. Once the `beer-item` element is registered with the browser, the `<beer-item>` is "upgraded" to a custom element, and the browser calls the `BeerElementPrototype.createdCallback()` function, which builds up the element in the DOM. The `createdCallback` function clones the template and sets the `textContent` value of "Porter" from the `<beer-item>` tag, inserting it as the span's `textContent`.

The script version for creating elements uses the `BeerItem` constructor returned from `document.register()`, which is defined in global scope. This lets you write code to create your custom element in script:

```
var beerElement = new BeerItem();
beerElement.textContent = "Wheat Beer";
document.body.appendChild(beerElement);
```

For this to work, you need to include the platform.js script. Listing 1.7 puts all this together, incorporating the code from listing 1.6, creating two beer item elements - one using HTML markup (the <beer-item> tag), and one using the `BeerItem` element in JavaScript.

Listing 1.7 beer-app.html - using the element

```
<!DOCTYPE html>
<html>
<head>
  <script src="platform.min.js"></script>              #A
</head>
<body>

  <template>                                             #B
   ... snip from listing 1.6 ...                         #B
  </template>                                            #B
  <script>
   ... snip from listing 1.6 ...                         #B
    BeerItem = document.register("beer-item", {          #B
      prototype: BeerItemPrototype                       #B
    });                                                  #B
  </script>

  <beer-item>Porter</beer-item>                    #C



  <script>                                         #D
    var beerEl = new BeerItem();                   #D
    beerEl.textContent = "Wheat Beer";          #D
    document.body.appendChild(beerEl);             #D
  </script>                                        #D
</body></html>
```

#A Including platform polyfill
#B Defining the beer-item element (from Listing 1.6)
#C HTMLElement is upgraded to a <beer-item>

#D Creating a BeerItem using script

That was a whistle stop tour of custom elements. We'll look at them again in more depth in chapter 3, but let's move on to another subject and see how the next technology, HTML Imports, complements custom elements, by making them reusable across multiple pages.

1.3.4 HTML Imports

In the previous section, our HTML contained both the custom element definition, and the custom element in use. That's fine for an example, but it doesn't promote great reuse. That's where HTML Imports come in. HTML imports let you take your custom elements, and any external CSS and script, and import them into another HTML page. That way, we can extract our custom element into a separate file: `beer-item-element.html`, and re-import it back into our host HTML page using an HTML `<link>` tag, with a property `rel="import"`:

```
<link rel="import" href="beer-item-element.html">
```

Figure 1.13 shows the new relationship between these files.

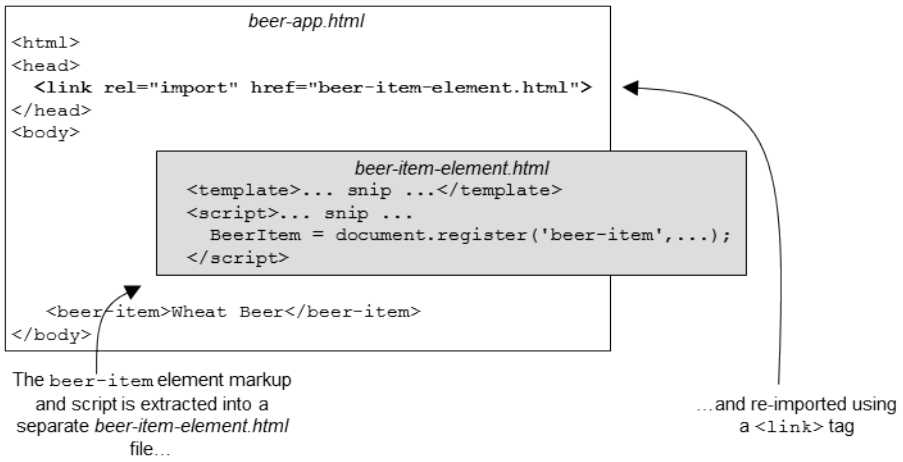

Figure 1.13 Extracting the `beer-item` element into an external file and `import`ing it

> **NOTE:** There is another hoop we need to jump through when using external imports. Because the `<template>` is no longer part of the `document` context. Instead it's part of the `link` element's context, so we need to query the DOM for the link resource in order to access the `<template>`. Listing 1.8b shows this, but you'll see, in chapter 3, how Polymer Elements (as opposed to basic custom elements) remove the need for this.

HTML imports do the same for HTML as the <script src=""> tag does for JavaScript; they allow you to reuse and import HTML elements across multiple pages, the the same way links to external CSS files work.

When you use HTML Imports to create elements with script, you need to wait for the `HTMLImportsLoaded` event to be fired, for example, a snippet from listing 1.7 would look like:

```
<script>
  window.addEventListener('HTMLImportsLoaded', function() {  #A
    var beer = new BeerItem();
    beer.textContent = "Wheat Beer";
    document.body.appendChild(beer);
  });
</script>
```

#A Waiting for HTML Imports to be loaded.

Serving HTML

For a number of the polyfills to work, including HTML imports, you need to serve the files from a web server rather than reading them from directly from the filesystem.

A quick and dirty way to do this is to install python (available for Windows, Linux and Mac). Python has a built-in web server that you can use to serve local files. Once python is installed, simply go to the command line, change to the folder you want to serve, and enter the command:

```
python -m SimpleHTTPServer
```

This will start a web server to serve files from the local folder that you can access by navigating to http://localhost:8000/

The new version of our code is now spread across two files, shown in listing 1.8a and 1.8b. Keep an eye out for the <link> element id, in the main html file, and where we use that to reference the <template> in the script by accessing the link as a resource.

Listing 1.8a

```
<!DOCTYPE html>
<html>
  <head>
    <script src="../../polymer-latest/platform.min.js"></script>
    <link id="beer-item-import"                                    #A
          rel="import" src="beer-item-element.html">               #A
  </head>
  <body>
    <beer-item>Porter</beer-item>                                  #B

    <script>
    window.addEventListener('HTMLImportsLoaded', function() {  #C
      var beer = new BeerItem();                                   #C
      beer.textContent = "Wheat Beer";                             #C
      document.body.appendChild(beer);                             #C
    });                                                            #C
    </script>
```

```
  </body>
</html>
```

#A Linking to the import file
#B Creating an instance declaratively
#C Creating an instance imperatively

Now that the `beerTemplate` is extracted in Listing 1.8b, the script now references the main document's `<link id='beer-item-import'...>` to access the actual `beerTemplate`.

Listing 1.8b

```
<template id="beerTemplate">                                    #A
    <img src="beer.png">                                        #A
    <span><!-- content goes here --></span><br />               #A
</template>                                                     #A

<script>                                                                #B
  var BeerItemPrototype = Object.create(HTMLElement.prototype);
  BeerItemPrototype.createdCallback = function() {

  link = document.querySelector('beer-item-import');                    #C
  var template = link.__resource.querySelector("#beerTemplate");        #C
  var beerItem = template.content.cloneNode(true);

  var beerItemSpan = beerItem.querySelector("span");
  beerItemSpan.textContent = this.textContent;
  this.textContent = '';
  this.appendChild(beerItem);

  Object.defineProperty(this, 'textContent', {
      set: function(inValue) {
        beerItemSpan.textContent = inValue;
      }
    });
  };

  BeerItem = document.register('beer-item', {
    prototype: BeerItemPrototype#A
  });

</script>
```

#A The original template
#B The original script
#C Now accessing the template via a the main document's linked resource

We'l play with these linked resources later in the book, and you'll also see how the Polymer framework provides some of this functionality out of the box.

HTML imports represent a breather before another more complex area in the form of the Shadow DOM. The Shadow DOM solves another problem with our custom element - that of hiding the implementation, so let's take a look.

1.3.5 Shadow DOM

When we add our beer-item tag in HTML, in the form:

```
<beer-item>Wheat Beer</beer-item>
```

we don't worry about what's going on internally, and we shouldn't need to know. Once the element is inserted into the DOM, however (visible when we inspect the element using chrome dev tools), our `<beer-item>` element has had the template's HTML copied into it, creating markup that looks like:

```
<beer-item>
  <img src="beer.png">              #A
  <span>Wheat Beer</span>           #A
  <br />                            #A
</beer-item>
```

#A Copied from the template

The problem exposing the internals of our `<beer-item>` element is that our implementation leaks into the host page, where other script or CSS might modify or style it in ways we'd not intended. Just like the built-in date-picker has its internals hidden, we too want to hide the internal elements of our component, as shown in figure 1.14.

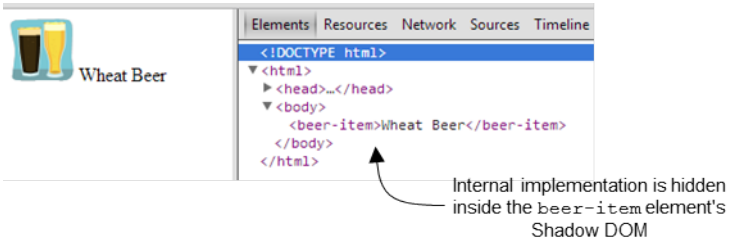

Figure 1.14 You use `beer-item`'s Shadow DOM to hide element's implementation.

SHADOW DOM VERSUS LIGHT DOM

This is where the Shadow DOM comes in. With Shadow DOM technology, each element has a child DOM that is visible, known as the *light DOM*, and a parallel, invisible DOM, known as the *shadow DOM*. By putting internal HTML elements of your custom element inside the Shadow DOM, you prevent users of your custom element accidentally modifying or styling them. This is great, because a developer embedding your custom element in their web page doesn't need to worry about namespace conflicts like duplicate element IDs from different parts of the page, or global CSS styles accidentally styling the imported element, which contains its own styles. This means that a Google +1 or Tweet This button can contain its own element IDs and styles without needing to be concerned with conflicts in the containing page.

By default, elements in the Shadow DOM aren't visible to the DOM, although the browser does process and render them. If you query the DOM for elements, any elements in the Shadow DOM won't appear. Figure 1.15 shows the relationship.

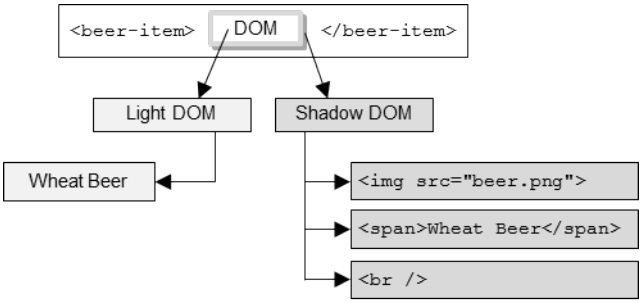

Figure 1.15 The beer-item has a public Light-DOM and an internal Shadow DOM

ADDING ELEMENTS TO THE SHADOW DOM

So how do we access the Shadow DOM? For our current example, it is quite straightforward. You use the `createShadowRoot()` function in the beer-item element's `createdCallback` function. Listing 1.8, below, shows the modified *beer-item-element.html*, which creates a "shadow root" on the `<beer-item>` element, and inserts the template elements directly into the element's shadow DOM instead of the element's public DOM.

Listing 1.8 beer-item-element.html: Using shadow root

```
...snip template...

<script>
  var BeerItemPrototype = Object.create(HTMLElement.prototype);
  BeerItemPrototype.createdCallback = function() {

    ...snip clone template...
    ...snip set textContent...

//      this.appendChild(beerItem);                    #A

      var shadow = this.createShadowRoot();       #B
      shadow.appendChild(beerItem);               #B

    };

    BeerItem = document.register('beer-item', {
      prototype: BeerItemPrototype
    });

  </script>
```

#A Don't attach the template clone to the <beer-item>
#B Instead, attach it to the <beer-item>'s shadow root

EXPLORING THE SHADOW DOM

It's useful to see that you can actually access the Shadow DOM's elements through JavaScript, by using the shadowRoot property, as shown in the following snippet that you can add into your host HTML page:

```
window.addEventListener('WebComponentsReady', function() {
  var item = document.querySelector("beer-item");
  console.log('light DOM: ', item.innerHTML);                    #A
  console.log('shadow DOM: ', item.shadowRoot.innerHTML);         #B
});
```

#A "Wheat Beer"
**#B "<img ...>...
"**

The chrome developer tools also provide some a useful setting to toggle viewing the Shadow DOM on or off, Figure 1.16 shows where you can find this setting, and what you see with the Shadow DOM visible.

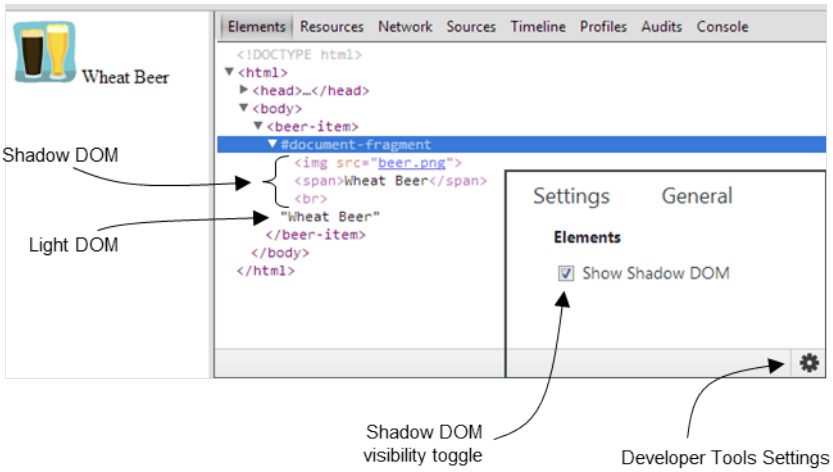

Figure 1.16 Viewing the Shadow DOM through the Chrome developer tools

In chapter 8, we'll look at some of the complexities that can occur with Shadow DOM, such as how browser events are propagated through the shadow, what happens if you have multiple, nested elements with shadow DOMs, and the Shadow DOM might affect performance.

In the next section, we'll look at Polymer elements. The Polymer framework takes the core technologies of templates, model driven views, custom elements and shadow DOM, and brings them together to eliminate much of the boilerplate and supercharge your custom elements.

1.4 Supercharged Polymer elements

Up to now, you've had a glimpse of the various technologies that make up the web components sphere. To get the benefit of them working together, you have to write some boilerplate, such as extracting template nodes and inserting them into the Shadow DOM, or adding an event listener to detect when the web component polyfills are ready, and then insert custom elements into the page. In this final short section in this chapter, you'll see how the Platform.js authors made an opinionated framework that helps you out in a number of ways.

The Polymer elements framework that builds on the platform.js polyfills brings all the technologies together, performing a number of boilerplate tasks and providing additional features, like letting you define custom elements declaratively using HTML Markup rather than JavaScript to clone the template, automating the browser registration process for custom elements, and automatically using the shadow DOM.

Figure 1.17, looks very much like our previous examples - the `beer-item`'s implementation details is hidden in the shadow DOM, and the beer names (Wheat Beer, India Pale Ale, Porter) are inserted into the correct child element.

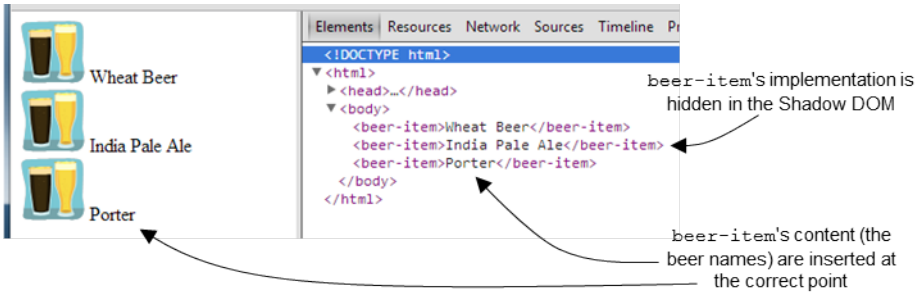

Figure 1.17 Using Polymer elements produces a similar end effect to the previous examples

If the output of using Polymer isn't so different to what you've seen previously, what does the framework provide? Earlier, I said that the Polymer library provides syntactic sugar to the web components technologies. That means (like jQuery does with JavaScript) you get a lot of things done for you, like element registration with the browser and model driven view data binding and Shadow DOM usage. It is an opinionated framework, which means that although there might be several ways to achieve the same goal, its designers have taken a stand about which way is best, and guides you to use that. Let's see how you create the `beer-item` using the Polymer framework.

In this case, instead of defining a custom element using script and a template, you define a `<polymer-element>`, which contains a template, shown in listing 1.9. This polymer element registers the `beer-item` element with the browser, inserts the template into the Shadow DOM, and puts the beer name (Wheat Beer, Porter, India Pale Ale) into the area defined by the `<content>` element. You'll see all this functionality comes declaratively, without needing to write any script.

Listing 1.9 beer-item-element.html - using polymer element

```
<polymer-element name="beer-item">                    #A
  <template>
    <img src="beer.png">
    <span><content></content></span>                  #B
    <br />
  </template>
```

```
  <script>Polymer('beer-item');</script>            #C
</polymer-element>
```

#A Defining a polymer-element
#B The <content> element is replaced with the beer-item tag's content
#C Registering the element with the browser

Other functionality in Polymer elements, like publishing properties and increasing the options for complex data binding are also available. As we go through the book, you'll see how to expose a published API for your element, letting separate elements communicate with each other.

1.5 Summary

Web component technologies build upon lessons learned from JavaScript component frameworks, by making the technology available natively in the browser. This lets you build your own elements and applications that take advantage of technologies previously only available internally to the browser vendors.

- Native support is appearing in evergreen browsers for Templating, Custom Elements, Shadow DOM, data binding with Model Driven Views, and componentization with HTML Imports.
- This lets you create modular, reusable HTML elements with encapsulated HTML Layout, CSS Style and Script.
- Polymer's platform.js polyfill library uses JavaScript to fill in the gaps where native support doesn't yet exist in all modern browsers.
- Using the Polymer framework on top of web components technology, you get web component boilerplate taken care of, and additional enhancements to the native technologies for more advanced data binding and publishing APIs.

As we go through the book, you'll see all these technologies in greater depth, as we build our Beer Rating application, a sample app that shows off how web component technologies work together. In the next chapter, you'll see templates and model driven views again, and learn how they work together to let you provide a declarative blueprint for dynamic HTML.