01

Introduction to Web Components

This chapter covers

* What is a Web Component
* What is the Shadow DOM?
* Structure and usage of a Template
* What is Polymer
* What is Firefox Web Components

This chapter describes the absolute basics of web components and Shadow DOM. It starts by giving you an overview of the new available techniques for web development using web components and Shadow DOM and it moves forward by showing various web components frameworks such as Polymer and Firefox. At the end of the chapter you will be aware of the various terminology and techniques used across the book and ready to take the first step into learning Polymer.

* 1. Web Components

First of all, what is a web component? A web component is a bundle composed of HTML, CSS and JavaScript files all packaged together to make it an atomic component capable to run independently and in the same exact way in any web page by carrying its own style, structure and intended behavior.

The standards adopted by a web component is a sort of new and specialized standard provided by W3C where we can finally introduce into the DOM (Document Object Model) personalized HTML tags. This approach can lead to a more semantic structure of the DOM where the tags introduced are self-explanatory and the complexity is hidden in the shadow. (We will see later in this chapter the meaning of Shadow DOM).

In summary, the web component technology lets you create your own HTML elements that can be reused over and over on all your applications and let you hide all implementations behind the hood.

1.1.1 A Practical Example

An HTML page is composed by a set of tags or elements. Each element carries a set of API (Application Programming Interface) and it is supported by a set of browsers. When you declare an element inside your HTML document, the browser analyzes that element and react accordingly by rendering the element on the screen in a specific way. Additionally, you use a scripting language like JavaScript to let the user interacts with your declared element or vice-versa.

Listing 1.1 Html page

<!DOCTYPE html>

<html>

<head>

<title>My First Polymer Page</title>

</head>

<body>

<div style=”color:#FF0000”>My First Polymer Page</div>

</body>

</html>

The markup shown in listing 1.1 represents a basic HTML page structure. An HTML page is always enclosed by the tags <html> and </html>. Inside these tags we have two main sections, the header <header> and the body <body>. In the header we include things like: metadata, stylesheets (CSS), JavaScript files and the title of the page. Inside the body area we add the content of our page.

The tag <div> has an attribute of type style where we declare the color of the text. In this case we have informed the browser to customize the text color of the content of the div element using the color RED.

Note about HTML code samples

In this chapter we are taking a look at how web components work and what are the most important characteristics. We will show multiple listings in this Chapter, if you want to try them just open your preferred editor such as Notepad, Notepad++, Sublime Text or similar. Copy/Paste the code in the listing and save it with an extension .html. Then load it into your preferred browser using a menu command like File/Open.

Let’s now take a step further by building a login screen where our user will be prompted to insert a username and password into two different input fields and a button to submit the login request to a remote server.

Listing 1.2 Html Login Form

1 <!-- Login form -->

2 <form action="/login" method="POST">

3 <label for="txtUsername">Username:</label>

4 <input id="txtUsername" type="text" placeholder="Insert Username">

5 <label for="txtPassword">Password:</label>

6 <input id="txtPassword" type="password" placeholder="Insert Password">

7 <button type="submit">Login</button>

8 </form>

The markup shown in listing 1.2 contains a <form> tag which is decorated with some special attributes. We used the attribute action to specify the url that will receive the data and an attribute method that specify the type of HTTP request (in this case an HTTP POST). Inside the form tag we have introduced two new elements: <label> and <input>. Let’s pay attention to the tag input for a moment. In the line 4 we specified that the browser should render a classic text input (using the attribute type=”text”) but in the line 6 we specified to render a password text input where the characters entered by the user are hidden with a pattern. This is an example of using the available APIs of a specific HTML element and how an element can react depending on how it is configured.

You may also have noticed that we are using two different terms, element and tag. The tag is just the opening or closing markup like <div> or </div> while an HTML element is the entire content including the opening and closing tag like <div>my context</div>.

A note about HTML tags documentation

The HTML is a markup language for describing web documents. HTML stands for Hyper Text Markup Language and is composed by a collection of tags. Each HTML tag describes different content inside a document and carries different APIs. All these APIs are constantly updated and documented by the W3C (World Wide Web Consortium) and available at the following website: https://www.w3.org/TR/html/.

1.1.2 How to import a Web Component

In the listing 1.2 we have created a form capable to send a login request to a remote server. We can consider this form a web component. If we want to reuse this form into our web pages, probably we would like to have it in the form of web component so that all boilerplate code should not be repeated over and over.

But how is possible to make a set of HTML elements a web component? Let’s have a look at how polymer provides to us a web component.

Listing 1.3 Sample usage of a web component

<html>

<head>

<meta>

<title>My First Polymer Page</title>

<!-- Import Polymer Framework -->

<script src="bower\_components/webcomponentsjs/webcomponents.js"></script>

<link rel="import" href="bower\_components/polymer/polymer.html">

<link rel="import" href="bower\_components/font-roboto/roboto.html">

<!-- Import Paper Card web component -->

<link rel="import" href="bower\_components/paper-card/paper-card.html">

<link rel="import" href="bower\_components/paper-button/paper-button.html">

</head>

<body>

<style>

html{

font-family: "Roboto",Helvetica,Arial,sans-serif;

background-color: #EFEFEF;

}

paper-card{

min-width: 350px;

}

</style>

<!-- Paper Card -->

<paper-card heading="My First paper card">

<div class="card-content">The content of my card.</div>

<div class="card-actions">

<paper-button>Action</paper-button>

</div>

</paper-card>

</body>

</html>

In the listing 1.3 we have created a web page that is using Polymer and more precisely it is using two web components available in Polymer, the paper-card and the paper-button. We also have created a custom <style> element to define some style in the web page. The result of this HTML markup is shown in the figure 1.1.

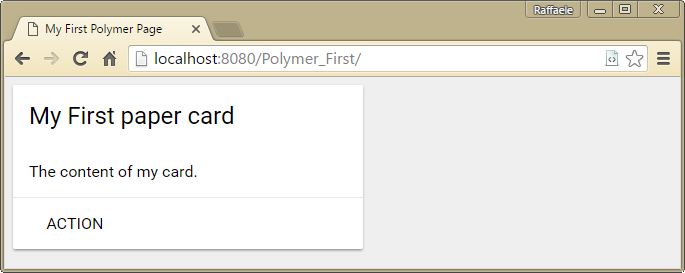


Figure 1.1 The result of the markup shown in listing 1.3

In order to start to work with Polymer web components we need to add two mandatory tags inside the <head> element, a <script> used to import webcomponents.js and a <link> used to import the polymer core component. Let’s focus our attention for a moment to the first link element. As you can see from Listing 1.3 we are declaring two different attributes into the element <link>. The first is rel and in our case we set it to import, the second one is href and we set it to our polymer web component HTML file.

Now that we have imported the polymer core component we can start to import extra components, in our sample we used <paper-card> and <paper-button>. If you browse the W3C documentation you will not find these two elements for the simple reason that they are not standard HTML elements but custom elements provided by polymer’s web components. This is the magic behind the web components, they encapsulate all boiler plate code and allows you to declare custom tags that hide the real markup using a technique called Shadow DOM. We will see later in this chapter what is the Shadow DOM, for now think about it as a sort of mask that allows you to hide the real markup that constitutes your web component.

1.1.3 Browser compatibility

The web components technique has been adopted since the beginning by Google Chrome and Mozilla Firefox teams, so it is absolutely safe to state that the current version of Chrome and Firefox fully support the web components programming technique. Also Opera is working to support as much as possible these techniques while Microsoft, with the new Edge browser, is supporting web components only partially and same apply to Safari.

Google Polymer (which is the main topic of this book) is a different story. The team of Polymer has worked really hard to provide as much support as possible across all browsers and at the time we are writing this book, this is the current status of support regarding Polymer 1.x.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Technique | Chrome | Firefox | IE10/Edge | Safari 8+ | Chrome (And.) | Safari (iOS) |
| Template | yes | yes | yes | yes | yes | yes |
| HTML Imports | yes | yes | yes | yes | yes | yes |
| Custom Elements | yes | yes | yes | yes | yes | yes |
| Shadow DOM | yes | yes | yes | yes | yes | yes |

Table 1.1 Polymer 1.0 browser compatibility

Polymer is also covering the gaps using a polyfills script called webcomponents.js, this is why the browser support of Polymer is wider than other web components frameworks.

If we look at the browser support for general web components techniques, without using the Polymer framework, the story is a bit different as shown in the table 1.2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Technique | Chrome/Opera | Firefox | Edge | Safari |
| Template | yes | yes | 13+ | 8+ |
| HTML Imports | yes | not yet | not yet | no |
| Custom Elements | yes | yes | not yet | partially |
| Shadow DOM | yes | yes | not yet | webkit |

Table 1.2 General web component browser support

The general idea is to support web components but framework like Polymer, as we said previously, takes a step further by adopting what is called the polyfills technique. 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. The polyfills for Polymer is called webcomponents.js.

1.2 Shadow DOM

We have seen that a web component is a set of HTML, CSS and JavaScript instructions all packaged together that can be used over and over on one or more web pages. We can consider a web component like a widget where all required code is encapsulated all together. Unfortunately, the HTML markup has some strict rules that we are forced to follow such as:

* An element of type <link> should be declared inside the element <head> of a web page because it must be loaded by the browser before the content of the page is loaded
* All external CSS files should be loaded with a <link> element inside the <head> element of a web page
* The ID of an element should be unique across the entire web page to avoid clash of duplicated IDs and unexpected behaviors
* And many more ...

Unfortunately, the DOM provided by a web component might get mixed with the rest of your web page, with things like CSS layout, JavaScript behaviors and other HTML elements’ rules such as duplicated IDs or wrong DOM tree, which may cause interference with the web component itself.

In order to address this problem, we need to use a technique called Shadow DOM. With Shadow DOM, elements can get a new kind of node associated with them. This new kind of node is called “shadow root”. An element that has a shadow root associated with it is called a “shadow host”. The content of a shadow host isn’t rendered; the content of the shadow root is rendered instead. Let’s make a practical example:

Listing 1.4 Example of Shadow DOM technique

<html>

<head>

<meta>

<title>My First Shadow DOM</title>

</head>

<body>

<button>Hello, world!</button>

<script>

var host = document.querySelector('button');

var root = host.createShadowRoot();

root.textContent = 'Hello, from Shadow DOM!';

</script>

</body>

</html>

In the listing 1.4 we have created a <button> element which should contain the text “Hello, world!” but instead it will be rendered with the text “Hello, from Shadow DOM”, as shown in the figure 1.2 below.

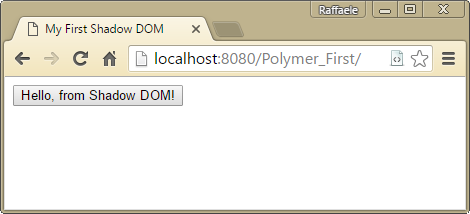


Figure 1.2 Button customized using Shadow DOM technique

The interesting finding is that if we inspect our DOM with a browser like Chrome, using the extension “Developer Toolbar” we will find out that the content of the <button> element is still “Hello, world!” because the DOM subtree under the shadow root is encapsulated. The figure 1.3 shows the HTML rendered by Chrome and highlight the shadow root and the shadow host contents.

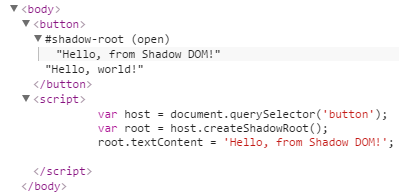


Figure 1.3 The HTML rendered by Chrome using the Shadow DOM technique

The main advantage of using Shadow DOM is given by the fact that each element has a child DOM that is visible, known as the “Shadow root”, and a parallel, invisible DOM, known as the “Shadow host”. Because your custom elements will be positioned into the Shadow DOM, the users (web developers) of your elements will not be able to modify or style them because the Shadow DOM is not accessible from the rest of the web page. It also avoids problems like duplicated IDs, for the same exact reason, because the content of your element is “hidden” by the Shadow DOM, so it can’t interfere with the rest of the web page content.

QuerySelector to select elements in a component

The figure 1.3 introduced a new JavaScript method exposed by the document object. The method document. querySelector([selector]). The querySelector() method returns the first element within the document that matches the specified group of selectors. It can be used in different ways, for example you can pass the name of the HTML tag like document.querySelector(‘div’) or the ID associated with the HTML element like document.querySelector(‘#myId’) or the CSS class associated to the HTML element like document.querySelector(‘.myCSSClass’). This method is very helpful, especially when working with the Shadow DOM, because it guarantees that the selection will happen only within the content of the shadow root.

In order to better understand this fundamental concept of the Shadow DOM we have create a more complex element in the listing 1.5.

Listing 1.5 Login Form as a Shadow DOM element

<html>

<head>

<meta>

<title>My First Shadow DOM</title>

</head>

<body>

<!-- Shadow host -->

<div id="orderForm"></div>

<!-- Shadow template -->

<template id="orderTemplate">

<form action="/submitOrder" method="POST">

<label for="txtQuantity">Quantity:</label>

<input id="txtQuantity" type="number"></input>

<button type="submit">Order</button>

</form>

</template>

<script>

var root = document.querySelector('#orderForm').createShadowRoot();

var template = document.querySelector('#orderTemplate');

var clone = document.importNode(template.content, true);

root.appendChild(clone);

</script>

</body>

</html>

In this code listing we have introduced a new element called <template> which is used to hold our Shadow root content. The first <div> is used to act as a Shadow host, so the content of the template will be rendered by the host. The JavaScript code at the end of the document is very simple and is informing the browser to:

1. Select the <div> host element and make it a Shadow root
2. Select the <template> element
3. Import the HTML content of the template
4. Load the HTML content of the template into the Shadow root element of the page

And the final result is shown in the figure 1.4.



Figure 1.4 HTML rendered by the Shadow root using a template

The <div> element is populated with the content of our <template> and it is rendered by the browser while the <template> element itself is not shown but it still available in the DOM.

1.2.1 Style encapsulation

We have already mention this concept, the most important feature of Shadow DOM is the automatic encapsulation of the template. What does it mean? It means that, also for CSS, you can feel sure that the style provided within an element that has been “Shadowed” will not be applied to the rest of your document.

When you work with Shadow DOM you have to keep this concept in mind, because the CSS rules defined inside a Shadow DOM are always “scoped” to the corresponding Shadow root element. Let’s have a look at a practical example.

Listing 1.6 Style Shadow DOM element

<html>

<head>

<meta>

<title>My First Shadow DOM</title>

</head>

<body>

<h3>Non styled area</h3>

<div>

<h3>Shadow host</h3>

</div>

<script>

var root = document.querySelector('div').createShadowRoot();

root.innerHTML =

'<style>' +

' h3 { color: red; text-transform: uppercase; }' +

'</style>' +

'<h3>Shadow root</h3>';

</script>

</body>

</html>

In this example we have created a Shadow host element with an <h3> element with the text “Shadow host” and later we inject into this element a different <h3> with some style. In the DOM we have also another <h3> element but this element is outside the Shadow root. What do you think it will happen here? The content of our Shadow root will get styled but the first <h3> will stay unaltered. This is happening because the style applied to the Shadow root has a boundary defined by the Shadow root itself. The figure 1.5 shows the final result and the generated DOM.

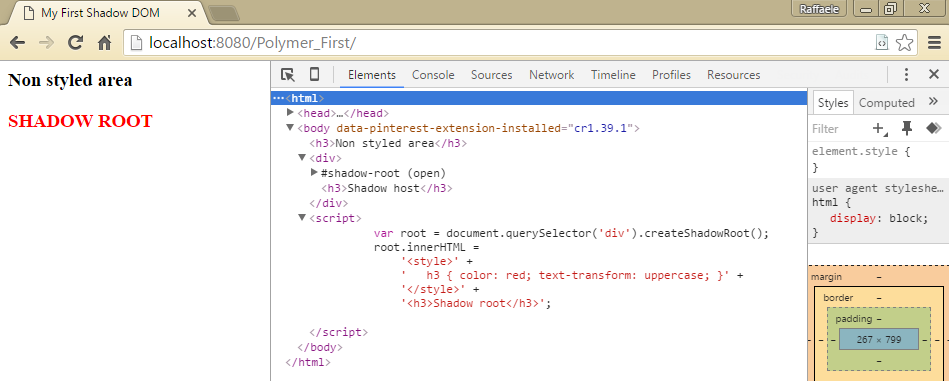


Figure 1.5 Custom style applied to Shadow DOM

Note about Google Chrome Developer toolbar

In the figure 1.5 we have introduced a new portion of the browser (Google Chrome) where we can inspect the DOM. This feature is available in all modern browsers (Chrome, Firefox, Safari, Edge) and it is called “developer toolbar” or similar. In Google Chrome or in any browser you can access the developer panel by pressing F12 or right click anywhere inside your web page and choose “view source” or “inspect element” depending on the browser you are working with. For the purpose of this book we kindly suggest you to use the latest Google Chrome browser.

1.2.2 Styles applied to the Shadow host

The Shadow host is the element that will contain the Shadow root element. In order to apply some styles to this element we need to use a specific CSS syntax. The :host keyword is used to target the Shadow host element and its content, while the syntax :host([selector]) allows you to target specific child elements inside the Shadow root.

Listing 1.7 Usage of the :host keyword

<html>

<head>

<meta>

<title>My First Shadow DOM</title>

</head>

<body>

<h3>Element to be transformed</h3>

<script>

var root = document.querySelector('h3').createShadowRoot();

root.innerHTML = '<style>' +

':host { text-transform: uppercase; }' +

'</style>' +

'<content></content>';

</script>

</body>

</html>

This time we have used a different method that we will see in details in the section “template” of this chapter. We have an <h3> element which is our Shadow host. We select the element and transform it into a Shadow root element. Then we inject a style and a new element called <content>. With the keyword :host we transform anything inside the Shadow host element to be upper-case. The <content> element in this case is used to display the entire content available in the Shadow host.

The :host keyword works in this way:

* Rules in the parent page container have higher priority than :host
* A style attribute defined in the Shadow host element has higher priority than the one defined in the parent page

The Shadow host can also be styled using specific selectors inside the :host keyword. For example, we may want to create a specific style for all <h3> inside the Shadow host and a specific one for all <h2> elements. The listing 1.8 contains this example.

Listing 1.8 :host selector keyword

Polymer

Firefox Web Components